From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday

Grossi's Festschrift, July 25, 2025, Venice, Italy

Edited by
Alessio Conte
Andrea Marino
Giovanna Rosone
Jeffrey Scott Vitter

Editors

Alessio Conte
University of Pisa, Italy
alessio.conte@unipi.it

Andrea Marino
University of Florence, Italy
andrea.marino@unifi.it

Giovanna Rosone
University of Pisa, Italy
giovanna.rosone@unipi.it

Jeffrey Scott Vitter
Tulane University, New Orleans, USA
University of Mississippi, Oxford, USA
jsv@vitter.org

ACM Classification 2012
Theory of computation → Design and analysis of algorithms; Theory of computation → Data structures design and analysis; Theory of computation → Data compression; Applied computing → Bioinformatics

ISBN 978-3-95977-391-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-391-1.

Publication date
August, 2025

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.Grossi.2025.0

ISBN 978-3-95977-391-1 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

A collection of articles written in honor of the 60th birthday of
Roberto Grossi, a great academic and much more.

Contents

Papers

From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday.
Editors: Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This Festschrift volume celebrates the 60th anniversary of Professor Roberto Grossi, in short Roberto in the following, accompanying a workshop scheduled on 25th July 2025, as a satellite workshop of SEA 2025, 23rd Symposium on Experimental Algorithms. The contributors of this volume, as well as the attendees to the workshop, are gathered from all over the world to express their admiration for Roberto as a researcher and educator, as well as their love for him as a person.

The contributions of Roberto to the academic world, particularly in the fields of algorithms and data structures for strings and graphs, have left an indelible mark on both his students and colleagues. Roberto's work is characterized by a profound commitment to research and teaching, and his ability to seamlessly integrate theory with practical applications has made his contributions invaluable. In this sense, we believe the collocation of the workshop, namely as a SEA satellite workshop, reflects the contamination of theory and practice which have pervaded the work of Roberto. His passion for knowledge and teaching extends beyond the classroom, fostering environments of intellectual curiosity and rigorous inquiry.

This collection of papers, gathered from colleagues, students, and collaborators, serves as both a tribute to his legacy and an exploration of the ideas and discussions that have shaped his academic journey. The variety of topics covered by this volume reflects the broad scope of his influence, showcasing the depth and breadth of his intellectual pursuits. It is our hope that this volume will serve as both a reflection of the profound impact Roberto has had on the academic community and as an inspiration for future generations to continue his work of intellectual exploration and discovery. Through this tribute, we honor a remarkable scholar, teacher, and mentor whose legacy will continue to inspire researchers over the years.

Beyond his scholarly achievements, Roberto is renowned for his unwavering kindness and generosity of spirit. Even during the busiest seasons of research and teaching, he always showed empathy, grace, and genuine care for those around him. His kindness has instilled a culture of respect, patience, and mentorship that will last beyond any publication or citation. The many contributions of this volume celebrate not only the academic achievements but also his kind and human impact on the community.

The story of Roberto: a very long CV made short

In the following, we are reporting a brief summary of Roberto's CV, mainly inspired by his home page and a very old CV we have found. This is far from being complete and updated as we realized that many of his achievements are not mentioned by him anywhere and we are aware of them only thanks to personal knowledge and the contributions will follow later in this volume.

Roberto got his degree (Laurea) summa cum laude in Computer Science (Scienze dell'Informazione) in the academic year 1987-1988 from Università di Pisa. He then obtained his Ph.D. degree in Computer Science (Dottorato di Ricerca in Informatica) from Università di Pisa in the academic year 1992-1993. He has been research and teaching assistant (Ricercatore) at Dipartimento di Sistemi e Informatica, Università di Firenze, from March 1993 to October 1998 and then Associate Professor (Professore Associato) at Dipartimento di Informatica, Università di Pisa, from November 1998 to October 2010. He is currently Full Professor (Professore Ordinario) at Dipartimento di Informatica, Università di Pisa, since November 2010.

From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday.
Editors: Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

He has been: Visiting scholar at Columbia University, Department of Computer Science, 1990-1991; Visiting researcher at AT&T Bell Laboratories, USA, 1993 and 1995; Visiting scientist at Berkeley, International Institute of Computer Science, USA, 1995; Visiting researcher at Aarhus University, Institute for Basic Research in Computer Science, Denmark, 1996; Visiting professor at the Universite de Marne-la-Vallee, Laboratoire de Informatique, France, 2001; Visiting researcher at Tohoku University, Graduate School for Information Sciences, Japan, 2002, and Haifa University, Department of Computer Science, 2005 and 2010; Visiting professor at the National Institute of Informatics, Japan, 2018-2019 and Tokyo University, Japan, 2024. He is member of ERABLE (European Research team in Algorithms and Biology, formaL and Experimental), INRIA Rhone Alpes, France, since 2005.

Roberto has been the advisor of the following Ph.D. students (in chronological order): Gianni Franceschini (U. di Roma "La Sapienza"), Ankur Gupta (Butler U., advisor Jeff Vitter), Iwona Bialynicka-Birula (Microsoft), Giovanni Battaglia (ION Trading, Dimoco), Alessio Orlandi (Google), Mauriana Pesaresi (1983-2008, passed away during studies), Rui Andre' Ferreira (Microsoft, Facebook, Spotify), Giuseppe Ottaviano (Facebook), Alessio Conte (NII Japan, U. di Pisa), Shima Moghtasedi (Milan), Luca Versari (Google), Giulia Punzi (U. Pisa), Giovanni Buzzega (U. Pisa).

Roberto has been (and still is) a very active member of the scientific and academic community, both nationally and internationally. He has been a member of the International Scientific Committee of the International Olympiad in Informatics. He has been the Treasurer of the Italian Chapter of the European Association for Theoretical Computer Science (EATCS). He has been the reviewer of international research projects for several Science Foundations, and international PhD Theses committees. He has been PC member of many prestigious conferences, to name a few ESA, SODA, FOCS, STACS, WWW, and CIKM. He is one of the few Italian authors cited in the second edition of the third volume of The Art of Computer Programming: Sorting and Searching by Donald Knuth for his contribution to the String B-Tree (along with P. Ferragina)

Roberto is internationally acknowledged for his outstanding and fundamental contributions to the area of the design and analysis of algorithms and data structures. His research interests are both in theoretical problems for core research and in applications and experimental work. Specifically, his interests are focused on algorithms for combinatorial pattern matching and mining on strings, sequences, trees, matrices, and graphs; design of algorithms and data structures for external and hierarchical memory; implicit, succinct and compressed data structures for (compressed) data sets; space- and time-efficient compressed indexing and fast searching in compressed text; text indexing and editing; multi-dimensional data structures; algorithm engineering for quick-access tables and dictionaries; routing algorithms for networks and robot.

Some testimonies

In this section, we report four extended testimonies about Roberto, respectively from his PhD Advisor, Fabrizio Luccio, from his long-term research collaborator Jeffrey Scott Vitter, from his colleague at Italian Olympiads in Informatics Luigi Laura, and from colleagues at the University of Pisa. In the resulting three orthogonal aspects of his work, first as a student, and later as a researcher, and educator, and colleague, Roberto is described as an extraordinary scholar, mentor, collaborator, and friend.

A great academic and much more

by Fabrizio Luccio

I met Roberto forty years ago in a classroom at the University of Pisa. He had enrolled to pursue a degree in Informatics, the Italian equivalent of a Master's degree in Computer Science, and was attending a course on Information Processing Systems that I was teaching. Roberto was a serious, responsible person, quickly mastering every aspect of the study material in all fields of CS and emerging as one of our top students. I was therefore particularly pleased when he asked me to supervise and discuss his final thesis for the Master's degree, in which the first signs of his groundbreaking research on the design and analysis of algorithms and data structures, for which he is now universally esteemed, began to emerge.

After graduating, Roberto was admitted to our PhD program under my supervision, then spent most of his academic life in our department, up to his current position as a highly regarded professor and an excellent research director. He initially joined a research group that I had led for many years. Eventually, the group split into two parts, and he took over the leadership of one of them. From the very beginning, our scientific work was carried out side by side, right up to the day of my retirement and, to some extent, even after that. As time went by, an unbreakable bond of friendship and mutual respect was established between us. Other colleagues, on this occasion, will present and discuss his research achievements and his commitment to teaching in various courses. I would like to highlight Roberto's fundamental role as a mentor to many young scholars in the PhD program and in guiding them toward research, as well as his openness to various activities where the experience gained during his academic career plays a crucial role.

First of all, Roberto has been and still is a great scientific leader, and this is perhaps the most important contribution that can be expected of a university professor. He has built relationships and collaborations with major research groups in Italy and abroad. Many excellent PhD students have studied or are studying under his guidance in our department, gaining top-level training and putting the acquired expertise and genuine enthusiasm at the service of the scientific and technical life of our country and abroad. Furthermore, several students who simply completed their Master's thesis under his supervision and then pursued professional or research careers have been able to approach their work on an excellent foundation of scientific knowledge.

Another activity of Roberto, which had significant international relevance, was the direction of training for the Informatics Olympiad, where young Italian university students had the opportunity to compete on equal footing with their peers from many other countries. Speaking with some of them years later, I appreciated their enthusiasm for that experience and the importance of the memory it left. Among other things, it is thanks to Roberto that a version of these Olympiads was organized in Italy. On a local level as well, he was among the organizers and evaluators of informatics contests for a large number of high school students eager to test their knowledge against others and to seek guidance on which university studies to pursue.

Finally, I want to highlight Roberto's commitment to initiatives aimed at enhancing both university and school-level teaching, as a contribution to education in the broader sense. Of particular importance is the ongoing professional development of teachers at all levels of education, achieved through courses that allow them to discuss the latest developments in their subjects and gain new inspiration for teaching.

In conclusion, Roberto has approached and developed his profession in the most intelligent and noble way, and everyone should be grateful to him for this.

My great friend and collaborator!

by Jeffrey Scott Vitter

I had the great fortune to meet Roberto 30 years ago. He and colleague Paolo Ferragina visited the USA to present their innovative way to index strings in external memory at the 1995 *STOC* conference. Much of my work was focused at that time on the field of external memory algorithms, in which the main bottleneck is the I/O communication between a small but fast memory (such as RAM) and a large but slow memory (such as disk, which utilizes block transfer). Roberto and Paolo's approach for strings met the optimal I/O bounds for processing strings, and they did it in a very elegant way. Shortly afterward, we had the opportunity along with our collaborator (the late) Lars Arge to work together on sorting strings in external memory, which we presented at the 1997 *STOC* conference.

Through the years, Roberto has truly become a great friend and collaborator. I don't think I know anyone who is more receptive and eager than Roberto when it comes to interacting with colleagues and developing new ideas. He gives of himself selflessly and is full of encouragement. And on a personal level, he is such a positive person. Sharon and our kids had great times visiting him and his family in Pisa during our sabbatical in 1998–1999, which was nearby at INRIA in Sophia-Antipolis, France.

Figure 1 After a fun dinner in August 1998 with our families in Pisa.

As with all my favorite reminiscences, food and wine play an important part. Roberto and Paolo introduced me during my sabbatical to the delicacy of *alici*, which we had for lunch one day during a visit to Pisa. I make sure to order them whenever I'm in Italy.

And I think I can legitimately credit Roberto as the person most responsible for my ever-growing interest in wine. During our sabbatical visits, Roberto shared that his wife Antonella's dad – who was an avid wine connoisseur and collector – had to give up drinking wine for health reasons. And in order to avoid temptation, he began giving away his extensive collection, much of it to Roberto and Antonella. In the process, Roberto gifted us with several bottles of stellar wine, including a 1967 Giordano Barolo, which figures prominently in our 1998 Christmas card.

Sharon and I still talk about it to this day; that's how memorable both Roberto and the Barolo were! Barolo wines are notorious for containing a lot of sediment, so the wine generally needs to be filtered first, and then it's best to decant it for an hour for each year of

Figure 2 At a 2009 dinner in Pisa with Roberto and family and Fabrizio Luccio.

Figure 3 Our 1998 Christmas photo from our terrace in Valbonne, France, with Roberto's gifted bottle of 1967 Giordano Barolo!

age. So given that it was 31 years old at the time, we filtered the wine and then aired it in a carafe for about a day and a half. And what an amazing wine it was!

On a professional level, Roberto is the person responsible for bringing me into the field of compressed text indexing. I was always interested in data compression – in the text, image, and video domains – but Roberto helped indoctrinate me into the fascinating world of stringology. I loved the elegant theory and was "all in." Roberto was the lead in our 2000 *STOC* work on compressed suffix arrays, in which we presented the first provably succinct text index whose space requirement was within a constant factor of the input size. The space savings was a factor of $\log_{|\Sigma|} n$ over the well-known suffix tree and suffix array data structures, where $|\Sigma|$ is the size of the alphabet.

Figure 4 At Bertinoro in 2006, with Gonzalo Navarro, Kunihiko Sadakane, Rahul Shah, and Roberto.

Roberto was also a great mentor to my graduate student Ankur Gupta. A couple of years later, the three of us developed the wavelet tree rank/select data structure, which generalized the famous RRR data structure for bit arrays in order to handle arbitrary alphabets Σ. Like RRR, the wavelet tree data structure provided 0th-order entropy compression with leading constant factor of 1, and when the portions of text sharing a common context of length k were each encoded in that manner, the resulting global structure provided kth-order entropy compression. Not only that, but the space bound had the constant factor of 1 in front of the leading nH_k term, which answered an open question of Giovanni Manzini and established asymptotic optimality for both compressed suffix arrays and the FM-index.

I feel very blessed to have had the opportunity to collaborate over the years with Roberto and witness his great understanding of the field as well as his scientific creativity. It was always fun to be at a conference with Roberto because the energy that developed led to many interesting collaborations. But mostly I am thankful for his friendship and for his being the person he is. Roberto, I hope this Festschrift lets you know how much you mean to us!

Twenty Years with Roberto: An Olympic Journey

by Luigi Laura

I have known Roberto for 25 years or slightly more. I probably met him when I was a PhD student in Sapienza, but our intense interaction started when Alberto Marchetti-Spaccamela came into my office in 2005, asking if I wanted to help with the Italian Olympiads in Informatics (OII): "it will not take you more than a few hours per year" Alberto said. It quite soon became my principal activity and still is today. Roberto was the head of the Italian organisation, for the scientific/educational part. In 2012 we also organized, in Italy, the International Olympiads in Informatics: 85 participating countries, with four students and two teachers (team leaders) per country. After that very tiring experience, Roberto stepped back (as much as he can, he has been also later very involved) and we switched roles.

Figure 5 Pisa, 2007: a consciousness-raising sessions; from left to right Sebastiano Maggiolo, Roberto Grossi, Nicola Pierazzo and Romeo Rizzi.

Figure 6 Volterra, 2010: Alessio Guerrieri and Roberto Grossi are distributing food to the students. (Photo by Giuseppe Ottaviano)

Thus, over the last 20 years we have worked together extensively, and I can proudly say that the overall OII organization grew a lot under "our" guide (or despite it, depending on the point of view). We spent a lot of time together, and I can tell dozens of stories about Roberto. A few of them can be reported here (don't worry Roberto, the couple that you fear will not be included, nor will the "few-lines-of-prolog-solving-every-problem" induced anger one).

Before diving into the stories, here's a quick crash course on the organization of OII: we start (first phase) with approximately 15,000 students in their own schools... then (second phase) the best 1-2k compete in 40-50 schools distributed across all the Italian territory, and (national final) the top 100 students participate in the final contest, in a school (or, sometimes, in a university) that changes every year. Then, the top 20 from the final are selected to participate in 4 periods of few days of training and selection, until we have four students to send to the International Olympiads in Informatics (IOI).

The majority of my interactions with Roberto have been in the training camps and when we went together to IOI (2007 Croatia, 2008 Egypt, 2010 Canada, and 2011 Thailand).

Figure 7 The Italian Team at IOI 2008 in Egypt: from left to right, first row: Matteo Boscariol, Massimo Cairo, Luigi Laura, one of the guides of the team, Marta Genoviè De Vita. Second row: the other guide, Roberto Grossi, Giovanni Mascellani, and Paolo Comaschi (Photo by Giuseppe Ottaviano)

A tradition of the training camps, started by Sebastiano Vigna, in charge of OII before Roberto, was the "Consciousness-raising sessions" (sedute di autocoscienza in italiano), a post dinner activity with all the students, where each participant explained their first experience with coding in an Alcoholics Anonymous-style format. In Figure 5 we can see Roberto in one of the consciousness-raising sessions from 2007 in Pisa.

The most amusing thing about Roberto regarding the training camps was that the technical team would often pull all-nighters to prepare the competition for the students the next morning. At a certain point in the evening or night, Roberto would finish his part and have nothing else to do, but he felt bad about abandoning the others, so he would sleep on a bench. Roberto spent quite a few nights in the early years of the training camps sleeping on benches. In Figure 6 we can see Roberto in Volterra during a training camp.

The most interesting single experience, together with Roberto, has been in IOI Egypt, in 2008. Egypt was under Mubarak's government. The venue of the IOI competition was the so called Mubarak Educational City, a complex of six large buildings, in a beautiful lawn (in the desert!), surrounded by a high wall, guarded by armed sentries, complete with the traditional changing of the guard. It was a suspicious location. We, as participants of IOI, had a map of the complex, and in the map one of the six buildings was not present! Most notably, whenever one of the participants tried to approach the *unmapped* building, someone would 'coincidentally' emerge from it, apparently to make a phone call or have a cigarette, who would politely invite the participant to move away from the building.

Now, if you know Roberto you might be aware that he has some inclination toward conspiracy theories (he would not state it in this form). So Roberto started developing a theory where the unmapped building was a missile base (in retrospect, he might be right). Instead of trying to reassure him, I would escalate with phrases like "you're right, but at this point, if they realize you know, they might decide to kidnap you or eliminate you. If you disappear, I'll immediately take our students to the Italian embassy. I already warned them that if both of us disappear they need to run and search for a safe place". I had a lot of fun playing the spy game with Roberto in those days.

Giuseppe Ottaviano, who shared the room in Egypt with me and Roberto, might focus on the fact that every morning he complained to us, saying something like "you both snored all night, sometimes in unison, sometimes taking turns, and I didn't sleep at all". In Figure 7 the Italian team at IOI 2008.

Over these years we have seen many young people grow: we saw them first participate in competitions and then, while studying at university, they collaborated with the Olympics as tutors; many did a PhD, continuing to collaborate. Some of them, even after completing their studies and finding work, have continued to collaborate. Among them, we mention Giorgio Audrito, soon to be an associate professor at the University of Turin, who has been the scientific coordinator of OII since 2020. Luca Foschini and Alessio Signorini met at OII in 2001 and few years ago founded Evidation Health, a startup now valued more than a billion dollars. We'll mention, in random order and with no claim to completeness, just a few others: Giuseppe Ottaviano works for Meta (formerly Facebook); Alessandro Dovis, Stefano Maggiolo, Andrea Ciprietti, Nicolò Mazzuccato, Alessio Orlandi, Marco Ribero and Luca Versari work at Google, the first two in London and the others in Zurich. Giovanni Campagna is in San Francisco, hired by Bardeen, while Gabriele Farina is an Assistant Professor at MIT, after working for a year as a Research Scientist at Meta. At MIT we also find Giada Franz, currently as an Instructor. Alice Cortinovis is an Assistant Professor at Stanford, where Matilde Padovano is completing a PhD in Machine Learning, while Federico Glaudo is a Postdoctoral Research Associate at Princeton. MIT, Stanford and Princeton: academically we can't complain, and on the corporate front as well, between Google, Meta, Microsoft (where Ottaviano worked in the past), DeepMind (Matilde Padovano's internship), Twitter (Emanuele Rossi, who now works for VantAI), we can say that the former Olympians are doing quite well for themselves.

Those young students might not have written a paper about succinct data structures or enumerating cliques in temporal graphs, but I am sure that at a certain point of their early careers they have been inspired, as much as I had, by the words and the works of Roberto. This is his legacy. Thank you Roberto!

Roberto: Master of Algorithms and Ceremonies

by Giovanni Buzzega, Alessio Conte, Veronica Guerrini, Andrea Marino, Giulia Punzi, Nadia Pisanti, Giovanna Rosone, Giorgio Vinciguerra

From tales of travels to work discussions, to fond memories of his students, Roberto always has a story to share, topped off with some good tea or coffee. His office has become famous for brewing traditional Japanese tea, a craft that he learned and perfected during his many travels to Japan, even attending a traditional tea ceremony. Recently, he moved on to specialty coffee as well, approaching it with a true scientific mindset and experimenting with different filters, techniques, and coffee varieties. During work breaks or meals, he is more than happy to prepare tea, infusions or coffee for anyone around. The kind and generous spirit that he shows in sharing with others is the same that he brings to research, teaching, and everyday life. Anyone who knows Roberto has seen his ability to make everyone feel welcome, at ease, and treated as equal, whether they are visiting scholars from around the world or first-year students.

It was during one of the shared meals in our department that we celebrated Roberto's 60th birthday. He brought some pastries to have a small celebration in the office, thinking that nobody knew about his birthday. Not only were we and our colleagues fully aware of it, but we had also prepared a surprise for him.

Figure 8 An overjoyed Roberto hugging his new grinder tightly, with some of his friends and colleagues. From left to right: (standing) Filippo Geraci, Roberto Grossi, Giovanni Buzzega, Veronica Guerrini, Alice Cortinovis, Francesco Landolfi, Giorgio Vinciguerra; (on the sofa) Alessio Conte, Linda Pagli, Giovanna Rosone, Giulia Punzi, Luca Mencarelli; (on call) Andrea Marino and Rossano Venturini.

There is a story behind it all, and it started during his trip to Tokyo in July 2024. Roberto was attending a seminar at Kanda lab, the lab of his collaborator and long-time friend Takeaki Uno. Kanda lab is Roberto's home-away-from-home when he is in Japan, so he was, as usual, offering to prepare coffee for the seminar participants. When approaching the kitchenette of the lab, he noticed a high-end coffee grinder belonging to the lab postdoc Yuta. It was, as Roberto himself later put it, "the Ferrari of coffee grinders". The postdoc, a fellow coffee enthusiast, saw Roberto's excitement and gladly agreed to let him prepare the ritual coffee using his grinder. It was only after Roberto had happily completed the grinding, boiling, filtering, and handing out of cups to his colleagues that he realized there was no more coffee left for him. At the time, he hid his disappointment quite well; still, in the many meals we shared in Pisa after that incident, he often expressed his regret at not having tasted such a premium-ground coffee when it had been so close at hand.

When his fellow coffee drinkers, colleagues, and friends found out that his birthday was approaching, a plan was set in motion. We asked Yuta for the specific coffee grinder's model and managed to have the same grinder delivered from a specialty coffee shop just in time for his birthday. On that day, imagine the surprise of Roberto, going from thinking that he was quietly announcing his birthday to oblivious people, to finally being able to prepare the coffee that he had been craving for months... the picture (Figure 8) says it all!

Roberto, this is for your $(2^6 - 4)$th birthday, with wishes for many joyful coffee-grinding moments ahead.

Open Messages to Roberto

In this Section we have collected some open messages to Roberto from some of his colleagues and his friends. They talk about their experience with Roberto, whether personal or professional. This collection of heartfelt tributes highlight his impactful contributions to theoretical computer science – especially in string algorithms, data structures, and succinct indexing – as well as his inspiring personality. In particular, he is described by many as kind, humorous, down-to-earth, and joyful in collaboration. Praised for making research fun, inspiring, and deeply human, often accompanied by thoughtful touches – like sharing fine Japanese tea or musical interests. These tributes show how his intellect and humanity have deeply shaped the lives and work of many across the globe.

I arrived in Florence in November 1997 and had to share a work room (not really an office) in an apartment with Roberto and Alberto Del Lungo. Roberto's surprise when he saw me enter the room with a tennis bag made me immediately understand that it was the beginning of a beautiful friendship. And so it was, but it was also the beginning of a superlative collaboration with one of the most brilliant researchers I have ever worked with. Our first paper together was about the problem of the IP address lookup: Leandro Dardini had a very nice idea, Roberto quickly formalized it and made a great result, and I .... well, I loved being part of it. And I remember that, when we discovered on the ESA website that the paper had been accepted (surprisingly, we had not yet received an acceptance letter), we started jumping and hugging each other in the room, as if Italy had won the world championship. Yes, that's what I miss the most: the enthusiasm that Roberto and other friends and colleagues allowed me to experience simply by proving theorems and writing code. Thanks, Robi!

Pilu Crescenzi

Dear Roberto,
The first memory that I have of you dates back to 2015, when I attended your course "Algoritmica 2" during my master's studies. The course had an unconventional structure that I enjoyed a lot, especially for the exam format. However, what struck me the most was your availability to students and your charming personality, which made student-professor interactions very smooth.
This trait of yours became even more evident to me when, a few years later, you became my first academic mentor, as I started working on my master's thesis under your supervision. I remember that, since day one, you asked me not to address you in a formal way by saying: "Dammi del tu, siamo colleghi adesso." You were able to make me feel like I was really talking to a peer, even though I was just a student, while you were a well established and acclaimed professor. I consider that to be at the very beginning my academic career, and I will always keep close to my heart the sweet memory of those spring mornings spent in your office, sipping from a cup of fine Japanese tea while tackling research questions on strings and graphs.
Years have passed since that spring in 2018, and if I grew into a more mature researcher and person it is also thanks to your teachings. I am thankful for your trust in my ideas but, above all, for your constructive criticism. Thank you for so explicitly telling me that I was wrong when you thought so. I always admired this ability of yours; not so much for spotting technical mistakes, but rather for identifying a sterile line of research, as it is way easier to fall in love with new ideas than it is to divorce them.
My warmest regards and wishes!

Massimo Equi

Epic times. I'll never forget those legendary mornings in my apartment while we were preparing our STOC '95 paper on String B-trees, taking inspiration from "the opening of a shutter" that, when discovered, brought both of us to be very very tight in the morning schedule .... I do not know whether those events sparked our creativity, or it was just good luck. Who knows – but man, those were epic times, Rob!

Paolo Ferragina

Dear Roberto,

We've watched you dance through **trees** and **strings**,
And build such clever **index** things!
You squeeze in **bits** where none should fit,
Then blaze new trails with endless grit.

From FOCS to SODA, peer review –
The world of **theory** knows what you do.
You set the tone with **Olympiad** flair –
While others blinked, you were just there.

You mapped out **patterns** hard to find,
And surely left no **clique** behind.
And in the land of cycles and paths,
You conquered even tricky **graphs**.

At Pisa's halls you made your stand,
With **succinct** structures close at hand.
Your **code** is clean, your **bounds** are tight,
Your papers? Always a delight!

And though we joke in rhyme today,
We mean each word – **chapeau**, we say!

Irene Finocchi

Hi everyone,
My connections to Italy are very strong. For over 30 years I spend part of every summer in Italy with one or two exceptions. I am actually in Italy right now. I have got to know and to interact with dozens of Italian computer scientists. Giorgio Ausioello sent me Pino Italiano, Alberto Apostolico sent me Raffaele Giancarlo. Pino and Raffaele were fantastic PhD students. Fabrizio Luccio sent me Roberto Grossi over thirty years ago. He was already in the middle of his PhD studies. If I remember correctly he spent a year at Columbia. Everybody liked Roberto. We have stayed in touch ever since and I have followed Roberto. He has had a very impressive career. He has had some beautiful results in stringology. Congratulations Roberto for your 60th birthday.

Zvi Galil

My passion for efficient algorithms has its roots in a beautiful algorithms course taught by Roberto. After the course I decided to ask Roberto to supervise my thesis. What I didn't know at the time was that that professor would later have become a good friend. In recent years, I've taught a few hours of one of Roberto's courses. To all the colleagues who asked me why I enjoyed doing it (at CNR we are not required to do teaching) I always replied that the payoff of that course was the lunch with Roberto after class.
Thank you, Roberto for the thousands of scientific and human things I learned from you. I've learned that I am more likely to prove P = NP (not sure about that) than I ever get to argue with you.

Filippo Geraci

Roberto personifies stringology – in fact, I can think of few others who deserve such a prominent recognition within the community. I am certain that others will speak of his well-deserved scholarly success, so I will talk more about how instrumental he was in my own career path. I remember Roberto fondly in his role as my second advisor. We worked extensively for several years on compressed suffix arrays and related work during my Ph.D. I had the pleasure of learning from his uncanny ability to attack problems in innovative ways until the solution was laid bare. Often, the elegance of the final result led to key insights in related problems. And you can 100% bet that is exactly what we did next.
Roberto also hosted my postdoc shortly after I was married. My wife (as of just a few days prior) and I stayed in Pisa for several months. I clearly recall one morning when we were working out how to represent the information-theoretic minimum space of the BWT on a whiteboard dense with combinatorics. It was electric fun, and there's no other way to describe it. Nevertheless, whenever I remember Roberto, I remember a warm, caring face, and a genuinely gracious host . . . and an Italian that did his best to avoid "Italian food" when visiting the United States. I have now eaten enough proper pasta from the experts to know that he was right all along. Congrats Roberto, on an amazing and awe-inspiring legacy!

Ankur Gupta

With Appreciation. Congratulations, Roberto, on your 60th birthday and such an influential career!
When I spent a year at KU as a visiting scholar with Jeff Vitter (too many years ago for me to say!), I learned so much from your research on compressed suffix arrays and graph algorithms. Your work has been a big influence on and inspiration for my research career.
I wish I could be in Venice in person, but unfortunately have conflicts that make that impossible. I hope my contribution with Jeff on graph edit distance heuristics will serve to memorialize the occasion. I look forward to meeting you sometime in person.

Hongwei Huo

Dear Roberto,
You made truly remarkable contributions to data structures, pattern matching theory and enumeration complexity, among other topics. Although we never collaborated on a common project, I keep a wonderful memory of our numerous meetings and discussions.
With my congratulations for your achievements, I wish you best of luck in all of your future scientific endeavors, and many happy returns with new beautiful discoveries!

Gregory Kucherov

To Roberto, a brilliant colleague and an even better friend – we've shared ideas, papers, research projects, and . . . sushi!

Giuseppe F. Italiano

In the (relatively short) time I've had the chance to work with Roberto, I discovered we share more than just a name – we've also got a common love for science and the sea. But here's the catch: we approach both with very different levels of bravery. When it comes to the sea, I stick to sunbathing and the occasional cautious swim – and only in summer, of course. Roberto? He has no problem throwing himself into pre-dawn winter dives, clad in nothing but a wetsuit. Just thinking about it makes me shiver. Same story with research. Roberto dives headfirst into algorithms, data structures, and code – no clue if he wears a wetsuit there too, but one thing's certain: he's not afraid to go deep. Meanwhile, I sit on the metaphorical shore, waiting for him to come back up with something useful that we poor biologists can use to save our skins. For his 60th birthday, I send him my loudest, most thunderous best wishes – loud enough to reach him all the way down in whatever deepness he's currently diving in his latest challenge with himself.

Roberto Marangoni

Figure 9 In Pisa, on the terrace of the office of Linda Pagli. At that time the Department of Computer Science of University of Pisa was in Corso Italia. With Fabrizio Luccio, Linda Pagli, Elena Lodi, Roberto, and Geppino Pucci.

It was 2003 where I was in Japan and Veli Mäkinen was emailing me about an idea he had. His enthusiasm was contagious, and I got engaged in his line of thought about a data structure that, by hierarchically partitioning the alphabet, could represent a sequence in succinct space and recover the symbols in logarithmic time. After a few days he sent me a final email like: "Never mind, look at the SODA paper of Grossi et al. this year". Yes, the wavelet tree was there, buried in a lot of other interesting stuff. Over time, I even managed to understand that paper. I remember when, again with Veli, we were writing in 2006 a survey on compressed full-text indexes, and I wrote the section for their paper. Veli said, impressed, "You actually went through it!".
That 2003 paper was my first "encounter" with Roberto. Over time, I had the pleasure of meeting him in person in many occasions, and we even wrote a paper together. Roberto was always easy to deal with, agreeable, and full of interesting ideas. I'm always eager to see him wherever life puts us together.
Happy Birthday, Roberto!

Gonzalo Navarro

Many wishes Roberto, colleague of many years but above all dear friend!

Linda Pagli

Dear Roberto, the three of us have walked side by side since the very beginning of our respective research journey. Indeed, we are siblings, since we share the same academic father and extended family. Due to this life-long acquaintance, we can authoritatively affirm that you are an outstanding researcher as well as an extraordinary human being, and a great friend. We wish you that the next 60 years of your life will be at least as successful and productive.
With love and affection,

Andrea Pietracaprina and Geppino Pucci

I came to know Roberto when I was a PhD student. This was longer ago than I like to admit... his hairs were longer than what I have now, and mine were all black!
Roberto was then just back in Pisa as an associate professor after having been assistant professor in Florence for a few years. Hence, we were both working in Pisa but - ironically - our collaboration started... in Paris!
I was visiting Marie-France Sagot at the Pasteur Institute for a few months, and he was visiting Maxime Crochemore in Marne-la-Vallée (at a University now named Paris Est). I was a regular visitor at Marne, where I had been master student a couple of years back, and in particular both Marie-France and I used to attend the weekly Tuesday seminars there. One of these Tuesdays Roberto was the speaker. I took the opportunity to introduce him and Marie-France to each other, and we shared with Roberto and Maxime the problem we were investigating. Working with him and Maxime was fun, and we end up with very nice results. That was the first one of many joint papers with many different co-authors, and also the beginning of a friendship.
In these years, I have been admiring Roberto's curiosity, passion, and brilliance when it's about doing research, and the way he conveys this to his students for whom he is an excellent mentor. I find it very significant that his former students, whether now in academia or industry, keep on coming to visit him even years after earning their degree.
Last but not least, conversations with him are not just about strings, trees or graphs. Roberto has many hobbies and passions: there is hi-fi sound, bread making, coffee grinding, swimming, green tea tasting... one never gets bored with him.
In Italy we have a saying that well applies to Roberto, I believe: *se non ci fosse bisognerebbe inventarlo!* (translatable into "if he weren't there, then someone should invent him").... Happy Birthday, Roberto!

Nadia Pisanti

Wishing you a very happy 60th, Roberto! You've reached a wonderful milestone with the curiosity, energy, and sense of FUN of someone half your age! Ever since we met at that memorable Dagstuhl seminar on data structures in the now-distant '90s, it's been a pleasure to share ideas, laughs, and inspiring conversations with you. May the next decades be just as rewarding – and just as much FUN!

Guido Proietti

With sincere esteem and affection, and deep gratitude for each opportunity shared to collaborate with fulfillment as educators and researchers.

Romeo Rizzi

La storia siamo noi,
Collaborazioni e
Amicizia anche

Marie France Sagot

My first impression of Roberto was that he was an awesome researcher. The Phi function and wavelet trees were these magical structure. But when I started to interact with him (working on some research and then a joint paper), he came across not only as one of the smartest persons, but also as a kind, wise, humorous and down-to-earth person. I believe he has influenced many and I am one of them. At this milestone, I express my warm wishes to Roberto for best of the health, so that we see many more cool things coming out from his endeavors.

Rahul Shah

I would like to wish Roberto the happiest birthday ever! I have been involved few times in organizing conferences with Roberto and it has always been a great experience! Roberto and I share another passion: high-end music! I wish Roberto many cable building experiences :-)

Fabrizio Silvestri

Dear Roberto,
Congratulations with your 60th birthday. And thanks a lot for what you are to me and lots of other people. It is an enrichment of my life to know people like you. Working with you has always been a pleasure and made me realize how privileged I am working in this field of applied mathematics. It has given me a wonderful. By the joy you radiate I am sure the same feeling holds for you. So you are heading for the last 10 years of your academic life in the Italian system. Enjoy it fully!!!
With friendship and admiration,

Leen Stougie

You have shown me how to make international communication fun, pleasant, and fruitful through high-quality research discussions and exchanges. Thanks to you, our lab has grown, and many of our members have now made good friends overseas and are able to enjoy top-level research. I am truly grateful for this.

Takeaki Uno

Working and interacting with Roberto has always been an inspiring experience: he brings his passion for research and his deep insights to every conversation. On top of that, his great sense of humor makes every collaboration not only productive but genuinely a lot of fun. Wishing you a fantastic birthday, Roberto!

Fabio Vandin

Happy 60th Birthday, Roberto!
In Japan, turning 60 is a very special milestone known as Kanreki (還暦). It's a time for celebration, symbolizing a rebirth and a fresh start. As part of the tradition, the person celebrating Kanreki often wears red items, such as a red hat or clothing, to signify this significant occasion and ward off evil.
Wishing you all the best on your Kanreki, Roberto!

Kunihiro Wasa

When I joined Dr. Jeff Vitter's group as a postdoc in 2009, compressed data structures for string search was a new and unfamiliar field to me. I began my learning with Roberto's groundbreaking work: his papers on the compressed suffix array (with Jeff Vitter, STOC2000) and the high-order compressed suffix array (with Ankur Gupta and Jeff Vitter, SODA2003). Later, I had the privilege of collaborating with Roberto and Jeff on an experimental study of the wavelet tree, exploring various coding techniques for the bit arrays on the tree nodes.
I was deeply impressed by Roberto's humility and meticulousness. He thoroughly reviewed every data table in the paper, asking insightful questions that often led to improvements or corrections. When a visa issue prevented me from traveling to Italy to present our paper, Roberto's support was unwavering. I vividly recall him taking multiple trains from northern to southern Italy to present the work himself. Roberto, thank you for your selfless support – it meant so much, especially as you had little to gain. My warmest congratulations on your remarkable career and your 60th birthday!

Bojian Xu

Organization of the volume

The papers included in this volume are peer-reviewed scientific contributions dedicated to Roberto, reflecting the diverse areas of his research interests. Of course, even though these papers are related to Roberto's research, they do not include him as an author since this is meant as a surprise for him. The papers fall in two categories: *research* and *education*. The research category includes papers related to the research fields Roberto has contributed to, while the education category includes papers related to Roberto's activities in teaching, training, and educational development.

July 2025
Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter

List of Authors

Jarno N. Alanko (6)
Department of Computer Science,
University of Helsinki, Finland

Marcella Anselmo (5)
Dipartimento di Informatica,
Università di Salerno, Italy

Giorgio Audrito (11)
University of Turin, Italy

Giulia Bernardini (9)
University of Milan, Italy

Anna Bernasconi (3)
Dipartimento di Informatica,
University of Pisa, Italy

Philip Bille (6)
Technical University of Denmark,
Lyngby, Denmark

Nathaniel K. Brown (10)
Department of Computer Science, Johns
Hopkins University, Baltimore, MD, USA

Giovanni Buzzega (20)
Department of Computer Science,
University of Pisa, Italy

Marie-Pierre Béal (14)
Univ. Gustave Eiffel, CNRS, LIGM,
Champs-sur-Marne, France

Giuseppa Castiglione (5)
Dipartimento di Matematica e Informatica,
Università di Palermo, Italy

Huiping Chen (9)
University of Birmingham, UK

Xiaoyang Chen (1)
Department of Computer Science, Xidian
University, Xi'an, China

Alessio Conte (19, 20)
University of Pisa, Italy

Nicola Cotumaccio (17)
University of Helsinki, Finland

Maxime Crochemore (14)
Univ. Gustave Eiffel, CNRS, LIGM,
Champs-sur-Marne, France;
King's College London, UK

Massimo Equi (7)
Aalto University, Finland

Paolo Ferragina (15)
Department L'EMbeDS, Sant'Anna School of
Advanced Studies, Pisa, Italy;
Department of Computer Science,
University of Pisa, Italy

Manuela Flores (5)
Dipartimento di Bioscienze e Territorio,
Università del Molise, Campobasso, Italy

Travis Gagie (10)
Faculty of Computer Science,
Dalhousie University, Halifax, Canada

Dora Giammarresi (5)
Dipartimento di Matematica,
Università Roma "Tor Vergata", Italy

Raffaele Giancarlo (15)
Department of Mathematics and Computer
Science, University of Palermo, Italy

Veronica Guerrini (20)
Department of Computer Science,
University of Pisa, Italy

Inge Li Gørtz (6)
Technical University of Denmark,
Lyngby, Denmark

Hongwei Huo (1)
Department of Computer Science,
Xidian University, Xi'an, China

Giuseppe F. Italiano (4)
LUISS University, Rome, Italy

Seungbum Jo (12)
Chungnam National University,
Daejeon, South Korea

Manas Jyoti Kashyop (4)
Indian Institute of Technology Bhubaneswar,
India

Yasuaki Kobayashi (8)
Faculty of Information Science and Technology,
Hokkaido University, Sapporo, Japan

Athanasios L. Konstantinidis (4)
University of Ioannina, Greece

Kazuhiro Kurita (19)
Nagoya University, Japan

Dominik Köppl (8)
Department of Informatics,
Yamanashi University, Japan

From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday.
Editors: Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Luigi Laura (11)
International Telematic University Uninettuno, Rome, Italy

Grigorios Loukides (9)
King's College London, UK

Maria Madonia (5)
Dipartimento di Matematica e Informatica, Università di Catania, Italy

Sabrina Mantaci (5)
Dipartimento di Matematica e Informatica, Università di Palermo, Italy

Giovanni Manzini (10, 15)
Department of Computer Science, University of Pisa, Italy

Roberto Marangoni (18)
Department of Biology, University of Pisa, Italy

Andrea Marino (19)
University of Florence, Italy

Yasuko Matsui (8)
Department of Mathematical Sciences, Tokai University, Japan

Veli Mäkinen (2)
Department of Computer Science, University of Helsinki, Finland

Gonzalo Navarro (6, 10)
Department of Computer Science, University of Chile, Santiago, Chile; Center for Biotechnology and Bioengineering (CeBiB), Santiago, Chile

Yakov Nekirch (16)
Michigan Technological University, Houghton, MI, USA

Hirotaka Ono (8)
Department of Mathematical Informatics, Nagoya University, Japan

Alessio Orlandi (11)
Google, Zürich, Switzerland

Dario Ostuni (11)
Università degli Studi di Milano, Italy

Linda Pagli (3)
Dipartimento di Informatica, University of Pisa, Italy

Nadia Pisanti (2)
Universitá di Pisa, Italy

Solon P. Pissis (9)
CWI, Amsterdam, The Netherlands; Vrije Universiteit, Amsterdam, The Netherlands

Simon J. Puglisi (6)
Department of Computer Science, University of Helsinki, Finland

Giulia Punzi (19, 20)
University of Pisa, Italy

Romeo Rizzi (11)
Università di Verona, Italy

Nicola Rizzo (2)
Department of Computer Science, University of Helsinki, Finland

Giovanna Rosone (15, 20)
Department of Computer Science, University of Pisa, Italy

Kunihiko Sadakane (13)
The University of Tokyo, Japan

Toshiki Saitoh (8)
School of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka, Japan

Srinivasa Rao Satti (12)
Norwegian University of Science and Technology, Trondheim, Norway

Marinella Sciortino (10)
Department of Mathematics and Computer Science, University of Palermo, Italy

Lorenzo Tattini (20)
EURECOM, Biot, France; CNRS UMR 7284, INSERM U 1081, Université Côte d'Azur, Nice, France

Sharma V. Thankachan (16)
North Carolina State University, Raleigh, NC, USA

Takeaki Uno (19)
National Institute of Informatics, Tokyo, Japan

Yushi Uno (8)
Graduate School of Informatics, Osaka Metropolitan University, Japan

Rossano Venturini (15)
Department of Computer Science, University of Pisa, Italy

Luca Versari (11)
Google, Zürich, Switzerland

Jeffrey Scott Vitter (1, 15)
Department of Computer Science,
Tulane University, New Orleans, LA, USA;
The University of Mississippi, MS, USA

Yujia Wang (1)
Department of Computer Science, Xidian
University, Xi'an, China

Kunihiro Wasa (19)
Hosei University, Tokyo, Japan

An Efficient Heuristic for Graph Edit Distance

Xiaoyang Chen
Department of Computer Science, Xidian University, Xi'an, China

Yujia Wang
Department of Computer Science, Xidian University, Xi'an, China

Hongwei Huo[1]
Department of Computer Science, Xidian University, Xi'an, China

Jeffrey Scott Vitter[1]
Department of Computer Science, Tulane University, New Orleans, LA, USA
The University of Mississippi, MS, USA

Abstract

The graph edit distance (GED) is a flexible distance measure widely used in many applications. Existing GED computation methods are usually based upon the tree-based search algorithm that explores all possible vertex (or edge) mappings between two compared graphs. During this process, various GED lower bounds are adopted as heuristic estimations to accelerate the tree-based search algorithm. For the first time, we analyze the relationship among three state-of-the-art GED lower bounds, label edit distance (LED), Hausdorff edit distance (HED), and branch edit distance (BED). Specifically, we demonstrate that $BED(G, Q) \geq HED(G, Q)$ and $BED(G, Q) \geq LED(G, Q)$ for any two undirected graphs G and Q. Furthermore, for BED we propose an efficient heuristic BED$^+$ for improving the tree-based search algorithm. Extensive experiments on real and synthetic datasets confirm that BED$^+$ achieves smaller deviation and larger solvable ratios than LED, HED and BED when they are employed as heuristic estimations. The source code is available online.

2012 ACM Subject Classification Information systems → Query optimization

Keywords and phrases Graph edit distance, Label edit distance, Hausdorff edit distance, Branch edit distance, Tree-based search, Heuristics

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.1

Category Research

Supplementary Material
Software (Source Code): `https://github.com/Hongweihuo-Lab/Heur-GED` [11]

Funding This work was supported in part by the National Natural Science Foundation of China under Grant No. 62272358.

1 Introduction

Graphs are frequently used to represent a wide variety of various objects, such as networks, maps, handwriting, molecular compounds, and protein structures. The process of evaluating the similarity of two graphs is referred to as error-tolerant graph matching, aiming to find a correspondence between their vertices. In this paper, we focus upon the similarity measure *graph edit distance (GED)* because it can be applied to all types of graphs and can precisely capture structural differences between the compared graphs. The GED of two graphs is defined as the minimum cost of transforming one graph into another through a sequence

[1] corresponding author

of edit operations (inserting, deleting and substituting vertices or edges). An edit cost is assigned to each edit operation to measure its strength, which can be obtained by combining specific knowledge of the domain or learning from a set of sample graphs [14].

However, computing the GED is an NP-hard problem [30] and usually based upon the tree-based search algorithm. This search tree enumerates all possible mappings between vertices (or edges) of two compared graphs, where the inner nodes denote partial mappings and the leaf nodes denote complete mappings. Most existing GED computation methods employ different search paradigms to traverse this search tree to seek for the *optimal mapping* that induces the GED. Riesen et al. [25, 26] proposed the standard method, A*-GED, based upon the best-first search paradigm. It needs to store numerous inner nodes, resulting in high memory consumption. To overcome this bottleneck, Abu-Aisheh et al. [3] proposed a depth-first search based algorithm, DF-GED, whose memory requirement increases linearly with the number of vertices of graphs. On the other hand, Chen et al. [9] introduced a method for the GED computation based upon beam-stack search [28], achieving a flexible tradeoff between memory consumption and the time overhead of backtracking in the depth-first search. Chang et al. [8] developed a unified framework that can be instantiated into either a best-first search approach or a depth-first search approach. Gouda et al. [17] proposed a novel edge-mapping based approach, CSI_GED, and also employed the depth-first search paradigm. CSI_GED works only for the uniform cost model, and Blumenthal et al. [4, 6] generalized it to cover the non-uniform cost model. Kim [19] developed an efficient GED computation algorithm using isomorphic vertices [9]. Liu et al. [22] explored a learning-based method for the approximate GED computation. Piao et al. [23] propose a deep learning method for the GED computation. It is worth mentioning that many researchers have proposed various indexing techniques [31, 8, 10] to accelerate graph similarity searches under the GED metric. They use the above GED computation methods as the final phase to verify the candidate graphs that satisfy the GED constraint.

In the tree-based search algorithm, a heuristic estimation is usually adopted to prune the useless search space to accelerate the search process. In order to ensure that the optimal mapping is not erroneously pruned, this heuristic function must be *admissible*; namely, it estimates the cost of a tree node that is less than or equal to the real cost. In the previous works A*-GED and DF-GED, they adopted the label edit distance (LED) as the heuristic, which calculates the minimum substitution cost of vertices and edges of two compared graphs. After that, Fischer et al. [15, 16] proposed the Hausdorff edit distance (HED) as a heuristic estimation. HED, based upon Hausdorff matching [29], performs a bidirectional matching between two graphs and allows multiple assignments between their vertices. Recently, Blumenthal et al. [5] proposed another effective GED lower bound, branch edit distance (BED), which also can be adopted as a heuristic estimation.

As observed in other studies [7, 15, 27], the higher the heuristic estimates the cost, the better the tree-based search algorithm performs. The following question naturally arises: Which of these three state-of-the-art GED lower bounds (namely, LED, HED, or BED) is more effective? In this paper, we first analyze the relationship among these three lower bounds and then propose an effective heuristic estimation. Our contributions are summarized as follows:

(1) We analyze the relationship among LED, HED and BED for the first time, and we derive that $BED(G, Q) \geq HED(G, Q)$ and $BED(G, Q) \geq LED(G, Q)$ for any two undirected graphs G and Q.

(2) We propose an efficient heuristic estimation BED$^+$ based upon BED, and demonstrate that BED$^+$ is still admissible.

(3) We conduct extensive experiments to confirm BED$^+$'s effectiveness on the real and synthetic datasets. The source code is available online [11].

The rest of this paper is organized as follows. In Section 2, we give the definition of the graph edit distance and revisit three state-of-the-art GED lower bounds. In Section 3, we theoretically analyze the relationship between LED, HED and BED. In Section 4, we propose the heuristic function BED$^+$ for improving the GED computation. In Section 5, we report the experimental results. Finally, we summarize this paper in Section 6.

2 Graph edit distance

In this paper, we consider *undirected*, *labeled* graphs without multi-edges or self-loops. A labeled graph is a triplet $G = (V_G, E_G, L)$, where V_G is the set of vertices, E_G is the set of edges, $L : V_G \cup E_G \to \Sigma$ is a labeling function that assigns a label to a vertex or an edge, and Σ is a set of labels. Also, we use a special symbol ε to denote a *dummy vertex* or a *dummy edge*.

Given two graphs G and Q, six edit operations [18, 25, 21, 5] can be used to transform G to Q (or vice versa): inserting or deleting a vertex or an edge, and substituting the label of a vertex or an edge. We denote the *label substitution* (or simply *substitution*) of vertices $u \in V_G$ and $v \in V_Q$ by $(u \to v)$, the *deletion* of u by $(u \to \varepsilon)$, and the *insertion* of v by $(\varepsilon \to v)$. For the three edit operations on edges, we use similar notation.

An *edit path* $\mathcal{P} = \langle p_1, p_2, \ldots, p_k \rangle$ between G and Q is a sequence of edit operations that transforms G to Q, such as $G = G^0 \xrightarrow{p_1} \ldots G^i \xrightarrow{p_{i+1}} G^{i+1} \ldots \xrightarrow{p_k} G^k = Q$, where graph G^{i+1} is obtained by performing the edit operation p_{i+1} on graph G^i, for $0 \le i \le k-1$. During this transformation, each edit operation p_i is assigned a penalty cost $c(p_i)$ to reflect whether it can strongly change a graph. Note that the cost of editing two dummy vertices (or edges) is 0; that is, $c(\varepsilon \to \varepsilon) = 0$. Thus, $\mathcal{P}$'s edit cost is defined as $\sum_{i=1}^{k} c(p_i)$. We define the *graph edit distance* as follows:

▶ **Definition 1.** *Given two graphs G and Q, the graph edit distance between them, denoted by $ged(G, Q)$, is defined as the minimum cost of transforming G to Q, namely,*

$$ged(G, Q) = \min_{\mathcal{P} \in \Upsilon(G,Q)} \sum\nolimits_{p_i \in \mathcal{P}} c(p_i) \tag{1}$$

where $\Upsilon(G, Q)$ is the set of all edit paths between G and Q, and $c(p_i)$ is edit operation p_i's cost.

Hereafter, for ease of presentation, we denote $V_G^\varepsilon = V_G \cup \overbrace{\{\varepsilon, \ldots, \varepsilon\}}^{|V_Q|}$ and $V_Q^\varepsilon = V_Q \cup \overbrace{\{\varepsilon, \ldots, \varepsilon\}}^{|V_G|}$ as the expanded sets of V_G and E_G, respectively, so that V_G^ε and V_Q^ε have the same number of vertices. Similarly, $E_G^\varepsilon = E_G \cup \overbrace{\{\varepsilon, \ldots, \varepsilon\}}^{|E_Q|}$ and $E_Q^\varepsilon = E_Q \cup \overbrace{\{\varepsilon, \ldots, \varepsilon\}}^{|E_G|}$ denote the expanded sets of E_G and E_Q, respectively.

2.1 State-of-the-art GED lower bounds

Below we introduce three state-of-the-art GED lower bounds, which can be used as heuristic estimations in the tree-based search algorithm to compute GED. Each of the methods gives a lower bound on GED because the operations are done in sets that do not have to be consistent with one another. For example, in the first method LED described below, the edit operations on the vertex labels can be done independently of the edit operations on the edge labels, and thus they may not be globally consistent.

Label Edit Distance. Riesen et al. [27, 26] proposed the label edit distance (LED), which is the minimum cost of substituting vertices and edges of two graphs.

▶ **Definition 2** (Label edit distance). *Given two graphs G and Q, the label edit distance between them is defined as $LED(G,Q) = \lambda_V(G,Q) + \lambda_E(G,Q)$, where $\lambda_V(G,Q) = \min_{\phi:V_G^\varepsilon \to V_Q^\varepsilon} \sum_{u \in V_G^\varepsilon} c(u \to \phi(u))$ is the minimum cost of substituting vertices of G and Q, and ϕ is a bijection from V_G^ε to V_Q^ε; and $\lambda_E(G,Q) = \min_{\varphi:E_G^\varepsilon \to E_Q^\varepsilon} \sum_{e(u,u') \in E_G^\varepsilon} c(e(u,u') \to \varphi(e(u,u')))$ is the minimum cost of substituting edges of G and Q, and φ is a bijection from E_G^ε to E_Q^ε.*

Hausdorff Edit Distance. Inspired by the Hausdorff distance [29] between two finite sets, Fischer et al. [16] proposed the Hausdorff edit distance (HED) between two graphs G and Q. The key ideas of HED are to perform a bidirectional matching between G and Q and to allow multiple assignments between their vertices.

▶ **Definition 3** (Hausdorff edit distance). *Given two graphs G and Q, their Hausdorff edit distance is defined as $HED(G,Q) = \sum_{u \in V_G} \min_{v \in V_Q \cup \{\varepsilon\}} f_H(u,v) + \sum_{v \in V_Q} \min_{u \in V_G \cup \{\varepsilon\}} f_H(u,v)$, where $f_H(u,v)$ is the Hausdorff cost of matching vertex u to vertex v.*

The Hausdorff vertex matching cost $f_H(u,v)$ considers not only the two vertices $u \in V_G$ and $v \in V_Q$ but also their neighboring edges.

▶ **Definition 4** (Neighboring edges). *Given graph G and a vertex $u \in V_G$, the* neighboring edges N_u *of u are defined as $N_u = \{e(u,u') : u' \in V_G \wedge e(u,u') \in E_G\}$.*

We define $f_H(u,v)$ as

$$f_H(u,v) = \begin{cases} c(u \to \varepsilon) + \sum_{e_1 \in N_u} \frac{c(e_1 \to \varepsilon)}{2} & \text{if } v = \varepsilon; \\ c(\varepsilon \to v) + \sum_{e_2 \in N_v} \frac{c(\varepsilon \to e_2)}{2} & \text{if } u = \varepsilon; \\ \frac{c(u \to v) + \frac{HED(N_u, N_v)}{2}}{2} & \text{otherwise.} \end{cases} \tag{2}$$

Similarly to Definition 3, the Hausdorff edit distance $HED(N_u, N_v)$ between N_u and N_v is defined as

$$HED(N_u, N_v) = \sum_{e_1 \in N_u} \min_{e_2 \in N_v \cup \{\varepsilon\}} f_H(e_1, e_2) + \sum_{e_2 \in N_v} \min_{e_1 \in N_u \cup \{\varepsilon\}} f_H(e_1, e_2) \tag{3}$$

where $f_H(e_1, e_2)$ is the cost of matching two edges such that

$$f_H(e_1,e_2) = \begin{cases} c(e_1 \to \varepsilon) & \text{if } e_2 = \varepsilon; \\ c(\varepsilon \to e_2) & \text{if } e_1 = \varepsilon; \\ \frac{c(e_1 \to e_2)}{2} & \text{otherwise.} \end{cases} \tag{4}$$

Branch Edit Distance. Blumenthal et al. [5] recently proposed the branch edit distance (BED), which computes the minimum cost of editing branch structures of two graphs.

▶ **Definition 5** (Branch structure). *The branch structure of vertex u in graph G is defined as $B_u = (u, N_u)$, where N_u is the set of neighboring edges of u.*

Given two branch structures B_u and B_v, the minimum cost of editing B_u into B_v is defined as

$$f_B(u,v) = c(u \to v) + \frac{1}{2} \min_{\varrho: N_u^\varepsilon \to N_v^\varepsilon} \sum_{e(u,u') \in N_u^\varepsilon} c(e(u,u') \to \varrho(e(u,u'))), \tag{5}$$

where $N_u^\varepsilon = N_u \cup \overbrace{\{\varepsilon, \ldots, \varepsilon\}}^{|N_v|}$ and $N_v^\varepsilon = N_v \cup \overbrace{\{\varepsilon, \ldots, \varepsilon\}}^{|N_u|}$ are expanded sets of N_u and N_v, respectively, and ϱ is a bijection from N_u^ε to N_v^ε.

▶ **Definition 6** (Branch edit distance). *Given two graphs G and Q, the branch edit distance between them is defined as $BED(G,Q) = \min_{\rho: V_G^\varepsilon \to V_Q^\varepsilon} \sum_{u \in V_G^\varepsilon} f_B(u, \rho(u))$, where ρ is a bijection from V_G^ε to V_Q^ε, and $f_B(\cdot,\cdot)$ is defined in (5).*

3 Tightness analysis

In this section, we analyze the tightness of the three GED lower bounds: LED, HED and BED. Specifically, we will prove that BED is the strongest of all; that is, for any two undirected graphs G and Q, we have $BED(G,Q) \geq LED(G,Q)$ and $BED(G,Q) \geq HED(G,Q)$.

3.1 Relation of LED and BED

▶ **Theorem 7.** *Given two graphs G and Q, we have $BED(G,Q) \geq LED(G,Q)$.*

Proof. For ease of proof, we insert dummy vertices and edges into G to make it become a complete graph with $(|V_G| + |V_Q|)$ vertices. Similarly, we transform Q into a complete graph that also has $(|V_G| + |V_Q|)$ vertices. Then, we can simplify (5) as

$$\begin{aligned} f_B(u,v) &= c(u \to v) + \frac{1}{2} \min_{\varrho: N_u^\varepsilon \to N_v^\varepsilon} \sum_{e(u,u') \in N_u^\varepsilon} c(e(u,u') \to \varrho(e(u,u'))) \\ &= c(u \to v) + \frac{1}{2} \min_{\zeta: V_G \setminus \{u\} \to V_Q \setminus \{v\}} \sum_{u' \in V_G \setminus \{u\}} c(e(u,u') \to e(v, \zeta(u'))) \\ &= c(u \to v) + \frac{1}{2} \sum_{u' \in V_G \setminus \{u\}} c(e(u,u') \to e(v, \zeta_{\min}^{u,v}(u'))) \end{aligned}$$

where $\zeta_{\min}^{u,v}$ is the bijection from $V_G \setminus \{u\}$ to $V_Q \setminus \{v\}$ for which $f_B(u,v)$ achieves the minimum value. Thus, we have

$$\begin{aligned} BED(G,Q) &= \min_{\rho: V_G \to V_Q} \sum_{u \in V_G} f_B(u, \rho(u)) \\ &= \sum_{u \in V_G} \left\{ c(u \to \rho_{\min}(u)) + \frac{1}{2} \sum_{u' \in V_G \setminus \{u\}} c(e(u,u') \to e(\rho_{\min}(u), \zeta_{\min}^{u,u'}(u')) \right\} \\ &= \sum_{u \in V_G} c(u \to \rho_{\min}(u)) + \frac{1}{2} \sum_{u \in V_G} \sum_{u' \in V_G \setminus \{u\}} c(e(u,u') \to e(\rho_{\min}(u), \zeta_{\min}^{u,u'}(u'))) \\ &= \sum_{u \in V_G} c(u \to \rho_{\min}(u)) + \sum_{e(u,u') \in E_G} c(e(u,u') \to \xi(e(u,u'))) \\ &\geq \min_{\phi: V_G \to V_Q} \sum_{u \in V_G} c(u \to \phi(u)) + \min_{\varphi: E_G \to E_Q} \sum_{e(u,u') \in E_G} c(e(u,u') \to \varphi(e(u,u')) \\ &= \lambda_V(G,Q) + \lambda_E(E_G, E_Q) = LED(G,Q) \end{aligned}$$

where $\rho_{\min}$ is the bijection from V_G to V_Q for which $BED(G,Q)$ achieves the minimum value, and ξ is the bijection from E_G to E_Q satisfying $e(\rho_{\min}(u), \zeta_{\min}^{u,u'}(u'))) = \xi(e(u,u'))$ for $\forall u \in V_G, u' \in V_G \backslash \{u\}$. ◀

3.2 Relation of HED and BED

▶ **Lemma 8.** *Given two vertices $u \in V_G^\varepsilon$ and $v \in V_Q^\varepsilon$, then we have*

$$f_H(u,v) \leq \begin{cases} f_B(u,v) & \text{if } u = \varepsilon \text{ or } v = \varepsilon; \\ \frac{1}{2} f_B(u,v) & \text{otherwise.} \end{cases}$$

where $f_H(u,v)$ and $f_B(u,v)$ are defined in (2) *and* (5)*, respectively.*

The proof of Lemma 8 is in Appendix A.

▶ **Theorem 9.** *Given two graphs G and Q, we have $BED(G,Q) \geq HED(G,Q)$.*

Proof. By $f_H(u,v)$'s definition in (2), we know that when $u = \varepsilon$, $\min_{v \in V_Q \cup \{\varepsilon\}} f_H(\varepsilon, v) = f_H(\varepsilon,\varepsilon) = 0$; and similarly when $v = \varepsilon$, $\min_{u \in V_G \cup \{\varepsilon\}} f_H(u,\varepsilon) = f_H(\varepsilon,\varepsilon) = 0$. We can rewrite $HED(G,Q)$ as

$$\begin{aligned} HED(G,Q) &= \sum_{u \in V_G} \min_{v \in V_Q \cup \{\varepsilon\}} f_H(u,v) + \sum_{v \in V_Q} \min_{u \in V_G \cup \{\varepsilon\}} f_H(u,v) \\ &= \sum_{u \in V_G^\varepsilon} \min_{v \in V_Q^\varepsilon} f_H(u,v) + \sum_{v \in V_Q^\varepsilon} \min_{u \in V_G^\varepsilon} f_H(u,v) \\ &= \sum_{u \in V_G^\varepsilon} f_H(u, \pi_1(u)) + \sum_{v \in V_Q^\varepsilon} f_H(\pi_2(v), v)) \\ &= \sum_{u \in V_G^\varepsilon} \Big\{ f_H(u, \pi_1(u)) + f_H\big(\pi_2(\rho_{\min}(u)), \rho_{\min}(u)\big) \Big\}, \end{aligned} \tag{6}$$

where π_1 is a *mapping* from V_G^ε to V_Q^ε satisfying $\pi_1(u) = \arg\min_{v \in V_Q^\varepsilon} f_H(u,v)$,; π_2 is a *mapping* from V_Q^ε to V_G^ε satisfying $\pi_2(v) = \arg\min_{u \in V_G^\varepsilon} f_H(u,v)$; and $\rho_{\min}$ is the bijection from V_G^ε to V_Q^ε for which $BED(G,Q)$ achieves the minimum value. We know that

$$BED(G,Q) = \sum_{u \in V_G^\varepsilon} f_B(u, \rho_{\min}(u)). \tag{7}$$

By (6) and (7), we can complete the proof by showing that

$$f_H\big(u, \pi_1(u)\big) + f_H\big(\pi_2(\rho_{\min}(u)), \rho_{\min}(u)\big) \leq f_B(u, \rho_{\min}(u)).$$

We do so by considering the following four exhaustive cases:

Case I. When $u = \varepsilon$ and $\rho_{\min}(u) = \varepsilon$, then $f_H(u, \pi_1(u)) + f_H(\pi_2(\rho_{\min}(u)), \rho_{\min}(u)) \leq f_H(\varepsilon,\varepsilon) + f_H(\varepsilon,\varepsilon) = f_B(\varepsilon,\varepsilon) = 0$, by the definitions of π_1, π_2, and $\rho_{\min}$.

Case II. When $u \neq \varepsilon$ and $\rho_{\min}(u) = \varepsilon$, then $f_H(u, \pi_1(u)) + f_H(\pi_2(\rho_{\min}(u)), \rho_{\min}(u)) \leq f_H(u,\varepsilon) + f_H(\varepsilon,\varepsilon) = f_H(u,\varepsilon)$, by the definitions of π_1, π_2, and $\rho_{\min}$. By Lemma 8, we know that $f_H(u,\varepsilon) \leq f_B(u,\varepsilon) = f_B(u, \rho_{\min}(u))$.

Case III. When $u = \varepsilon$ and $\rho_{\min}(u) \neq \varepsilon$, the analysis is similar to that of Case II.

Case IV. When $u \neq \varepsilon$ and $\rho_{\min}(u) \neq \varepsilon$, then we have $f_H(u, \pi_1(u)) \leq f_H(u, \rho_{\min}(u))$ and $f_H(\pi_2(\rho_{\min}(u)), \rho_{\min}(u)) \leq f_H(u, \rho_{\min}(u))$, by the definitions of π_1, π_2, and $\rho_{\min}$. By Lemma 8, we know that $f_H(u, \rho_{\min}(u)) \leq \frac{1}{2} f_B(u, \rho_{\min}(u))$. Thus, we have $f_H(u, \pi_1(u)) + f_H\big(\pi_2(\rho_{\min}(u)), \rho_{\min}(u)\big) \leq 2 \times \frac{1}{2} f_B(u, \rho_{\min}(u)) = f_B(u, \rho_{\min}(u))$. ◀

4 Tree-based search algorithm

The previous section showed that BED achieves the tightest GED lower bound. In this section, based upon BED we propose an efficient heuristic estimation to improve the tree-based search algorithm [2, 3] for the GED computation.

4.1 Search tree

Computing the GED of graphs G and Q is typically based upon a tree-based search procedure that explores all possible graph mappings from G to Q. Starting from a dummy node, root $= \emptyset$, we logically create the search tree layer by layer by iteratively generating successors using BasicGenSuccr [9]. This search space can be organized as an ordered search tree, where the inner nodes denote partial graph mappings and the leaf nodes denote complete graph mappings. Such a search tree is created dynamically at runtime by iteratively generating successors linked by edges to the currently considered node. For more details, please refer to Section 2 in the reference [9].

4.2 Heuristic cost estimation

For a node r in the search tree, let $h(r)$ be the estimated cost from r to its descendant leaf node that is less than or equal to the real cost. Based upon BED, we introduce how to estimate $h(r)$ in the tree-based search algorithm.

4.2.1 Heuristic function

Consider an inner node $r = \{(u_1 \to v_{j_1}), \ldots, (u_l \to v_{j_\ell})\}$, where v_{j_k} is u_k's mapped vertex, for $1 \le k \le \ell$. We divide G into two subgraphs G_r^1 and G_r^2, where G_r^1 is the mapped part of G such that $V_{G_r^1} = \{u_1, \ldots, u_l\}$ and $E_{G_r^1} = \{e(u,v) : u, v \in V_{G_r^1} \wedge e(u,v) \in E_G\}$, and G_r^2 is the unmapped part such that $V_{G_r^2} = V_G \backslash V_{G_r^1}$ and $E_{G_r^2} = \{e(u,v) : u, v \in V_{G_r^2} \wedge e(u,v) \in E_G\}$. We obtain Q_r^1 and Q_r^2 similarly.

Clearly, the lower bound $BED(G_r^2, Q_r^2)$ can be used to estimate r's cost. However, $BED(G_r^2, Q_r^2)$ has not covered the potential edit cost on the edges between G_r^1 (resp., Q_r^1) and G_r^2 (resp., Q_r^2). Recently, [8, 9] proposed two different methods to cover this potential cost; nevertheless, these two methods only worked for the *uniform cost function* (i.e., for which the cost of each edit operation is 1). We expand the method in [8] to support for any cost function.

▶ **Definition 10.** *Given vertices $u \in G_r^2$ and $v \in Q_r^2$, we define the cost of matching u to v as $f_B^+(u,v) = f_B(u,v) + \sum_{u' \in V_{G_r^1}} c(e(u,u') \to e(v,v'))$, where v' is the mapped vertex of the already processed vertex u', and $f_B(u,v)$ is the minimum cost of transforming B_u to B_v, which we defined in* (5).

When there is no edge between u and u', we set $e(u,u') = \varepsilon$, and similarly for $e(v,v')$. Based upon $f_B^+(u,v)$, we define the improved lower bound BED$^+$ as

$$BED^+(G_r^2, Q_r^2) = \min_{\rho: V_{G_r^2}^\varepsilon \to V_{Q_r^2}^\varepsilon} \sum_{u \in V_{G_r^2}^\varepsilon} f_B^+(u, \rho(u)) \tag{8}$$

▶ **Theorem 11.** *Given a node r in the GED tree of graphs G and Q, then $BED^+(G_r^2, Q_r^2) \ge BED(G_r^2, Q_r^2)$, where G_r^2 and Q_r^2 are the unmapped subgraphs of G and Q, respectively, $BED^+(\cdot,\cdot)$ and $BED(\cdot,\cdot)$ are defined in* (8) *and Definition 6, respectively.*

Proof. We trivially obtain this theorem since $f_B^+(u,v) \geq f_B(u,v)$ for $\forall u \in V_{G_r^2}, v \in V_{Q_r^2}$. ◀

▶ **Theorem 12.** *Given a descendant leaf node s of r, the heuristic estimation $h(r) = BED^+(G_r^2, Q_r^2)$ is admissible; that is, $h(r) \leq g(s) - g(r)$, where $g(\cdot)$ gives the incurred cost from the root node to the currently considered node.*

Proof. For ease of proof, we insert dummy vertices and edges into G to transform it to a complete graph with $(|V_G| + |V_Q|)$ vertices. Similarly, we transform Q to a complete graph that also has $(|V_G| + |V_Q|)$ vertices.

Consider an internal node $r = \{(u_1 \to v_{j_1}), \ldots, (u_\ell \to v_{j_\ell})\}$ in the search tree, where v_{j_k} is the mapped vertex of u_k, for $1 \leq k \leq \ell$. For easy presentation, hereafter we use $r(u_k)$ to denote u_k's mapped vertex, i.e., $v_{j_k} = r(u_k)$. Given a descendent leaf node s (i.e., s is a complete vertex mapping from G to Q) of r, then the incurred cost of s is

$$\begin{aligned} g(s) &= \sum_{u \in V_G} c(u \to s(u)) \ + \ \frac{1}{2} \sum_{u \in V_G} \sum_{u' \in V_G \backslash \{u\}} c(e(u,u') \to e(s(u), s(u'))) \\ &= \sum_{u \in V_G} c(u \to s(u)) \ + \sum_{e(u,u') \in \binom{V_G}{2}} c(e(u,u') \to e(s(u), s(u'))) \end{aligned} \tag{9}$$

As we know, r induces an edit path transforming G_r^1 to Q_r^1, where G_r^1 and Q_r^1 are the already mapped subgraphs of G and Q, respectively, and $V_{G_r^1} = \{u_1, \ldots, u_\ell\}$ and $V_{Q_r^1} = \{r(u_1), \ldots, r(u_\ell)\}$. According to (9), we know that

$$g(r) = \sum_{u \in V_{G_r^1}} c(u \to r(u)) \ + \sum_{e(u,u') \in \binom{V_{G_r^1}}{2}} c(e(u,u') \to e(r(u), r(u')))$$

Let $\omega = s \backslash r$ be the partial mapping that contains the vertex mapping pairs belong to s but not r; namely, $\omega = \{(u \to s(u)) : u \in V_G \backslash V_{G_r^1}\}$. We can obtain that

$$\begin{aligned} g(s) - g(r) &= \sum_{u \in V_G} c(u \to s(u)) \ + \sum_{e(u,u') \in \binom{V_G}{2}} c(e(u,u') \to e(s(u), s(u'))) \\ &\qquad - \left\{ \sum_{u \in V_{G_r^1}} c(u \to r(u)) \ + \sum_{e(u,u') \in \binom{V_{G_r^1}}{2}} c(e(u,u') \to e(r(u), r(u'))) \right\} \\ &= \sum_{u \in V_{G_r^2}} c(u \to \omega(u)) \ + \sum_{e(u,u') \in \binom{V_{G_r^2}}{2}} c(e(u,u') \to e(\omega(u), \omega(u'))) \\ &\qquad + \sum_{u \in V_{G_r^2}} \sum_{u' \in v_{G_r^1}} c(e(u,u') \to e(\omega(u), r(u'))) \\ &= \sum_{u \in V_{G_r^2}} \left\{ c(u \to \omega(u)) + \frac{1}{2} \sum_{u' \in V_{G_r^2} \backslash \{u\}} c(e(u,u') \to e(\omega(u), \omega(u'))) \right. \\ &\qquad \left. + \sum_{u' \in V_{G_r^1}} c(e(u,u') \to e(\omega(u), r(u'))) \right\} \\ &= \sum_{u \in V_{G_r^2}} f_B^+(u, \omega(u)) \geq \min_{\rho : V_{G_r^2} \to V_{Q_r^2}} f_B^+(u, \rho(u)) = BED^+(G_r^2, Q_r^2) = h(r). \end{aligned}$$

where $V_{G_r^2} = V_G \backslash V_{G_r^1}$ and $V_{Q_r^2} = V_Q \backslash V_{Q_r^1}$. The second equality is due to $\binom{V_G}{2} = \binom{V_{G_r^1}}{2} \cup \binom{V_{G_r^2}}{2} \cup (V_{G_r^1} \times V_{G_r^2})$ when V_G is partitioned into two disjoint sets $V_{G_r^1}$ and $V_{G_r^2}$. ◀

We give an example in Appendix B of computing three GED lower bounds: LED, HED and BED. The same optimization that produces BED+ from BED can be applied to LED and HED to achieve enhanced heuristics LED+ and HED+, but we do not include them in this paper.

4.3 Algorithm

In this section, we show how to incorporate the heuristic estimation BED^+ into the anytime-based GED computation algorithm [2]. The reason we consider the anytime-based algorithm is that it is flexible and can control the algorithm to output tighter and tighter GED upper bounds until the exact GED value by setting more and more running time.

Algorithm 1 gives the anytime-based algorithm for computing the GED, where $t_{\max}$ is the user-defined maximum running time. We perform a depth-first search over the GED search tree of G and Q to find better and better GED upper bounds until the running time $\#t$ reaches $t_{\max}$. To accomplish this, we first employ the BP [25] algorithm to fast compute an initial GED upper bound ub; then, we adopt a stack $\mathcal{S}$ to finish the depth-first search. Each time we pop a node q from $\mathcal{S}$. If q is a leaf node, then we find a better solution. Otherwise, we call *procedure* BasicGenSuccr [9] (see Appendix C) to generate q's successors and then insert them into $\mathcal{S}$. During this process, we adopt the *branch-and-bound* strategy to prune the useless space: for each successor r, if $g(r) + h(r) \geq ub$, we can safely prune it, where $h(r) = BED^+(G_r^2, Q_r^2)$ is defined in (8).

Algorithm 1 Anytime-based GED computation.

Input : G and Q, and $t_{\max}$
Output : best GED upper bound for $t_{\max}$

```
1  initialize ub with the BP algorithm
2  root ← {}, S ← {}
3  S.push(root)
4  while S ≠ ∅ and #t ≤ t_max do
5  |  q ← S.pop()
6  |  if q is a complete mapping then
7  |  |  ub ← g(q), export ub and q
8  |  |  continue
9  |  succ ← BasicGenSuccr(q)
10 |  foreach r ∈ succ do
11 |  |  if g(r) + h(r) < ub then
12 |  |  |  S.push(r)
13 export ub and q
```

5 Experiments

5.1 Datasets and settings

Datasets. We chose four real (GREC, MUTA, PRO, and CMU) and one synthetic (SYN) datasets in the experiments. The datasets GREC, MUTA, and PRO were taken from the *IAM Graph Database Repository* [24]; the CMU dataset could be found at the *CMU website* [13]; and the SYN dataset was generated by the synthetic graph generator *GraphGen* [12]. Following the same procedure in [2, 21], we selected some subsets of GREC, MUTA, and PRO as the tested datasets, respectively, where each subset consists of graphs that have the same

Table 1 Summary of characteristics of datasets and cost functions used.

Dataset	#Graphs	$\|V\|$	$\|E\|$	vertex labels	edge labels	c_v	c_e	α	c_{vs}	c_{es}
GREC	40	12.5	17.5	(x, y) coord.	Line type	90	15	0.5	Ext. ED	Dirac
MUTA	70	40	41.5	Chem. symbol	Valence	11	1.1	0.25	Dirac	Dirac
PRO	30	30	58.6	Type/AA-seq.	Type/length	11	1	0.75	Ext. SED	Dirac
CMU	111	30	79.1	None	Distance	∞	–	0.5	0	L1 norm
SYN	100	14.5	20	Symbol	Symbol	0.3	0.5	0.75	Dirac	Dirac

number of vertices. Specifically, the subsets of CREC contain 5, 10, 15, and 20 vertices, respectively; the subsets of MUTA contain 10, 20, ..., 70 vertices, respectively; the subsets of PRO contain 20, 30, and 40 vertices, respectively; and each subset consists of 10 graphs.

Table 1 summarizes the characteristic and applied cost function of each dataset. ED and SED are short for *Euclidean distance* and *string edit distance* functions, respectively. c_v is the cost of inserting/deleting a vertex; c_e is the cost of inserting/deleting an edge; c_{vs} and c_{es} are the costs of substituting a vertex and an edge, respectively. In addition, we introduce a parameter α to control whether edit operations on vertices or edges are more important.

Settings. We conducted all the experiments on a HP Z800 PC running the Ubuntu 12.04 LTS operating system and equipped with a 2.67GHz CPU and 24 GB of memory. We implemented the algorithm in C++, using -O3 to compile and run it.

5.2 Evaluation metrics

We discuss two metrics to evaluate algorithm performance: *deviation* (*dev*) [1] and *solvable ratio* (*sr*) [9]. The metric *dev* measures the deviation generated by an algorithm. Formally, given two graphs G and Q, the deviation of the two graphs can be computed as $deviation(G,Q) = |dis(G,Q) - R(G,Q)|/R(G,Q)$, where $dis(G,Q)$ is the (approximate) GED value produced by the algorithm, and $R(G,Q)$ is the best GED value produced in all the experiments done on the graph database repository in [1]. Based upon the pairwise comparison model, the deviation on the dataset $\mathcal{G}$ can be computed as

$$dev = \frac{1}{|\mathcal{G}| \times |\mathcal{G}|} \sum\nolimits_{G \in \mathcal{G}} \sum\nolimits_{Q \in \mathcal{G}} deviation(G,Q) \tag{10}$$

The metric *sr* measures how often the exact GED value is obtained when reaching the maximum running time threshold $t_{\max}$. Formally, let $slove(G,Q)$ indicate whether an algorithm outputs the exact GED of G and Q within $t_{\max}$ time; in other words, if the algorithm requires less than $t_{\max}$ time to output the GED, $slove(G,Q) = 1$; otherwise, $slove(G,Q) = 0$. The solvable ratio (*sr*) on the dataset $\mathcal{G}$ can be computed as

$$sr = \frac{1}{|\mathcal{G}| \times |\mathcal{G}|} \sum\nolimits_{G \in \mathcal{G}} \sum\nolimits_{Q \in \mathcal{G}} slove(G,Q) \tag{11}$$

Obviously, a smaller *dev* and a larger *sr* reflects a better performance of an algorithm.

5.3 Experimental results

As described earlier in this paper, we first analyzed the relation of three GED lower bounds (i.e., LED, HED and BED). Then based upon BED we proposed an efficient heuristic estimation BED$^+$. Thus, it is necessary to evaluate the contribution of these lower bounds to the GED computation.

5.3.1 Tightness of LED, HED and BED

In this section, we evaluate the tightness of three GED lower bounds LED, HED and BED as well as their running time. Table 2 shows the obtained results, where the abbreviation "ms" represents milliseconds.

As shown in Table 2, BED achieves the smallest *dev*, which means that BED is closest to the exact GED value. This result is consistent with the analysis in Section 3, i.e., $BED(G, Q) \geq HED(G, Q)$ and $BED(G, Q) \geq LED(G, Q)$ for any two graphs G and Q. We also find in most cases tthat LED performs better than HED; the reason is that HED allows multiple assignments between vertices of G and Q and greedily selects matched vertices with the lowest cost.

We also list the running time of each method in Table 2. It can be seen from this table that HED runs faster than LED and LED runs faster than BED. The reason is that HED runs in quadratic time, while both LED and BED run in cubic time. LED independently considers the cost of substituting vertices and edges and ignores the structures, thus it has a better running time than BED.

Table 2 Deviation (%) and running time (ms) of LED, HED, and BED.

Datasets	LED		HED		BED	
	dev	*time*	*dev*	*time*	*dev*	*time*
GREC	4.41	0.38	17.45	**0.29**	**3.54**	0.52
MUTA	12.54	1.07	30.13	**0.47**	**11.49**	2.72
PRO	4.61	3.75	21.31	**2.84**	**3.25**	6.07
CMU	61.56	9.53	57.6	**3.77**	**25.1**	13.2
SYN	67.41	0.28	91.92	**0.14**	**46.61**	0.45

5.3.2 Effect of heuristic

Observing that BED produces the tighter lower bound than LED and HED, we propose BED$^+$ as a heuristic estimation to improve the GED computation. To achieve the comparison, we adopted LED, HED, BED, and BED$^+$ as the heuristic estimations, respectively, and fixed the running time $t_{\max} = 10^4$ ms. Table 3 lists the obtained deviation *dev* and solvable ration *sr*.

Table 3 Deviation (%) and solvable ratio (%) of of LED, HED, BED, and BED$^+$.

Datasets	LED		HED		BED		BED$^+$	
	dev	*sr*	*dev*	*sr*	*dev*	*sr*	*dev*	*sr*
GREC	0.36	69.87	0.56	54.38	0.22	67.88	**0.01**	**90**
MUTA	5.56	3.47	4.85	3.27	4.49	3.51	**2.6**	**22.33**
PRO	1.27	4	0.71	3.56	0.68	4.22	**0.06**	**4.22**
CMU	109.89	19.06	49.18	19.06	52.83	31.16	**2.97**	**75.79**
SYN	8.79	8.4	9.48	4.18	7.96	7.48	**0.17**	**94.34**

From Table 3, we know that using BED$^+$ as a heuristic can produce the smallest *dev*. This is due to the fact that BED$^+$ produces a higher estimated bound. For the solvable ratio *sr*, we also find that BED$^+$ achieves the best performance.

We also varied the running time $t_{\max}$ from 10^1 ms to 10^5 ms in order to evaluate the above heuristic estimations under different running times. From Figure 1, as the running time $t_{\max}$ increases, using the above four heuristic estimations we obtain lower and lower

deviation *dev* as well as higher and higher solvable ratio *sr*. Also, we find in most cases that BED$^+$ achieves the best *dev* and *sr* under both small (e.g., 10^2 ms) and large (e.g., 10^5 ms) running times. Compared with the widely used heuristic estimation LED, using BED$^+$ can decrease the *dev* by 72.2%, 34.1%, 48.1%, 39.4%, and 52.1% on average on the GREC, MUTA, PRO, CMU, and SYN datasets, respectively. Using BED$^+$ can increase the *sr* by 54.3%, 293.2%, 3.4%, 113.1%, and 702.8% on average on the five above datasets, respectively. Thus, we conclude that using BED$^+$ as a heuristic estimation can greatly improve the GED computation.

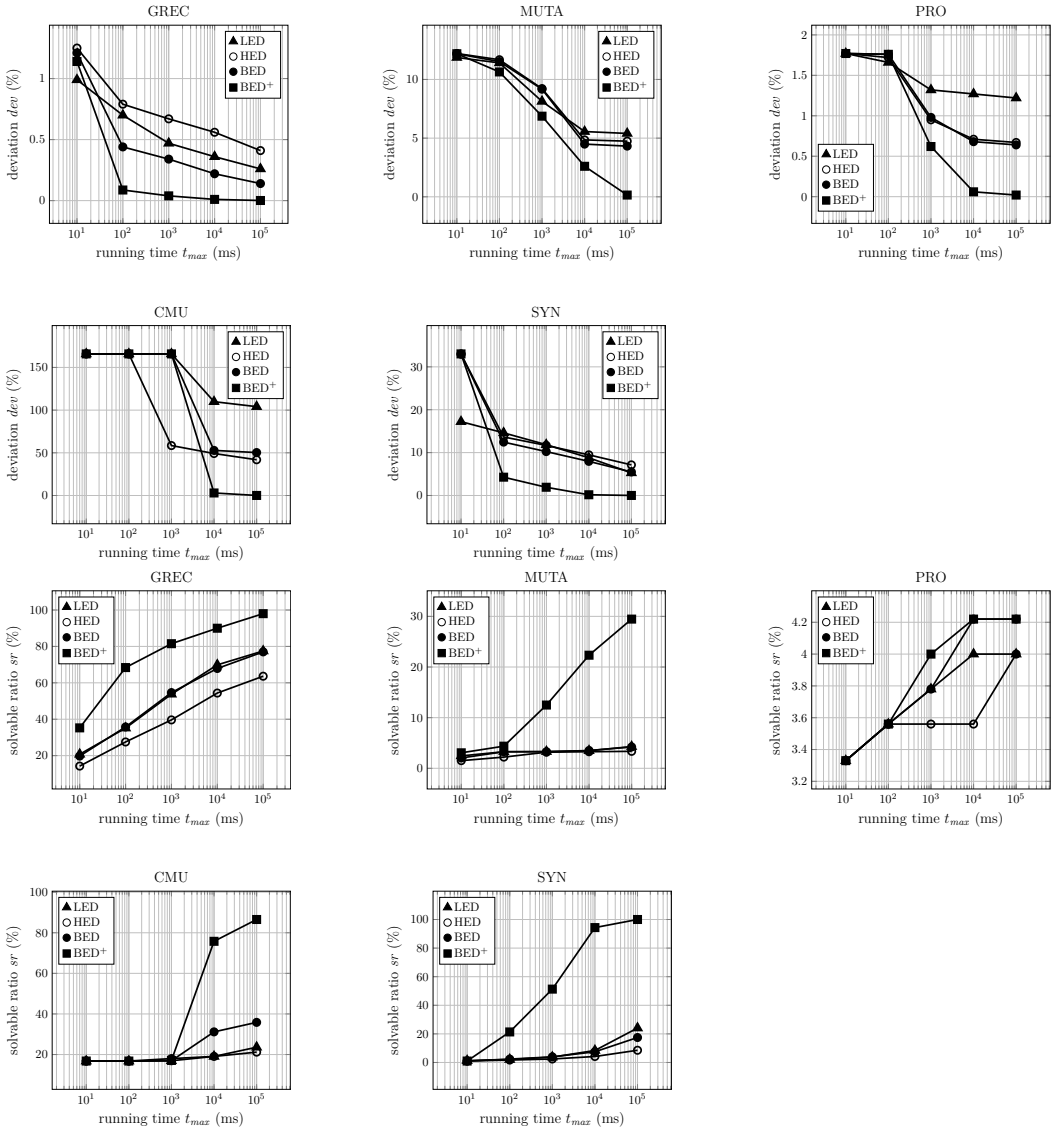


Figure 1 The *dev* (top two rows) and *sr* (bottom two rows) under different running time.

6 Conclusion and future works

In this paper, we analyze the relationship among three state-of-the-art GED lower bounds that are widely used as heuristic estimations in the tree-based search algorithm for the GED computation. Specifically, we demonstrate that $BED(G, Q) \geq LED(G, Q)$ and $BED(G, Q) \geq$

$HED(G, Q)$ for any two undirected graphs G and Q. Furthermore, based upon BED we propose an efficient heuristic estimation BED^+ and demonstrate that BED^+ still estimates a cost that is not greater than the real cost. Experimental results on four real and one synthetic datasets confirm that BED^+ can achieve the best performance under both small and large running time.

When calculating the heuristic estimation $BED^+(G_r^2, Q_r^2)$, we first compute the transformation cost (i.e., $f_B(\cdot, \cdot)$) of two compared branch structures. In fact, the transformation cost of these two branch structures may have been calculated many times in the previous traversal of the search tree. Future work will consider how to build a suitable index structure to maintain the transformation cost of these traversed branch structures in order to accelerate the computation of $BED^+(G_r^2, Q_r^2)$.

References

1 Z. Abu-Aisheh, R. Raveaux, and J. Y. Ramel. A graph database repository and performance evaluation metrics for graph edit distance. In *GbRPR*, pages 138–147, 2015.

2 Z. Abu-Aisheh, R. Raveaux, and J. Y. Ramel. Anytime graph matching. *Pattern Recogn Lett.*, 84:215–224, 2016. doi:10.1016/J.PATREC.2016.10.004.

3 Z. Abu-Aisheh, R. Raveaux, J. Y. Ramel, and P. Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In *ICPRAM*, pages 271–278, 2015.

4 D. B. Blumenthal and J. Gamper. Exact computation of graph edit distance for uniform and non-uniform metric edit costs. In *GbRPR*, pages 211–221, 2017.

5 D. B. Blumenthal and J. Gamper. Improved lower bounds for graph edit distance. *IEEE Trans. Knowl Data Eng.*, 30(3):503–516, 2018. doi:10.1109/TKDE.2017.2772243.

6 D. B. Blumenthal and J. Gamper. On the exact computation of the graph edit distance. *Pattern Recogn Lett.*, 134:46–57, 2020. doi:10.1016/J.PATREC.2018.05.002.

7 B. Bonet and H. Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001. doi:10.1016/S0004-3702(01)00108-4.

8 L. Chang, X. Feng, X. Lin, L. Qin, and W. Zhang. Efficient graph edit distance computation and verification via anchor-aware lower bound estimation. *CoRR*, 2017. arXiv:1709.06810.

9 X. Chen, H. Huo, J. Huan, and J. S. Vitter. An efficient algorithm for graph edit distance computation. *Knowl.-Based Syst.*, 163:762–775, 2019. doi:10.1016/J.KNOSYS.2018.10.002.

10 X. Chen, H. Huo, J. Huan, J. S. Vitter, W. Zheng, and L. Zou. MSQ-Index: A succinct index for fast graph similarity search. *IEEE Trans. Knowl Data Eng.*, 33(6):2654–2668, 2021. doi:10.1109/TKDE.2019.2954527.

11 X. Chen, Y. Wang, H. Huo, and J. S. Vitter. An efficient heuristic for graph edit distance [source code], June 2019. URL: https://github.com/Hongweihuo-Lab/Heur-GED.

12 James Cheng, Yiping Ke, and Wilfred Ng. GraphGen — a synthetic graph data generator. URL: https://cse.hkust.edu.hk/graphgen/.

13 CMU house and hotel datasets. URL: https://github.com/dbblumenthal/gedlib/blob/master/data/datasets/CMU-GED.

14 X. Cortés and F. Serratosa. Learning graph-matching edit-costs based on the optimality of the oracle's node correspondences. *Pattern Recogn Lett.*, 56:22–29, 2015. doi:10.1016/J.PATREC.2015.01.009.

15 A. Fischer, R. Plamondon, Y. Savaria, K. Riesen, and H. Bunke. A Hausdorff heuristic for efficient computation of graph edit distance. *Structural, Syntactic, and Statistical Pattern Recognition*, LNCS 8621:83–92, 2014.

16 A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H. Bunke. Approximation of graph edit distance based on Hausdorff matching. *Pattern Recogn.*, 48(2):331–343, 2015. doi:10.1016/J.PATCOG.2014.07.015.

17 K. Gouda and M. Hassaan. CSI_GED: An efficient approach for graph edit similarity computation. In *ICDE*, pages 256–275, 2016.

18 D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Trans. Pattern Anal Mach Intell.*, 28(8):1200–1214, 2006. doi:10.1109/TPAMI.2006.152.

19 J. Kim. Efficient graph edit distance computation using isomorphic vertices. *Pattern Recogn Lett.*, 168(2023):71–78, 2023. doi:10.1016/J.PATREC.2023.03.002.

20 H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955. doi:10.1002/nav.3800020109.

21 J. Lerouge, Z. Abu-Aisheh, R. Raveaux, P. Héroux, and S. Adam. New binary linear programming formulation to compute the graph edit distance. *Pattern Recogn.*, 72:254–265, 2017. doi:10.1016/J.PATCOG.2017.07.029.

22 J. Liu, M. Zhou, S. Ma, and L. Pan. MATA*: Combining learnable node matching with A* algorithm for approximate graph edit distance computation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23)*, pages 1503–1512, 2023.

23 Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. Computing graph edit distance via neural graph matching. *Proceedings of the VLDB Endowment*, 16(8):1817–1829, 2023. doi:10.14778/3594512.3594514.

24 K. Riesen and H. Bunke. IAM graph database repository for graph based pattern recognition and machine learning. *Structural, Syntactic, and Statistical Pattern Recognition*, pages 287–297, 2008.

25 K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, 27(7):950–959, 2009. doi:10.1016/J.IMAVIS.2008.04.004.

26 K. Riesen, S. Emmenegger, and H. Bunke. A novel software toolkit for graph edit distance computation. In *GbRPR*, pages 142–151, 2013.

27 K. Riesen, S. Fankhauser, and H. Bunke. Speeding up graph edit distance computation with a bipartite heuristic. In *MLG*, pages 21–24, 2007.

28 S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach (2nd ed.)*. Prentice-Hall, New Jersey, USA, 2002.

29 O Schütze, X. Esquivel, A. Lara, and C. A. C. Carlos. Using the averaged Hausdorff distance as a performance measure in evolutionary multiobjective optimization. *IEEE Trans. Evol. Comput.*, 16(4):504–522, 2012. doi:10.1109/TEVC.2011.2161872.

30 Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009. doi:10.14778/1687627.1687631.

31 W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. Efficient graph similarity search over large graph databases. *IEEE Trans. Knowl Data Eng.*, 27(4):964–978, 2015. doi:10.1109/TKDE.2014.2349924.

A Proof of Lemma 8

Proof. We prove this lemma by considering the following two cases:

Case I. when $u = \varepsilon$ or $v = \varepsilon$. We first discuss the case $u = \varepsilon$. It is trivial to know that $f_H(u, \varepsilon) = f_B(u, \varepsilon) = c(u \to \varepsilon) + \frac{1}{2}\sum_{e \in N_u} c(e \to \varepsilon)$. Similarly, when $v = \varepsilon$, we also have $f_H(u, v) = f_B(u, v)$. Thus, when $u = \varepsilon$ or $v = \varepsilon$, the lemma follows.

Case II. when $u \neq \varepsilon$ and $v \neq \varepsilon$. Then, we know that

$$f_B(u, v) = c(u \to v) + \frac{1}{2} MLS(N_u, N_v), \tag{12}$$

$$f_H(u, v) = \frac{1}{2}\left\{ c(u \to v) + \frac{1}{2} HED(N_u, N_v) \right\} \tag{13}$$

where $MLS(N_u, N_v) = \min_{\varrho: N_u^\varepsilon \to N_v^\varepsilon} \sum_{e \in N_u^\varepsilon} c(e \to \varrho(e))$, and ϱ is a bijection from N_u^ε to N_v^ε.

In order to prove $f_H(u,v) \leq \frac{1}{2} f_B(u,v)$, it suffices from (12) and (13) to prove

$$HED(N_u, N_v) \leq MLS(N_u, N_v),$$

which we do as follows:

(i) Rewriting $MLS(N_u, N_v)$ and $HED(N_u, N_v)$:

$$MLS(N_u, N_v) = \sum_{e \in N_u^\varepsilon} c(e \to \varrho_{\min}(e)) = \sum_{e \in N_u^\varepsilon} c(e \to y),$$

where $\varrho_{\min}$ is the *bijection* from N_u^ε to N_v^ε that $MLS(N_u, N_v)$ achieves the minimum value; $y = \varrho_{\min}(e) \in N_v^\varepsilon$ is e's mapped edge under the bijection $\varrho_{\min}$, for $\forall e \in N_u^\varepsilon$;

$$HED(N_u, N_v) = \sum_{e \in N_u^\varepsilon} \Big\{ f_H(e, \chi_1(e)) + f_H(\chi_2(y), y) \Big\},$$

where χ_1 is the *mapping* from N_u^ε to N_v^ε satisfying $\chi_1(e) = \arg\min_{e' \in N_v^\varepsilon} f_H(e, e')$, for $\forall e \in N_u^\varepsilon$; and χ_2 is the *mapping* from N_v^ε to N_u^ε satisfying $\chi_2(y) = \arg\min_{e \in N_u^\varepsilon} f_H(e, y)$.

(ii) Proving $f_H(e, \chi_1(e)) + f_H(\chi_2(y), y) \leq c(e \to y)$: According to the definition of χ_1 and χ_2, $f_H(e, \chi_1(e)) \leq \min\{f_H(e, y), f_H(e, \varepsilon)\}$ and $f_H(\chi_2(y), y) \leq \min\{f_H(e, y), f_H(\varepsilon, y)\}$. We discuss the following cases (a)–(d):

- **(a)** When $e = \varepsilon$ and $y = \varepsilon$, then $f_H(e, \chi_1(e)) + f_H(\chi_2(y), y) \leq f_H(\varepsilon, \varepsilon) + f_H(\varepsilon, \varepsilon) = c(\varepsilon \to \varepsilon) = 0$;
- **(b)** When $e \neq \varepsilon$ and $y = \varepsilon$, then $f_H(e, \chi_1(e)) + f_H(\chi_2(y), y) \leq f_H(e, \varepsilon) + f_H(\varepsilon, \varepsilon) = c(e \to \varepsilon)$;
- **(c)** When $e = \varepsilon$ and $y \neq \varepsilon$, the analysis is similar to that of (b);
- **(d)** When $e \neq \varepsilon$ and $y \neq \varepsilon$, then $f_H(e, \chi_1(e)) + f_H(\chi_2(y), y) \leq f_H(e, y) + f_H(e, y) = 2 f_H(e, y) = 2 \times \frac{1}{2} c(e \to y) = c(e \to y)$.

(iii) Combining both (i) and (ii), we have

$$HED(N_u, N_v) = \sum_{e \in N_u^\varepsilon} \Big\{ f_H(e, \chi_1(e)) + f_H(\chi_2(y), y) \Big\} \leq \sum_{e \in N_u^\varepsilon} c(e \to y) = MLS(N_u, N_v)$$

Therefore, $f_H(u,v) \leq \frac{1}{2} f_B(u,v)$ when $u \neq \varepsilon$ and $v \neq \varepsilon$. This completes the proof. ◀

B Examples of computing LED, HED and BED

In this section, we give an example of calculating three GED lower bounds, LED, HED and BED.

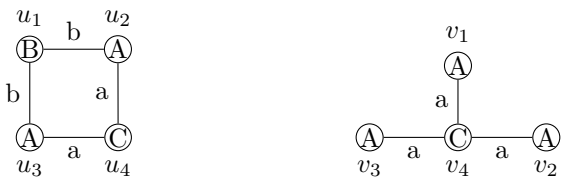


Figure 2 Graphs G (left) and Q (right).

Figure 2 shows two graphs G and Q, where "A", "B" and "C" denote vertex labels, and "a" and "b" denote edge labels. Consider the cost function c satisfying: (i) the cost of each vertex edit operation is 2, that is, $c(u \to v) = 2$ when two vertices $u \in V_G^\varepsilon$ and $v \in V_Q^\varepsilon$ have different labels, and $c(u \to v) = 0$ otherwise; (ii) the cost of each edge edit operation is 1, that is, $c(e_1 \to e_2) = 1$ when two edges $e_1 \in E_G^\varepsilon$ and $e_2 \in E_Q^\varepsilon$ have different labels, and $c(e_1 \to e_2) = 0$ otherwise. Based upon this cost function c, we discuss how to compute $LED(G, Q)$, $HED(G, Q)$ and $BED(G, Q)$ below using the examples shown in Figure 2.

(1) Computing $LED(G, Q)$

In $LED(G, Q)$ (see Definition 2 in main text), we need to compute the minimum substitution cost of vertices and edges of G and Q, i.e., $\lambda_V(G, Q)$ and $\lambda_E(G, Q)$. For $\lambda_V(G, Q) = \min_{\phi: V_G^\varepsilon \to V_Q^\varepsilon} \sum_{u \in V_G^\varepsilon} c(u \to \phi(u))$, we seek for a bijection ϕ from V_G^ε to V_Q^ε to minimize the linear sum $\lambda_V(G, Q)$; this is a well-investigated *linear sum assignment problem* (LSAP) and can be solved by the Hungarian algorithm [20] through the following two steps:

(1) Construct the vertex substitution cost matrix W^V, such that $W^V_{u,v} = c(u \to v)$ is the cost of substituting vertices $u \in V_G$ and $v \in V_Q$; $W^V_{u,\varepsilon} = c(u \to \varepsilon)$ is the cost of deleting u; and $W^V_{\varepsilon,v} = c(\varepsilon \to v)$ is the cost of inserting v. In this example, we compute W^V as

$$W^V = \begin{array}{c} \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ \varepsilon \\ \varepsilon \\ \varepsilon \\ \varepsilon \end{array} \begin{array}{c} \begin{array}{cccccccc} v_1 & v_2 & v_3 & v_4 & \varepsilon & \varepsilon & \varepsilon & \varepsilon \end{array} \\ \begin{pmatrix} 2 & 2 & 2 & 2 & 2 & \infty & \infty & \infty \\ 0 & 0 & 0 & 2 & \infty & 2 & \infty & \infty \\ 0 & 0 & 0 & 2 & \infty & \infty & 2 & \infty \\ 2 & 2 & 2 & 0 & \infty & \infty & \infty & 2 \\ 2 & \infty & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & 2 & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & 2 & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & \infty & 2 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

(2) Find the *optimal assignment* $\phi_{\min}$ that minimizes the linear sum on W^V. In this example, we find that $\phi_{\min} = \{(u_1 \to v_1), (u_2 \to v_2), (u_3 \to v_3), (u_4 \to v_4), (\varepsilon \to \varepsilon)\}$ is the optimal assignment, and then obtain $\lambda_V(G, Q) = W^V_{u_1,v_1} + W^V_{u_2,v_2} + W^V_{u_3,v_3} + W^V_{u_4,v_4} + W^V_{\varepsilon,\varepsilon} = 2$.

Similar to the above process, we can compute the edge substitution cost matrix W^E as follows:

$$W^E = \begin{array}{c} \\ e(u_1, u_2) \\ e(u_1, u_3) \\ e(u_2, u_4) \\ e(u_3, u_4) \\ \varepsilon \\ \varepsilon \\ \varepsilon \end{array} \begin{array}{c} \begin{array}{cccccccc} e(v_1, v_4) & e(v_2, v_4) & e(v_3, v_4) & \varepsilon & \varepsilon & \varepsilon & \varepsilon \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & \infty & \infty & \infty \\ 1 & 1 & 1 & \infty & 1 & \infty & \infty \\ 0 & 0 & 0 & \infty & \infty & 1 & \infty \\ 0 & 0 & 0 & \infty & \infty & \infty & 1 \\ 1 & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & 1 & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

With the Hungarian algorithm, we know that the optimal assignment on W^E is $\varphi_{\min} = \{(e(u_1, u_2) \to \varepsilon), (e(u_1, u_3) \to e(v_1, v_4)), (e(u_2, u_4) \to e(v_2, v_4)), (e(u_3, u_4) \to e(v_3, v_4)), (\varepsilon \to \varepsilon)\}$. Then, $\lambda_E(G, Q) = W^E_{e(u_1,u_2),\varepsilon} + W^E_{e(u_1,u_3),e(v_1,v_4)} + W^E_{e(u_2,u_4),e(v_2,v_4)} + W^E_{e(u_3,u_4),e(v_3,v_4)} + W^E_{\varepsilon,\varepsilon} = 2$. Combing $\lambda_V(G, Q)$ and $\lambda_E(G, Q)$, we have $LED(G, Q) = \lambda_V(G, Q) + \lambda_E(G, Q) = 2 + 2 = 4$.

(2) Computing $HED(G, Q)$

According to the definition of $HED(G, Q)$ (see Definition 3 in main text), we need to calculate the hausdorff matching cost $f_H(u, v)$ between two vertices $u \in V_G^\varepsilon$ and $v \in V_Q^\varepsilon$, and then perform a bidirectional matching between G and Q. When performing a matching from G to Q, we greedily seek for the minimum matching cost $\min_{v \in V_Q^\varepsilon} f_H(u, v)$ of each vertex u; then, the sum of these minimum costs is the matching cost from G to Q, i.e., $\sum_{u \in V_G} \min_{v \in V_Q^\varepsilon} f_H(u, v)$. Similarly, the matching cost from Q to G is $\sum_{v \in V_Q} \min_{u \in V_G^\varepsilon} f_H(u, v)$. Finally, the sum of the above two matching costs is $HED(G, Q)$. We can summarize the computation process of $HED(G, Q)$ as two steps:

(1) Construct the hausdorff matching cost matrix W^H, such that $W^H_{u,v} = f_H(u, v)$ is the hausdorff cost of matching vertex $u \in V_G$ to vertex $v \in V_Q$; $W^H_{u,\varepsilon} = f_H(u, \varepsilon)$ is the hausdorff cost of deleting u; and $W^H_{\varepsilon,v} = f_H(\varepsilon, v)$ is the hausdorff cost of inserting v, where $f_H(\cdot, \cdot)$ is defined in (2) in main text. In this example, we can compute W^H as

$$W^H = \begin{array}{c} \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ \varepsilon \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & \varepsilon \end{array} \\ \begin{pmatrix} 1.375 & 1.375 & 1.375 & 1.625 & 3 \\ 0.125 & 0.125 & 0.125 & 1.125 & 3 \\ 0.125 & 0.125 & 0.125 & 1.125 & 3 \\ 1 & 1 & 1 & 0 & 3 \\ 2.5 & 2.5 & 2.5 & 3.5 & 0 \end{pmatrix} \end{array}$$

(2) Based upon W^H, compute $\sum_{u \in V_G} \min_{v \in V_Q^\varepsilon} W^H_{u,v}$ and $\sum_{v \in V_Q} \min_{u \in V_G^\varepsilon} W^H_{u,v}$. In this example, we trivially obtain $HED(G, Q) = \sum_{u \in V_G} \min_{v \in V_Q^\varepsilon} W^H_{u,v} + \sum_{v \in V_Q} \min_{u \in V_G^\varepsilon} W^H_{u,v} = (W^H_{u_1,v_1} + W^H_{u_2,v_1} + W^H_{u_3,v_1} + W^H_{u_4,v_4}) + (W^H_{u_2,v_1} + W^H_{u_2,v_2} + W^H_{u_3,v_2} + W^H_{u_4,v_4}) = (1.375 + 0.125 + 0.125 + 0) + (0.125 + 0.125 + 0.125 + 0) = 2$.

Note that when calculating $W^H_{u,v}$ (i.e., $f_H(u, v)$), we need to calculate $HED(N_u, N_v)$ (see Equation (3) in main context), where N_u and N_v are the sets of edges adjacent to u and v, respectively. The computation of $HED(N_u, N_v)$ is similar to the above process of computing $HED(G, Q)$; and thus, we omit the detailed calculation here.

(3) Computing $BED(G, Q)$

The process of calculating $BED(G, Q)$ is similar to that of calculating $\lambda_V(G, Q)$, which is also looking for a bijection ρ from V_G^ε to V_Q^ε to minimize the linear sum $\sum_{u \in V_G^\varepsilon} f_B(u, \rho(u))$. The computation contains two steps:

(1) Construct the branch matching cost matrix W^B, such that $W^B_{u,v} = f_B(u, v)$ is the branch cost of matching vertex $u \in V_G$ to vertex $v \in V_Q$; $W^B_{u,\varepsilon} = f_B(u, \varepsilon)$ is the branch cost of deleting u; and $W^B_{\varepsilon,v} = f_B(\varepsilon, v)$ is the branch cost of inserting v, where $f_B(\cdot, \cdot)$ is defined in (5) in main text. In this example, we can compute W^B as

$$W^B = \begin{array}{c} \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ \varepsilon \\ \varepsilon \\ \varepsilon \\ \varepsilon \end{array} \begin{array}{c} \begin{array}{cccccccc} v_1 & v_2 & v_3 & v_4 & \varepsilon & \varepsilon & \varepsilon & \varepsilon \end{array} \\ \begin{pmatrix} 3 & 3 & 3 & 3.5 & 3 & \infty & \infty & \infty \\ 0.5 & 0.5 & 0.5 & 3 & \infty & 3 & \infty & \infty \\ 0.5 & 0.5 & 0.5 & 3 & \infty & \infty & 3 & \infty \\ 2.5 & 2.5 & 2.5 & 0.5 & \infty & \infty & \infty & 3 \\ 2.5 & \infty & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & 2.5 & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & 2.5 & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & \infty & 3.5 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

(2) Find the *optimal assignment* $\rho_{\min}$ that minimizes the linear sum on W^B. In this example, we find that $\rho_{\min} = \{(u_1 \to v_1), (u_2 \to v_2), (u_3 \to v_3), (u_4 \to v_4), (\varepsilon \to \varepsilon)\}$ is the optimal assignment, and then obtain $BED(G,Q) = W^B_{u_1,v_1} + W^B_{u_2,v_2} + W^B_{u_3,v_3} + W^B_{u_4,v_4} + W^B_{\varepsilon,\varepsilon} = 4.5$.

Note that when calculating $W^B_{u,v}$ (i.e., $f_B(u,v)$), we need to calculate the minimum edge substitution cost between N_u and N_v, which is similar to the process of calculating $\lambda_E(,)$; and thus, we omit the detailed computation here.

For graphs G and Q in Figure 2, we finally obtain that $LED(G,Q) = 4$, $HED(G,Q) = 2$ and $BED(G,Q) = 4.5$. Clearly, $BED(G,Q) \geq LED(G,Q)$ and $BED(G,Q) \geq HED(G,Q)$.

C Successor generation

We discuss how to generate successors of each node in the GED search tree with Algorithm 2.

Consider an inner node $r = \{(u_1 \to v_{j_1}), \ldots, (u_\ell \to v_{j_\ell})\}$, where v_{j_k} is the mapped vertex of u_k in the GED search tree, for $1 \leq k \leq \ell$. BasicGenSuccr generates all the possible successors of r. First, we compute the sets of unmapped vertices in G and Q, respectively, i.e., $V^r_G = V_G \backslash \{u_1, \ldots, u_l\}$ and $V^r_Q = V_Q \backslash \{v_{j_1}, \ldots, v_{j_l}\}$. If $|V^r_G| > 0$, then we select a vertex z from $V^r_Q \cup \{\varepsilon\}$ as the mapped vertex of $u_{\ell+1}$, and consequently, obtain a successor *child* of r such that $child = r \cup \{(u_{\ell+1} \to z)\}$. Otherwise, all the vertices of G are processed; trivially, we obtain a leaf node $s = r \cup \bigcup_{z \in V^r_Q} \{(\varepsilon \to z)\}$.

Algorithm 2 BasicGenSuccr(r).

```
1  succ ← {}
2  V_G^r ← V_G \ {u_1, ..., u_ℓ}
3  V_Q^r ← V_Q \ {v_{j_1}, ..., v_{j_ℓ}}
4  if |V_G^r| > 0 then
5  |   foreach z ∈ V_Q^r ∪ {ε} do
6  |   |   child ← r ∪ {(u_{ℓ+1} → z)}
7  |   |   succ ← succ ∪ {child}
8  else
9  |   s ← r ∪ ⋃_{z ∈ V_Q^r} {(ε → z)}
10 |   succ ← succ ∪ {s}
11 return succ
```

On the Construction of Elastic Degenerate Strings

Nicola Rizzo
Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen
Department of Computer Science, University of Helsinki, Finland

Nadia Pisanti
Universitá di Pisa, Italy

Abstract

An *elastic degenerate string* (EDS) is a sequence of sets of strings. In the context of bioinformatics, EDSes can be used to represent the variations observed in a population from its consensus genome. Pattern matching and comparison problems on EDSes have been widely studied in the literature, but their construction has been largely omitted. We fill this gap by showing how algorithms originally developed for related problems of founder reconstruction can be adapted to minimize the total cardinality of the EDS sets and total length of the EDS strings in linear time, given suitable multiple alignments representing the input data.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Sorting and searching; Theory of computation → Dynamic programming; Applied computing → Genomics

Keywords and phrases multiple sequence alignment, pattern matching, data structures, segmentation algorithms, founder reconstruction, dynamic programming, semi-dynamic range minimum queries, positional Burrows–Wheeler transform

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.2

Category Research

Funding This work was partially supported by the PANGAIA, ALPACA, and TeamPerMed projects that received funding from the European Union's Horizon 2020 research and innovation programme with two first under the Marie Skłodowska-Curie grant agreements No. 872539 and 956229, respectively, and the third under grant agreement No. 101060011. NP was also partially supported by MUR PRIN 2022 YRB97K PINC.

1 Introduction

Computing an optimal partition of an interval $[1..c]$, that is, a *segmentation*, is a problem that arises naturally when dealing with the transformation of sequences that are aligned into c positions, after trivial and contradicting constraints are imposed on the desired features of the solution. Figure 1 gives an example of such segmentation on a *multiple sequence alignment* (MSA) of 6 sequences. In an MSA, the r input sequences are made to be of equal length c by adding gaps "$-$" forming a matrix of r rows and c columns which we will denote $\mathsf{MSA}[1..r, 1..c]$. The quality of such MSA can be measured in many ways, but already the problem of placing gaps such that the number of columns containing a single symbol is maximized equals the problem of finding the length of the longest common subsequence (LCS) of the input sequences, which is NP-hard [19]. Nevertheless, MSAs play a vital role in bioinformatics, and practical heuristics have been developed to obtain high quality MSAs [23]. In this paper, we assume an MSA to be given as input, and we look for rigorous ways to convert the MSA into a sequence of sets of strings, that is, into an *elastic degenerate string*

(EDS) [18] (also known as elastic-degenerate text). In Figure 1, we can observe that a segmentation of the MSA directly induces an EDS by interpreting the strings of each segment to be part of a set.

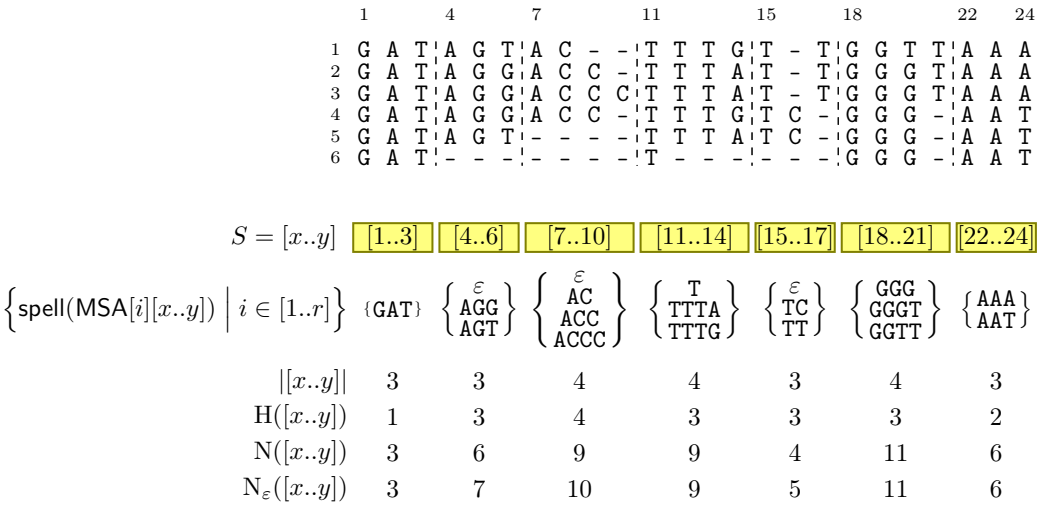


Figure 1 Example of an $\mathsf{MSA}[1..6, 1..24]$ and a segmentation S containing $k = 7$ segments. The maximum length $\mathrm{L}(S)$ is 4, the height $\mathrm{H}(S)$ is 4, the cardinality $\mathrm{m}(S)$ is 19, the size $\mathrm{N}(S)$ is 43, and the gap-aware size $\mathrm{N}_\varepsilon(S)$ is 51.

The transformation from an MSA to an EDS through segmentation can be seen as lossy compression: we merge together equal strings inside each segment and we ignore the exact connections between segments. Such lossy compression can be a desired feature, for example, for hiding the sensitive full sequence input by publishing only the EDS. A more rudimentary reason for such conversion is that EDSes and related pangenome graphs do compress well and enable fast processing and analysis of variation in populations [9]. A notable feature of this scheme is that if a query string occurs in a row of the MSA, then it will also occur in the EDS, but not the other way around. Various ways to optimize the transformation to minimize false positives while not sacrificing privacy could be considered. Minimizing a specific parameter is useful when the downstream analyses of the resulting EDS employ algorithms whose complexity depends from that parameter.

This paper is initiating the study of EDS optimization by looking at some selected optimization criteria (Section 3) for which we can modify algorithms from related literature. We give linear-time algorithms (Section 4) to minimize the total cardinality of the EDS sets (*cardinality*) and total length of the strings in the EDS sets (*size*) under specific assumptions: the algorithms directly work for gapless MSAs, and they serve as heuristics when gaps are interpreted as part of the alphabet. In the latter case, the optimization criteria should be adjusted accordingly: see Section 5 for the multiple problems that we leave open for general MSAs. We start by reviewing the related work in Section 2.

2 Related work

Founder reconstruction. Ukkonen [25] studied the problem of explaining a given set of r aligned sequences representing c haplotype sites (thus no insertions or deletions) with the recombination of a few *founder sequences*, under the simple *crossover* model of recombination (sequence switching at aligned positions); minimizing the founder set is

trivial if the recombination can happen at any position, but if the recombinations are allowed only at selected positions, then the resulting segmentation problems – minimizing the maximum cardinality of a segment (the number of founders in [25]) given a lower bound on the segment length, or given a target average segment length size to surpass – can be solved in polynomial time.

Norri et al. [22] solved one of the problems posed by Ukkonen – the segmentation of aligned sequences minimizing the maximum height given a segment-length lower bound L – in linear $O(rc)$ time and $O(r + L)$ space. The solution proceeds in a dynamic programming fashion, from left to right, and for each aligned position y it uses the *positional Burrows–Wheeler Transform* [10], enhanced by a few other arrays, to efficiently compute all the possible heights of a segment ending at y and their optimal recursive values.

Cazaux et al. [8] studied the related problem of maximizing the minimum segment length or the average segment length of any segmentation that has height of at most a given H, motivated by the minimization of the size of the resulting pangenomic index based on founder sequences. To do so, they recognize that the approach by Norri et al. [22] fits into a more general left-to-right *column-stream* model of computation, and use *range maximum queries* to efficiently compute the recursive values. For this last task, they modify existing techniques for the semi-dynamic setting where an array is indexed for amortized constant-time range queries in an online fashion.

Algorithms on EDSes. There is a wide literature on pattern search and comparison on (elastic) degenerate strings [18, 17, 5, 6, 1, 21, 4, 2, 3] where algorithmic results exhibit complexity that depend on both the cardinality and the size of input EDSes. Moreover, the latest algorithms for EDS-based pangenome comparison (intersection, matching statistics, similarity and distance measures) also depend on the cardinality and size metrics [15, 14, 16]. These results give further motivation for our focus on optimizing these measures.

Indexable Founder Graphs. *Elastic Founder Graphs* (EFGs), introduced by Equi et al. [13], take the simplified recombination model of founder sequences (recombination at selected positions only) to transform an MSA of r sequences and c aligned positions – with insertions and deletions and thus gaps – into an acyclic Elastic Block Graph: the segmentation results into consecutive blocks of nodes. In the general case, the resulting graph is still hard to index just like more general graphs [12, 11], so Equi et al. [13] prove that if each chosen segment spells strings that exclusively occur from the segments' starting column, then this property is conserved in the resulting graph, which is easy to index. This results in a framework of linearithmic (i.e. $O(rc \log(rc))$) time constructions of indexable EFGs, which support fast pattern matching.

Rizzo et al. [24] improved some construction solutions to linear time (i.e. $O(rc)$) while studying the problem of minimizing the maximum segment length or maximum height of a segmentation resulting in an indexable EFG.

MSA covers. While segmentation induces a limited class of acyclic pangenome graph representations, there is an analogous approach by Cartes et al. [7] to cover MSAs and generate arbitrary acyclic graphs; the related optimization problems are much harder, although small instances can be solved using integer linear programming.

3 Preliminaries

We denote integer interval $\{1,\ldots,n\}$ as $[1..n]$. Let $\mathsf{MSA}[1..r,1..c] \in (\Sigma \cup \{-\})^{r\times c}$ be a *multiple sequence alignment* of r rows and c columns representing the input sequences (or *strings*) and the aligned positions, respectively. It is built from a finite integer alphabet $\Sigma = [1..\sigma]$ ($\Sigma = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ in our examples) augmented with the *gap symbol* $- \notin \Sigma$ to represent insertions and deletions. It is immediate to see that the alignment takes at most $O(rc)$ words of space, or $O(rc\log|\Sigma|)$ bits of space. We denote the i-th row of the MSA as $\mathsf{MSA}[i,1..c]$, and the concatenation of its characters from position x to y, with $x \leqslant y$, as $\mathsf{MSA}[i,x..y]$. We say that $\mathsf{MSA}[1..r,1..c]$ is *gapless* if no gap symbol is used. The operator $\mathrm{spell}(T)$ removes all the gap symbols from a given string $T \in (\Sigma\cup\{-\})^c$: for example, $\mathrm{spell}(\mathsf{MSA}[i,1..c])$ is the i-th (unaligned) sequence. If T contains only gaps then $\mathrm{spell}(T) = \varepsilon$, the empty string. Consider the following definition alongside Figure 1.

▶ **Definition 1** (Segmentation). *Given* $\mathsf{MSA}[1..r,1..c] \in (\Sigma\cup\{-\})^{r\times c}$, *a* segmentation *of the MSA is a partition* $S = S_1, S_2, \ldots, S_k$ *of* $[1..c]$, *i.e. a sequence of contiguous and non-overlapping intervals covering* $[1..c]$. *In symbols*

$$S_1 = [x_1..y_1],\ \ldots,\ S_k = [x_k..y_k] \qquad \text{with} \qquad \begin{array}{c} x_1 = 1, \\ y_k = c, \text{ and} \\ y_j + 1 = x_{j+1}\ \forall j \in [1..k-1]. \end{array}$$

We denote:

- *the* number of segments *of* S *as* $|S| = k$*;*
- *the* maximum segment length $\max_{j\in[1..k]}|S_j|$ *of* S *as* $\mathrm{L}(S)$*, where* $|[x..y]| = y - x + 1$*;*
- *the* height $\max_{j\in[1..k]}\mathrm{H}(S_j)$ *of* S *as* $\mathrm{H}(S)$*, where the height* $\mathrm{H}(S_j)$ *of a segment* $S_j = [x..y]$ *is defined as*

$$\mathrm{H}([x..y]) = \left|\left\{\, \mathrm{spell}(\mathsf{MSA}[i,x..y]) \;\middle|\; i \in [1..r] \right\}\right|$$

  (note that the empty string is counted by H*);*
- *the* cardinality $\sum_{j\in[1..k]}\mathrm{H}(S_j)$ *as* $\mathrm{m}(S)$*;*
- *the* size $\sum_{j\in[1..k]}\mathrm{N}(S_j)$ *as* $\mathrm{N}(S)$*, where the size* $\mathrm{N}(S_j)$ *of a segment* $S_j = [x..y]$ *is defined as*

$$\mathrm{N}([x..y]) = \sum_{T\in\,\mathrm{spell}(\mathsf{MSA}[1..r,x..y])} |T| \qquad \text{with}$$

$$\mathrm{spell}(\mathsf{MSA}[1..r,x..y]) = \left\{\, \mathrm{spell}(\mathsf{MSA}[i,x..y]) \;\middle|\; i\in[1..r] \right\}$$

  (note that the empty string ε *contributes* 0 *units to* $\mathrm{N}([x..y])$*); and*
- *the* gap-aware size $\mathrm{N}_\varepsilon(S)$ *as above, substituting* $\mathrm{N}([x..y])$ *with*

$$\mathrm{N}_\varepsilon([x..y]) = \begin{cases} \mathrm{N}([x..y]) + 1 & \text{if } \varepsilon \in \mathrm{spell}(\mathsf{MSA}[1..r,x..y]), \\ \mathrm{N}([x..y]) & \text{otherwise.} \end{cases}$$

Consider the problem of finding a segmentation S of minimum cardinality $\mathrm{m}(S)$. If the MSA contains few but very long sequences, that is, $r \ll c$, the trivial segmentation $S^{\equiv} = [1..c]$ might be optimal. We can encourage a certain level of recombination in the corresponding EDS by setting an upper bound U on the length of the accepted segments.

▶ **Problem 2** (min-U-cardinality). Given $\mathsf{MSA}[1..r,1..c]$ and an upper bound U on the segment length, find a segmentation S containing segments of length at most U and minimizing the cardinality $\mathrm{m}(S)$.

On the other hand, consider finding a segmentation S of minimum gap-aware size $\mathrm{N}_\varepsilon(S)$. In Section 4.1 we show that the trivial segmentation $S^{|||} = [1], \ldots, [c]$ has always minimum size in the gapless setting, resulting in a highly recombinant EDS. We can symmetrically discourage recombination in regions of low sequence similarity by fixing a lower bound L on the length of the accepted segments.

▶ **Problem 3** (min-L-size)**.** Given $\mathsf{MSA}[1..r, 1..c]$ and a lower bound L on the segment length, find a segmentation S containing segments of length at least L and minimizing the gap-aware size $\mathrm{N}_\varepsilon(S)$.

One crucial data structure that we use in Section 4 is based on sorting iteratively the MSA rows while reading the alignment from left to right, column by column. Let Σ have an implicit total order $\leqslant$ on its characters. Given strings $S, T \in \Sigma^c$, we say that S is *lexicographically smaller* than T, in symbols $S \prec_{\text{lex}} T$, if $S[1..x] = T[1..x]$ and $S[x+1] < T[x+1]$ for some $x \in [0..c-1]$ (where $S[1..0]$ is equal to the empty string ε). Symmetrically, S is *co-lexicographically smaller* than T, in symbols $S \prec_{\text{colex}} T$, if $S[x..c] = T[x..c]$ and $S[x-1] < T[x-1]$ for some $x \in [1..c+1]$ (where $S[c+1..c] = \varepsilon$). Moreover, we call $S[1..x]$ a *prefix* of S and $S[1..x]$ a *suffix* of S. They are non-empty if $x \geqslant 1$ and $x \leqslant c$, respectively.

▶ **Definition 4** (Positional Burrows–Wheeler transform [10])**.** *The* positional Burrows–Wheeler transform *(pBWT) of a* gapless $\mathsf{MSA}[1..r, 1..c]$ *for position* $x \in [1..c]$ *consists of array* $\mathtt{a}_x[1..r]$*, representing the prefixes* $\mathsf{MSA}[i, 1..x]$ *for* $i \in [1..r]$ *sorted co-lexicographically, and array* $\mathtt{d}_x[1..r]$*, describing the length of the longest common suffixes of adjacent prefixes in the sorted order. More specifically:*

- $\mathtt{a}_x[1..r]$ *is the permutation of* $[1..r]$ *such that*

$$\mathsf{MSA}[\mathtt{a}_x[1], 1..x] \preceq_{\text{colex}} \mathsf{MSA}[\mathtt{a}_x[2], 1..x] \preceq_{\text{colex}} \cdots \preceq_{\text{colex}} \mathsf{MSA}[\mathtt{a}_x[r], 1..x];$$

- $\mathtt{d}_x[i]$ *is an integer in* $[1..x+1]$ *such that the longest common suffix between* $\mathsf{MSA}[\mathtt{a}_x[i], 1..x]$ *and* $\mathsf{MSA}[\mathtt{a}_x[i-1], 1..x]$ *has length* $\mathtt{d}_x[i]$*, if* $i > 1$ *and such suffix is non-empty, otherwise* $\mathtt{d}_x[i] = x + 1$.

▶ **Lemma 5** ([10, 20])**.** *Let* $\mathsf{MSA}[1..r, 1..c]$ *be a gapless MSA over integer alphabet* Σ *of size* $O(r)$*. Given the positional Burrows–Wheeler transform for position* $x \in [1..c-1]$*, that is, arrays* $\mathtt{a}_x[1..r]$ *and* $\mathtt{d}_x[1..r]$*, arrays* $\mathtt{a}_{x+1}[1..r]$ *and* $\mathtt{d}_{x+1}[1..r]$ *can be computed in* $O(r)$ *time.*

Finally, one last data structure that we use in this paper indexes an integer array to later find out the minimum value inside a given range of array positions.

▶ **Lemma 6** (Semi-dynamic range minimum query data structure [8, Lemma 7])**.** *There exists a data structure that maintains an integer array* $\mathtt{I}[1..c]$ *and supports: the append query, which adds a new element to the end of* $\mathtt{I}$ *and increments* c*, in* $O(1)$ *amortized time; and the Range Minimum Query, which for any given* $x, y \in [1..c]$ *with* $x \leqslant y$*, computes a position* $k \in [1..c]$ *such that* $\mathtt{I}[k] = \min\{Q[\ell] : x \leqslant \ell \leqslant y\}$*, in* $O(1)$ *time.*

4 Solutions

In Section 4.1 we remark useful properties of gapless MSAs. Then, in Sections 4.2 and 4.3 we provide linear-time solutions to min-U-cardinality and min-L-size for MSAs under this setting.

4.1 Basic properties and non-trivial settings

Before investigating under which setting problems min-U-cardinality and min-L-size are trivial, consider the following properties of MSAs without gaps that we will exploit in this section.

▶ **Observation 7** (Monotonicity of left extensions [22])**.** *Given* $\mathsf{MSA}[1..r, 1..c]$*, for any* x, y *with* $1 \leqslant x \leqslant y \leqslant c$ *we say that* $[x..y]$ *is a* left extension *of suffix* $\mathsf{MSA}[1..r, y+1..c]$*. If the MSA is* gapless*, the following monotonicity property holds for any left extension* $[x..y]$ *of* $\mathsf{MSA}[1..r, y+1..c]$*:*

$$r \geqslant \mathrm{H}([x'..y]) \geqslant \mathrm{H}([x..y]) \qquad \forall x' < x.$$

▶ **Observation 8.** *Given a* gapless $\mathsf{MSA}[1..r, 1..c]$*, the (gap-aware) size of any segment* $[x..y]$ *is equal to the height times the length of the segment, in symbols* $\mathrm{N}_\varepsilon([x..y]) = \mathrm{N}([x..y]) = |x..y| \cdot \mathrm{H}([x..y])$*. Indeed, if the MSA does not contain the gap symbol then* $\mathrm{spell}(\mathsf{MSA}[i, x..y]) = \mathsf{MSA}[i, x..y]$ *for all* $i \in [1..r]$ *and all strings spelled by segment* $[x..y]$ *have length* $y - x + 1$*.*

The simplest version of min-L-size occurs when the MSA contains no gap symbol and there is no lower bound on the segment length, that is, when $L = 1$. Intuitively, this setting presents the same trivial solution as in the founder reconstruction problem (see [25, Theorem 1]), since transforming each column into its own segment achieves the best possible compression. Let $[x]$ denote the singleton range $[x..x] = \{x\}$.

▶ **Theorem 9.** *An optimal solution to min-*1*-size for a* gapless $\mathsf{MSA}[1..r, 1..c]$ *is the trivial segmentation* $S^{|||} = [1], [2], \ldots, [c]$*.*

Proof. Consider any segment $[x..y]$ with $|[x..y]| > 1$. Then $[x..y-1]$ and $[y]$ are also valid segments, and since the MSA is gapless

$$\begin{aligned} \mathrm{N}[x..y] &= |[x..y-1]| \cdot \mathrm{H}([x..y]) + 1 \cdot \mathrm{H}([x..y]) && \text{(Observation 8)} \\ &\geqslant |[x..y-1]| \cdot \mathrm{H}([x..y-1]) + |[y]| \cdot \mathrm{H}([y]) && \text{(Observation 7)} \\ &= \mathrm{N}([x..y-1]) + \mathrm{N}([y]). \end{aligned}$$

Thus breaking down segments longer than 1 never increases the size and the (gap-aware) size of $S^{|||}$ is less than or equal to the size of any other segmentation. ◀

However, the trivial solution is not valid if $L > 1$ or it can be suboptimal if the MSA contains gaps: in the former setting, greedily picking segments of length exactly L can be suboptimal, and in the latter case each empty string ε contributes 1 to the total size of the segmentation so the parts of the MSA with long runs of gaps need to be carefully considered. See Figure 2.

Regarding the problem of minimizing the cardinality of the resulting segmentation, min-U-card behaves differently than the other segmentation problems, as it does not admit a trivial optimal segmentation even if there is no upper bound on the segment length (i.e. $U = c$) and the MSA does not have any gap symbol.

▶ **Observation 10.** *Trivial segmentations* $S^{|||} = [1], [2], \ldots, [c]$ *and* $S^{\equiv} = [1..c]$ *are not optimal with respect to min-c-card for some* gapless $\mathsf{MSA}[1..r, 1..c]$*: on one hand, in regions of high similarity longer segments can be preferable as they can spell very few strings; on the other hand, shorter segments generate less strings in regions of high diversity. See the example* $\mathsf{MSA}[1..10, 1..4]$ *in Figure 3.*

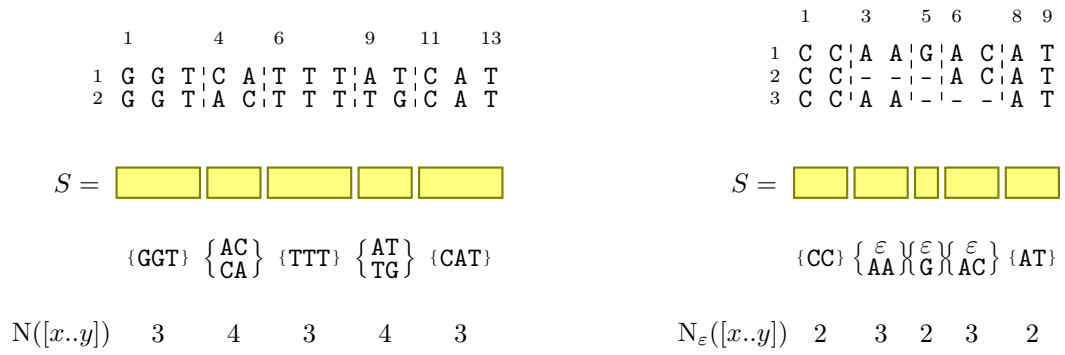


Figure 2 On the left, a gapless MSA where the optimal solution to min-2-size (i.e. the lower bound on the minimum segment length is 2) has size 17 and is non-trivial. On the right, an MSA with gaps for which the optimal solution to min-1-size is non-trivial and has gap-aware size 12.

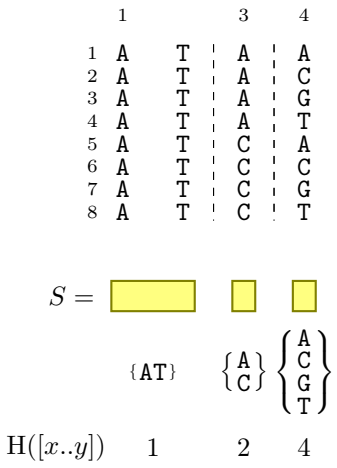


Figure 3 Example of a gapless MSA$[1..8, 1..4]$ for which the trivial segmentations $S^{\equiv} = [1..c] = [1..4]$ and $S^{|||} = [1], [2], [3], [4]$ are not a solution to min-4-size, since their cardinality is equal to 8. Instead, $S = [1..2], [3], [4]$ (shown) is such that $\mathrm{m}(S) = 7$.

4.2 Minimizing the cardinality

Given $\mathsf{MSA}[1..r, 1..c]$ and $U \in [1..c]$, for any $y \in [1..c]$ we define m_y as the size of an optimal solution of min-U-cardinality on instance $\mathsf{MSA}[1..r, 1..y]$ respecting segment upper bound U. Then, the following recursion holds, since the cardinality of a segmentation S is the sum of the individual cardinalities of its segments:

$$\mathrm{m}_0 = 0,$$
$$\mathrm{m}_y = \min_{x \in [0..y-1]} \Big(\mathrm{m}_x + \mathrm{H}([x+1..y]) \Big) \qquad \forall y \in [1..c], \tag{1}$$

and it is easy to see that m_c is equal to the cardinality $\mathrm{m}(S)$ of an optimal segmentation S.

Norri et al. [22] recognized that, at least for gapless MSAs, all changes in segment height can be described compactly and in a range fashion by fixing an end column y and considering values $\mathrm{H}([x..y])$ for $x \in [1..y]$. This concept was restated and extended to MSAs with gaps by Rizzo et al. [24] into the *extensions* of the MSA suffix $\mathsf{MSA}[1..r, y+1..c]$ that are *meaningful*. Consider the following definition alongside Figure 4.

▶ **Definition 11** (Meaningful left extensions [22, 24]). *Given* $\mathsf{MSA}[1..r, 1..c]$*, let* $L, U \in [1..c]$ *be some given lower bound and upper bound on the segment length*[1]*, with* $L \leqslant U$*. For any* $y \in [1..c]$ *we denote with* $\mathcal{L}_y = \ell_{y,1}, \ell_{y,2}, \ldots, \ell_{y,d_y}$ *the* meaningful left extensions *of* $\mathsf{MSA}[1..r, y+1..c]$*, meaning the strictly decreasing sequence of all positions smaller than or equal to* y *such that:*

1. $y - U + 1 \leqslant \ell_{y,d_y} < \cdots < \ell_{y,2} < \ell_{y,1} = y - L + 1$*, so that* $\mathcal{L}_y$ *captures the left extensions of* $\mathsf{MSA}[1..r, y+1..c]$ *of length at least* L *and at most* U*; and*
2. $\mathrm{H}([\ell_{y,j}..y]) \neq \mathrm{H}([\ell_{y,j}+1..y])$ *for* $2 \leqslant j \leqslant d_y$*, so that each* $\ell_{y,j}$ *marks a column where the height of the left extension changes.*

If $y < L$*, i.e.* $\mathsf{MSA}[1..r, y+1..c]$ *has no left extension of size at least* L*, we define* $\mathcal{L}_y = ()$ *and* $d_y = 0$*. Otherwise, for completeness, we define* $\ell_{y,d_y+1} = \max(0, y - U)$ *(see how this is used later in Equation* (2)*).*

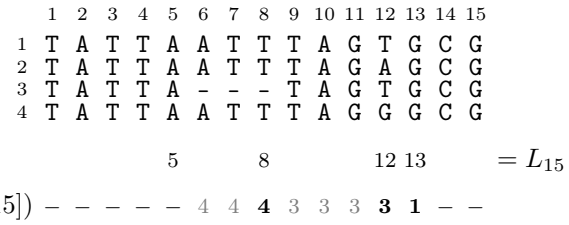


Figure 4 Example of $\mathsf{MSA}[1..4, 1..15]$ and of meaningful left extensions $\mathcal{L}_{15} = 13, 12, 8$ (represented from right to left in the figure), given lower bound $L = 3$ and upper bound $U = 10$.

The meaningful left extensions $\mathcal{L}_y$ are indeed a compact description of how the height of $[x..y]$ changes when fixing y and moving x: all segments $[x..y]$ with $x \in [\ell_{y,j+1}+1..\ell_{y,j}]$ have height $\mathrm{H}([\ell_{y,j}..y])$; if the monotonicity of left extensions (Observation 7) holds, then $d_y \leqslant r$,

[1] For simplicity, the definition accepts both a lower bound and an upper bound on the segment length, even though min-U-cardinality and min-L-size require only one of the two. If either is unspecified, we simply assume that $L = 1$ or $U = c$ accordingly, that is, there is no lower or upper bound.

that is, $|\mathcal{L}_y|$ is at most r for each $y \in [1..c]$, and the cumulative count of all meaningful left extensions and their height values is $O(rc)$. Regardless of the number of meaningful left extensions, Equation (1) can be rewritten as

$$\mathrm{m}_y = \min_{j \in [1..d_y]} \Big(\mathrm{H}([\ell_{y,j}..y]) + \min_{x \in [\ell_{y,j+1}+1..\ell_{y,j}]} \mathrm{m}_{x-1} \Big), \tag{2}$$

where m_{x-1} is used due to the different meaning of x in Equation (2) and Definition 11. Given pairs $(\ell_{y,j}, \mathrm{H}([\ell_{y,j}..y]))$ in input for each $y \in [1..c]$, we can compute values m_y in a dynamic programming fashion and we can solve operation $\min_{x \in [\ell_{y,j+1}+1..\ell_{y,j}]} \mathrm{m}_{x-1} = \min_{x \in [\ell_{y,j+1}..\ell_{y,j}-1]} \mathrm{m}_x$ by indexing the array of values m_y for Range Minimum Queries with Lemma 6, as shown in Algorithm 1.

Algorithm 1 Segmentation of $\mathsf{MSA}[1..r, 1..c]$ with segment-length upper bound U minimizing the cardinality. Note that bound U is implicitly used in value $\ell_{y,d_y+1} = \max(0, y - U)$ (see Definition 11 and Equation (2)).

input : integers $r, c \in \mathbb{N}$,
segment length upper bound $U \in [1..c]$, and
meaningful left extensions $(\ell_{y,j}, h_{y,j})$ for $y \in [1..c]$, $j \in [1..d_y]$
output : minimum cardinality of a segmentation $S_1, \ldots, S_k$ such that $|S_i| \leqslant U$ for $i \in [1..k]$

```
Create array m[0..c] holding values in [0..rc];
Preprocess m for semi-dynamic Range Minimum Queries;
m[0] ← 0;
for y ← 1 to c do
    m[y] ← +∞;
    for j ← 1 to d_y do
        x ← RangeMinQuery(m, [ℓ_{y,j+1}..ℓ_{y,j} − 1]);
        m[y] ← min(m[y], h_{y,j} + m[x]);
    end
    Update m for Range Minimum Queries over incremented range [1..y];
end
return m[c]
```

Finally, for gapless MSAs the meaningful left extensions can be computed efficiently with a modification of the positional Burrows–Wheeler Transform (Definition 4). Norri et al. correctly realized that the pBWT for position y already describes $\mathcal{L}_y$ and all changes in height [22, Lemma 4] and modified the pBWT to obtain values $(\ell_{y,j}, \mathrm{H}([\ell_{y,j}..y]))$ in $O(r)$ time per each column y.

▶ **Lemma 12** ([22, Lemmas 5 and 6]). *Let $\mathsf{MSA}[1..r, 1..c]$ be a gapless multiple sequence alignment over alphabet Σ, with $|\Sigma| \in O(r)$. The meaningful left extensions $\mathcal{L}_y = \ell_{y,1}, \ldots, \ell_{y,d_y}$ and their corresponding segment height values $\mathrm{H}([\ell_{y,j}..y])$ for $j \in [1..d_y]$ can be computed for $y = 1, \ldots, c$ in $O(rc)$ time and in $O(r + c)$ working space. Specifically, the values are computed column-wise in a streaming fashion, from $y = 1$ to $y = c$, and the space bound does not include storing the computed values.*

By interleaving the computation from Lemma 12 and Algorithm 1 we obtain a linear-time solution to min-U-cardinality in the case with no gap.

▶ **Theorem 13.** *Let* $\mathsf{MSA}[1..r, 1..c]$ *be a gapless multiple sequence alignment over alphabet* Σ*, with* $|\Sigma| \in O(r)$*, and* $U \in [1..c]$ *be an upper bound on the maximum segment length. We can solve* min-U-cardinality *(Problem 2) in* $O(rc)$ *time and* $O(c + r)$ *space by finding the optimal segmentation* S *respecting* U *and minimizing the cardinality* $\mathrm{m}(S)$*.*

Proof. The final algorithm is provided by combining Algorithm 1, that utilizes the meaningful left extensions $\mathcal{L}_y$ and their height values from $y = 1$ to $y = c$, with Lemma 12, that computes and can provide pairs $(\ell_{y,j}, \mathrm{H}([\ell_{y,j}..y]))$ in the same order. The correctness of Algorithm 1 follows from Equation (2). Since the MSA is gapless, the total number of iterations of the inner for-loop in Algorithm 1 is $O(rc)$ due to the monotonicity of left extensions (Observation 7), and by using the Range Minimum Query data structure from Lemma 6 on array `m` we obtain the stated time and space complexity. Algorithm 1 can be augmented with standard backtracking techniques to obtain the actual segmentation, as shown by Rizzo et al. [24, Section 4.5]. ◀

4.3 Minimizing the size

Similarly to Section 4.2, given $\mathsf{MSA}[1..r, 1..c]$ and $L \in [1..c]$, for any $y \in [1..c]$ we define N_y as the size of an optimal solution of min-L-size on instance $\mathsf{MSA}[1..r, 1..y]$ respecting lower bound L. Then the following recursion holds, since the gap-aware size of a segmentation S is the sum of the individual gap-aware segment sizes:

$$\mathrm{N}_0 = 0,$$
$$\mathrm{N}_y = \min_{x \in [0..y-1]} \Big(\mathrm{N}_x + \mathrm{N}_\varepsilon([x+1..y]) \Big) \qquad y \in [1..c], \tag{3}$$

and it is easy to see that N_c is equal to the gap-aware size of an optimal segmentation.

We now concentrate on gapless MSAs. Recall that, in such setting, segment size and gap-aware segment size coincide, and they are equal to the length of the segment times its height (Observation 8). Also, consider how the meaningful left extensions (Definition 11) are a compact description of all possible segment heights, as discussed in Section 4.2. Then, in the gapless setting, Equation (3) for $y \in [1..c]$ can be rewritten as

$$\begin{aligned}
\mathrm{N}_y &= \min_{x \in [0..y-1]} \Big(\mathrm{N}_x + \mathrm{H}([x+1..y]) \cdot (y - x) \Big) && \text{(Observation 8)} \\
&= \min_{x \in [0..y-1]} \Big(\mathrm{N}_x - \mathrm{H}([x+1..y]) \cdot x + \mathrm{H}([x+1..y]) \cdot y \Big) \\
&= \min_{j \in [1..d_y]} \Big(\mathrm{H}([\ell_{y,j}..y]) \cdot y + \min_{x \in [\ell_{y,j+1}..\ell_{y,j}-1]} \Big(\mathrm{N}_x - \mathrm{H}([\ell_{y,j}..y]) \cdot x \Big) \Big), && (4)
\end{aligned}$$

where the last step holds because the meaningful left extensions $\mathcal{L}_y$ partition by construction range $[1..y - L + 1]$ according to the segment height $\mathrm{H}([x..y])$ for variable x, or in other words, $\mathrm{H}([x+1..y]) = \mathrm{H}([\ell_{y,j}..y])$ for all $x + 1 \in [\ell_{y,j+1} + 1..\ell_{y,j}]$, $j \in [1..d_y]$. Equation (4) is suitable for an efficient dynamic programming approach, shown in Algorithm 2:

- value $\mathrm{H}([\ell_{y,j}..y]) \cdot y$ inside the outer min operator depends on y and on the height considered (i.e. the meaningful left extension), as in Equation (2) from Section 4.2;
- the addends in the internal min operation depend on x and on the segment height $\mathrm{H}([\ell_{y,j}..y]) \in [1..r]$, and since the number of possible height values is $O(r)$ we can store and maintain these recursive values for each column $x \in [1..y-1]$ and each height $h \in [1..r]$.

Algorithm 2 Segmentation of $\mathsf{MSA}[1..r, 1..c]$ with segment-length lower bound L minimizing the cardinality. Note that bound L is implicitly used in value $\ell_{y,1} = y - L + 1$ (see Definition 11).

input : integers $r, c \in \mathbb{N}$,
segment length lower bound $L \in [1..c]$, and
meaningful left extensions $(\ell_{y,j}, h_{y,j})$ for $y \in [1..c]$, $j \in [1..d_y]$
output : minimum size of a segmentation $S_1, \ldots, S_k$ such that $|S_i| \leqslant U$ for $i \in [1..k]$
Initialize array $\mathtt{N}[1..r]$ with values in $[0..rc]$;
Initialize matrix $\mathtt{M}[1..r, 0..c]$ with values in $[0..rc]$;
Preprocess the rows of $\mathtt{M}$ for semi-dynamic Range Minimum Queries;
for $h \leftarrow 1$ **to** r **do**
  $\mathtt{M}[h, 0] \leftarrow 0$;
end
for $y \leftarrow 1$ **to** c **do**
  $\mathtt{N}[y] \leftarrow +\infty$;
  for $j \leftarrow 1$ **to** d_y **do**
    $x \leftarrow \text{RangeMinQuery}\big(\mathtt{M}[h_{y,j}], [\ell_{y,j+1}..\ell_{y,j} - 1]\big)$;
    $\mathtt{N}[y] \leftarrow \min\Big(\mathtt{N}[y], \quad h_{y,j} \cdot y + \mathtt{M}[h_{y,j}, x]\Big)$;
  end
  for $h \leftarrow 1$ **to** r **do**
    $\mathtt{M}[h, y] \leftarrow \mathtt{N}[y] - h \cdot y$;
    Update the Range Minimum Query data structure of row h to cover
     incremented range $\mathtt{M}[h, 1..y]$;
  end
end
return $\mathtt{N}[c]$

▶ **Theorem 14.** *Given a* gapless $\mathsf{MSA}[1..r, 1..c]$ *and a lower bound* $L \in [1..c]$ *on the minimum segment length, we can solve* min-L-size *(Problem 3) in* $O(rc)$ *time and* $O(rc)$ *space by finding the optimal segmentation* S *respecting* L *and minimizing the size* $\mathrm{N}(S)$.

Proof. The final algorithm takes the output of Lemma 12, pairs $(\ell_{y,j}, h_{y,j})$ for $y \in [1..c]$ and $j \in [1..d_y]$, and directly runs Algorithm 2, whose correctness follows from Equation (4). The total number of iterations of the two inner for-loops is $O(rc)$, since the number of meaningful left extensions is $O(rc)$ (Observation 7) and the possible height values are in the range $[1..r]$. The stated time and space complexity follows by applying Lemmas 6 and 12. Similarly to Algorithm 1, Algorithm 2 can be augmented with standard backtracking techniques to obtain the actual segmentation, as shown in [24, Section 4.5]. ◀

5 Future work

This initial study on EDS construction opens many interesting directions for further study:

Gaps as symbols yield upper bounds Even though Equations (2) and (3) (but not Equation (4)) hold for MSAs with gaps, Theorems 13 and 14 work only on MSAs without gaps. A straightforward way to extend them is to treat them as normal symbols in the alphabet. After segmentation, the gap symbols can be removed to obtain the final EDS. With all the quality measures we proposed, this works as an upper bound: for example, consider the height of a segment that contains `A-` and `-A`; height is reduced by one after gaps are removed. Thus, the removal of gaps cannot increase the height, and it is of interest to study experimentally how much the measures change after gap removal.

Gaps as symbols yield exact solutions? For height optimization, it seems possible to characterize the class of MSAs with gaps where the approach above yields an optimal solution, and not just an upper bound. Such characterization should be possible through forbidding local sub-optimal alignments like `A-` and `-A`. This raises several natural questions: What is the exact form of the characterization? How to efficiently detect if a given MSA is part of this class? Do all optimal alignments under some column-wise scoring function belong to this class? Are there efficient algorithms to convert an MSA into one in this class without sacrificing the alignment score?

Tailored algorithms with gaps The presented Algorithms 1 and 2 do not find the optimal segmentation of arbitrary MSAs, so it is natural to ask if there are efficient algorithms to directly optimize the studied measures on general MSAs. Alternatively, there may be slight adjustments to the measures that are amenable to efficient algorithms: in a different context, Rizzo et al. [24] introduced the metric of prefix-aware height, which was optimized in linear time, and also optimized in linear time the metric of maximum segment length.

References

1 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.

2 Rocco Ascone, Giulia Bernardini, Alessio Conte, Massimo Equi, Estéban Gabory, Roberto Grossi, and Nadia Pisanti. A unifying taxonomy of pattern matching in degenerate strings and founder graphs. In Solon P. Pissis and Wing-Kin Sung, editors, *24th International Conference on Algorithms in Bioinformatics, WABI*, volume 312 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.WABI.2024.14.

3 Giulia Bernardini, Estéban Gabory, Solon P. Pissis, Leen Stougie, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string matching with 1 error or mismatch. *Theory Comput. Syst.*, 68(5):1442–1467, 2024. doi:10.1007/S00224-024-10194-8.

4 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM J. Comput.*, 51(3):549–576, 2022. doi:10.1137/20M1368033.

5 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2017. doi:10.1007/978-3-319-67428-5_7.

6 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. doi:10.1016/J.TCS.2019.08.012.

7 Jorge Avila Cartes, Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, and Luca Denti. Pangeblocks: customized construction of pangenome graphs via maximal blocks. *BMC Bioinform.*, 25(1):344, 2024. doi:10.1186/S12859-024-05958-5.

8 Bastien Cazaux, Dmitry Kosolobov, Veli Mäkinen, and Tuukka Norri. Linear time maximum segmentation problems in column stream model. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2019. doi:10.1007/978-3-030-32686-9_23.

9 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, October 2016. doi:10.1093/bib/bbw089.

10 Richard Durbin. Efficient haplotype matching and storage using the positional burrows-wheeler transform (PBWT). *Bioinform.*, 30(9):1266–1272, 2014. doi:10.1093/BIOINFORMATICS/BTU014.

11 Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. *Theor. Comput. Sci.*, 975:114128, 2023. doi:10.1016/J.TCS.2023.114128.

12 Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Trans. Algorithms*, 19(3):21:1–21:25, 2023. doi:10.1145/3588334.

13 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing founder graphs. *Algorithmica*, 85(6):1586–1623, 2023. doi:10.1007/S00453-022-01007-W.

14 Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Pangenome comparison via ed strings. *Frontiers Bioinform.*, 4, 2024. doi:10.3389/fbinf.2024.1397036.

15 Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Comparing elastic-degenerate strings: Algorithms, lower bounds, and applications. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France*, volume 259 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CPM.2023.11.

16 Estéban Gabory, Njagi Moses Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba. Elastic-degenerate string comparison. *Inf. Comput.*, 304:105296, 2025. doi:10.1016/J.IC.2025.105296.

17 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-line pattern matching on similar texts. In *Proc. CPM 2017*, volume 78 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.CPM.2017.9.

18 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Inf. Comput.*, 279:104616, 2021. doi:10.1016/J.IC.2020.104616.

19 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978. doi:10.1145/322063.322075.

20 Veli Mäkinen and Tuukka Norri. Applying the positional Burrows–Wheeler transform to all-pairs Hamming distance. *Inf. Process. Lett.*, 146:17–19, 2019. doi:10.1016/J.IPL.2019.02.003.

21 Njagi Moses Mwaniki and Nadia Pisanti. Optimal sequence alignment to ED-strings. In Mukul S. Bansal, Zhipeng Cai, and Serghei Mangul, editors, *Bioinformatics Research and Applications - 18th International Symposium, ISBRA 2022, Haifa, Israel, November 14-17, 2022, Proceedings*, volume 13760 of *Lecture Notes in Computer Science*, pages 204–216. Springer, 2022. doi:10.1007/978-3-031-23198-8_19.

22 Tuukka Norri, Bastien Cazaux, Dmitry Kosolobov, and Veli Mäkinen. Linear time minimum segmentation enables scalable founder reconstruction. *Algorithms Mol. Biol.*, 14(1):12:1–12:15, 2019. doi:10.1186/S13015-019-0147-6.

23 Cédric Notredame. Recent evolutions of multiple sequence alignment algorithms. *PLoS Comput. Biol.*, 3(8), 2007. doi:10.1371/JOURNAL.PCBI.0030123.

24 Nicola Rizzo, Massimo Equi, Tuukka Norri, and Veli Mäkinen. Elastic founder graphs improved and enhanced. *Theor. Comput. Sci.*, 982:114269, 2024. doi:10.1016/J.TCS.2023.114269.

25 Esko Ukkonen. Finding founder sequences from a set of recombinants. In Roderic Guigó and Dan Gusfield, editors, *Proceedings of 2nd International Workshop on Algorithms in Bioinformatics WABI*, volume 2452 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2002. doi:10.1007/3-540-45784-4_21.

“Strutture Di Dati e Algoritmi. Progettazione, Analisi e Visualizzazione”, a Book Beating Its Own Drum

Anna Bernasconi
Dipartimento di Informatica, University of Pisa, Italy

Linda Pagli
Dipartimento di Informatica, University of Pisa, Italy

Abstract

In this contribution, we discuss the innovative approach taken by Roberto Grossi and his co-authors in their book “Strutture di dati e algoritmi. Progettazione, analisi e visualizzazione,” which aims to bridge the gap between the theoretical and practical aspects of algorithms. Unlike traditional texts that either focus on formal mathematical analysis or practical implementation, this book adopts an intermediate approach that emphasizes both programming skills and theoretical understanding. This contribution reflects our experience as instructors of the introductory algorithms course in the Computer Science degree program at the University of Pisa.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Algorithms, C-Programming, Basics of Computational Thinking

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.3

Category Education

Dedicated to Roberto Grossi

The book “Strutture di dati e algoritmi. Progettazione, analisi e visualizzazione” by Pierluigi Crescenzi, Giorgio Gambosi, and Roberto Grossi was first published in 1999. Focusing on algorithms – a foundational topic in computer science – the authors chose not to simply follow in the footsteps of other renowned texts. Instead, they aimed to offer a fresh and innovative perspective, taking into account the most recent developments in algorithm research at the time.

Before this volume, there were basically two main approaches, and they were pretty different from each other. The first one looks at algorithms, their correctness, and their computational complexity from a more theoretical and formal angle, rich in mathematics. It goes through the history of classic algorithms, organizing them by topic, comparing them based on their theoretical complexity, and figuring out their computational limits. This approach uses a high-level language to highlight the key features of an algorithm without getting distracted by implementation details. A great example of this is the famous book “Introduction to Algorithms” by Cormen, Leiserson, and Rivest, which first came out in 1990 and was updated to include Stein in 2002. It has been adopted in many computer science courses around the world, including at MIT, and has become a classic in the field. Many Italian books that followed have pretty much followed this line without altering its approach.

On the other hand, the second approach concentrates solely on the implementation of algorithmic problems, with the goal of optimizing the computer’s capabilities for maximum efficiency. This approach is less formal, and the study of efficiency is based more on experimentation than on theoretical analysis. It is a practical approach focused on identifying

Figure 1 The covers of the first and second edition of the book "Strutture di dati e algoritmi. Progettazione, analisi e visualizzazione".

efficient algorithms based on the actual conditions under which they operate. Quality code and experimentation are the goals of this approach, perhaps more suited to experienced programmers.

Roberto Grossi et al have followed an intermediate approach, aimed at students who are tackling programming for the first time but are also beginning to orient themselves in theory, studying classic problems and learning to analyze solutions. Special attention is given to the code of an algorithm, which is not yet complete and is almost entirely lacking parameter testing and input/output instructions. However, it is very close to actual code (C or Java) and can be implemented with only a few minor additions. Programming skills are placed on the same level as theoretical analysis. This type of training seems particularly suitable for first-year computer science students. We, the authors of this note, were at the time instructors of the two twin courses in Algorithms in the Computer Science degree program at the University of Pisa. We were already experienced in teaching this subject, as well as in using the Cormen's book, but upon learning about the new volume, we did not miss the opportunity to adopt it, curious about the new approach and with the convenience of being able to consult directly with one of the authors in case of doubts.

The book also presents other attractive features: it offers students the opportunity to visualize, through a system called Alvie accessible online, the execution of almost all the algorithms presented in the book, with input data specified by the student. Following the execution of an algorithm as things become more complicated is a very effective teaching tool for both the student and the teacher, who can avoid the often tedious manual execution on the board. Another interesting feature is the integration of theoretical topics with practical applications, with the dual aim of providing students with a clear idea of the importance of the concepts studied and giving teachers a set of useful motivations to present the more challenging topics. These integration topics between theory and practice are referred to in the volume as *opus libri*.

Lastly, we liked the cover of the first edition, where little robots in the shape of a colorful, stylish personal computer, very popular in the 1990s, work together to build a treehouse and make a computer function.

Adopting a newly released book and following its framework in teaching also means seeing some limitations and finding inevitable calculation errors, printing mistakes, and even some subtler ones. For us, it was a discovery, a challenge, and also a struggle, but we believe we have prepared students well in the field of algorithms and have contributed operationally to

the production of a better second edition of the volume. The students appreciated the book, and with the help of lessons and exercises, they also learned the most theoretical concepts and complicated proofs; in particular, we will mention some of the most educationally formative topics, enjoyable to teach and well-received by the students.

The book begins with difficult problems, the distinction between P and NP, and the important concept that for many interesting problems, no better solution algorithms are known than those that explore the whole set of possible solutions. These algorithms, known as *brute force*, require exponential time in the worst case with respect to the input size of the problem. To explain to students the complexity and length of an exponential algorithm that tests all possibilities, the game of Sudoku is used as an example. Sudoku emerged in the 1990s and became very popular during that time. It is a useful tool for explaining the definition of the NP class, which includes problems that can be easily verified in polynomial time. Indeed, in Sudoku, having the solution to a specific puzzle (which is published in newspapers shortly after the puzzle is proposed) and placing the numbers in the correct spots allows one to easily verify that the solution satisfies all the specified constraints. Furthermore, in solving games proposed as difficult, one proceeds exactly as in brute force algorithms, going through multiple decisions, where different values can be associated with a generic cell. By choosing one of these, one can proceed for one or more steps or even to the end, or one must go back to the most recent step where a choice was made and move on to the next choice if it exists. With a nice example that is not too large, students, almost all familiar with the game, can quickly grasp the meaning of a brute force algorithm and have a practical idea of time exponential in the dimensions of the starting grid.

In our opinion, one very educational topic introduced in the volume regarding data structures is that of variable-sized arrays. The array is studied along with its memory allocation. When the array of size d is dynamic, meaning it undergoes many insertions and deletions, care is taken to ensure that the memory space reserved for it is always adequate. In particular, if, for a new insertion, the number of elements in the array reaches its size, the array is duplicated, leading to the elements being copied into a new array of size $2d$. Conversely, if, following a new deletion, the number of stored elements becomes equal to $d/4$, the array is halved, which also involves copying the elements. It is shown that n operations of insertion or deletion in a dynamic array can be performed in $O(n)$ time. The proof is very brief and neat but not trivial; it simultaneously shows that halving when the number of elements is equal to $d/2$, a choice that seems the most natural, does not work, which helps to better understand why the choice to halve when $n = d/4$ is made. The analysis of dynamic arrays introduces the concept of *amortized complexity*: the cost of copying the array is spread over previous and subsequent operations that do not require any action, and these are sufficiently numerous to absorb the cost of copying the elements of the array.

Amortized complexity is introduced and discussed even more explicitly in the chapter dedicated to lists, where the study of self-organizing lists is explored, particularly the Move-to-Front strategy. This strategy is familiar to students because it is used in the call logs on their phones (which were not yet smart at the time). The analysis of the computational cost of this strategy is not simple and requires nearly an entire lesson. It was especially rewarding to complete the lesson without losing the students' attention, and even more so when they chose this topic as one they were eager to discuss during the oral exam.

Another innovative topic, never encountered in other similar texts, is that of recursive algorithms on binary trees. A general paradigm is presented, as an extension of divide and conquer on arrays, to be applied to binary trees. The typical divide and conquer scheme – solve recursively on the left subset of $n/2$ elements, solve recursively on the right subset

of $n/2$ elements, and then recombine – applies almost directly to binary trees, replacing the subset with the left and right subtrees, respectively. This type of approach allows for very easy solutions to classic problems on trees that would otherwise be quite complicated. Students easily grasp this tool and use it to effectively solve problems that, when approached without this framework, would traditionally be challenging.

A basic algorithm for the sorting problem is Quick-Sort, which is super efficient in the average case, while in the worst case it is quadratic like the most basic algorithms. Since it is ultimately the most used algorithm in practice and, with some variations, is present in the libraries of many programming languages, it is important to demonstrate to students its average case cost. As is well known, proofs in this case are much more complicated than those considering the worst case, where identifying the situation that leads to the highest complexity often involves elementary mathematical study. The average case, on the other hand, requires considering and evaluating all possible inputs, often making use of probability, a topic that first-year algorithm students will tackle the following year. The text offers an alternative approach that, while not trivial, simplifies the analysis and is particularly well-suited to students.

Dynamic Programming is a rather original algorithm design technique that is typically applied to optimization problems, where the so-called principle of optimality holds. This principle states that, for the problem at hand, the optimal solution can be derived from the optimal solutions of smaller subproblems. The algorithms are generally complex and not very intuitive, to the point that sometimes it is not at all straightforward to even understand how they work; designing a solution using this technique is even more challenging and requires a deep mastery of the subject. Thus, the book provides a sort of guide for defining an algorithm based on four fundamental aspects:

1. The formulation of the problem structure that is valid for the problem in question and its subproblems.
2. The identification of elementary subproblems and the assignment of their solution values, obtained through direct inspection.
3. The definition of the recursive rule that allows the problem to be solved as a composition of the solutions of subproblems.
4. The derivation of an ordering of the subproblems for efficient computation and the storage of partial solutions in a table.

These rules, which may seem obscure at first, accompanied by various examples, provide important guidance and assistance to those who wish to tackle the definition of a dynamic programming algorithm. Students struggle with this topic, but in the end, with step-by-step guidance, they manage to master it. We can assert that the four-step technique is quite effective and provides important guidelines in the design of a dynamic programming algorithm.

The implementation of sorted dictionaries on balanced binary search trees is another topic that Roberto's book presents in a clear and effective way. The chosen balancing technique is that of 1-balanced binary search trees, known as *AVL trees* (named after the initials of the Russian authors Adelson-Velsky and Landis, who proposed them in 1962). A binary tree is 1-balanced if the heights of the two child subtrees of any node differ by at most one. The connection between being 1-balanced and having logarithmic height is not immediate and involves Fibonacci trees, which are the most unbalanced among 1-balanced trees. More specifically, a Fibonacci tree is a 1-balanced tree that has the fewest nodes for a given height. Starting with Fibonacci trees, the text walks the reader through a detailed process that

uses concepts such as minimality, recursive definitions, closed-form Fibonacci numbers, and exponential growth. This all comes together to prove that a generic 1-balanced tree has a logarithmic height. After this somewhat theoretical discussion, the student is quickly rewarded with code to implement dictionary operations on AVL trees. The code is clear, comprehensive in all its stages (including the rebalancing of the structure through rotations), and practically ready to be executed.

In all algorithm books, graphs are treated as a static structure: memory representations rely on keeping all the nodes of the graph in a direct access array, which requires knowing all the nodes of the graph in advance to ensure constant time access to any node. All traversal algorithms are based on this assumption, which is never explicitly stated, allowing their complexity to remain linear in the number of nodes and edges, precisely because access to any node is achieved in constant time. Classic algorithms on graph structure also always assume that the entire graph is known a priori. Networks like the internet, naturally represented as graphs, are instead very dynamic structures, where important information must be calculated, such as degree, eccentricity, shortest paths, etc. In Roberto's book, dynamic graphs are given separate treatment, where nodes are allocated in a dictionary, that is, in a flexible structure, for example organized as a balanced tree or a hash table, which guarantees efficient access, insertion, and deletion operations, but not executable in constant time. Consequently, graph algorithms can be transformed, providing students with immediate use of all variations on the organization of a dictionary, introduced earlier, and giving instructors an inexhaustible source of problems to propose as exercises.

When the Algorithms course was accompanied by a separate course on Practical Programming Laboratory, the implementation part of basic algorithms was almost completely transferred to that course, and the Algorithms course inevitably shifted a bit more towards theory. Thus, we returned to the original course setup, resuming Cormen et al as the textbook. Nevertheless, almost all the topics mentioned above retained the approach and important concepts encountered in Roberto et al book during the lessons.

As a conclusion, let us say that Roberto is an innovator, and this attitude is reflected in all his activities: from scientific research to teaching and writing textbooks, from finding the best leavening for homemade bread to creating cooking recipes inspired by Japanese culture, and from preparing students for the *Olympiad in Informatics* to educating and protecting them as if they were his own children.

On Graph Burning and Edge Burning

Giuseppe F. Italiano
LUISS University, Rome, Italy

Athanasios L. Konstantinidis
University of Ioannina, Greece

Manas Jyoti Kashyop
Indian Institute of Technology Bhubaneswar, India

Abstract

Graph burning is a deterministic, discrete-time process that models how influence or contagion spreads in a graph. Initially, all vertices are unburned. At each round, one new vertex is chosen to burn. Once a vertex is burned, in the next round each of its unburned neighbors become burned. The process ends when all vertices are burned. The burning number of a graph is the minimum number of rounds needed for the process to end. Very recently, a variant called edge burning was introduced, where instead of vertices we burn edges: at each round one new edge is burned. Once an edge is burned, in the next round all its unburned incident edges become burned. The edge burning number is the minimum number of rounds that are needed to burn all the edges. In this paper, we present a systematic study of edge burning and provide some new results for graph burning. First, we show a tight relationship between the edge burning number and the burning number of a given graph: specifically, their absolute difference is at most 1. Moreover, we show that the edge burning number of a graph is equal to the graph burning number of its line graph. On the computation complexity side, we show that the edge burning problem is NP-complete, but can be solved in linear time on paths, split graphs, and cographs. Furthermore, we give an XP algorithm when the edge burning problem is parameterized by the diameter of the input graph and a linear kernel when parameterized by the neighborhood diversity. For the graph burning problem, we provide 2-approximation algorithms when either the solution is part of the input or forced to form a path.

2012 ACM Subject Classification Theory of computation → Algorithm design techniques

Keywords and phrases Burning Number, Graph Burning, Edge Burning, Approximation

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.4

Category Research

Funding *Giuseppe F. Italiano*: Giuseppe F. Italiano was partially supported by MUR, the Italian Ministry for University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).

Acknowledgements The work was done when the second and the third authors were postdocs at LUISS University Rome, Italy.

1 Introduction

Graph burning was introduced as a model to study spread of an influence or a contagion in a social network in [4]. Graph burning is a deterministic process and it advances in discrete time steps called *rounds*. Input to the graph burning process is a simple, undirected, and unweighted graph. Initially, all the vertices in the graph are marked as unburned. In the first round a vertex is selected, it is marked as burned, and denoted as the first *activator*. In round i, where $i \geq 2$, an unburned vertex (if such a vertex is available) is selected and marked as burned (called the i-th *activator*). Further, in round i, where $i \geq 2$, all the unburned neighbors of the vertices which are marked as burned till round $i-1$ are marked as burned. The goal in the graph burning process is to mark all the vertices in the graph as burned using

the minimum number of rounds. Bonato et al. [4] called the minimum number of rounds required to mark all the vertices as burned in the input graph as *burning number* $b(G)$ of graph G. The set of activators selected in every round forms a sequence, which is called the *burning sequence* of the graph.

Graph burning marks vertices from unburned to burned and selects a sequence of vertices as a sequence of activators. Mondal et al. [16] proposed the process of marking edges instead of vertices and selecting a sequence of edges as a sequence of activators. They referred to this process as *edge burning*. In other words, edge burning is a discrete deterministic process where, instead marking all the vertices of the input graph from unburned to burned, the goal is to mark all the edges of the graph from unburned to burned. Initially, all the edges in the graph are marked as unburned. In the first round an edge is selected and it is marked as burned: this edge is referred to as the first *source*. In round i, $i \geq 2$, an unburned edge (if such an edge is available) is selected and marked as burned, and it is referred to as the i-th *source*. Further, in round i, $i \geq 2$, all the unburned neighbors of the edges which are marked as burned till round $(i-1)$ are marked as burned. The goal in the edge burning process is to mark all the edges in the graph as burned using the minimum number of rounds. The *edge burning number* of a graph, denoted by $Eb(G)$, is the minimum number of rounds required to mark all the edges in the graph as burned. The set of sources selected in every round forms a sequence called the *edge burning sequence* of the graph. Throughout the paper, we will use the notation *burn a vertex* instead of *marking a vertex as burned*, *burn an edge* instead of *marking an edge as burned*. Similarly, we will refer to *burned (unburned) vertex/vertices* and to *burned (unburned) edge/edges*.

Previous work. The graph burning problem has received significant attention in recent years. Bessy et al. [2] showed that computing the burning number is NP-complete even for trees with maximum degree 3 but optimally solvable in linear time for paths. In addition, this problem remains NP-hard even on caterpillar graphs [10], interval graphs [9] and spider graphs [2], but it is polynomial time solvable on split graphs and cographs [11]. In [5], Bonato et al. presented a very elegant 3-approximation algorithm for graph burning on general graphs and a 2-approximation algorithm for trees. Moreover, Mondal et al. [16] proved that graph burning is APX-hard and provided a 3-approximation algorithm with a different approach from [5]. From the parameterized complexity viewpoint, graph burning is W[2]-hard when parameterized by the burning number, and thus unlikely to admit an FPT algorithm [12]. Furthermore, the problem has been studied under several structural parameters [11, 12].

As for the edge burning problem, Mondal et al. [16] observed that the burning number can be smaller than the edge burning number by giving the wheel graph as an example, as shown in Figure 1 (left): W_5, the wheel graph of 6 vertices, has burning number equal to 2, and edge burning number equal to 3. It is easy to see that the edge burning number can be smaller than the burning number, as shown in Figure 1 (right): the graph G has burning number equal to 3, and edge burning number equal to 2. As pointed out by Mondal et al. [16], an interesting question is to determine the relationship between the burning number and the edge burning number of a given graph.

Our results. In this paper, we present a systematic study of edge burning and provide some new results for graph burning. First, we address the question posed by Mondal et al. [16] and show that there is indeed a tight relationship between the edge burning number and the burning number of a given graph: specifically, we prove that their absolute difference can be

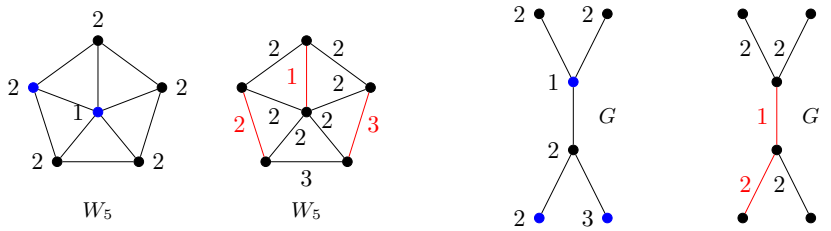


Figure 1 We denote activators for graph burning by the blue vertices and sources for edge burning by the red edges. The numbers over the vertices or edges show the number of rounds of an optimal solution. The graph W_5 is an example from [16] in which the graph burning number is less than the edge burning ($b(W_5) = 2 < Eb(W_5) = 3$). On the other hand, we give the graph G as an example that shows the opposite, the edge burning number of G is less than the burning number of G ($Eb(G) = 2 < b(G) = 3$).

at most 1. In addition, we show that the edge burning number of a graph is equal to the graph burning number of its line graph. The latter result appears to be of broad interest. On the one hand, we use it to establish new upper bounds for the edge burning number using known results for the burning number. On the other hand, it allows us to prove also new results for the burning number, such as the NP-completeness of the graph burning problem on the restricted class of connected claw-free graphs.

On the computation complexity side, we show that the edge burning problem is NP-complete on general graphs. This is not surprising, given the tight relationship that we have proved between the edge burning number and the burning number of a graph. On the positive side, we show that edge burning can be solved in linear time on paths, split graphs, and cographs. We next consider the parameterized complexity of the edge burning problem; first we show a trivial XP algorithm for the edge burning problem when parameterized by the diameter of the graph, followed by a linear kernel when parameterized by the neighborhood diversity.

Next, we focus our attention to two variants of the graph burning problem. The first variant is defined as follows. Consider a network in which one wishes to spread influence in the minimum number of rounds by selecting some nodes as activators. The nodes selected as activators must be operational until the influence reaches every node in the network. Suppose this requirement forces the nodes selected as activator to be connected to an emergency service line. Therefore, all the activator nodes in the network must lie in a path which models the emergency service line. With this motivation, we define a variant of the graph burning problem in which all the activators must be on a path. We call the problem as *Graph burning with Activators on a Path (GbAP)* problem. Our motivation for the GbAP problem resembles the motivation for studying well-known clustering problems like the k-center problem with all the centers on a line [6]. Potential applications include setting up kiosks for medical emergencies, where these kiosks must be connected to an uninterrupted power supply line, in an event that attracts large gatherings and many stalls. Note that the GbAP problem reduces to the graph burning problem when we restrict the graph class to paths, which can be solved optimally in linear time. We show that GbAP problem is NP-complete even for trees with maximum degree 3; on the positive side, we present a 2-approximation algorithm for the GbAP problem on trees.

The second variant was defined by Mondal et al. [16], who considered graph burning with additional advice: more specifically, together with the graph, an (unordered) set of activators used by an optimal solution is provided as input. In [16] it is shown that finding an

optimal burning sequence is NP-hard even in this special case. We refer to this variant as the *Graph burning with Activators as Input (GbAI)* problem. For the GbAI problem, we present a deterministic algorithm that guarantees a 2-approximation and a randomized algorithm that guarantees better than 2-approximation. We also show that the GbAI problem is fixed parameter tractable when parameterized by the solution size (the burning number). To put our results in perspective, we recall that for the graph burning problem no better than 3-approximation is known for general graphs [5, 16] and the problem is W[2]-hard when parameterized by the burning number [12].

Related work. Independently from our work, Antony et al. [1] very recently studied the relationship between the burning number and the edge burning number and the computational complexity of the edge burning by showing NP-hardness for it. Also, there are some results on edge burning number in very restrictive graphs [13, 14].

Organization of the paper. The remainder of the paper is organized as follows. We start by introducing some preliminary terminology and definitions in Section 2, while in Section 3 we present our new results on the edge burning problem. Next, we consider our two variants of the graph burning problem: Section 4 deals with graph burning with all activators on a path (GbAP problem), where all activators must be on a path of the input graph, while Section 5 deals with graph burning with known activators (GbAI problem), where a set of activators used by an optimal solution is given as input. Finally, Section 6 lists some concluding remarks and directions for further research.

2 Preliminaries

All graphs considered throughout the paper are simple, undirected and connected, unless explicitly stated otherwise. Given an input graph $G = (V, E)$, we denote by $n = |V|$ the number of vertices, and by $m = |E|$ the number of edges. The *neighborhood* of a vertex v of G is $N(v) = \{x \mid vx \in E\}$ and the *closed neighborhood* of v is $N[v] = N(v) \cup \{v\}$. For $S \subseteq V$, $N(S) = \bigcup_{v \in S} N(v) \setminus S$ and $N[S] = N(S) \cup S$. For $X \subseteq V$, the subgraph of G *induced* by X, $G[X]$, has vertex set X, and for each vertex pair u, v from X, uv is an edge of $G[X]$ if and only if $u \neq v$ and uv is an edge of G. We denote the edge burning number of G by $Eb(G)$ and the burning number of G by $b(G)$. We use K_n to denote a complete graph with n vertices and P_n to denote a path with n vertices. The complete bipartite graph $K_{1,3}$ is also called as *claw* and a *claw-free graph* is a graph in which no induced subgraph is a claw. A *clique* of a graph is a set of pairwise adjacent vertices of the graph and an *independent set* of a graph is a set of pairwise non-adjacent vertices of the graph. We use the notion of line graph of a graph.

▶ **Definition 1.** *The* line graph *of a graph G is a graph $L(G)$ with the following properties:*
1. *For every edge in G, there is a corresponding vertex in $L(G)$.*
2. *Two vertices in $L(G)$ are adjacent if and only if the corresponding edges in G are adjacent.*

We define the distance between two edges e_1 and e_2 as the minimum number of edges between the two edges plus 1. The distance between two vertices v_1 and v_2, denoted as $\mathsf{dist}(\mathsf{v}_1, \mathsf{v}_2)$, is the length of the shortest path between v_1 and v_2 (the minimum number of edges). The *eccentricity* of a vertex u, denoted as $\mathsf{ecc}(\mathsf{u})$, is $\mathsf{ecc}(\mathsf{u}) = \max_{\mathsf{v} \in \mathsf{V}}\{\mathsf{dist}(\mathsf{u}, \mathsf{v})\}$. The *diameter* d of G is the maximum length of a shortest path in G, that is $\mathsf{d} = \max_{\mathsf{u},\mathsf{v} \in \mathsf{V}}\{\mathsf{dist}(\mathsf{u}, \mathsf{v})\} = \max_{\mathsf{v} \in \mathsf{V}}\{\mathsf{ecc}(\mathsf{v})\}$. The radius rad of G is $\min_{\mathsf{v} \in \mathsf{V}}\{\mathsf{ecc}(\mathsf{v})\}$. For a vertex v, the set $\mathsf{N}_\mathsf{k}[\mathsf{v}]$ denotes the set of vertices at a distance at most k, that is $\mathsf{N}_\mathsf{k}[\mathsf{v}] = \{\mathsf{u} | \mathsf{dist}(\mathsf{u}, \mathsf{v}) \leq \mathsf{k}\}$.

Problem statements. Next, we formalize the decision version of the graph burning problem. We recall here that the set of activators selected in every round forms a sequence, which is called the burning sequence of the graph, and the set of sources selected in every round forms a sequence called the edge burning sequence of the graph.

Graph Burning
Input: An undirected graph G and an integer $k \geq 1$.
Question: Does G have a burning sequence that burns all the vertices of G in at most k rounds?

Similarly, we formalize the decision version of the edge burning problem.

Edge Burning
Input: An undirected graph G and an integer $k \geq 1$.
Question: Does G have an edge burning sequence that burns all the edges of G in at most k rounds?

Finally, we present some concepts from parameterized complexity which are essential for our results.

Parameterized complexity. A problem with input size n and parameter k is fixed parameter tractable (FPT), if it can be solved in time $f(k) \cdot n^{O(1)}$ for some computable function f. A kernelization for a parameterized problem is a polynomial time algorithm that maps each instance (I, k) of a parameterized problem to an instance (I', k') of the same problem such that (I, k) is a yes-instance if and only if (I', k') is a yes-instance, and $|I'| + k'$ is bounded by $f(k)$ for some computable function f. The output (I', k') is called a kernel. The function f is said to be the size of the kernel. It is well-known that every decidable parameterized problem is FPT if and only if it admits a kernel. The class XP contains all problems that can be solved in $O(|I|^{f(k)})$ time for some computable function f. It is common to build an FPT algorithm or a kernel for a parameterized problem by constructing a series of *reduction rules*, that is, polynomial algorithms that either solve the problem or produce instances of the problem that, typically, have lesser sizes or lesser values of the parameter. Respectively, it is said that a rule is *safe* or *sound* if it either correctly solves the problem or constructs an equivalent instance.

3 Edge Burning Number

In this section we study the edge burning problem. Since in the edge burning problem edges are selected as sources in place of vertices with the objective to burn all the edge instead of the vertices, the immediate question, as posed in [16], that arises is the relationship between the edge burning number and the burning number for a given graph. The examples in Figure 1 show that there are graphs in which the difference between the edge burning number and the burning number is one. In the following theorem we show that the difference between these two numbers is at most 1 for any graph.

▶ **Theorem 2.** *For any graph G without isolated vertices $|b(G) - Eb(G)| \leq 1$.*

Proof. First, we show that $Eb(G) \leq b(G) + 1$. Given a graph G we assume a solution for the graph burning problem of size k. This means that there is a burning sequence A of k activators that burns all the vertices of G. Based on A we give an edge burning sequence S of k sources that burns all the edges of G in $k + 1$ rounds. For the activator v_i of A we

choose an incident edge to v_i as a source, let e_i be this source. Thus, S contains k edges of G and every one of them is incident to an activator. Observe that the activator v_i burns all the vertices at distance at most $k-i$, denote W all these vertices. Thus, there is a path from v_i to every vertex $u \in W$ that contains at most $k-i$ edges. By considering an incident edge to v_i as source for the edge burning problem and by the definition of the distance between two edges, we get that all the edges of the paths between v_i and every vertex $u \in W$ will be burned by the source e_i in next $k-i$ rounds. Since this holds for every activator v_i and source e_i, in k rounds we have that every vertex of G has at least one burned incident edge. Hence, in the next round, $k+1$, all the unburned edges will be burned.

Next, we show that $b(G) \leq Eb(G)+1$. Assume a solution for the edge burning problem of size k of a given graph G. This means that there is an edge burning sequence S of k sources that burns all the edges of G. Based on S we give a burning sequence A of k activators that burns all the vertices of G in $k+1$ rounds. For the source e_i of S we choose one of the two endpoints as an activator, let v_i be this activator. Thus, A contains k vertices of G, each of them is endpoint of some source of S. Let F be the set of the edges that are burned by the source e_i. This means that the edges of F are at distance at most $k-i$ from e_i. Consider now the corresponding activator v_i. By the definition of the distance between two edges, we get that v_i is at distance at most $k-i+1$ from every endpoint of the edges in F. Thus, we may need one more round to burn the vertices of F by using v_i as activator. Notice that this holds for every source e_i and activator v_i. Since the sources in S burn all the edges of G in k rounds, we have that all the vertices of G can be burned by A in at most $k+1$ rounds. ◀

Our next theorem provides another relationship between the edge burning number and the burning number through the notion of line graph.

▶ **Theorem 3.** *For any graph G, $Eb(G) = b(L(G))$ where $L(G)$ is the line graph of G.*

Proof. Assume that (G,k) is a yes-instance for Edge Burning and let $(e_1, e_2, ..., e_k)$ be the edge burning sequence for G, i.e. $Eb(G) = k$. Let also $\mathcal{A} = (A_1, A_2, ..., A_k)$ be the set of edges that can be burned by the edges $(e_1, e_2, ..., e_k)$ respectively. This means that the edges that belong to the sets of $\mathcal{A}$ are exactly all the edges of the graph if we ignore the edges $(e_1, e_2, ..., e_k)$. Without loss of generality, let f be an edge that belongs to A_1. By the definition of edge burning, the distance between the edges e_1 and f is at most $k-1$. Now consider the line graph $L(G)$. Then the edges of the edge burning sequence correspond to some vertices $U = \{v_1, v_2, ..., v_k\}$ of $L(G)$. We show that the vertices of U form a burning sequence with the same order for the graph burning problem, this means $b(L(G)) = k$. Let v_f be the corresponding vertex in $L(G)$ of the edge $f \in G$. Since there is a path with (at most) $k-2$ edges between the edges e_1 and f, there is a path with (at most) $k-2$ vertices between the vertices v_1 and v_f. Thus, we get that v_f will be burned in at most $k-1$ steps. Notice that this holds for every edge source e_i and edge $f \in A_i$. Hence, the burning number of $L(G)$ is at most the edge burning number of G, i.e., $b(L(G)) \leq Eb(G)$.

Assume now that $(L(G), k')$ is a yes-instance for graph burning and let $U = \{v_1, v_2, ..., v_{k'}\}$ be the burning sequence for $L(G)$. With the same argumentation, for every vertex v_f there is some source v_i that burns it in at most $k'-i-1$ steps. In other words, there is a path with at most $k'-i-2$ intermediate vertices in the path from v_i to v_f. This means, there is a path that contains $k'-i-2$ edges between the corresponding edges, e_i and f in G. Therefore, the edge burning number of G is at most the burning number of $L(G)$, i.e., $Eb(G) \leq b(L(G))$. By combining the above inequalities, we get that $Eb(G) = b(L(G))$. ◀

The burning number of a path or a cycle with n vertices is known to be $\lceil \sqrt{n} \rceil$ by [4]. Thus, using Theorem 3, we get the following bounds.

▶ **Corollary 4.** *If G is a path with m edges and n vertices then $Eb(G) = \lceil \sqrt{m} \rceil = \lceil \sqrt{n-1} \rceil$.*

▶ **Corollary 5.** *If G is a cycle with m edges and n vertices then $Eb(G) = \lceil \sqrt{m} \rceil = \lceil \sqrt{n} \rceil$.*

For a complete graph K_n, $n \geq 2$, we have the following cases for the edge burning number:

$$Eb(K_n) = \begin{cases} 1 & \text{if } n = 2 \\ 2 & \text{if } n = 3 \text{ or } 4 \\ 3 & \text{if } n \geq 5 \end{cases}$$

Observe that K_2 is just an edge and so $Eb(K_n) = 1$. For $n = 3$ *or* 4, by choosing any edge as the first source, all other edges will be burned in the second round for $n = 3$. For $n = 4$, only one edge is not incident to the first source, so we choose this edge as the second source. Thus, $Eb(K_n) = 2$. For $n \geq 5$, observe that there are at least 2 non-incident edges to the first source. Thus, with the same procedure as previously, we need three rounds to burn all the edges. On the other hand, the burning number of K_n, $n \geq 2$, is always 2. Note that a simple algorithm for computing the edge burning number on a path can be obtained by adapting the corresponding algorithm to find the burning sequence on paths [3]:

Finding an edge burning sequence on paths. Let $P_n = \{\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_\mathsf{m}\}$ denote a path with m edges and $Eb(P_n) = k$. Let $\{\mathsf{x}_1, \mathsf{x}_2, \cdots, \mathsf{x}_\mathsf{k}\}$ denote an edge burning sequence. We compute the sources as follows.

- for $0 \leq \mathsf{i} \leq \mathsf{k} - 2$, $\mathsf{x}_{\mathsf{k}-\mathsf{i}} = \mathsf{e}_{\mathsf{m}-\mathsf{i}^2-\mathsf{i}}$.
- if $\mathsf{m} \geq (\mathsf{k}-1)^2 + \mathsf{k}$, we get $\mathsf{x}_1 = \mathsf{e}_{\mathsf{m}-(\mathsf{k}-1)^2-(\mathsf{k}-1)}$, otherwise $\mathsf{x}_1 = \mathsf{e}_1$.

3.1 NP-Hardness of Edge Burning

In this section, we study the computational complexity of the edge burning problem (Edge Burning). First, observe that given a graph and a sequence of edges as sources we can check in polynomial time if the sequence is a solution for the edge burning. Thus, Edge Burning is in NP. Next, we show that Edge Burning is NP-hard. We obtain our result by using a reduction from the Vertex Cover problem, a well-known NP-complete problem. Vertex Cover: Given graph G and an integer k. Is there a subset S of vertices in G of size k or less such that every edge of G has at least one of its endpoints in S? Our constructions for the NP-hardness are based on [16].

▶ **Theorem 6.** *Edge Burning is NP-complete.*

Proof. Given an instance $(G = (V, E), k)$ of Vertex Cover we construct a graph G' as follows:

- for every vertex v of V, we add two adjacent vertices v^1 and v^2 in G',
- for every edge $\{u, v\} \in E$,
  - we add two vertices uv and vu in G', additionally, we add the edges $\{u^1, uv\}$, $\{v^1, vu\}$ and $\{uv, vu\}$ in G'. We then replace the edge $\{uv, vu\}$ by a path with $2n + 3$ vertices, where uv and vu are the endpoints of the path. Let w be the middle vertex of the path.
  - we add a path T_{uv} with $n + 1$ vertices and make one of its endpoints adjacent to w.
- we add $2n + 5$ pairs of adjacent vertices. Every pair is isolated with respect to the rest of the graph.

Notice that G' can be constructed in polynomial time.

We claim that (G, k) is a yes-instance for Vertex Cover if and only if G' has a burning sequence of length at most $k + 2n + 5$.

Assume that G has a vertex cover C of size at most k. We find an edge burning sequence F for G'. First, we add to F the edges $\{v_i^1, v_i^2\}$ of G' that correspond to the vertices of C in arbitrary order. We denote this set of edges as C'. This means in the first k rounds all the edges of C' are burned. After that we choose the $2n + 5$ isolated edges of G' as burning sources for F. Hence, we need $k + 2n + 5$ rounds in order to burn these edges. Now we show that the rest edges can be burned within $k + 2n + 5$ rounds. As we said in the first k rounds all the edges of C' are burned. Moreover, for every edge $\{u^1, u^2\}$ that does not belong to C' there is an edge of C' that is in distance $2n + 5$ since C is vertex cover of G. This means all the edges $\{u^1, u^2\}$ can be burned in the next $(2n + 5)$ rounds. Observe also that all the edges between an edge of C' and an edge $\{u^1, u^2\} \notin C'$ are in distance at most $2n + 4$ from some edge of C'. Furthermore, the edges of a path T_{uv} are in distance at most $2n + 4$ from some edge $\{v^1, v^2\} \in C'$. Thus, all the edges of G' can be burned in $k + 2n + 5$ rounds.

For the opposite direction, assume that G' has a burning sequence F of length at most $k + 2n + 5$. We show how to build a vertex cover C for G of size at most k. For every edge $\{u, v\} \in E$, we define H_{uv} to be a subgraph of G' that contains the edges $\{u^1, u^2\}$ and $\{v^1, v^2\}$, all the edges in distance at most $n + 2$ either from $\{u^1, u^2\}$ or $\{v^1, v^2\}$, all the edges in the path of u^1 to v^1 and the edges of the path T_{uv}. For every H_{uv} and for each burning source e in it, we check whether the distance between e and $\{u^1, u^2\}$ is smaller than the distance between e and $\{v^1, v^2\}$. If the former holds then we put u in C otherwise we put v in C. We break ties arbitrarily. Next we prove that C is a vertex cover of G. Assume that there is an edge $\{u, v\} \in E$, where neither u nor v belongs to C. This means that every burning source $e \in G'$ is closer to some $\{x, y\} \in G'$ other than $\{u^1, u^2\}$ and $\{v^1, v^2\}$. Thus, H_{uv} does not contain any edge burning source. Observe that the distance between an edge e outside of H_{uv} and the last edge of T_{uv} is at least $n + 2n + 5$. Hence, we need at least $n + 2n + 6$ rounds in order to burn the edges of H_{uv} which is strictly larger than $k + 2n + 5$. This leads to a contradiction, and so C is a vertex cover since for every edge $\{u, v\} \in E$ at least one of the endpoints belong to C. Finally, since we need $2n + 5$ rounds to burn the isolated edges, we get that the vertex cover is at most k. ◀

By construction, the graph G' in the proof of Theorem 6 is disconnected. Next, we can adjust the proof of Theorem 6 and show that the Edge Burning problem is also NP-hard on connected graphs. More precisely, let G be a graph and let v be a vertex of G. We extend this graph by adding a pair of adjacent vertices $\{x, y\}$ and make the vertex x adjacent to v of G. Observe that the minimum vertex cover of the new graph, H, is one more than in G, since we have added the edge $\{x, y\}$ and so one of the endpoints has to belong to the vertex cover. Let ℓ the minimum vertex cover of H. Next, we follow the construction of Theorem 6 but instead of using the isolated pair of adjacent vertices, we add a path $P = (x^2, Q, Q', e_1, Q', e_2, Q', \ldots, Q', e_{2n+5})$, where Q is a sequence of $(\ell + 2n + 4)$ vertices, Q' is a sequence of $(2n + 4)$ vertices, and $e_1, e_2, \cdots, e_{2n+5}$ are pairs of adjacent vertices that correspond to the isolated pairs of adjacent vertices of Theorem 6. Notice that the number of the edges of $P \setminus \{x^2, Q\}$ is $(2n + 4)(2n + 5) + (2n + 5) = (2n + 5)^2$. By Corollary 4 we get that any edge burning sequence for $P \setminus \{x^2, Q\}$ contains $2n + 5$ edges.

Again, we wish to show that the extended graph has a vertex cover of size at most ℓ if and only if there is an edge burning sequence with at most $\ell + 2n + 5$ sources. Observe that any vertex cover in the extended graph contains either x or y. Without loss of generality assume that x belong to vertex cover, otherwise we can swap y and x. For one direction,

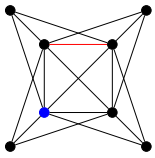


Figure 2 A complete split graph. By choosing the vertex as the first activator for the graph burning problem in two rounds all the vertices will be burned. On the other hand, by choosing the edge as the first source for the edge burning problem all the edges will be burned in three rounds.

we can burn all the edges by burning first the edge $\{x^1, x^2\}$, next the remaining edges that correspond to the vertices of the vertex cover and finally the edges of the path $P \setminus \{x^2, Q\}$ by using the algorithm for finding an edge burning sequence on a path previously described. For the other direction, assume that there is an edge burning sequence of size $\ell + 2n + 5$. Then, since the number of the edges of the path $P \setminus \{x^2, Q\}$ is $(2n + 5)^2$ we need $2n + 5$ sources to burn it. Moreover, the distance between these sources and $\{x^1, x^2\}$ is $\ell + 2n + 5$. Thus, the remaining ℓ sources need to be in H, and so, the corresponding vertices will be the vertex cover of size ℓ.

Hence, combining Theorems 3 and 6 and since the line graph of a connected graph is a connected claw-free graph, we get the following result.

▶ **Theorem 7.** *Graph Burning is NP-hard on connected claw-free graphs.*

3.2 Algorithms for edge burning on special graphs

The NP-hardness result for the edge burning problem makes it interesting to look for graph classes for which the edge burning problem admits a polynomial-time algorithm. In this section, we provide linear-time algorithms to compute the edge burning number on two graph classes: split graphs and cographs. We start with the class of split graphs. A graph is a split graph if its vertices can be partitioned into an independent set I and a clique C, where (C, I) is called a split partition of the graph. It is known that split graphs are self-complementary, that is, the complement of a split graph remains a split graph. First, we present the following observation about the edge burning number on a graph.

▶ **Observation 8.** *A graph G with at least two edges has $Eb(G) \geq 2$. Moreover, G has $Eb(G) = 2$ if and only if there is an edge e such that all the remaining edges are incident to e except for at most one of them.*

Proof. The first claim is straightforward since in the first round we can burn only one edge. Assume now that $Eb(G) = 2$, this means that there are two sources, (e_1, e_2). If G has only the edges e_1 and e_2 then we have the claim. Assume that there are more than two edges in the graph. Since $Eb(G) = 2$ and e_1 is the first source, the edges of $E(G) \setminus \{e_1, e_2\}$ are incident to e_1 in order to be burned in the second round. Hence, there is at most one edge non incident to e_1 and that can be the edge e_2. In the other direction, we assume first there is an edge e such that every other edge of the graph is incident to e. Then if we burn e in the first round then all the other edges will be burned in the second round. Observe that if there is exactly one edge non-incident to e we can also burn this edge at the end of the second round. Thus, in both cases the $Eb(G) = 2$. ◀

Using Observation 8, we present an algorithm to compute the edge burning number for split graphs in the following theorem.

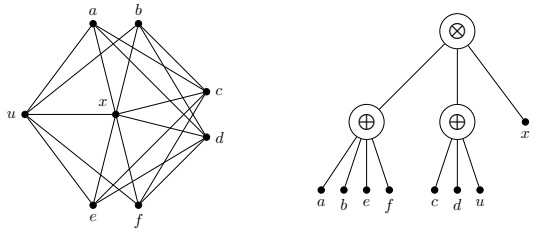


Figure 3 A cograph with its cotree. Observe that by burning the edge $\{x, u\}$, all the edges of the cograph will be burned in three rounds.

▶ **Theorem 9.** *The edge burning number of a split graph is at most* 3 *and it can be computed in linear time.*

Proof. Let $G = (V, E)$ be a split graph with clique partition (C, I). Based on Observation 8, we can check in linear time whether $Eb(G) = 2$; otherwise, we show that it must be $Eb(G) = 3$. In the first round we arbitrarily burn an edge from the clique C. In the second round all the incident edges will be burned and we burn again one of the remaining unburned edges. At this point every vertex of the clique has at least one burned incident edge. Thus, in the third round all the edges of the clique will be burned as well as all the edges between the vertices of the clique and the vertices of the independent set. Since there is no edge between the vertices of I, all the edges of the graph have been burned. Hence, $Eb(G) = 3$. Clearly, the above procedure can be implemented in linear time. ◀

Even in the case where the graph is a complete split graph, i.e., every vertex of the independent set is adjacent to every vertex of the clique, the edge burning number is at most 3, while the burning number of a complete split graph is 2. For example, see Figure 2.

Next, we consider the class of cographs. Let $G = (V, E)$ and $H = (U, F)$ be two undirected graphs with $V \cap U = \emptyset$. The *disjoint union* of G and H is the graph obtained from the union of G and H, denoted by $G \oplus H = (V \cup U, E \cup F)$. The *complete join* of G and H is the graph obtained from the union of G and H and adding edges between every pair of vertices that belong to different graphs, denoted by $G \otimes H = (V \cup U, E \cup F \cup \{vu \mid v \in V, u \in U\})$. A graph is a *cograph* if it can be generated from single-vertex graphs and recursively applying the disjoint union and complete join operations. By definition, the complement of a cograph is also a cograph. Cographs are exactly the graphs that do not contain chordless paths on 4 vertices, P_4, as induced subgraphs [7], and they can be recognized in linear time [8]. In the following theorem, we present an algorithm to compute the edge burning number for a cograph.

▶ **Theorem 10.** *The edge burning number of a cograph is at most* 3 *and it can be computed in linear time.*

Proof. Let $G = (V, E)$ be a connected cograph, then the last operation need to be complete join. Let $G = H_1 \otimes H_2$. Using Observation 8 in linear time we can determine if $Eb(G) = 2$; otherwise we show that the $Eb(G) = 3$. We consider as a first source an edge vu such that $v \in H_1$ and $u \in H_2$. By definition, v is adjacent to every vertex H_2 and u is adjacent to every vertex H_1, and so, at the end of the second round all the vertices of G have at least one burned incident edge. Thus, in the third round all the edges will be burned. The above steps can be implemented in linear time by using a suitable data structure, called *cotree* [8], (see for e.g., Figure 3). The construction of the cotree of G takes time $O(n + m)$ [8]. ◀

3.3 Parameterized complexity for edge burning

One way to cope with the NP-hardness of the edge burning problem is to study the problem in the context of parameterized complexity. We start with the following observation.

▶ **Observation 11.** *For a graph G with radius* rad *and diameter* d *we have that:* $\sqrt{\mathsf{d}+1}-1 \leq Eb(G) \leq \mathsf{rad}+2$.

Proof. For any graph G it is known that $\sqrt{\mathsf{d}+1} \leq b(G) \leq \mathsf{rad}+1$ [4]. Moreover, by Theorem 2 we have $b(G)-1 \leq Eb(G) \leq b(G)+1$. The observation yields from combining these two results. ◀

Based on the observation above, we can obtain a trivial XP algorithm for the edge burning problem parameterized by the diameter d of the graph. We compute all the subsets of edges of size at most d and for each such subset we check all the permutations. Every permutation can be considered as a sequence for the edge burning problem. Observe that this can be done in $O(\mathsf{d}! \cdot \mathsf{n}^{2(\mathsf{d}+1)})$ time. An XP algorithm has also been given in [12] for the graph burning problem parameterized by the diameter. An interesting direction to explore next is to find a parameter with respect to which the edge burning problem admits an FPT algorithm. We consider graphs with bounded *neighborhood diversity* and show that the edge burning problem is FPT and admits a linear kernel when parameterized by neighborhood diversity.

The *neighborhood diversity* has been defined by Lampis [15]. For a graph $G(V,E)$, two vertices x and y in V have the *same type* if $N(x) \setminus \{y\} = N(y) \setminus \{x\}$.

▶ **Definition 12.** *A graph $G=(V,E)$ has neighborhood diversity at most* nd*, if there exists a partition of $V(G)$ into at most* nd *sets, such that all vertices in each set have the same type.*

Observe that the vertices of a given type not only have the same (closed) neighborhood in G, but also form either a clique or an independent set in G. Further, there exists a polynomial time algorithm that finds a minimum partition of V into neighborhood types [15]. In the following theorem, we present a linear kernel for the edge burning problem parameterized by the neighborhood diversity of the input graph, similar to [11] for the graph burning problem.

▶ **Theorem 13.** *Edge burning admits a* 5nd*-kernel, where* nd *is the neighborhood diversity of the input graph.*

Proof. Let $G=(V,E)$ be a graph with neighbohood diversity, nd, and let $C_1, C_2, \cdots, C_{\mathsf{nd}}$ be the type classes of G. We show that edge burning admits a 5nd-kernel. To do that, we apply one of the following reduction rules for every type class in G:

▶ **Rule 13.1.** *If a class C_i forms a clique in G and $|C_i| > 5$ then we remove all the vertices apart from 5 vertices.*

To show that the rule is sound, observe that if C_i forms a clique and contains at least 5 vertices then in at most 3 rounds all the incident edges to the vertices of C_i can be burned. More precisely, if an edge incident to a vertex of C_i is burned, then in the next round all edges of $G[C_i]$ that are incident to that vertex will be burned. Thus, in the following round all the edges of $G[C_i]$ will be burned as well as all the incident edges from others classes to the vertices of C_i. On the other hand, if we set fire on an edge of $G[C_i]$ then in two rounds we have that every vertex of $G[C_i]$ have at least one burned incident edge, and so, in the next round all the rest incident edges to C_i will be burned. Hence, it is enough to keep 5 vertices for such a class C_i since a clique with 5 vertices needs 3 rounds in order to be burned.

▶ **Rule 13.2.** *If a class C_i forms an independent set in G and $|C_i| > 2$ then we remove all the vertices apart from 3 vertices.*

To show that the rule is sound, observe that if we pick an edge incident to a vertex of C_i as a source, then in the next round all the vertices of C_i will have at least one burned incident edge. This is because the endpoint of the source edge that is not in C_i is adjacent to every vertex of C_i. Thus, in the following round all incident edges to the vertices of C_i will be burned. Now, observe that the same holds even if an edge incident to a neighbor of C_i burned, since the vertices of C_i have the same neighborhood. Therefore, it is enough to keep 3 vertices. Finally, Rules 13.1 and 13.2 can be applied in polynomial time and the resulting graph has nd classes with at most 5 vertices in each class. ◀

In the following sections, we present our results on graph burning.

4 Graph burning with all the activators on a path

In this section, we consider our first variant of the graph burning problem in which all activators are on a path in the input graph. We remind the reader that our motivation for considering this variant comes from the scenario in social networks where a set of high performing nodes, termed as activators, must be connected to an emergency service line which enforces all the activators to be on path. Formally, we define the problem as follows.

▶ **Definition 14** (Graph burning with Activators on a Path (GbAP))**.** *The input is a graph G. The goal is to design an optimal burning sequence such that all the activators are on a path in G.*

Note that if we restrict the graph class to paths, then the GbAP problem reduces to the graph burning problem and it is solvable optimally in polynomial time. We first prove that GbAP problem is NP-complete even for trees with degree at most 3.

4.1 NP-completeness for GbAP

Given a sequence $\{x_1, x_2, \ldots, x_g\}$ of sources for the GbAP problem for trees, in time $O(n)$ we can compute $\mathsf{N_{g-i}[x_i]}$ for $1 \leq i \leq g$ and verify if $\cup_{\mathsf{i=1}}^{\mathsf{g}} \mathsf{N_{g-i}[x_i]} = \mathsf{V}$. Therefore, GbAP problem for trees is in NP. To show that GbAP problem is NP-hard, we show that there is a polynomial time reduction from the Distinct 3-Partition problem to the GbAP problem on trees with maximum degree 3. In fact, the reduction presented by Bessy et al. [2] to prove that graph burning is NP-hard for trees with maximum degree 3 also proves that GbAP problem is NP-hard for trees with maximum degree 3. To prove the NP-hardness result, Bessy et al. [2] presented a polynomial time reduction from the Distinct 3-Partition problem. An instance of the Distinct 3-Partition problem consists of the following: a finite set $X = \{a_1, a_2, \ldots, a_{3n}\}$ of positive distinct integers and a positive integer B where $\sum_{i=1}^{3n} a_i = nB$, and $B/4 < a_i < B/2$, for $1 \leq i \leq 3n$. The goal is to decide if there is a partition of X into n triples such that the elements in each triple add up to B. Let $m = \max\{a | a \in X\}$. For an instance for the Distinct 3-Partition problem as mentioned above, Bessy et al. [2] constructed a tree T consisting of $(2m+1)^2 + \frac{3(m^2+m)}{2}$ vertices with maximum degree 3 and proved that the given instance of the Distinct 3-Partition problem is an yes-instance if and only if burning number of T is $2m+1$. Further, they proved that because of the construction of T, any optimal solution for the graph burning problem must place all the activators on a path in T. Therefore, the implication of the reduction in [2] is

stronger. It proves that graph burning is NP-hard for trees with maximum degree 3 even in the restricted case when all the activators must be placed on a path. We get the following theorem.

▶ **Theorem 15.** *GbAP problem is NP-complete for trees with maximum degree three.*

4.2 A 2-approximation algorithm for GbAP on trees

We present a 2-approximation algorithm for the GbAP problem on trees. To design our algorithm, we crucially use the properties of a *diametral path* and *center* of a graph. Let d be the diameter of a graph G. Any path P in G is called a diametral path if the length of P is equal to d. A vertex with minimum eccentricity is called center of G. When the graph G is a tree, the following theorem is well known.

▶ **Theorem 16** (Jordan theorem [17])**.** *A tree has at most* 2 *centers. If a tree has* 2 *centers, they must be adjacent.*

We first prove the following lemma about diametral paths and centers of a tree.

▶ **Lemma 17.** *Every diametral path in a tree must contain every center.*

Proof. Let P be any diametral path in the tree, d be the diameter of the tree, and c be a center of the tree. Assume by contradiction that c is not contained in P. Let x_1 and x_2 be the two endpoints of the path P. Let $\mathsf{v_{mid}}$ be a vertex contained in P and is at a distance at least $\lfloor \mathsf{d}/2 \rfloor$ from x_1 and x_2. Therefore, $\mathsf{ecc}(\mathsf{v_{mid}}) \leq \lceil \mathsf{d}/2 \rceil$. Otherwise, it will contradict that the length of the diameter of the tree is d. Further, it also implies that the eccentricity of any center for the tree is $\leq \lceil \mathsf{d}/2 \rceil$. Next, we consider the unique path between $\mathsf{v_{mid}}$ and c in the tree.

1. $\mathsf{dist}(\mathsf{c}, \mathsf{v_{mid}}) > 1$: Consider the following paths: from c to x_1 and from c to x_2. At least one of the two paths must contain $\mathsf{v_{mid}}$. Without loss of generality, suppose the path from c to x_1 contains $\mathsf{v_{mid}}$. This implies that $\mathsf{dist}(\mathsf{c}, \mathsf{x_1}) = \mathsf{dist}(\mathsf{c}, \mathsf{v_{mid}}) + \mathsf{dist}(\mathsf{v_{mid}}, \mathsf{x_1}) > 1 + \lfloor \mathsf{d}/2 \rfloor \geq \lceil \mathsf{d}/2 \rceil$. Therefore, $\mathsf{ecc}(c) > \lceil \mathsf{d}/2 \rceil$. But this contradicts that c is a center of the tree because there exists a vertex $\mathsf{v_{mid}}$ with $\mathsf{ecc}(\mathsf{v_{mid}}) \leq \lceil \mathsf{d}/2 \rceil$.
2. $\mathsf{dist}(\mathsf{c}, \mathsf{v_{mid}}) = 1$: In this case both the paths, from c to x_1 and from c to x_2, must contain $\mathsf{v_{mid}}$. We consider the following two scenarios.
   a. Diameter d is even: For $x_i, i \in \{1, 2\}$, we have $\mathsf{dist}(\mathsf{c}, \mathsf{x_i}) = 1 + \mathsf{dist}(\mathsf{v_{mid}}, \mathsf{x_i}) = 1 + \mathsf{d}/2 > \lceil \mathsf{d}/2 \rceil$. Therefore, $\mathsf{ecc}(c) > \lceil \mathsf{d}/2 \rceil$ and it contradicts that c is a center.
   b. Diameter d is odd: For $x_i, i \in \{1, 2\}$, consider the x_i for which $\mathsf{dist}(\mathsf{x_i}, \mathsf{v_{mid}}) = (1 + \mathsf{d})/2$. Then, $\mathsf{dist}(\mathsf{c}, \mathsf{x_i}) = 1 + \mathsf{dist}(\mathsf{v_{mid}}, \mathsf{x_i}) = 1 + \frac{\mathsf{d}+1}{2} > \lceil \mathsf{d}/2 \rceil$. Therefore, $\mathsf{ecc}(c) > \lceil \mathsf{d}/2 \rceil$ and it contradicts that c is a center.

Therefore, if c is a center of the tree and P is any diametral path of the tree, then P must contain c. Hence the lemma. ◀

Next we make the following observation about two paths on a tree.

▶ **Observation 18.** *Let P_1 and P_2 be two paths in a tree and P_1 and P_2 are not vertex disjoint. Then all the common vertices between P_1 and P_2 induce a path $\hat{P}$ and $\hat{P}$ is a subpath of P_1 and P_2. Note that $\hat{P}$ may contain only one vertex.*

Proof. The proof follow immediately from the fact that a tree is an acyclic graph. Otherwise, if the common vertices do not form an induced path, then a subset of vertices belonging to only P_1, a subset of vertices belonging to only P_2, and a subset of vertices common to P_1 and P_2 form a cycle. Furthermore, since the common vertices induce a path, it must be a subpath of the two overlapping paths. ◀

Before presenting the algorithm, we prove some useful properties.

▶ **Lemma 19.** *For the GbAP problem on trees, if the tree has at least two diametral paths with only the centers as common vertices, then any optimum solution must place the first activator on one of the centers.*

Proof. Suppose not, and there exists an optimum solution that places the first activator on a vertex other than the centers. Since we are constrained to place all the activators on a path, a center can be burned only in the second or higher rounds. Since we assumed that the diametral paths have only centers as the common vertices and at least two diametral paths are there, the solution would require $\geq 2 + \lceil \mathsf{d}/2 \rceil$ rounds to burn the tree completely. However, from Lemma 17, every diametral path in a tree must contain every center. Therefore, any solution that places the first activator in a center would require $\leq 1 + \lceil \mathsf{d}/2 \rceil$ rounds to completely burn the tree, contradicting that the solution we started with is an optimum solution. Hence the Lemma. ◀

▶ **Lemma 20.** *For trees containing at least two diametral paths with only the centers as common vertices, an optimum solution for the GbAP problem needs* $1 + \lceil \mathsf{d}/2 \rceil$ *rounds.*

Proof. From Lemma 19, the first activator must be on a center. After that the optimum may place the remaining activators on one of the diametral paths. But it will take $\lceil \mathsf{d}/2 \rceil$ rounds to burn the other diametral paths. Therefore, any optimum solution takes $1 + \lceil \mathsf{d}/2 \rceil$ rounds. ◀

▶ **Lemma 21.** *For trees there exists an optimum solution for the GbAP problem that places all the activators on a diametral path.*

Proof. Let P' be a non diametral path and assume that all the activators in an optimum solution are on the path P'. Let P be a diametral path and d be the length of the diameter. Note that any optimum solution takes at most $1 + \lceil \mathsf{d}/2 \rceil$ rounds. We have the following two cases.

1. P' and P are edge disjoint: In this case, P' must contain one of the centers, say c, and the first activator must be on that center. Otherwise, the solution will take $> 1 + \lceil \mathsf{d}/2 \rceil$ rounds contradicting that it is optimal. Placing all the activators on P' starting with the first activator on the center c takes $1 + \lceil \mathsf{d}/2 \rceil$ rounds. Let x'_1 and x'_2 be the endpoints of path P'. Clearly, $\mathsf{dist}(\mathsf{c}, \mathsf{x}'_1) \leq \lceil \mathsf{d}/2 \rceil$ and $\mathsf{dist}(\mathsf{c}, \mathsf{x}'_2) \leq \lceil \mathsf{d}/2 \rceil$. Therefore, placing all the activators on P starting with the first activator on the center c yields another optimal solution.
2. P' and P are not edge disjoint: From Observation 18, the common vertices between P and P' induce a subpath. Let $\hat{P}$ be that subpath. Let $\hat{\mathsf{x}_1}$ and $\hat{\mathsf{x}_2}$ be the endpoints of $\hat{P}$. Let x'_1 be the endpoint of P' nearest to $\hat{\mathsf{x}_1}$ and x'_2 be the endpoint of P' nearest to $\hat{\mathsf{x}_2}$. Similarity, let x_1 be the endpoint of P nearest to $\hat{\mathsf{x}_1}$ and x_2 be the endpoint of P nearest to $\hat{\mathsf{x}_2}$. Clearly, $\mathsf{dist}(\hat{\mathsf{x}_1}, \mathsf{x}_1) \geq \mathsf{dist}(\hat{\mathsf{x}_1}, \mathsf{x}'_1)$ and $\mathsf{dist}(\hat{\mathsf{x}_2}, \mathsf{x}_2) \geq \mathsf{dist}(\hat{\mathsf{x}_2}, \mathsf{x}'_2)$. Therefore, placing all the activators on P' will contribute $\max\{\mathsf{dist}(\hat{\mathsf{x}_1}, \mathsf{x}_1), \mathsf{dist}(\hat{\mathsf{x}_1}, \mathsf{x}_2)\}$ many rounds in addition to burning the subpath $\hat{P}$ and the rest of the tree. Instead, placing all the activators on P will contribute $\max\{\mathsf{dist}(\hat{\mathsf{x}_1}, \mathsf{x}'_1), \mathsf{dist}(\hat{\mathsf{x}_2}, \mathsf{x}'_2)\}$ many rounds in addition to burning the subpath $\hat{P}$ and rest of the tree. Thus, placing all the activators on P is another optimal solution. ◀

Based on the above lemmas, we present a simple algorithm (Algorithm 1) for the GbAP problem on trees. In the following lemma we prove the approximation guarantee of Algorithm 1.

Algorithm 1 GbAP for input tree T.

Find a diametral path P, length d of P, and one center c of P
if T has at least two diametral paths with only centers as common vertices **then**
Output c followed by any arbitrary sequence of $\lceil \mathsf{d}/2 \rceil$ vertices on P as the sequence of activators for T
else
Find the sequence activators on P using the optimal algorithm for burning a path and output that sequence as the sequence of activators for T
end if

▶ **Lemma 22.** *Algorithm 1 finds a 2-approximation for the GbAP problem on trees.*

Proof. From Lemma 19 in the case when the tree has two diametral paths with only the centers as common vertices, the first activator must be on a center. Furthermore, by Lemma 20 any optimum solution takes $1 + \lceil \mathsf{d}/2 \rceil$ rounds in this case. Therefore, c followed by any arbitrary sequence of $\lceil \mathsf{d}/2 \rceil$ vertices on the diametral path P as activators is an optimal solution. In the other case, the tree has either a unique diametral path or overlapping diametral paths with length of the common subpath strictly greater than 2. In that case, Lemma 21 states that there exists an optimum solution that places all the activators on a diametral path. In this case, Algorithm 1 finds a sequence to optimally burn the diametral path P and outputs the same sequence as the burning sequence for the GbAP problem on T. Let ALG be the number rounds required by Algorithm 1 and let OPT be the number of rounds required by an optimum solution. It is straight forward to argue that after burning the path P in $\sqrt{\mathsf{d}+1}$ rounds, Algorithm 1 will take at most OPT many additional rounds. Therefore, $\mathsf{ALG} \leq \sqrt{\mathsf{d}+1} + \mathsf{OPT}$. But number of rounds required by any optimal solution is at least the number of rounds required to burn the diametral path. Therefore, $\mathsf{OPT} \geq \sqrt{\mathsf{d}+1}$ and $\frac{\mathsf{ALG}}{\mathsf{OPT}} \leq 2$. Hence, the lemma follows. ◀

Finally, in Lemma 23 we prove the running time of Algorithm 1.

▶ **Lemma 23.** *The running time of Algorithm 1 is $O(n)$*

Proof. In a tree, a diametral path P, the length of P, and the centers of P can be found using two invocations of a breadth first search algorithm. Any diametral path P' other than P must contain at least all the centers. Therefore, the existence of such a path can be checked using another breadth first search from one of the (at most two) centers. Finally, finding an optimal burning sequence on a path takes $O(n)$ time. Therefore, the running time of Algorithm 1 is $O(n)$. ◀

Combining Lemma 22 and Lemma 23 yields the following theorem.

▶ **Theorem 24.** *There exists a deterministic 2-approximation algorithm for the GbAP problem on trees with running time $O(n)$.*

5 Graph burning with known activators

In this section, we consider our second variant of the graph burning problem where a set of activators used by an optimal solution is given as input. We remind the reader that a motivation to consider this variant is to develop better approximation algorithms or efficient heuristic based algorithms. We call the set of activators used by an optimal algorithm as the optimal activator set. Formally, we define the following variant of the graph burning problem.

▶ **Definition 25** (Graph burning with Activators as Input (GbAI)). *The input is a graph G and an optimal activator set. The goal is to design an optimal burning sequence.*

The GbAI problem is known to be NP-hard [16]. In this section, we present a deterministic 2-approximation and a randomized better than 2-approximation algorithm for the GbAI problem. We also show that the GbAI problem is fixed parameter tractable parameterized by the size of the optimal activator set.

5.1 A deterministic 2-approximation algorithm

Let $\mathcal{S}$ be the optimal activator set given as input. Our approach is to burn the activators in the set $\mathcal{S}$ in any arbitrary order. A simple argument shows that the above straightforward deterministic approach results in a 2-approximation for the GbAI problem.

▶ **Theorem 26.** *GbAI admits a deterministic 2-approximation algorithm.*

Proof. Let $k = |\mathcal{S}|$. Let ALG be the number of rounds required by our approach and let OPT be the number of rounds required by an optimum solution. Clearly, $\mathsf{OPT} \geq k$. It is easy to observe that after burning the activators in the set $\mathcal{S}$, our approach will require at most OPT many additional rounds. Therefore, $\mathsf{ALG} \leq \mathsf{k} + \mathsf{OPT}$ and $\frac{\mathsf{ALG}}{\mathsf{OPT}} \leq 2$. Hence the theorem. ◀

5.2 A randomized better than 2-approximation algorithm

Let $\mathcal{S}$ be the optimal activator set given as input. Our approach is to select the activators one by one uniformly at random from the set $\mathcal{S}$ without replacement and burn them. Again, a simple argument shows that the above randomized approach yields a better than 2-approximation for the GbAI problem.

▶ **Theorem 27.** *GbAI admits a randomized better than 2-approximation algorithm.*

Proof. Let $k = |\mathcal{S}|$. Among the $k!$ different sequences of the activators in $\mathcal{S}$, at least one results in the optimal burning sequence. Therefore, probability that we obtain an optimal burning sequence by sampling uniformly at random without replacements is at least $\frac{1}{k!}$. Let ALG be the number of rounds required by our approach and let OPT be the number of rounds required by an optimum solution. Clearly, $\mathsf{OPT} \geq k$. In the case our sampling does not obtain an optimal burning sequence, it is easy to observe that after burning the activators, our approach will require at most OPT many additional rounds. Therefore, In the case our sampling does not obtain an optimal burning sequence, total rounds required is $\leq 2 \cdot \mathsf{OPT}$. Hence, $\mathsf{ALG} \leq \frac{1}{\mathsf{k!}} \cdot \mathsf{OPT} + (1 - \frac{1}{\mathsf{k!}}) \cdot (2 \cdot \mathsf{OPT})$ and $\frac{\mathsf{ALG}}{\mathsf{OPT}} \leq 2 - \frac{1}{\mathsf{k!}}$. Hence the theorem. ◀

5.3 GbAI is fixed parameter tractable

Let $\mathcal{S}$ be the optimal activator set given as input and $k = |\mathcal{S}|$. Also, let $\mathsf{perm}(\mathcal{S})$ denote the set of all permutations of the elements in $\mathcal{S}$. We present Algorithm 2.

▶ **Lemma 28.** *Algorithm 2 outputs an optimal burning sequence.*

Proof. Let $\{o_1, o_2, \ldots, o_k\} \in \mathsf{perm}(\mathcal{S})$ be an optimal burning sequence. Therefore, such a sequence satisfies the condition $\mathsf{N_{k-1}[o_1] \cup N_{k-2}[o_2] \cup N_{k-3}[o_3] \ldots N_0[o_k]} = \mathrm{V}$. Since Algorithm 2 uses an exhaustive search on the set $\mathsf{perm}(\mathcal{S})$, it must provide an optimal burning sequence. ◀

▶ **Lemma 29.** *For an input graph G with n vertices and an optimal activator set of size k, Algorithm 2 takes $O(2^{k \log k} \cdot n^2)$ time.*

Algorithm 2 GbAI for input graph G(V,E) and optimal activator set $\mathcal{S}$.

```
1: for a in S do
2:     for i in {0, 1, 2, ..., k − 1} do                ▷ k = |S|
3:         Compute N_i[a]
4:     end for
5: end for
6: for {a_1, a_2, ..., a_k} in perm(S) do
7:     if N_{k−1}[a_1] ∪ N_{k−2}(a_2) ∪ N_{k−3}[a_3] ... N_0[a_k] = V then
8:         Output {a_1, a_2, ..., a_k} as the optimal burning sequence and Stop
9:     end if
10: end for
```

Proof. Let m be the number of edges and n be the number of vertices in graph G. Computing the $\mathsf{N_r[v]}$ data structure requires a single breadth first search for vertex v. Therefore, the first for loop (Line 1 to 5) takes $O((m+n)\cdot k)$ times. Each set $\mathsf{N_r[v]}$ is of size at most n. The second for loop (Line 6 to 10 in Algorithm 2) takes $O(n)$ time for every sequence in the set $\mathsf{perm}(\mathcal{S})$. Therefore, total time required by the second for loop is $O(n \cdot k^k)$. Thus, total running time of Algorithm 2 is $O((m+n)\cdot k + n \cdot k^k)$. For $m = O(n^2)$, running time is $O(k^k \cdot n^2) = O(2^{k \log k} \cdot n^2)$. Hence, the lemma follows. ◀

Combining Lemma 28 and Lemma 29 yields the following theorem.

▶ **Theorem 30.** *The GbAI problem is FPT when parameterized by the size of the optimal activator set.*

6 Conclusions

In this paper we have studied the edge burning problem, and provided a collection of interesting results. After showing that it is NP-complete, we have provided linear-time algorithms for split graphs and cographs and we have shown that the problem admits a linear kernel when parameterized by neighborhood diversity. Furthermore, we have shown that the absolute difference between the edge burning number and the burning number for a graph is at most 1. We believe that this relationship between the edge burning number and the burning number for a graph can be of independent interest. We have also proved another relationship between the edge burning number and the burning number through the line graph and used this relationship to prove several properties of edge burning and graph burning on special graph classes. Our work raises some interesting and perhaps intriguing questions. One interesting direction is to find graph classes for which edge burning and graph burning have different computational complexities, say one problem is NP-hard and the other is polynomially solvable. Are there any such problems? Another interesting direction is to find graph classes for which the edge burning number is exactly the same as the burning number.

References

1 Dhanyamol Antony, Anita Das, Shirish Gosavi, Dalu Jacob, and Shashanka Kulamarva. Graph burning: Bounds and hardness, 2025. doi:10.48550/arXiv.2402.18984.

2 Stéphane Bessy, Anthony Bonato, Jeannette C. M. Janssen, Dieter Rautenbach, and Elham Roshanbin. Burning a graph is hard. *Discret. Appl. Math.*, 232:73–87, 2017. doi:10.1016/j.dam.2017.07.016.

3 Anthony Bonato, Jeannette Janssen, and Elham Roshanbin. How to burn a graph. *Internet Mathematics*, 12(1-2):85–100, 2016. doi:10.1080/15427951.2015.1103339.

4 Anthony Bonato, Jeannette C. M. Janssen, and Elham Roshanbin. Burning a graph as a model of social contagion. In Anthony Bonato, Fan Chung Graham, and Pawel Pralat, editors, *Algorithms and Models for the Web Graph - 11th International Workshop, WAW 2014, Beijing, China, December 17-18, 2014, Proceedings*, volume 8882 of *Lecture Notes in Computer Science*, pages 13–22. Springer, 2014. doi:10.1007/978-3-319-13123-8_2.

5 Anthony Bonato and Shahin Kamali. Approximation algorithms for graph burning. In T. V. Gopal and Junzo Watada, editors, *Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019, Kitakyushu, Japan, April 13-16, 2019, Proceedings*, volume 11436 of *Lecture Notes in Computer Science*, pages 74–92. Springer, 2019. doi:10.1007/978-3-030-14812-6_6.

6 Peter Brass, Christian Knauer, Hyeon-Suk Na, Chan-Su Shin, and Antoine Vigneron. The aligned k-center problem. *Int. J. Comput. Geom. Appl.*, 21(2):157–178, 2011. doi:10.1142/S0218195911003597.

7 D.G. Corneil, H. Lerchs, and L.K. Stewart. Complement reducible graphs. *Discrete Applied Mathematics*, 3:163–174, 1981. doi:10.1016/0166-218X(81)90013-5.

8 D.G. Corneil, Y. Perl, and L.K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14:926–934, 1985. doi:10.1137/0214065.

9 Arya Tanmay Gupta, Swapnil A Lokhande, and Kaushik Mondal. Burning grids and intervals. In *Algorithms and Discrete Applied Mathematics: 7th International Conference, CALDAM 2021, Rupnagar, India, February 11–13, 2021, Proceedings 7*, pages 66–79. Springer, 2021. doi:10.1007/978-3-030-67899-9_6.

10 Michaela Hiller, Arie MCA Koster, and Eberhard Triesch. On the burning number of p-caterpillars. In *Graphs and Combinatorial Optimization: from Theory to Applications: CTW2020 Proceedings*, pages 145–156. Springer, 2020.

11 Anjeneya Swami Kare and I Vinod Reddy. Parameterized algorithms for graph burning problem. In *Combinatorial Algorithms: 30th International Workshop, IWOCA 2019, Pisa, Italy, July 23–25, 2019, Proceedings*, pages 304–314. Springer, 2019. doi:10.1007/978-3-030-25005-8_25.

12 Yasuaki Kobayashi and Yota Otachi. Parameterized complexity of graph burning. *Algorithmica*, 84(8):2379–2393, 2022. doi:10.1007/S00453-022-00962-8.

13 S. Komala and U. Mary. Burning, edge burning & chromatic burning classification of some graph family. *Journal of Discrete Mathematical Sciences and Cryptography*, 27(2-B):665–674, 2024.

14 S. Komala and U. Mary. Edge burning: Relationship between edge and vertex burning of grid graph. *Journal of Dynamics and Control*, 8:1–12, 2024.

15 Michael Lampis. Algorithmic meta-theorems for restrictions of treewidth. *Algorithmica*, 64(1):19–37, 2012. doi:10.1007/S00453-011-9554-X.

16 Debajyoti Mondal, Angelin Jemima Rajasingh, N. Parthiban, and Indra Rajasingh. Apx-hardness and approximation for the k-burning number problem. *Theor. Comput. Sci.*, 932:21–30, 2022. doi:10.1016/j.tcs.2022.08.001.

17 Douglas B. West. *Introduction to Graph Theory, page 72.* Prentice Hall, Upper Saddle River, N.J., second edition, 2001.

Generalized Fibonacci Cubes Based on Swap and Mismatch Distance

Marcella Anselmo
Dipartimento di Informatica, Università di Salerno, Italy

Giuseppa Castiglione
Dipartimento di Matematica e Informatica, Università di Palermo, Italy

Manuela Flores
Dipartimento di Bioscienze e Territorio, Università del Molise, Campobasso, Italy

Dora Giammarresi
Dipartimento di Matematica, Università Roma "Tor Vergata", Italy

Maria Madonia
Dipartimento di Matematica e Informatica, Università di Catania, Italy

Sabrina Mantaci
Dipartimento di Matematica e Informatica, Università di Palermo, Italy

Abstract

The hypercube of dimension n is the graph with 2^n vertices associated to all binary words of length n and edges connecting pairs of vertices with Hamming distance equal to 1. Here, an edit distance based on swaps and mismatches is considered and referred to as *tilde-distance*. Accordingly, the *tilde-hypercube* is defined, with edges linking words having tilde-distance equal to 1. The focus is on the subgraphs of the tilde-hypercube obtained by removing all vertices having a given word as factor. If the word is 11, then the subgraph is called *tilde-Fibonacci cube*; in the case of a generic word, it is called *generalized tilde-Fibonacci cube*. The paper surveys recent results on the definition and characterization of those words that define generalized tilde-Fibonacci cubes that are *isometric* subgraphs of the tilde-hypercube. Finally, a special attention is given to the study of the tilde-Fibonacci cubes.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words; Theory of computation → Design and analysis of algorithms; Theory of computation → Formal languages and automata theory

Keywords and phrases Swap and mismatch distance, Isometric words, Hypercube

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.5

Category Research

Funding Partially supported by INdAM-GNCS Project 2025 - CUP E53C24001950001, FARB Project ORSA240597 of University of Salerno, TEAMS Project and PNRR MUR Project PE0000013-FAIR University of Catania, FFR fund University of Palermo, MUR Excellence Department Project MatMod@TOV, CUP E83C23000330006, awarded to the Department of Mathematics, University of Rome Tor Vergata, "ACoMPA – Algorithmic and Combinatorial Methods for Pangenome Analysis" (CUP B73C24001050001) funded by the NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem (Spoke 6).

1 Introduction

The n-dimensional *hypercube*, Q_n, is the well-known graph whose vertices are in correspondence with the 2^n words of length n over the binary alphabet $\{0, 1\}$ and two vertices are connected by an edge if the corresponding words differ in one position, that is, if their

Hamming distance is 1. Hence, the distance (number of edges in a minimal path) between two vertices in the graph is the Hamming distance of the corresponding words. The notion of hypercube has been extensively investigated because it is used to design interconnection networks (cf. [13, 17]) and also finds applications in theoretical chemistry (cf. [27] and [20, 24] for surveys). However, the critical limitation of the hypercube lies in its exponential growth in size, specifically in the number of vertices. Exploring some of its isometric subgraphs can improve efficiency. A subgraph of the n-dimensional hypercube is *isometric* to Q_n if the distance between any pair of vertices in such subgraphs is the same as the distance in the complete hypercube. With this aim, in 1993, Hsu introduced the *Fibonacci cubes* [21], obtained from Q_n by selecting only those vertices that do not contain 11 as a factor. They have a Fibonacci number of vertices making them useful in applications to reduce the complexity and to limit resources. They have many remarkable properties, also related to Fibonacci numbers (cf. [15]).

In 2012, Generalized Fibonacci cubes have been defined by means of a binary word f; the graph $Q_n(f)$ is a subgraphs of Q_n whose vertices do not contain f as a factor, i.e. the vertices are f-free binary words [22]. Then, the property of $Q_n(f)$ being an isometric subgraph of Q_n is related to some combinatorial properties of the avoided word f that in such cases is called *isometric*. More formally, a binary word f is isometric (or Ham-isometric) when, for any $n \geq 1$, $Q_n(f)$ is an isometric subgraph of Q_n, and *non-isometric*, otherwise [25]. The definition can also be done without mentioning graphs and only in terms of the Hamming distance. A word f is Ham-isometric, if for any pair of f-free words u and v of the same length, u can be transformed into v by a minimal sequence of symbol replacements that each time produce an f-free word, as well. Binary Ham-isometric words have been characterized in [23, 25, 35, 38, 39] and research on the topic remains very active [11, 36, 37, 12, 6, 7, 16, 8, 9, 4, 10].

Over the years, many variations of the hypercube have been introduced to enhance certain features. For example, folded hypercubes (cf. [18]) and enhanced hypercubes (cf. [32]) have been defined by adding some edges to the original structure, providing several advantages in terms of topological properties. Motivated by the same reasons, we introduced a type of generalized hypercube that employs a different distance, enabling the definition of isometric subgraphs. The inspiration and motivation came from many applications of computational biology, where many processes involve complex transformations and it is natural to consider not only replacement operations but also *swap* operations that exchange two adjacent different symbols in a word. The edit distance based on swaps and replacements is worth considering [1, 19, 29] instead of the Hamming distance. Actually, this distance was defined in the 70s by Wagner and Fischer [34, 33] who proved that it can be efficiently computed.

In [3] this distance is referred to as *tilde-distance*, since the $\sim$ symbol somehow evokes the swap operation. In [2], the tilde-distance is taken as the base to define the *tilde-hypercube*, $\tilde{Q}_n$; it has again all the n-binary strings as vertices, but two vertices are adjacent if the tilde-distance is equal to 1. This implies that $\tilde{Q}_n$ has more edges than Q_n; in particular, since a swap corresponds to two replacements of consecutive characters, some vertices at distance 2 in Q_n, become adjacent in $\tilde{Q}_n$.

This paper surveys most of the recent results on tilde-isometric words and generalized tilde-Fibonacci cubes. In particular, we collect the main definitions to give a recursive construction of tilde-hypercubes and enumerate their edges and vertices. Then, the subgraphs $\tilde{Q}_n(f)$ of the tilde-hypercubes are considered. They are obtained by selecting the vertices corresponding to f-free words, for a given word f. It is also reported the characterization of words f such that $\tilde{Q}_n(f)$ is an isometric subgraph of $\tilde{Q}_n$ as proved in [5]. We also exhibit an infinite family of tilde-isometric words that are not Hamming-isometric and vice versa. Finally, the last part

of the paper focuses on the word $f = 11$, that is both a Hamming- and a tilde-isometric word; the subgraph $\tilde{Q}_n(11)$ is referred to as the *tilde-Fibonacci cube*. We describe its recursive construction and present new simple results on diameter and radius that prove that the tilde-Fibonacci cubes are self-centered graphs.

2 Basic on Strings and Fibonacci Cubes

In this paper we focus only on the binary alphabet $\Sigma = \{0, 1\}$. A word (or string) w over Σ of length $|w| = n$, is $w = a_1 a_2 \cdots a_n$, where $a_1, a_2, \ldots, a_n$ are symbols in Σ. The set of all words over Σ is denoted Σ^*. Finally, ϵ denotes the *empty word* and $\Sigma^+ = \Sigma^* - \{\epsilon\}$. For any word $w = a_1 a_2 \cdots a_n$, the *reverse* of w is the word $w^{rev} = a_n a_{n-1} \cdots a_1$. If $x \in \Sigma$, $\overline{x}$ denotes the opposite of x, i.e $\overline{x} = 1$ if $x = 0$ and vice versa. Then we define the *complement* of w the word $\overline{w} = \overline{a}_1 \overline{a}_2 \cdots \overline{a}_n$.

Let $w[i]$ denote the symbol of w in position i, i.e. $w[i] = a_i$. Then, $w[i..j] = a_i \ldots a_j$, for $1 \leq i \leq j \leq n$, denotes a *factor* u of w of length $j - i + 1$ placed from the i-th to the jth position of w. We say that $I = [i..j]$ is the *interval* where the factor u occurs in w. The *prefix* (resp. *suffix*) of w of length l, with $1 \leq l \leq n - 1$ is $\mathrm{pre}_l(w) = w[1..l]$ (resp. $\mathrm{suf}_l(w) = w[n - l + 1..n]$). When $\mathrm{pre}_l(w) = \mathrm{suf}_l(w) = u$ then u is here referred to as an *overlap* of w of length l; in other frameworks, it is also called the border, or bifix of w (cf. [30]). A word w *avoids a word* f if w does not contain f as a factor: we also say that w is f*-free*.

An *edit operation* is a function $O : \Sigma^* \to \Sigma^*$ that transforms a word into another one. Let OP be a *set of edit operations*. The *edit distance* of words $u, v \in \Sigma^*$, with respect to the set OP, is the minimum number of edit operations in OP needed to transform u into v.

In this paper, we consider the edit distance that uses only *swap* and *replacement* operations. Note that these operations preserve the length of the word.

▶ **Definition 1.** *Let $w = a_1 a_2 \ldots a_n$ be a word over Σ.*
The replacement operation (or replacement, for short) on w at position i, with $i = 1, \ldots, n$, is defined by

$$R_i(a_1 a_2 \ldots a_{i-1} \boldsymbol{a_i} a_{i+1} \ldots a_n) = a_1 a_2 \ldots a_{i-1} \boldsymbol{\overline{a}_i} a_{i+1} \ldots a_n.$$

The swap operation (or swap, for short) on w at position i, with $i = 1, \ldots, n-1$ and $a_i \neq a_{i+1}$, is defined by

$$S_i(a_1 a_2 \ldots a_{i-1} \boldsymbol{a_i a_{i+1}} a_{i+2} \ldots a_n) = a_1 a_2 \ldots a_{i-1} \boldsymbol{a_{i+1} a_i} a_{i+2} \ldots a_n.$$

Note that one swap corresponds to two replacements of consecutive symbols.

The *Hamming distance* $\mathrm{dist}_H(u, v)$ of equal-length words $u, v \in \Sigma^*$ is defined as the minimum number of replacements needed to obtain v from u.

Let $G = (V(G), E(G))$ be a graph, $V(G)$ be the set of its nodes and $E(G)$ be the set of its edges. The distance of $u, v \in V(G)$, $\mathrm{dist}_G(u, v)$, is the length of the shortest path that connects u and v in G. A subgraph $S = (V(S), E(S))$ of a (connected) graph G is an *isometric subgraph* if for any $u, v \in V(S)$, $\mathrm{dist}_S(u, v) = \mathrm{dist}_G(u, v)$.

The *n-hypercube*, or binary n-cube, Q_n, is a graph with 2^n vertices, labeled with the words of length n and edges connecting two vertices u and v in Q_n when their labels differ exactly in 1 position, i.e. when $\mathrm{dist}_H(u, v) = 1$. Therefore, $\mathrm{dist}_{Q_n}(u, v) = \mathrm{dist}_H(u, v)$.

Denote by f_n the n-th Fibonacci number, defined by $f_1 = 1, f_2 = 1$ and $f_i = f_{i-1} + f_{i-2}$, for $i \geq 3$. The *Fibonacci cube* (cf. [22]) F_n of order n is the subgraph of Q_n whose vertices are binary words of length n avoiding the factor 11. It is well known that F_n is an isometric subgraph of Q_n (cf. [24]).

One of the main properties of Q_n and F_n is their *recursive structure* that have been extensively studied (cf. [21, 28, 26, 24]).

The following results are well-known, but are hereby stated for future reference.

▶ **Proposition 2.** *Let Q_n be the hypercube of order n. Then*
- $|V(Q_n)| = 2^n$,
- $|E(Q_n)| = n2^{n-1}$.

▶ **Proposition 3.** *Let F_n be the Fibonacci cube. Then:*
- $|V(F_n)| = f_{n+2}$,
- $|E(F_n)| = \frac{2(n+1)f_n + nf_{n+1}}{5}$.

In other terms, if the number of vertices $N = 2^n$ of the hypercube is taken as main parameter, the number of edges of a hypercube with N vertices is $(N \log N)/2$.

The sequence $|E(F_n)|$ is Sequence A001629 in [31]. Hence, the number of edges of a Fibonacci cube with N vertices, $N = f_{n+2}$, is $O(N \log N)$, asymptotically equal to the number of edges of a hypercube with the same number of vertices.

In order to generalize the notion of Fibonacci cube, one can consider the subgraphs of the n-hypercube whose nodes are f-free, for some word $f \in \Sigma^*$, denoted by $Q_n(f)$, called *generalized Fibonacci cubes*. Of course, if $n < |f|$, then $Q_n(f) = Q_n$; so it makes sense to consider $n \geq |f|$. Unfortunately, not all the words f make $Q_n(f)$ an isometric subgraph of Q_n. The words having this property have been widely studied (cf. [23, 25, 35, 38, 39]). These words are in fact called isometric words, because they reflect on this isometry property on the corresponding graphs.

Under the combinatorial point of view, isometric words are defined as follows. A word $f \in \Sigma^*$ is *Ham-isometric* (or simply *isometric*) if and only if $Q_n(f)$ is an isometric subgraph of Q_n. A word that is not Ham-isometric is said to be *Ham-non-isometric*. In terms of transformations, isometric words can be characterized by the following:

▶ **Proposition 4.** *A word f is isometric iff for any pair of f-free words u and v, there exists a sequence of $\mathrm{dist}_H(u, v)$ replacements that transforms u into v where all the intermediate words are also f-free.*

A word w has a 2-*error overlap* if there exists $l \leq |w|$ such that $\mathrm{pre}_l(w)$ and $\mathrm{suf}_l(w)$ have Hamming distance 2. Then, the following characterization of Ham-non-isometric words is proved (cf. [35]).

▶ **Proposition 5** ([35])**.** *A word f is Ham-non-isometric if and only if f has a 2-error overlap.*

3 Tilde-distance and Tilde-hypercube

In this section, the edit distance based on swap and replacement operations is considered. In [3] it is called tilde-distance and denoted by $dist_\sim$. According to this definition one can define the n-tilde-hypercube. We start with the definition of tilde-distance.

▶ **Definition 6.** *Let $u, v \in \Sigma^*$ be words of equal length. The tilde-distance $\mathrm{dist}_\sim(u, v)$ between u and v is the minimum number of replacements and swaps needed to transform u into v.*

Note that for all $u, v \in \Sigma^*$, $\mathrm{dist}_\sim(u, v) \leq \mathrm{dist}_H(u, v)$, since a swap is a shortcut for two adjacent replacements.

▶ **Example 7.** The words $u = 1011$ and $v = 0110$ have tilde-distance $\mathrm{dist}_\sim(u, v) = 2$. In fact, v can be obtained from u with a swap S_1 of the first and the second bits, and a replacement R_4 in the fourth position. Note that, in order to compute the Hamming distance, three replacements are needed in positions $1, 2$ and 4, therefore $\mathrm{dist}_H(u, v) = 3$.

In order to describe the sequence of the operations that are used to compute the tilde-distance of two words, we need the following definition of a tilde-transformation:

▶ **Definition 8.** *Let $u, v \in \Sigma^*$ be words of equal length and $dist_\sim(u, v) = d$. A tilde-transformation τ from u to v is a sequence of $d+1$ words $(w_0, w_1, \ldots, w_d)$ such that $w_0 = u$, $w_d = v$, and for any $k = 0, 1, \ldots, d-1$, $\mathrm{dist}_\sim(w_k, w_{k+1}) = 1$. Further, τ is f-free if for any $i = 0, 1, \ldots, d$, word w_i is f-free.*

A tilde-transformation $(w_0, w_1, \ldots, w_d)$ from u to v with $dist_\sim(u, v) = d$ is associated to a sequence of d operations $(O_{i_1}, O_{i_2}, \ldots, O_{i_d})$ such that, for any $k = 1, \ldots, d$, $O_{i_k} \in \{R_{i_k}, S_{i_k}\}$ and $w_k = O_{i_k}(w_{k-1})$; it can be graphically represented as follows:

$$u = w_0 \xrightarrow{O_{i_1}} w_1 \xrightarrow{O_{i_2}} \cdots \xrightarrow{O_{i_h}} w_h = v.$$

With a little abuse of notation, in the sequel we will refer to a tilde-transformation both as a sequence of words and as a sequence of operations. Furthermore, as a consequence of the definition, in a tilde-transformation, the positions i_1, i_2, …, i_d are all distinct.

In the following we give some examples that show some special features of tilde-transformations, which never occur when transformations using only replacements are considered. This highlights, on the one hand, the richness of new situations that arise from introducing the swap operation, and on the other hand, it anticipates the need for new and more sophisticated techniques to handle these increasingly complex scenarios. First of all, we point out that when only replacements are used, the different transformations from u to v use the same set of operations, possibly applied in a different order. This is not the case for the different tilde-transformations between two words. The following two examples show two singular cases of different tilde-transformations which use different sets of operations.

▶ **Example 9.** Let $u = 100$, $v = 001$. In this case $\sigma_1 = (S_1, S_2)$ and $\sigma_2 = (R_1, R_2)$ are two tilde-transformations from u to v on different sets of operations. In particular, observe that σ_1 flips twice the bit in the second position, whereas in σ_2 the second bit is not involved.

$$\sigma_1 : 100 \xrightarrow{S_1} 010 \xrightarrow{S_2} 001 \qquad \sigma_2 : 100 \xrightarrow{R_1} 000 \xrightarrow{R_3} 001$$

▶ **Example 10.** Consider $u' = 010$ and $v' = 101$ and the tilde-transformations $\rho_1 = (S_1, R_3)$ and $\rho_2 = (S_2, R_1)$ from u' to v'. Here, different sets of operations are used and, differently from Example 9 in both transformations each symbol is changed just once:

$$\rho_1 : 010 \xrightarrow{S_1} 100 \xrightarrow{R_3} 101 \qquad \rho_2 : 010 \xrightarrow{S_2} 001 \xrightarrow{R_1} 101$$

The variety of situations described above translates into a higher degree of difficulty of the tilde-transformations (compared to the Hamming transformations) when handling some property like, for instance, isometricity.

▶ Remark 11. Let $u, v \in \Sigma^m$ and τ be a tilde-transformation from u to v. Referring to Example 9, if τ contains two swaps, S_i and S_{i+1}, at consecutive positions i and $i+1$ of u, then such swaps can be substituted by two replacements, namely R_i and R_{i+2}, still obtaining a tilde-transformation from u to v of length equal to their distance. Hence, in particular, if $u = 00u'$ and $v = 10v'$, among all sequences of swaps and replacements of minimal length that transform u into v there is one starting with the replacement R_1.

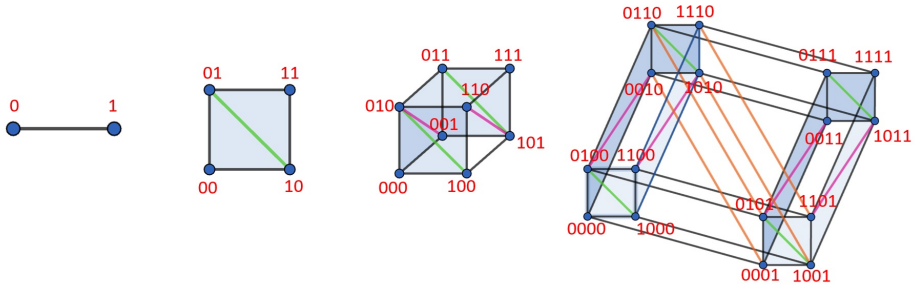


Figure 1 Tilde-hypercubes with n=1,2,3,4 (the colored edges are those added to the traditional hypercube).

▶ Remark 12. Let $u, v \in \Sigma^m$ and τ be a tilde-transformation from u to v. Referring to Example 10. If τ contains a swap S_{i+1} and a replacement R_i then they can be substituted by S_i and R_{i+2}. As a consequence, if $u = 01u'$ and $v = 10v'$ among all sequences of swaps and replacements of minimal length that transform u into v there is one starting with the replacement R_1.

Based on the definition of tilde-distance, a natural extension of the concept of n-hypercube is given in [2] as follows:

▶ **Definition 13.** *The n-tilde-hypercube $\tilde{Q}_n$, or tilde-hypercube of order n, is a graph with 2^n vertices, labeled with binary words of length n. Two vertices in $\tilde{Q}_n$, are adjacent whenever the tilde-distance of their labels is* 1.

Figure 1 shows the tilde-hypercubes of order $1, 2, 3, 4$.

▶ Remark 14. Q_n is a proper subgraph of $\tilde{Q}_n$. In fact, for $u, v \in \Sigma^*$, $\mathrm{dist}_H(u, v) = 1$ implies $\mathrm{dist}_\sim(u, v) = 1$. Note that Q_n is not an isometric subgraph of $\tilde{Q}_n$. Indeed, for any $n \geq 2$, there exists a pair of words (u_n, v_n) of length n such that $\mathrm{dist}_\sim(u_n, v_n) = 1$ and $\mathrm{dist}_H(u_n, v_n) \neq 1$. For instance, for any $0 \leq k, h \leq n-2$, $h + k = n - 2$, consider the words $u_n = 0^h 010^k$ and $v_n = 0^h 100^k$; then $\mathrm{dist}_\sim(u_n, v_v) = 1$ and $\mathrm{dist}_H(u_n, v_n) = 2$, therefore (u_n, v_n) is an edge in $\tilde{Q}_n$ but not in Q_n.

The following lemma is the main tool for exhibiting a recursive definition of the tilde-hypercube, in analogy with the classical hypercube.

▶ **Lemma 15.** *For any $u, v \in \Sigma^{n-1}$, $\mathrm{dist}_\sim(u0, v0) = \mathrm{dist}_\sim(u, v) = \mathrm{dist}_\sim(u1, v1)$ and $\mathrm{dist}_\sim(u0, u1) = 1$. Moreover, for any $u' \in \Sigma^{n-2}$, $\mathrm{dist}_\sim(u'01, u'10) = 1$.*

▶ **Proposition 16** ([2])**.** *The n-tilde-hypercubes $\tilde{Q}_n$, with $n \geq 1$, can be recursively defined.*

Proof. If $n = 1$, $\tilde{Q}_1$ has just two vertices 0 and 1 connected by an edge.

Suppose all the tilde-hypercubes of dimension smaller than n have been defined. The hypercube $\tilde{Q}_n$ is recursively constructed as follows. Start with a copy of $\tilde{Q}_{n-1}$ where every vertex u is replaced by $u0$, and denote this copy by $\tilde{Q}_{n-1}0$ and a second copy where every vertex u is replaced by $u1$, and denoted by $\tilde{Q}_{n-1}1$.

By Lemma 15, if $(u, v) \in E(\tilde{Q}_{n-1})$, then $(u0, v0), (u1, v1) \in E(\tilde{Q}_{n-1})$, this means that $\tilde{Q}_{n-1}0$ and $\tilde{Q}_{n-1}1$ are subgraphs of $\tilde{Q}_n$. Moreover, for any $u \in \Sigma^{n-1}$, there is an edge $(u0, u1)$ in $\tilde{Q}_n$ with $u0 \in V(\tilde{Q}_{n-1}0)$ and $u1 \in V(\tilde{Q}_{n-1}1)$. Finally, for any $v \in \Sigma^{n-2}$ $(v10, v01) \in E(\tilde{Q}_n)$ with $v10 \in V(\tilde{Q}_{n-1}0)$ and $v01 \in V(\tilde{Q}_{n-1}1)$. In Fig. 1, these latter edges added in the fourth step of recursion, are depicted with orange edges. For any other pair of words $u, v \in \{0, 1\}^n$, $\mathrm{dist}_\sim(u, v) > 1$, then (u, v) is not an edge of $\tilde{Q}_n$. ◀

▶ **Corollary 17.** *Let $\tilde{Q}_n$ be the tilde-hypercube of order n. Then, $|E(\tilde{Q}_1)| = 1$ and, for any $n \geq 2$ $|E(\tilde{Q}_n)| = 2|E(\tilde{Q}_{n-1})| + 2^{n-1} + 2^{n-2}$.*

By solving the above recurrence, we find the exact formula $|E(\tilde{Q}_n)| = (3n-1) \cdot 2^{n-2}$ (Sequence A053220 in [31]). Let $\tilde{EQ}(N)$ be the number of edges of the tilde-hypercube with N vertices, $N = 2^n$. Then,

$$\tilde{EQ}(N) = \frac{3N(\log N - 1)}{4}. \tag{1}$$

4 Tilde-hypercube Avoiding a Word and Tilde-isometric Words

In analogy with the n-hypercube avoiding a word based on the Hamming distance, referred to as generalized Fibonacci cube in [22], in this section, we consider the n-tilde-hypercube avoiding a word, based on the tilde-distance, here named generalized tilde-Fibonacci cubes.

In [2] the following definition is given:

▶ **Definition 18.** *The n-tilde-hypercube avoiding a word f, or the tilde-hypercube of order n avoiding a word f, denoted $\tilde{Q}_n(f)$, is the subgraph of $\tilde{Q}_n$ obtained by removing those vertices which contain f as a factor.*

We are interested in those words f such that $\tilde{Q}_n(f)$ is an isometric subgraph of $\tilde{Q}_n$, i.e. the distance of two vertices of $\tilde{Q}_n(f)$ is equal to the tilde-distance of the corresponding labels.

▶ **Definition 19.** *A word $f \in \Sigma^*$ is tilde-isometric if and only if for all $n \geq |f|$, $\tilde{Q}_n(f)$ is an isometric subgraph of $\tilde{Q}_n$.*

The following proposition characterizes isometric words re-stating the definition of isometric word under a combinatorial point of view.

▶ **Proposition 20.** *Let $f \in \Sigma^*$ be a word of length n with $n \geq 1$. The word f is tilde-isometric if and only if for any pair of f-free words u and v of equal length $m \geq n$, there exists a tilde-transformation from u to v that is f-free. It is tilde-non-isometric if it is not tilde-isometric.*

In order to show that a word is tilde-non-isometric it is sufficient to exhibit a pair (u, v) of f-free words such that all the tilde-transformations from u to v are not f-free. Such pair of words is called *a pair of tilde-witnesses*. More challenging is to prove that a word is tilde-isometric.

▶ **Example 21.** The word $f = 1010$ is tilde-non-isometric. In fact, let $u = 11000$ and $v = 10110$; then (u, v) is a pair of tilde-witnesses for f. In fact u and v are f-free; moreover there are only two possible tilde-transformations from u to v, namely $11000 \xrightarrow{S_2} 10100 \xrightarrow{R_4} 10110$ and $11000 \xrightarrow{R_4} 11010 \xrightarrow{S_2} 10110$, and in both cases 1010 appears as factor after the first step. On the other side, observe that f is Ham-isometric by Proposition 5.

The following straightforward property of tilde-isometric binary words is very helpful to simplify proofs and computations.

▶ Remark 22. A word f is tilde-isometric iff $\overline{f}$ is tilde-isometric iff f^{rev} is tilde-isometric.

In view of Remark 22 , we will focus on words starting with 1.

5 Characterization of Tilde-isometric Words

The characterization of Ham-isometric words given in [38] and here reported as Proposition 5, uses the notion of 2-error overlap. In this section we introduce the corresponding definition that refers to the tilde-distance. Tilde-error overlaps will have a main role in the characterization of tilde-isometric words but the presence of swap operations will force us to handle them with care.

▶ **Definition 23.** *Let $f \in \Sigma^n$. Then, f has a q-tilde-error overlap of length ℓ and shift $r = n - \ell$, with $1 \leq \ell \leq n-1$ and $0 \leq q \leq \ell$, if* $\mathrm{dist}_\sim(\mathrm{pre}_\ell(f), \mathrm{suf}_\ell(f)) = q$.

▶ **Example 24.** The word $f = 1101110101101$ has a 2-tilde-error overlap of length 6 and shift 7. Indeed, $\mathrm{pre}_6(f) = 110111$, $\mathrm{suf}_6(f) = 101101$ and $\mathrm{dist}_\sim(110111, 101101) = 2$.

For our proofs, given a word f with a q-tilde-error overlap of length ℓ, we will study the tilde-transformations τ from $\mathrm{pre}_\ell(f)$ to $\mathrm{suf}_\ell(f)$ with q operations. For this reason, it is useful to refer to the alignment of the two strings $\mathrm{pre}_\ell(f)$ to $\mathrm{suf}_\ell(f)$. Furthermore, for our purpose, it is relevant to consider also the bits adjacent to a tilde-error overlap of a word. For this reason, we introduce the following notation.

Let f be a word in $\{0,1\}^*$ and \$ be a symbol different from $0, 1$, here used as delimiter of a word, that by definition "matches" any symbol of the word. Consider f with its delimiters $\$f\$$. A q-tilde-error overlap of length ℓ is denoted by $\binom{\$xb}{ay\$}$ where xb, ay are a prefix and a suffix, respectively, of f, $a, b \in \Sigma$, $x, y \in \Sigma^*$ with $|x| = |y| = \ell$, and $\mathrm{dist}_\sim(\$xb, ay\$) = \mathrm{dist}_\sim(x, y) = q$. This notation makes evident the fact that in f the prefix x is followed by b and the suffix y is preceded by a. Moreover, a q-tilde-error overlap $\binom{\$xb}{ay\$}$ is sometimes factorized into blocks to highlight the significant parts. For example, the 2-tilde-error overlap in Example 24 is denoted by $\binom{\$1}{01}\binom{1011}{0110}\binom{10}{1\$}$ because $\mathrm{dist}_\sim(110111, 101101) = 2 = \mathrm{dist}_\sim(1011, 0110)$.

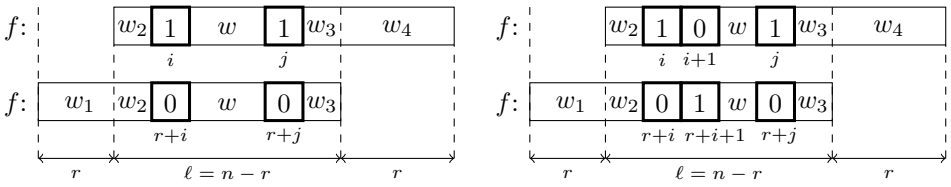


Figure 2 A word f and its 2-tilde-error overlap of shift r and length $\ell = n - r$, with tilde-transformation $(O_i, O_j) = (R_i, R_j)$ (left), and $(O_i, O_j) = (S_i, R_j)$ (right).

In the sequel, we will be interested in the specific case of 1-tilde-error overlap where the single error in the alignment is a swap and in all the cases of 2-tilde-error overlaps. Consider a 2-tilde-error overlap of f of shift r, length $\ell = n - r$, and let (O_i, O_j), $1 \leq i < j \leq \ell$, be a tilde-transformation from $\mathrm{pre}_\ell(f)$ to $\mathrm{suf}_\ell(f)$. Observe that the positions in $\mathrm{pre}_\ell(f)$ modified by O_i are either i or both i and $i+1$, following that O_i is a replacement or a swap. Hence, the number of the positions modified by O_i and O_j may be 2, 3 or 4. Fig. 2 shows a word f with its 2-tilde-error overlap of shift r and length $\ell = n - r$. With our notation, the 2-tilde-error overlap is $\binom{\$w_2 1 w 1 w_3 b}{a w_2 0 w 0 w_3 \$}$ in the figure on the left and $\binom{\$w_2 10 w 1 w_3 b}{a w_2 01 w 0 w_3 \$}$ in the figure on the right, where b is the first letter of w_4 and a the last letter of w_1. A tilde-transformation from $\mathrm{pre}_\ell(f)$ to $\mathrm{suf}_\ell(f)$ is given by $(O_i, O_j) = (R_i, R_j)$ in the first case and by $(O_i, O_j) = (S_i, R_j)$ in the second case. We say that a 2-tilde-error overlap has *non-adjacent errors* when there is at least one character interleaving the positions modified by O_i and those modified by O_j.

The 2-tilde-error overlap $\binom{\$x}{ax}\binom{100}{001}\binom{yb}{y\$}$ is also considered as having non-adjacent errors because it admits the tilde-transformation $(O_i, O_j) = (R_i, R_{i+2})$, despite it has also the other tilde-transformation $(O_i, O_j) = (S_i, S_{i+1})$.

In all the other cases, we say that the 2-tilde-error overlap has *adjacent errors*.

Let us state the characterization of tilde-isometric words in terms of special configurations in their overlap proved in [5].

▶ **Theorem 25.** *A word $f \in \Sigma^n$ is tilde-non-isometric if and only if one of the following cases occurs (up to complement, reverse and inversion of rows):*

(C0) *f has a 1-tilde-error overlap $\binom{\$x}{ax}\binom{01}{10}\binom{yb}{y\$}$ with $x, y \in \Sigma^*$, $a, b \in \Sigma$;*

(C1) *f has a 2-tilde-error overlap with non-adjacent errors, different from $\binom{\$x}{ax}\binom{000}{101}\binom{yb}{y\$}$ with $x, y \in \Sigma^+$, $a, b \in \Sigma$;*

(C2) *f has a 2-tilde-error overlap $\binom{\$x}{ax}\binom{0101}{1010}\binom{yb}{y\$}$ or $\binom{\$x}{ax}\binom{0110}{1001}\binom{yb}{y\$}$ with $x, y \in \Sigma^*$, $a, b \in \Sigma$;*

(C3) *f has a 2-tilde-error overlap $\binom{\$x}{ax}\binom{010}{101}\binom{yb}{y\$}$ with $x, y \in \Sigma^*$, $a, b \in \Sigma$;*

(C4) *f has a 2-tilde-error overlap $\binom{\$x}{ax}\binom{011}{100}\binom{0}{\$}$ with $x \in \Sigma^*$, $a \in \Sigma$;*

(C5) *f has a 2-tilde-error overlap $\binom{\$}{0}\binom{00}{11}\binom{1}{\$}$.*

Thanks to Theorem 25, we can classify any word as isometric or non-isometric. In the following, several examples are given. The following are two examples of words with a 2-tilde-error overlap with adjacent errors; the first one is tilde-isometric, the second one is tilde-non-isometric.

▶ **Example 26.** The word $f = 010110000$ is tilde-isometric; indeed its unique 2-tilde-error overlap has shift 5 and length 4, $\binom{\$0}{10}\binom{101}{000}\binom{1}{\$}$. Note this is the case of non-adjacent errors but of the type prohibited by the condition in (C1).

▶ **Example 27.** The word $f = 1011000$ is tilde-non-isometric; indeed it has the 2-tilde-error overlap, of shift 4 and length 3, $\binom{\$}{1}\binom{101}{000}\binom{1}{\$}$, that satisfies (C1). Note that the pair $(u, v) = (10110011000, 10101001000)$ is a pair of tilde-witnesses for f.

The following example shows an infinite family of words that are tilde-isometric and Ham-non-isometric.

▶ **Example 28.** All the words $f = 1^n0^m$ (and their complement $f = 0^n1^m$) for $n, m > 2$ are Ham-non isometric, by Proposition 5, and tilde-isometric. In fact, for $n, m > 2$, $f = 1^n0^m$ has only two 2-tilde-error overlaps and none of them fall into a case in the statement of Theorem 25. The first one is the tilde-error overlap with shift 2, $\binom{\$1^{n-2}}{11^{n-2}}\binom{11}{00}\binom{0^{n-2}0}{0^{n-2}\$}$, the other one has shift $n + m - 2$, and it is $\binom{\$}{0}\binom{11}{00}\binom{1}{\$}$.

Instead, if $n = m = 2$, $f = 1100$ is both tilde-non-isometric and Ham-non-isometric. In fact, f has only one 2-tilde-error overlap $\binom{\$}{1}\binom{11}{00}\binom{0}{\$}$, corresponding to case $(C5)$ of Theorem 25. Moreover, one can verify that $(u, v) = (110100, 101010)$ is a pair of tilde-witnesses for f.

The next corollary shows that there exist Ham-isometric words that are not tilde-isometric. This fact, together with Example 5, proves that the families of Ham-isometric and tilde-isometric words are incomparable.

▶ **Corollary 29.** *The word $f = 1010$ is Ham isometric and tilde non-isometric.*

Proof. The word 1010 has no 2-error overlap, therefore is isometric. Instead, it has a 2-tilde-error overlap $\binom{1}{\$}\binom{010}{101}\binom{\$}{0}$ corresponding to case $(C3)$ of Theorem 25. ◀

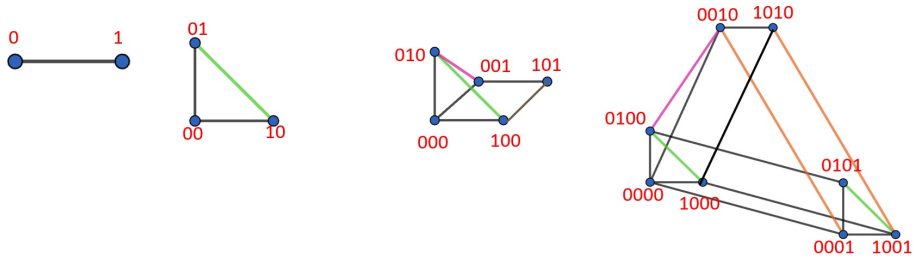


Figure 3 Tilde-Fibonacci cubes with n=1,2,3,4 (the colored edges are those added to the traditional hypercube).

6 Tilde-Fibonacci Cubes

Theorem 25 allows also to give light to isometric subgraphs of $\tilde{Q}_n$ avoiding a word. In fact they are those avoiding a tilde-isometric word.

In analogy with the definition of Fibonacci cubes introduced by Hsu [21], we give the following:

▶ **Definition 30.** *Let $n \geq 1$ and $\tilde{Q}_n$ be the tilde-hypercube of dimension n. The n-tilde-Fibonacci cube, or tilde-Fibonacci cube of dimension n is $\tilde{F}_n = \tilde{Q}_n(11)$.*

Figure 3 shows the tilde-Fibonacci cube of order $1, 2, 3, 4$.

From Theorem 25 and Proposition 19, the following corollary derives.

▶ **Corollary 31.** *The tilde-Fibonacci cube $\tilde{F}_n = \tilde{Q}_n(11)$ is an isometric subgraph of $\tilde{Q}_n$.*

Further, about other hypercubes avoiding words of length 2, we have the following:

▶ Remark 32. For each $n \geq 1$, $\tilde{Q}_n(10)$ and $Q_n(10)$ coincide. In fact, $V(\tilde{Q}_n(10)) = \{0^h1^k | \ h, k \geq 0, \ \ h + k = n\} = V(Q_n(10))$ and $E(\tilde{Q}_n(10)) = \{(0^i1^j, 0^{i-1}1^{j+1}) | \ 1 \leq i \leq n, \ 0 \leq j \leq n-1, \ i + j = n\} = E(Q_n(10))$.

By complement, also $\tilde{Q}_n(01)$ and $Q_n(01)$ coincide (see Remark 22).

Note also that $\tilde{Q}_n(01)$ is obtained from $\tilde{Q}_n(10)$ by complementing all the bits in the vertices labels, i.e. they are isomorphic.

The tilde Fibonacci cube admits also a recursive construction that allows one to enumerate its edges.

By Proposition 2, $|V(\tilde{F}_n)| = |V(F_n)| = f_{n+2}$. Among these vertices, f_{n+1} end with a 0 and f_n end with a 1. Figure 3 shows the tilde-Fibonacci cube of order 4.

▶ Remark 33. Let $u \in V(F_{n-1})$, $x \in \Sigma$. If u ends with 1, then $ux \in V(\tilde{F}_n)$ iff $x = 0$. If u ends with 0 then $ux \in V(\tilde{F}_n)$, for any $x \in \{0, 1\}$.

▶ **Proposition 34.** *The n-tilde-Fibonacci cubes $\tilde{F}_n$, with $n \geq 1$, can be defined recursively.*

Proof. If $n = 1$, $\tilde{F}_1$ has two vertices 0 and 1 connected by an edge. If $n = 2$, $\tilde{F}_2$ has three vertices 00, 01 and 10 and $E(\tilde{F}_2) = \{(00, 10), (00, 01), (01, 10)\}$.

Let $n \geq 3$ and suppose that $\tilde{F}_i$ are defined for all $i < n$. Then, $\tilde{F}_n$ can be constructed from a copy of $\tilde{F}_{n-1}$ where each vertex u is replaced by $u0$, denoted by $\tilde{F}_{n-1}0$, and a copy of $\tilde{F}_{n-2}$, where each vertex v is replaced by $v01$, and denoted by $\tilde{F}_{n-2}01$. If there is an edge (u, v) in $\tilde{F}_{n-1}$, then there is en edge $(u0, v0)$ $\tilde{F}_n$, i.e. $\tilde{F}_{n-1}0$ is a subgraph of $\tilde{F}_n$. For similar reasons $\tilde{F}_{n-2}01$ is a subgraph of $\tilde{F}_n$. Further, for any $u0 \in V(\tilde{F}_{n-1})$, then there is an edge in

$\tilde{F}_n$ connecting $v00$ in $\tilde{F}_{n-1}0$ and $u01$ in $\tilde{F}_{n-2}01$ and for any $u1 \in V(\tilde{F}_{n-1})$ there is an edge linking $u10$ in $\tilde{F}_{n-1}0$ and $u01$ in $\tilde{F}_{n-2}01$ (see the orange edges in Fig. 3). By Remark 33 and Lemma 15 no further edges exist in $\tilde{F}_n$. ◀

▶ **Corollary 35.** *Let $\tilde{F}_n$ be the n-tilde-Fibonacci cube. Then, $|E(\tilde{F}_1)| = 1, |E(\tilde{F}_2)| = 3$ and, for any $n \geq 2$*

$$|E(\tilde{F}_n)| = |E(\tilde{F}_{n-1})| + |E(\tilde{F}_{n-2})| + f_{n+1}.$$

Hence, we can give the following exact formula:

$$|E(\tilde{F}_n)| = \frac{(n+1)f_{n+3} + (n-2)f_{n+1}}{5},$$

(Sequence A023610 in [31] for $|E(\tilde{F}_{n+1}|)$. Since the number of vertices of $\tilde{F}_n$ is f_{n+2}, from the previous formula it follows that the tilde-Fibonacci cube has $O(N \log N)$ edges, where N is the number of vertices, as for the tilde-hypercube (see Equation (1)). Let us conclude the section with some structural properties of tilde-Fibonacci cubes such as the diameter and the radius.
The *eccentricity* of a vertex v of a connected graph G is defined as

$$e(v) = \max_{u \in V(G)} d_G(u, v),$$

where $d_G(u, v)$ is the length of a shortest path from u to v in G. The *diameter* and the *radius* of G are respectively defined by

$$d(G) = \max_{v \in V(G)} e(v) \quad \text{and} \quad r(G) = \min_{v \in V(G)} e(v).$$

In [21] it is proven that $d(F_n) = n$ and that the maximal distance involves the words $(10)^{n/2}$ and $(01)^{n/2}$ for even n, and $(01)^{\lfloor n/2 \rfloor}0$ and $(10)^{\lfloor n/2 \rfloor}1$ for odd n. Note that, since the swap operation adds new edges, it shortens the distances between vertices. Moreover, the distances can even be halved because a swap replaces two replacement operations. More precisely, we have the following proposition.

▶ **Proposition 36.** *Let $\tilde{F}_n$ be the n-tilde-Fibonacci cube. Then, for any $n \geq 1$, $d(\tilde{F}_n) = r(\tilde{F}_n) = \lceil n/2 \rceil$.*

Proof. First, we prove that for any $n \geq 1$ and $u, v \in V(\tilde{F}_n)$, $dist_\sim(u, v) \leq \lceil n/2 \rceil$ by induction on n.
The case where $n = 1$ is trivial. If $n = 2$, then for any $u, v \neq 11$, we have $\text{dist}_\sim(u, v) = 1$. Let now u, v be 11-free words of same length n, with $n > 2$. Then, $u = xyu'$ and $v = ztv'$, with $x, y, z, t \in \{0, 1\}$ and $u', v' \in \{0, 1\}^*$. Recall that $\text{dist}_\sim(u, v)$ is the minimal number of swaps and replacements to transform u into v. Since xy can be transformed into zt with at most a single swap or replacement, a possible way to transform u in v is to transform xy into zt and then u' into v'. Therefore, $\text{dist}_\sim(u, v) \leq 1 + \text{dist}_\sim(u', v') \leq 1 + \lceil (n-2)/2 \rceil \leq \lceil n/2 \rceil$.

Moreover, for each 11-free word u of length n, with $n \geq 1$, there exists a 11-free word v of length n such that $dist_\sim(u, v) = \lceil n/2 \rceil$. In fact, for each word u of length n, the word v is obtained by replacing in u, from left to right, the blocks 00 with 10, the blocks 10 with 00, the blocks 01 with 10 and finally, for odd n, the last bit with its complement. Trivially, if u is 11-free then v is 11-free, as well. We prove that $\text{dist}_\sim(u, v) = \lceil n/2 \rceil$ by induction on n. The case $n = 1$ is trivial. If $n = 2$ then $\text{dist}_\sim(01, 10) = \text{dist}_\sim(10, 00) = \text{dist}_\sim(00, 10) = 1$. If

$n > 2$ then either $u = 00u'$ ($v = 10v'$) or $u = 01u'$ ($v = 10v'$). In the first case, by Remark 11, one has $dist_{\sim}(u, v) = 1 + \lceil (n-2)/2 \rceil = \lceil n/2 \rceil$. In the second case, by Remark 12, one has $dist_{\sim}(u, v) = 1 + \lceil (n-2)/2 \rceil = \lceil n/2 \rceil$. In any case, $dist_{\sim}(u, v) = \lceil n/2 \rceil$. Finally, since $\tilde{F}_n$ is an isometric subgraph of $\tilde{Q}_n$ then $d_{\tilde{F}_n}(u, v) = dist_{\sim}(u, v)$ and $r(\tilde{F}_n) = d(\tilde{F}_n) = \lceil n/2 \rceil$. ◀

The previous proposition proves that $\tilde{F}_n$ is a *self-centered* graph, that is, a graph in which radius and diameter coincide (cf. [14] for a survey).

7 Conclusions

The paper surveys some recent results on isometric words with respect to the edit distance that allows swap and replacement operations, here referred to as tilde-distance. Moreover, the tilde-hypercube and the tilde-Fibonacci cube are presented as a generalization of the corresponding classical notions, with the tilde-distance in place of the Hamming distance.

Compared with the setting of Hamming distance, all the problems appear to be more complicated since, when using swaps, the order of performing the operations does matter. Nevertheless, isometric words and generalized Fibonacci cubes based on this tilde-distance open up new scenarios and present interesting new situations that surely deserve further investigation as it can serve as base for string and graph algorithmic developments.

References

1 Amihood Amir, Estrella Eisenberg, and Ely Porat. Swap and mismatch edit distance. *Algorithmica*, 45(1):109–120, 2006. doi:10.1007/978-3-540-30140-0_4.

2 Marcella Anselmo, Giuseppa Castiglione, Manuela Flores, Dora Giammarresi, Maria Madonia, and Sabrina Mantaci. Hypercubes and isometric words based on swap and mismatch distance. In *Descriptional Complexity of Formal Systems. DCFS23*, volume 13918 of *Lect. Notes Comput. Sci.*, pages 21–35. Springer Nature, 2023. doi:10.1007/978-3-031-34326-1_2.

3 Marcella Anselmo, Giuseppa Castiglione, Manuela Flores, Dora Giammarresi, Maria Madonia, and Sabrina Mantaci. Isometric words based on swap and mismatch distance. In *Developments in Language Theory. DLT23*, volume 13911 of *Lect. Notes Comput. Sci.*, pages 23–35. Springer Nature, 2023. doi:10.1007/978-3-031-33264-7_3.

4 Marcella Anselmo, Giuseppa Castiglione, Manuela Flores, Dora Giammarresi, Maria Madonia, and Sabrina Mantaci. Isometric sets of words and generalizations of the Fibonacci cubes. In *Computability in Europe. CIE24*, volume LNCS 14773 of *Lect. Notes Comput. Sci.*, pages 1–14. Springer, 2024. doi:10.1007/978-3-031-64309-5_35.

5 Marcella Anselmo, Giuseppa Castiglione, Manuela Flores, Dora Giammarresi, Maria Madonia, and Sabrina Mantaci. Characterization of isometric words based on swap and mismatch distance. *International Journal of Foundations of Computer Science*, 36(03):221–245, 2025. doi:10.1142/S0129054125430051.

6 Marcella Anselmo, Manuela Flores, and Maria Madonia. Fun slot machines and transformations of words avoiding factors. In *Fun with Algorithms*, volume 226 of *LIPIcs*, pages 4:1–4:15, 2022. doi:10.4230/LIPIcs.FUN.2022.4.

7 Marcella Anselmo, Manuela Flores, and Maria Madonia. On k-ary n-cubes and isometric words. *Theor. Comput. Sci.*, 938:50–64, 2022. doi:10.1016/j.tcs.2022.10.007.

8 Marcella Anselmo, Manuela Flores, and Maria Madonia. Density of Ham- and Lee- non-isometric k-ary words. In *ICTCS'23 Italian Conference on Theor. Comput. Sci.*, volume 3587 of *CEUR Workshop Proceedings*, pages 116–128, 2023. URL: https://ceur-ws.org/Vol-3587/3914.pdf.

9 Marcella Anselmo, Manuela Flores, and Maria Madonia. Computing the index of non-isometric k-ary words with Hamming and Lee distance. *Computability, IOS press*, 13(3-4):199–222, 2024. doi:10.3233/COM-230441.

10 Marcella Anselmo, Manuela Flores, and Maria Madonia. Density of k-ary words with 0, 1, 2 - error overlaps. *Theor. Comput. Sci.*, 1025:114958, 2025. doi:10.1016/j.tcs.2024.114958.

11 Jernej Azarija, Sandi Klavžar, Jaehun Lee, Jay Pantone, and Yoomi Rho. On isomorphism classes of generalized Fibonacci cubes. *Eur. J. Comb.*, 51:372–379, 2016. doi:10.1016/j.ejc.2015.05.011.

12 Marie-Pierre Béal and Maxime Crochemore. Checking whether a word is Hamming-isometric in linear time. *Theor. Comput. Sci.*, 933:55–59, 2022. doi:10.1016/j.tcs.2022.08.032.

13 Laxmi N. Bhuyan and Dharma P. Agrawal. Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on Computers*, C-33(4):323–333, 1984. doi:10.1109/TC.1984.1676437.

14 Fred Buckley. Self-centered graphs. *Ann. New York Acad. Sci*, 576:71–78, 1989. doi:10.1111/j.1749-6632.1989.tb16384.x.

15 Sergio Cabello, David Eppstein, and Sandi Klavzar. The Fibonacci dimension of a graph. *Electron. J. Comb.*, 18(1), 2011. doi:10.37236/542.

16 Giuseppa Castiglione, Manuela Flores, and Dora Giammarresi. Isometric words and edit distance: Main notions and new variations. In *Cellular Automata and Discrete Complex Systems. AUTOMATA 2023*, volume 14152 of *Lect. Notes Comput. Sci.*, pages 6–13. Springer, 2023. doi:10.1007/978-3-031-42250-8_1.

17 W.J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, 1990. doi:10.1109/12.53599.

18 Ahmed El-Amawy and Shaharam Latifi. Properties and performance of folded hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):31–42, 1991. doi:10.1109/71.80187.

19 Simone Faro and Arianna Pavone. An efficient skip-search approach to swap matching. *Comput. J.*, 61(9):1351–1360, 2018. doi:10.1093/comjnl/bxx123.

20 Frank Harary, John P. Hayes, and Horng J. Wu. A survey of the theory of hypercube graphs. *Comput. Math. Appl.*, 15(4):277–289, 1988. doi:10.1016/0898-1221(88)90213-1.

21 Wen-Jing Hsu. Fibonacci cubes-a new interconnection topology. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):3–12, 1993. doi:10.1109/71.205649.

22 Aleksandar Ilić, Sandi Klavžar, and Yoomi Rho. Generalized Fibonacci cubes. *Discrete Math.*, 312(1):2–11, 2012. doi:10.1016/j.disc.2011.02.015.

23 Aleksandar Ilić, Sandi Klavžar, and Yoomi Rho. The index of a binary word. *Theor. Comput. Sci.*, 452:100–106, 2012. doi:10.1016/j.tcs.2012.05.025.

24 Sandi Klavžar. Structure of Fibonacci cubes: A survey. *J. Comb. Optim.*, 25(4):505–522, 2013. doi:10.1007/s10878-011-9433-z.

25 Sandi Klavžar and Sergey V. Shpectorov. Asymptotic number of isometric generalized Fibonacci cubes. *Eur. J. Comb.*, 33(2):220–226, 2012. doi:10.1016/j.ejc.2011.10.001.

26 Sandi Klavžar. On median nature and enumerative properties of Fibonacci-like cubes. *Discrete Mathematics*, 299(1):145–153, 2005. doi:10.1016/j.disc.2004.02.023.

27 Sandi Klavžar and Petra Žigert. Fibonacci cubes are the resonance graphs of fibonaccenes. *The Fibonacci Quarterly*, 43(3):269–276, 2005. doi:10.1080/00150517.2005.12428368.

28 Emanuele Munarini and Norma Zagaglia Salvi. Structural and enumerative properties of the Fibonacci cubes. *Discrete Math.*, 255(1-3):317–324, 2002. doi:10.1016/S0012-365X(01)00407-1.

29 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

30 P. Tolstrup Nielsen. A note on bifix-free sequences (corresp.). *IEEE Transactions on Information Theory*, 19(5):704–706, 1973. doi:10.1109/TIT.1973.1055065.

31 N.J.A. Sloane. On-line encyclopedia of integer sequences. URL: http://oeis.org/.

32 Nian-Feng Tzeng and Sizheng Wei. Enhanced hypercubes. *IEEE Trans. Computers*, 40(3):284–294, 1991. doi:10.1109/12.76405.

33 Robert A. Wagner. On the complexity of the extended string-to-string correction problem. In *Symposium on Theory of Computing*, pages 218–223, 1975. doi:10.1145/800116.803771.

34 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

35 Jianxin Wei. The structures of bad words. *Eur. J. Comb.*, 59:204–214, 2017. doi:10.1016/j.ejc.2016.05.003.

36 Jianxin Wei. Proof of a conjecture on 2-isometric words. *Theor. Comput. Sci.*, 855:68–73, 2021. doi:10.1016/j.tcs.2020.11.026.

37 Jianxin Wei, Yujun Yang, and Guangfu Wang. Circular embeddability of isometric words. *Discret. Math.*, 343(10):112024, 2020. doi:10.1016/j.disc.2020.112024.

38 Jianxin Wei, Yujun Yang, and Xuena Zhu. A characterization of non-isometric binary words. *Eur. J. Comb.*, 78:121–133, 2019. doi:10.1016/j.ejc.2019.02.001.

39 Jianxin Wei and Heping Zhang. Proofs of two conjectures on generalized Fibonacci cubes. *Eur. J. Comb.*, 51:419–432, 2016. doi:10.1016/j.ejc.2015.07.018.

Compact Data Structures for Collections of Sets

Jarno N. Alanko
Department of Computer Science, University of Helsinki, Finland

Philip Bille
Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
Technical University of Denmark, Lyngby, Denmark

Gonzalo Navarro
Department of Computer Science, University of Chile, Santiago, Chile
Center for Biotechnology and Bioengineering (CeBiB), Santiago, Chile

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland

Abstract

We define a new entropy measure $L(\mathcal{S})$, called the *containment entropy*, for a set $\mathcal{S}$ of sets, which considers the fact that some sets can be contained in others. We show how to represent $\mathcal{S}$ within space close to $L(\mathcal{S})$ so that any element of any set can be retrieved in logarithmic time. We extend the result to predecessor and successor queries and show how some common set operations can be implemented efficiently.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Compressed data structures, entropy of sets, data compression

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.6

Category Research

Funding *Jarno N. Alanko*: Funded by the Helsinki Institute for Information Technology (HIIT).
Philip Bille: Danish Research Council grant DFF-8021-002498.
Inge Li Gørtz: Danish Research Council grant DFF-8021-002498.
Gonzalo Navarro: Basal Funds FB0001 and AFB240001, ANID, Chile.
Simon J. Puglisi: Academy of Finland grant 339070.

Acknowledgements This work was initiated at the NII Shonan Meeting no. 187 on "Theoretical Foundations of Nonvolatile Memory."

1 Introduction

We consider the problem of representing a collection of sets $\mathcal{S} = \{S_1, \ldots, S_s\}$ from a universe $\mathcal{U}$ of size u while supporting basic queries, including retrieving the kth element and predecessor and successor queries. The goal is to store the sets compactly while supporting fast queries. This problem has important applications, including the representation of postings lists in inverted indexes and adjacency-list representations of graphs.

To measure space, we often consider the *worst-case entropy* defined as $H(\mathcal{S}) = \sum_{i=1}^{s} \lg \binom{u}{|S_i|}$ as a natural information-theoretic worst-case lower bound on the number of bits needed to store $\mathcal{S}$. Using standard techniques [6, 5, 9], we can store $\mathcal{S}$ within $H(\mathcal{S}) + O(n + s \log n)$ bits and support retrieval (i.e., accessing any kth element of any set) in constant time.

In this paper, we propose a finer measure of entropy for $\mathcal{S}$ that can take advantage of the fact that some sets may be subsets of others. If $S_i \subset S_j$, we can encode S_i using $\log \binom{|S_j|}{|S_i|}$ bits, by indicating which elements in S_j are also in S_i. We show how to construct a hierarchy from $\mathcal{S}$ such that children are subsets of their parent. This leads to a new notion of entropy called the *containment entropy*, defined as

$$L(\mathcal{S}) = \sum_{i=1}^{s} \lg \binom{|p(S_i)|}{|S_i|},$$

where $p(S_i)$ is the parent of S_i (see Section 3). It is easy to see that the containment entropy is a finer notion of entropy since $L(\mathcal{S}) \leq H(\mathcal{S})$. Our main result is that we efficiently represent $\mathcal{S}$ in space close to its containment entropy while supporting queries efficiently.

▶ **Theorem 1.** *Let $\mathcal{S}$ be a set of s sets of total size n, elements of which are drawn from a universe of size u. We can construct a data structure in $O(sn \log n)$ time that uses $L(\mathcal{S}) + O(n + s \log n)$ bits of space and supports retrieval, predecessor, and successor queries on any set $S \in \mathcal{S}$ in time $O(\log(u/|S|))$.*

Thus, compared to the above-mentioned standard data structures, we replace the worst-case entropy $H(\mathcal{S})$ with $L(\mathcal{S})$ in our space bound in Theorem 1.

We also obtain several corollaries of Theorem 1. We show how to implement the common operations of set intersection, set union, and set difference. By combining Theorem 1 with techniques from Barbay and Kenyon [2] we obtain fast implementations of these operations in terms of the *alternation bound* [2] directly on our representation. We also show how to apply Theorem 1 to efficiently store a collection of bitvectors while supporting access, rank, and select queries.

Technically, the result is obtained by constructing a tree structure, where each node represents a set, and children represent subsets of their parent. We then represent each set as a sparse bitvector that indicates which of the elements of the parent are present in the child. This leads to a simple representation that achieves the desired space bound. Unfortunately, if we directly implement a retrieval query on this representation, we have to traverse the path from the set S to the root leading to a query time of $\Omega(h)$, where h is the height of the tree. We show how to efficiently *shortcut* paths in the tree without asymptotically increasing the space, yielding the $O(\log(u/|S|))$ time. We then extend these techniques to handle predecessor and successor queries in the same time, leading to the full result of Theorem 1.

Relation with wavelet trees. We note that the idea is reminiscent of wavelet trees [7], which represent a sequence of symbols as a binary tree where each node stores a bitvector. In the root's bitvector, 0s/1s mark the sequence positions with symbols in the first/second half of the alphabet, and left/right children recursively represent the subsequences formed by the symbols in the first/second half of the alphabet. Nodes handling single-symbol subsequences are wavelet tree leaves. If we define $\mathcal{S}$ as the sets of positions in the sequence holding the symbols of each subalphabet considered in the wavelet tree, then our data structure built on $\mathcal{S}$ has the same shape of the wavelet tree. The wavelet tree uses less space, however, because it corresponds to a particular case where the maximal subsets of each set partition it into two: while the wavelet tree can be compressed to the zero-order entropy of the sequence [7], $L(\mathcal{S})$ doubles that entropy.

2 Basic Concepts

A *bitvector* $B[1..u]$ is a data structure that represents a sequence of u bits and supports the following operations:

- $access(B, i)$ returns $B[i]$, the ith bit of B.
- $rank_b(B, i)$, where $b \in \{0, 1\}$, returns the number of times bit b occurs in the prefix $B[1..i]$ of the bitvector. We assume $rank_b(B, 0) = 0$.
- $select_b(B, j)$ returns the position in B of the jth occurrence of bit $b \in \{0, 1\}$. We assume $select_b(B, 0) = 0$ and $select_b(B, i) = n + 1$ if b does not occur i times in B.

Note that $rank_0(B, i) = i - rank_1(B, i)$. By default we assume $b = 1$. It is possible to represent bitvector B using $u + o(u)$ bits and solve all the operations in $O(1)$ time [3, 8]. If n is the number of 1s in B, it is possible [6, 5, 9] to represent B using

$$n \lg \frac{u}{n} + 2n \quad = \quad \lg \binom{u}{n} + O(n + \log u)$$

bits,[1] so that $select_1$ is implemented in constant time, while $access$ and $rank_b$ are solved in time $O(1 + \log \frac{u}{n})$. Operation $select_0$ can be solved in time $O(\log n)$ by binary search on $select_1$. This is called the *sparse bitvector representation* in this paper.

We note that $\lg \binom{u}{n}$ is the entropy of the set of positions where B contains n 1s, or equivalently, the entropy of the sets of n elements in a universe of size u. We will indistinctly speak of the operations $access$, $rank$, and $select$ over sets S on the universe $[1..u]$, referring to the corresponding bitvector B where $B[i] = 1$ iff $i \in S$. For example, $select(S, k)$ is the kth smallest element of S.

3 Containment Entropy

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$ be a set of s distinct sets, and $\mathcal{U} = S_1 \cup S_2 \cup \cdots \cup S_s$ be their union. For simplicity we assume $\mathcal{U} = [1..u]$, so $u = |\mathcal{U}|$. Let $n_i = |S_i|$ and $n = \sum_{i=1}^{s} n_i$ be the total size of all sets; note $n \geq u$.

A simple notion of entropy for $\mathcal{S}$ is its *worst-case entropy*

$$H(\mathcal{S}) \quad = \quad \sum_{i=1}^{s} \lg \binom{u}{n_i}.$$

It is not hard to store $\mathcal{S}$ within $H(\mathcal{S}) + O(n + s \log n)$ bits, by using the sparse bitvector representation of Section 2, which stores each S_i within $\lg \binom{u}{n_i} + O(n_i + \log u)$ bits, and offers constant-time retrieval of any element of S_i via $select(S_i, k)$. We use $O(s \log n)$ additional bits to address the representations of the s sets.

We now propose a finer measure of entropy for $\mathcal{S}$, which exploits the fact that some sets can be subsets of others. If $S_i \subset S_j$, we can describe S_i using $\lg \binom{n_j}{n_i}$ bits, which indicate which elements of S_j are also in S_i. We define a tree structure whose nodes are the sets S_i, and the parent of S_i, $p(S_i)$, is any *smallest* S_j such that $S_i \subset S_j$. If S_i is not contained in any other set, then its parent is the tree root, which represents $\mathcal{U}$. We then define the following measure of entropy, which we call the *containment entropy*.

[1] The formula on the left is zero if $n = 0$.

Algorithm 1 Retrieving the kth element of a set S in a hierarchy.

Input : Set S and rank k of the element to be retrieved, with $1 \leq k \leq |S|$.
Output : The kth smallest element in S.

```
1 function retrieve(S, k)
2     if S is the root then return k
3     else return retrieve(p(S), select(S, k)) // use c(S) on contracted hierarchy
```

▶ **Definition 2.** *Let $\mathcal{S}$ be a set of sets S_i. Its* hierarchy *has $\mathcal{U} = \cup_i S_i$ at the root, and the parent $p(S_i)$ of S_i is any smallest set containing S_i (or $\mathcal{U}$ if no such set exists). Let $p_i = |p(S_i)|$. The* containment entropy *of $\mathcal{S}$ is*

$$L(\mathcal{S}) \quad = \quad \sum_{i=1}^{s} \lg \binom{p_i}{n_i}.$$

Clearly $L(\mathcal{S}) \leq H(\mathcal{S})$ because $p_i \leq u$ for every i. Note that $L(\mathcal{S})$ is arguably the optimal space we can achieve by representing sets as subsets of others in $\mathcal{S}$, but we could obtain less space if other arrangements were possible. For example, if many sets are subsets of both S_i and S_j, it could be convenient to introduce a new set $S_i \cap S_j$ in the hierarchy, even if it is not in $\mathcal{S}$. An advantage of $L(\mathcal{S})$ is that it is easily computed in polynomial time from $\mathcal{S}$, whereas more general notions like the one that allows the creation of new sets may be not.

4 A Containment Entropy Bounded Representation with Fast Retrieval

It is not hard to represent $\mathcal{S}$ within space close to $L(\mathcal{S})$: we can use the sparse bitvector representation of Section 2 to store each S_i relative to its parent $p(S_i)$, within $\lg \binom{p_i}{n_i} + O(n_i + \log p_i)$ bits, which add up to $L(\mathcal{S}) + O(n + s \log u)$ bits. The problem is that now $select(S_i, k)$ gives the position of the kth element of S_i within those of $p(S_i)$, not within $\mathcal{U}$, and thus $select(S_i, k)$ is not directly the identity of the kth element of S_i. In order to obtain the identity of the element, we must compute $select(p(S_i), select(S_i, k))$ and so on, consecutively following the chain of ancestors of S_i until the root $\mathcal{U}$; see Algorithm 1. This may take time proportional to the height of the hierarchy, which can be up to $O(s)$. We now show how to reduce this time to logarithmic, by introducing *shortcuts* in the hierarchy.

▶ **Definition 3.** *The* contracted hierarchy *of $\mathcal{S}$ has $\mathcal{U}$ as its root, and the parent $c(S_i)$ of S_i is the highest ancestor S_t of $p(S_i)$ in the hierarchy of $\mathcal{S}$ such that $n_t \leq 2n_i$. If no such ancestor exists, then $c(S_i) = p(S_i)$.*

We now prove a couple of relevant properties of this contracted hierarchy.

▶ **Lemma 4.** *The depth of node S_i in the contracted hierarchy of $\mathcal{S}$ is $O(\log(u/n_i))$. The height of the contracted hierarchy is $O(\log u)$.*

Proof. By definition, the grandparent of node S_i, if it exists, has size $> 2n_i$; thus the path towards the root (whose size is u) cannot be longer than $2 \lg(u/n_i)$. An obvious consequence is that the height of the tree cannot be longer than $2 \lg u$. ◀

If we change $p(S)$ to $c(S)$ in Algorithm 1, then, the retrieval time becomes $O(\log(u/|S|))$. We now show that the space is not asymptotically affected by contracting the hierarchy.

▶ **Lemma 5.** *The contracted hierarchy of $\mathcal{S}$ can be represented within $L(\mathcal{S}) + O(n + s \log n)$ bits.*

Algorithm 2 Finding the position of the predecessor of k in the set S of a hierarchy.

Input : Set S and element k of the universe, $1 \leq k \leq u$.
Output : The position of the predecessor of k in S.

```
1 function predecessor(S, k)
2     if S is the root then return k
3     else return rank(S, predecessor(c(S), k))
```

Proof. We start from our representation that uses $L(\mathcal{S})+O(n+s\log n)$ bits, and show how it changes when we replace $p(S_i)$ by $c(S_i)$. In case $p(S_i) \neq c(S_i)$, it holds that $|c(S_i)| < 2|p(S_i)|$, because $S_i \subset p(S_i)$ and thus $|c(S_i)| \leq 2|S_i| < 2|p(S_i)|$. Then, changing $p(S_i)$ to $c(S_i)$ increases the size of the representation from $n_i \lg \frac{p_i}{n_i} + 2n_i$ to at most $n_i \lg \frac{2p_i}{n_i} + 2n_i = n_i \lg \frac{p_i}{n_i} + 3n_i$. Thus, the total increase produced by changing all $p(S_i)$ to $c(S_i)$ is bounded by n; we still use $L(\mathcal{S}) + O(n + s\log n)$ bits. The parent pointers that allow climbing paths also fit in $O(s\log n)$ bits. ◀

In summary, we have the following result.

▶ **Lemma 6.** *A set $\mathcal{S}$ of s sets of total size n and universe size u can be represented within $L(\mathcal{S}) + O(n + s\log n)$ bits so that any element of any set S can be retrieved in time $O(\log(u/|S|))$.*

5 Other Operations

5.1 Predecessor and Successor in a Set

Given an element identifier k and a set S_i, the *predecessor* is the largest $v \leq k$ such that $v \in S_i$. Conversely, the *successor* is the smallest $v \geq k$ such that $v \in S_i$. We can find the predecessor in $O(\log u)$ time, as follows. We start at the node S_i and walk up the path to the root $\mathcal{U}$, so as to determine the nodes in the path. In the return from the recursion, we start at the position $v \leftarrow k$ in $\mathcal{U}$, and whenever returning from a node S to its child S', we compute $v \leftarrow rank(S', v)$, which gives the number of elements from S' up to the element v in S, that is, the predecessor of v among the elements of S'. By the time we return to S_i again, v is the position in S_i of the predecessor of k; see Algorithm 2. We then find out the identity of v in $\mathcal{U}$ using Algorithm 1. To find the successor, we use $v \leftarrow 1 + rank(S', v-1)$ instead.

Operation *rank* takes time $O(1 + \log(|S|/|S'|))$ in the sparse bitvector representation of S', as seen in Section 2. Therefore, the sum of the times along the path to S_i telescopes to $O(\log(u/n_i))$. This yields the following result.

▶ **Lemma 7.** *On the same representation of Lemma 6, we can compute the predecessor and successor of any element of any set S in time $O(\log(u/|S|))$.*

5.2 Set Operations

These operations are useful to compute set operations, by mimicking any standard algorithm that traverses both ordered sets. In particular, we can implement an intersection algorithm that is close to the alternation complexity lower bound δ [2]: any algorithm that can find the successor of any element in either list in time t, can intersect both lists in time $O(\delta t)$. Predecessor and successor on the complement of a set can also be solved in time $O(t)$, by using $rank_0$ instead of *rank* (and managing the base case accordingly).

▶ **Corollary 8.** *On the representation of Lemma 6, we can compute the intersection between any two sets S_i and S_j in time $O(\delta \log u)$, where δ is the alternation complexity of both sets. We can also compute their union in time $O(|S_i \cup S_j| \log u)$. The difference $S_i \setminus S_j$ can be implemented in time $O(\delta' \log u)$, where δ' is the alternation complexity of S_i and S_j^c.*

5.3 Back to Bitvectors

Returning to the bitvector semantics, our results allow us store a set of sparse bitvectors $B_i[1..u]$, representing subsets S_i of a universe $\mathcal{U}$ of size u, so that operations $access(B_i, k)$, $rank(B_i, k)$ and $select(B_i, k)$ take time $O(\log(u/n_i))$. To implement $access(B_i, k)$, we return 1 if the predecessor of k in S_i is k, or else 0. To implement $rank(B_i, k)$, we compute the position v of the predecessor of k in the ordered set S_i and return $rank(S_i, v)$. To implement $select_1(B_i, k)$, we retrieve the kth element of S_i, and for $select_0$ we do binary search on $select_1$.

▶ **Corollary 9.** *A set of s bitvectors $B_i[1..u]$ can be represented in $L(\mathcal{S}) + O(n + s \log u)$ bits, where $\mathcal{S}$ is the set of sets $S_i = \{k,\ B_i[k] = 1\}$, n_i is the number of 1s in B_i and $n = \sum_i n_i$. This representation supports operations access, $rank_b$, and $select_1$ on any B_i in time $O(\log(u/n_i))$, and $select_0$ in time $O(\log n_i \log(u/n_i))$.*

This is to be compared with storing each bitvector directly [9], which gives total space $H(\mathcal{S}) + O(n + s \log u)$ and supports *access* and $rank_b$ in the same time $O(\log(u/n_i))$, and $select_b$ in the better times $O(1)$ for $b = 1$ and $O(\log n_i)$ for $b = 0$.

6 Construction

We can build the hierarchical representation by adding one set S at a time, *in decreasing order of size*, to ensure it is correct to insert S as a leaf. To insert S in $\mathcal{S}$, we first find a smallest set $S' \in \mathcal{S}$ such that $S \subset S'$. The sets that contain S form a connected subgraph of the hierarchy that includes the root. So we can traverse the hierarchy from the root, looking for the lowest nodes that contain S, and retain the smallest of those. For each hierarchy node S_i to check, we take each element of S and verify that it exists in S_i via a predecessor query, which takes time $O(\log(u/|S_i|)) \subseteq O(\log(u/|S|))$, because $|S| \leq |S_i|$. We then find a smallest set S' containing S in time $O(s'|S| \log(u/|S|))$, where $s' \leq n/|S|$ is the current number of sets in $\mathcal{S}$ and n is its final sum of set sizes.

We then insert S in the hierarchy by setting $p(S) = S'$. To find $c(S)$, we find the highest ancestor S'' of S' whose size is $|S''| \leq 2|S|$ (or let $S'' = S'$ if no such ancestor exists), and set $c(S) = S''$. Finally, we build the representation of S relative to S'', in time $O(|S|)$ [9].

Note that, when we find the ancestor S'', we traverse the upward path defined by the parent function $p(\cdot)$, not $c(\cdot)$. We still use $c(\cdot)$ during construction to answer the predecessor queries, to determine inclusion of S, in logarithmic time.

▶ **Lemma 10.** *The representation of Lemma 6 can be built in time $O(sn \log n)$.*

Combining Lemmas 6, 7, and 10 we have shown Theorem 1.

7 Concluding Remarks

We have described the containment entropy, a new entropy measure for collections of sets, which is sensitive to subset relationships between sets. To our knowledge, this idea has not before been considered in the vast literature on efficient set representations that has emerged

primarily from efficiency concerns in information retrieval systems [11]. One could consider dictionary-based compression of sets (see, e.g., [10, 4]) as implicitly capturing some aspect of subset relationships, but the representations and analysis we have described here consider subset relationships explicitly.

An interesting direction for future work is to explore the practicality of these hierarchical set representations in various application contexts. An immediate concern is that of hierarchy construction. Our initial experiments with a collection of sets taken from the genomic search engine Themisto [1] applied to a set of 16 thousand bacterial genomes (over 80GB of data) indicate that, even for large set collections, hierarchy construction is tractable and scales almost linearly with the total length of the sets in practice. On a collection of 10.55 million sets of average size 3,607 (10.52 million of which were subsets of at least one other set) of more than 38 billion elements in total, we were able to find the smallest super set of every set in less than 40 hours in total, using just 16 threads of execution. The resulting representation used just 0.18 bits per element, compared to the 0.32 bits per element used by Themisto's representation, which selects between different set representations based on set density.

References

1 Jarno N. Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J. Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(Supplement-1):260–269, 2023. doi:10.1093/BIOINFORMATICS/BTAD233.

2 J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1):1–18, 2008. doi:10.1145/1328911.1328915.

3 D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.

4 F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):article 16, 2010.

5 P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974. doi:10.1145/321812.321820.

6 R. Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts, 1971.

7 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

8 J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

9 D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

10 Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. Fast dictionary-based compression for inverted indexes. In *Proc. 12th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 6–14. ACM, 2019. doi:10.1145/3289600.3290962.

11 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):125:1–125:36, 2021. doi:10.1145/3415148.

Conditional Lower Bounds for String Matching in Labelled Graphs

Massimo Equi
Aalto University, Finland

Abstract

The problem of *String Matching in Labelled Graphs (SMLG)* is one possible generalization of the classic problem of finding a string inside another of greater length. In its most general form, SMLG asks to find a match for a string into a graph, which can be directed or undirected. As for string matching, many different variations are possible. For example, the match could be exact or approximate, and the match could lie on a path or a walk. Some of these variations easily fall into the NP-hard realm, while other variants are solvable in polynomial time. For the latter ones, fine-grained complexity has been a game changer in proving quadratic conditional lower bounds, allowing to finally close the gap with those upper bounds that remained unmatched for almost two decades.

If the match is allowed to be approximate, SMLG enjoys the same conditional quadratic lower bounds shown for example for edit distance (Backurs and Indyk, STOC '15). The case that really requires ad hoc conditional lower bounds is the one of finding an *exact* match that lies on a *walk*. In this work, we focus on explaining various conditional lower bounds for this version of SMLG, with the goal of giving an overall perspective that could help understand which aspects of the problem make it quadratic. We will introduce the reader to the field of fine-grained complexity and show how it can successfully provide the exact type of lower bounds needed for polynomial problems such as SMLG.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Problems, reductions and completeness

Keywords and phrases conditional lower bounds, strong exponential time hypothesis, fine-grained complexity, string matching, graphs

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.7

Category Research

Funding This work was supported in part by the Research Council of Finland, Grant 359104.

1 Introduction

The classic problem of string matching consists in finding a match for a shorter pattern string P into a longer text string T. This problem has been extensively studied throughout the years, but the core fundamental complexity result was already discovered in the '70s, when the problem was proven to be solvable in linear time [19]. The problem of *String Matching in Labelled Graphs* (SMLG) is a generalization of the string matching problem, where we look for matches of P not in a text but in a graph with nodes labelled by single characters. The SMLG problem first received attention in the '90s [20, 3, 4] due to potential applications in network searches and, later on, applications such as bioinformatics [15], graph databases [5] and heterogeneous networks [24] motivated it anew. Finding quadratic upper bounds of the form $O(|E||P|)$ was possible for different variations of the problem [4, 21, 25], and those results were later proven to be optimal under some complexity hypotheses for other problems [10, 11]. This conditional lower bounds falls into the field of the so-called

fine-grained complexity, and in this work we aim to showcase how fine-grained complexity lower bounds are achieved and why they changed the game for polynomial problems, in particular for SMLG.

The journey of fine-grained complexity begins in the early 2000s, when Impagliazzo, Paturi and Zane [17, 18] put forward a hypothesis on the complexity of SAT in conjunctive normal form (CNF-SAT). The hypothesis was named *Strong Exponential Time Hypothesis* (SETH), and states that no algorithm can solve a CNF-SAT instance over n variables in time $O(2^{\alpha n})$, where $\alpha < 1$. This hypothesis still holds its ground, since no one has been able to disprove it as of the writing of this work. A few years later, Williams [26] showed how SETH can be used to prove conditional lower bounds for polynomial problems. Among others, this work presented a fine-grained reduction from CNF-SAT to the *Orthogonal Vectors* (OV) problem, that is a reduction that requires subexponential time, which in this case was $O(2^{\frac{n}{2}})$[1]. In the OV problem, we are given sets X and Y of n binary vectors of dimension d, and we are asked to find a pair (x, y) of orthogonal vectors $x \in X$ and $y \in Y$. So far, no algorithm was able to solve OV in $O(n^{2-\varepsilon}\text{poly}(d))$ time, and this fact is referred to as the *Orthogonal Vectors Conjecture* (OVC). Thus, the reduction between CNF-SAT and OV shows that a $O(n^{2-\varepsilon}\text{poly}(d))$ time algorithm for OV would disprove SETH, or SETH $\Rightarrow$ OVC, and this is an important connection between exponential and polynomial complexities. Indeed, this allows to prove SETH-based lower bounds by reducing from OV instead of CNF-SAT, a choice that usually makes reductions cleaner and the lower bounds more reliable as, in principle, OVC might be true even if SETH fails.

After this initial connection, many other lower bounds for polynomial problems conditioned on SETH have been shown. Among these, one of the most celebrated examples is probably the lower bound for the problem of computing *edit distance* (EDIT) between two strings A and B [6]. The textbook algorithm for EDIT solves the problem in $O(n^2)$ time, where $n = |A| = |B|$, and has been known for decades, but the first lower bound was achieved only thanks to a fine-grained reduction from OV. This brings us back to SMLG, as the conditional lower bound for EDIT has consequences also for matching strings in graphs. Indeed, there are different variations of SMLG, and one thing to specify is whether the matches for pattern P in graph G should be exact or approximate, and whether walks are admitted (same nodes can be matched multiple times) or only paths (every node must be matched ones). To allow for approximate matches, one has to specify what an approximate match is. We could be very permissive, and allowing edit operations to occur both in the pattern and in the graph. This possibility was explored and proven to be an NP-hard problem [4]. If we then ask for a path spelling a string at minimum edit distance from P, then we incur in the following problem: if we label every node with character a and we query for pattern of length $|P| = |V|$, then we are solving the Hamiltonian Path problem, known to be NP-hard. Thus, let us look for a *walk* spelling a path at minimum edit distance from P. In this setting, an $O(|E||P|)$ algorithm exists [21, 25], and since a string can be viewed as a chain of nodes, EDIT is a special case of this problem, and thus the conditional quadratic lower bound matches the upper bound.

As previously mentioned, the version of SMLG where we ask for a walk in G spelling an exact match for P has also received a conditional quadratic lower bound [10, 11]. This result however could not be achieved as a simple adaptation of other lower bounds, and needed a

[1] We acknowledge that this use of the term "subexponential" is improper, as a real subexponential complexity should be of the form $O(2^{o(n)})$. Nonetheless, we will use this wording for lack of a better term.

custom fine-grained reduction from OV. Moreover, this highlights also a remarkable difference between the text and graph case: while the problem is linear for texts, it is quadratic for graphs, while in the approximate case it is quadratic for both texts and graphs.

At this point, one natural approach to find upper bounds could be to change the setting to try to play around the lower bound. For example, one could ask if queries to the pattern could be solved efficiently, that is in subquadratic time, after spending quadratic or even polynomial time indexing the graph. Unfortunately, also this question was answered negatively, as Equi, Mäkinen and Tomescu [11] proved that superpolynomial indexing time is needed in order to achieve subquaratic time queries. This result was given as a special case of a more general technique, which allows to claim that the same type of indexing lower bound holds for any problem with a fine-grained reduction from OV respecting a certain structure.

All these fine-grained results where further strengthened after the introduction of new hardness hypotheses [1] believed to be more reliable than SETH. Using this framework, Gibney, Hoppenworth, and Thankachan [16] showed that the hardness of SMLG can be based on these hypotheses, and this improved the lower bound proving that even shaving logarithmic factors from the quadratic time complexity is hard. This enhanced lower bound is given by reducing from a problem called Formula-SAT, a generalisation of CNF-SAT, and due to the nature of the problem the design of the reduction had to be modified into a structure of recursively-defined gadgets.

As a final note, although this work is centred around the quadratic lower bounds for SMLG, it is important to remember that there are graphs for which the problem is easier to solve, and even linear time complexity can be achieved. This is the case for instance for Wheeler graphs [14], certain (Elastic) Founder graphs [23, 13], trees [22] and Funnels [7]. More in general, it is also possible to parametrize the complexity in a parameter expressing the "sortability" of the graph [9, 8].

In the sections that follow, we will introduce SMLG more formally as well as the fine-grained complexity hypotheses. As a warm up, we show how to reduce CNF-SAT to OV in subexponential time, as the reduction is very central to the field and very concise at the same time. Then, we give an overview of the main ideas behind different fine-grained reductions for SMLG that were given throughout the years, with the goal of showcasing the different features among them.

2 Problem Definition

A pattern string P is a string of characters drawn from an alphabet Σ, has length $|P|$, and $P[i]$ is its i-th character. We say that $G = (V, E, \ell)$ is a labeled graph if V is a set of nodes, E a set of edges over V, and $\ell : V \to \Sigma$ is a labeling function that associate a character of alphabet Σ to every node in V. Moreover, we say that P has a *match* in G if there is a walk of nodes $v_1, \ldots, v_m$ in G such that $P = \ell(v_1) \cdots \ell(v_m)$. Then, SMLG is formally defined as follows.

▶ **Problem 1** (String Matching in Labelled Graphs (SMLG)).
INPUT: *A labeled graph* $G = (V, E, \ell)$ *and a pattern string* P*, both over alphabet* Σ*.*
OUTPUT: True *if and only if* P *has at least one match in* G*.*

Note that in this definition of SMLG we ask for a *walk* to exist and not a *simple path*. This is because, if we are not allowed to repeat nodes, there is a straightforward reduction from the Hamiltonian path problem, making the problem NP-Complete. To see this, simply define $\Sigma = \{\mathtt{A}\}$, $P = \mathtt{A\ A} \cdots \mathtt{A}$, $|P| = |V|$ and $\ell(v) = \mathtt{A}$ for every $v \in V$. In other words, we are asking to find a match for a path as long as the number of nodes, and if we can match every node only once, we are finding an Hamiltonian path.

2.1 Some Upper Bounds for SMLG

When faced with an algorithmic problem, one of the first questions we want to answer is often how bad does the brute force algorithm performs. If by brute force we just mean trying all possibilities, that is check all paths of length $|P|$, then we get very poor results, as in general there are $O(|V|^{|P|})$ many such paths. However, better alternatives exit, so let us briefly give a bird's-eye view of these approaches, without going into details. A quadratic algorithm for solving SMLG in the exact setting in general graphs has been known since the '90s [3, 4]. This algorithm constructs the product graph obtained between the graph and the pattern, which is obtained by repeating every node in the graph as many times as there are characters in the patterns, and then placing edges according to the original structure of the graph plus the constraint of having node labels appear in the pattern at specific positions. Then, the algorithm performs a DFS visit of the graph which, if successful in reaching a certain target node, reveals a match of P in G. Not long after, a first quadratic solution [21], which was later refined [25], was found also for the approximate setting, when we are looking for a walk spelling a string at minimum edit distance from P. Moreover, there are linear time algorithms for a rooted (thus directed) tree [2], and even when the roots of many trees are connected in a cycle [11]. This highlights already some important topology features: SMLG is linear in trees, while in DAGs it is quadratic in general. Non-topological characterization can also be given. For instance, in Wheeler graphs [14] a node can be sorted according to the sets of strings that it represents, that is the set of all possible strings spelled by paths reaching that node. This implies a total order on the nodes, which leads to linear time matching algorithms. This idea can be generalized and the time complexity can be parametrized as a function of the sortability of the graph [8].

3 Fine-Grained Complexity

Quadratic algorithms have been the best we could achieve for general formulations of SMLG, and the reason is because better solutions are unlikely to exist. In order to show that we have reached the optimum, we of course need lower bound techniques, and here is where fine-grained complexity comes into play, has it provides us the means of proving such lower bounds via reductions. Thus, before presenting the conditional lower bounds for SMLG, we first introduce fine-grained complexity tools and basic notions needed to understand the later reductions. The central hypothesis in fine-grained complexity is the *Strong Exponential Time Hypothesis* (SETH).

▶ **Definition 2** (SETH). *For every $\varepsilon > 0$, there exists $k \geq 3$ such that no deterministic or randomized algorithm can solve an instance of CNF-SAT over n variables with clauses of size at most k in $O(2^{(1-\varepsilon)n})$ time.*

This hypothesis is called *strong* to remark that it is a stronger statement when compared to the more forgiving *Exponential Time Hypothesis* (ETH), which forbids the existence of algorithms solving CNF-SAT in $O(2^{o(n)})$ time. To give an example, if an algorithm solves CNF-SAT in $O(2^{\frac{n}{2}})$, SETH is violated but ETH is not.

There are polynomial problems for which hardness conjectures are independently believed, and for which there also exist efficient reductions from CNF-SAT. When possible, it is preferred finding a reduction from these problems instead of reducing directly from CNF-SAT. This way, multiple hypotheses have to fail to invalidate the conditional lower bound, and moreover we have to deal only with polynomial complexities in the analysis, instead of having both exponential and polynomial complexities.

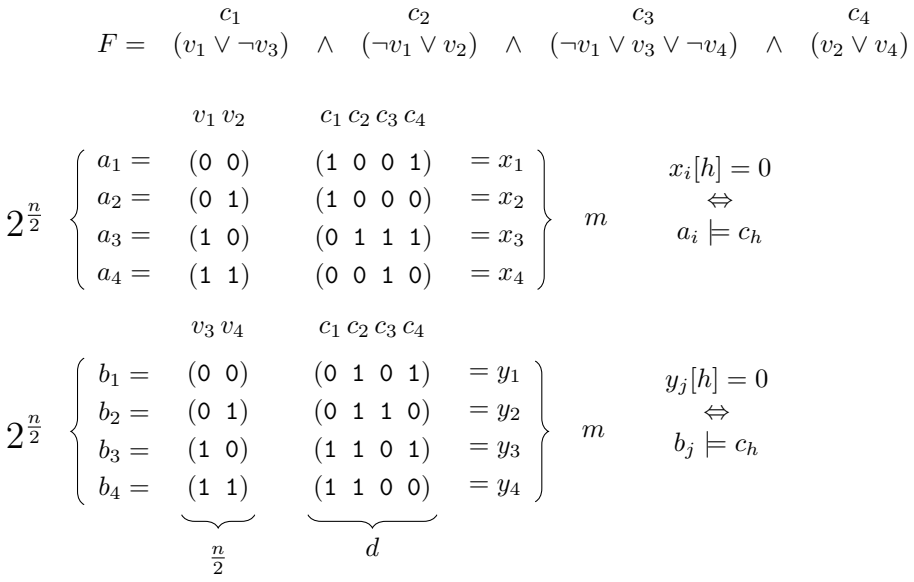


Figure 1 An example of the reduction from CNF-SAT to OV. Formula F has $n = 4$ variables and $d = 4$ clauses, and we construct two sets of $m = 2^{\frac{n}{2}} = 4$ vectors of size $d = 4$.

The polynomial problems most commonly used as a base for reductions to other polynomial problems are Orthogonal Vectors, All Pairs Shortest Paths, and 3-SUM [27]. In this work, we focus solely on *Orthogonal Vectors* (OV) which, intuitively, asks the following: given two sets of binary vectors, answer whether it is possible to find a vector in the first set and a vector in the second set so that they are orthogonal.

▶ **Definition 3** (OV). *Let $X, Y \subseteq \{0,1\}^d$ be two sets of $n = |X| = |Y|$ binary vectors each of length $d = \omega(\log n)$. Determine whether there exist $x \in X$ and $y \in Y$ such that $x \cdot y = \sum_{i=1}^{d} x[i] \cdot y[i] = 0$.*

The notation $x[i] \cdot y[i]$ indicates the scalar product when used for two single entries of vectors x and y, while it refers to the dot product $x \cdot y$ when applied on the vectors themselves. Currently, it is conjectured that no algorithm can solve OV in truly subquadratic time.

▶ **Definition 4** (OVC). *For every constant $\varepsilon > 0$, no deterministic or randomized algorithm can solve OV over two sets of n binary vectors of size d in $O(n^{2-\varepsilon} poly(d))$ time.*

One central result in fine-grained complexity is that SETH and OVC are connected via a subexponential reduction. We give a proof of this result, as we find it easy to follow and very instructive at the same time.

▶ **Lemma 5.** *SETH implies OVC.*

Proof. We can prove the contrapositive by showing a reduction from CNF-SAT to OV, an example of which is given in Figure 1. In other words, we prove that a subquadratic-time algorithm for OV implies a subexponential-time algorithm for CNF-SAT. Consider an instance of CNF-SAT where formula F has d clauses over n variables $v_1, \ldots, v_n$. The idea is to generate two sets of vectors that can represent partial truth assignments of F using only $O(2^{\frac{n}{2}})$ space, imposing the property that finding a pair of orthogonal vectors reveals how to combine two partial truth assignments into an actual truth assignment satisfying F.

To this end, we evenly split the variables into two sets $V_1 = \{v_1, \ldots, v_{\frac{n}{2}}\}$ and $V_2 = \{v_{\frac{n}{2}+1}, \ldots, v_n\}$. Then, we define sets A and B as the set of every possible partial truth assignment for the variables in V_1 and V_2, respectively. Given that $|V_1| = |V_2| = \frac{n}{2}$, we have that $|A| = |B| = 2^{\frac{n}{2}}$. Now we can define our two sets of vectors X and Y, consisting of $m = \frac{n}{2}$ vectors each. For every assignment $a_i \in A$, we construct a vector in X as follows. We evaluate the variables in V_1 under a_i, and if this is enough to satisfy clause c_h (i.e. at least one literal evaluates to 1), then the h-th entry of vector x_i shall be $x_i[h] = 0$, otherwise $x_i[h] = 1$. We perform the same construction for Y using B and V_2.

The logic of the reduction is the following. Our goal is to find a truth assignment satisfying F combining two partial truth assignments $a_i \in A$ and $b_j \in B$ encoded as vectors $x_i \in X$ and $y_j \in Y$. Since the h-th entry of vector x_i corresponds to the h-th clause of F, if the entry is $x_i[h] = 0$ then the clause is satisfied by a_i, meaning that whatever value $y_j[h]$ is, it does not change the end result. Conversely, if $x_i[h] = 1$, then a_i does not satisfy the h-th clause, and we need $y_j[h] = 0$ to guarantee that b_j satisfies it instead.

Now observe that the reduction takes $O(2^{\frac{n}{2}}\text{poly}(d))$ time, as there are $2 \cdot 2^{\frac{n}{2}}$ of size d, where d is the number of clauses. At this point, it is easy to see that, if vectors x_i and y_j are orthogonal, and thus for every h either $x_i[h] = 0$ or $y_j[h] = 0$, then at least one of the two partial truth assignments $a_i \in A$ or $b_j \in B$ satisfies clause c_h, and vice versa. Thus, F is satisfied if and only if there exist $x \in X$ and $y \in Y$ such that $x \cdot y = 0$. To conclude the proof, assume OVC is false, namely there exists an algorithm that solves OV in $O(m^{2-\alpha}\text{poly}(d))$ time, for some $\alpha > 0$. Then, this would provide an algorithm for CNF-SAT running in time

$$O(2^{\frac{n}{2}(2-\alpha)}\text{poly}(d)) = O(2^{n(1-\frac{\alpha}{2})}\text{poly}(d)),$$

Taking $\varepsilon = \frac{\alpha}{2}$ proves SETH false. ◀

The connection between SETH and OVC shown by Lemma 5 allow us to obtain lower bounds conditioned on both by only reducing from OV. This is advantageous, because conditional lower bounds obtained in this way stop being valid only when both SETH and OVC fail. Moreover, for polynomial problems, reducing from OV makes proofs cleaner as we can reason only in terms of polynomial time complexities, without having to juggle them together with exponential-time complexities.

4 The Lower Bound for String Matching in Labelled Graphs

Using fine-grained reductions, we can provide a quadratic lower bound for *exact* SMLG conditioned on SETH and OVC. In this section, we state this lower bound in its simplest form, focusing on giving the intuition behind the structure of the reduction. In Section 5, we explain a few different ways of strengthening this result.

The conditional lower bound for *exact* SMLG is formally stated as follows.

▶ **Theorem 6.** *The String Matching in Labelled Graphs (SMLG) problem on pattern string P and graph $G = (V, E)$ cannot be solved neither in $O(|E|^{1-\varepsilon}\,|P|)$ nor in $O(|E|\,|P|^{1-\varepsilon})$ time for any constant $\varepsilon > 0$, unless SETH fails.*

We now sketch the idea of the reduction from OV used to show the lower bound [12, 10]. Starting from sets of vectors X and Y, the reduction builds pattern P and graph G in $O(nd)$ time, such that P matches in G if and only if there is a pair of orthogonal vectors between X and Y. We first describe how to construct the pattern, and then the graph.

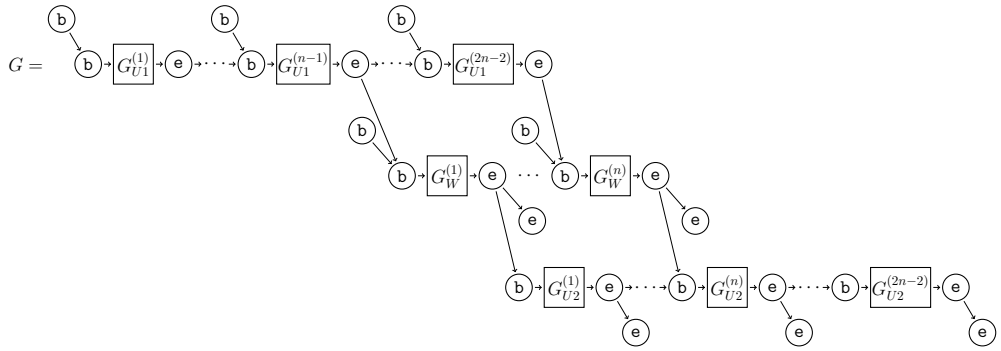


Figure 2 A high-level view of graph G constructed by the reduction from OV. The top and bottom rows consists of universal gadgets, while in the middle rows there are only vector gadgets, which match only patterns corresponding to orthogonal vectors. Here, we also add an orientation on the edges, even if it is not explicitly mentioned in the statement of Theorem 6. Indeed, this is allowed since the `b`- and `e`-nodes already force an orientation, as stated in Theorem 10. Figure adapted from [12].

Patter P is defined over alphabet $\Sigma = \{\mathtt{b}, \mathtt{e}, \mathtt{0}, \mathtt{1}\}$, has length $|P| = O(nd)$, and can be built in $O(nd)$ time from the first set of vectors $X = \{x_1, \ldots, x_n\}$. Namely, we define

$$P = \mathtt{bb}P_{x_1}\mathtt{e}\,\mathtt{b}P_{x_2}\mathtt{e}\ldots\mathtt{b}P_{x_n}\mathtt{ee}$$

where P_{x_i} is a string of length d that is just a copy of $x_i \in X$, that is $x_i[h] = 0 \Rightarrow P_{x_i}[h] = \mathtt{0}$ and $x_i[h] = 1 \Rightarrow P_{x_i}[h] = \mathtt{1}$, for $1 \leq i \leq n$ and $1 \leq h \leq d$. Here we are using characters `b` and `e` to mark the beginning and the end of each subpattern P_{x_i}, respectively. Substrings `bb` and `ee` instead respectively mark the beginning and the end of the entire pattern, forcing it to start and end the match at specific locations in the graph.

Graph G is built on top of the second set of vectors $Y = \{y_1, \ldots, y_n\}$, and consists of different substructures hierarchically organized: gadgets encoding single entries of the vectors are combined into gadgets encoding entire vectors, which are further combined to form the whole graph. At a macroscopic level, we want to structure the graph in three conceptual rows stacked on top of each other, as exemplified in Figure 2. In the graph, characters `b` and `e` force the subpatterns to correctly align to the gadgets encoding entire vectors, and thus forcing the entire match to synchronize at the subpattern level. Then, the idea is to make the top and bottom rows able to match any properly-synchronizing prefix and suffix of the pattern, respectively, while the middle row shall match only those subpatterns encoding vectors that are orthogonal to some vector in Y. Thus, if we force a match to start from the top or middle row, and end in the middle or bottom row, at least one subpattern must match in the middle row, which will be possible only if the pair of vectors encoded by that subpattern and the graph gadget it matches in are orthogonal. As shown in Figure 2, we can force this behaviour by introducing paths of two nodes spelling the string `bb` in the top and middle row. In pattern P, this string is present only as a prefix, marking the beginning of the pattern. The same logic applies to paths of two nodes spelling the string `ee` in the middle and bottom row, since that string is present in P only as a suffix.

As already mentioned, the single graph gadgets in the middle row G_W must provide the following property.

▶ **Lemma 7.** *Subpattern* $\mathtt{b}P_{x_i}\mathtt{e}$ *has a match in* $G^{(j)}_W$ *if and only if* $x_i \cdot y_j = 0$.

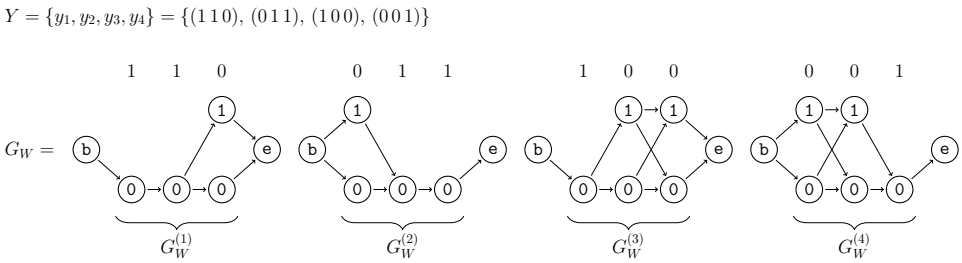


Figure 3 Vector gadgets matching only patterns corresponding to orthogonal vectors. For instance, $G_W^{(3)}$ will match 010 or 011, but not 110, as $(110) \cdot (100) \neq 0$. Figure adapted from [12].

Let us see how to construct such gadgets. Since $\sum_{h=1}^{d} x[h] \cdot y[h] = 0$ when every term of the sum is 0, the idea is to make the graph force the pattern to have character $S[h] = 0$ when $y[h] = 1$, while letting $S[h]$ be either `0` or `1` when $y[h] = 0$. To achieve this, first we place a path of h nodes all with label `0`, then we add a node with label `1` for those positions such that $y[h] = 0$, and finally we fully connect nodes encoding the entry at position h to those for position $h + 1$, for $1 \leq h \leq d$. Figure 3 shows an example of the entire middle row G_W, where gadget $G_W^{(j)}$ encodes vector y_j, for $1 \leq j \leq n$, using nodes with labels `b` and `e` to mark the beginning and the end of the gadgets. It then holds the following.

▶ **Lemma 8.** *Subpattern* $\texttt{b}P_{x_i}\texttt{e}$ *has a match in* G_W *if and only if there exists* $y_j \in Y$ *such that* $x_i \cdot y_j = 0$.

In order to complete the reduction, it remains only to build the universal gadgets for the top and bottom rows of G. This is easily achieved by following the same construction scheme as for gadgets $G_W^{(j)}$, and just placing both a `0`-node and a `1`-node at every position. Finally, we remark that in order to make all possible "shifts" of the pattern possible, the top and bottom rows of G must have $2(n-1)$ universal gadgets. We can now claim the following.

▶ **Lemma 9.** *Pattern* P *has a match in* G *if and only if a subpattern* $\texttt{b}P_{x_i}\texttt{e}$ *of* P *has a match in subgraph* G_W.

Notice that every gadget $G_W^{(j)}$ or universal gadget consists of $O(d)$ nodes and edges, and there are $O(2(n-1) + n + 2(n-1)) = O(n)$ such gadgets, for a total size of $O(nd)$. This consideration together with Lemma 7, 8 and 9 let us claim Theorem 6.

5 Tighter Lower Bounds

After having established a first quadratic conditional lower bound for SMLG, we now want to see how far we can push it. What if there are constraints on the graph, or on the alphabet? Can we make up for the quadratic complexity if we first build an index? Can we condition the lower bound on other hypotheses to make it even stronger? Let us explore these questions.

5.1 DAGs, Determinism and Bounded Degree

In this section, we study how strong assumptions we can make on the graph while still having the conditional quadratic lower bound hold. Starting from the graph structure, we remark that the nodes labelled with `b` and `e` already force the pattern to follow a specific direction, that is the pattern never needs to visit a node twice to make the reduction work. Indeed, directing the edges from left to right and from top to bottom as in Figure 4 does not compromise the construction.

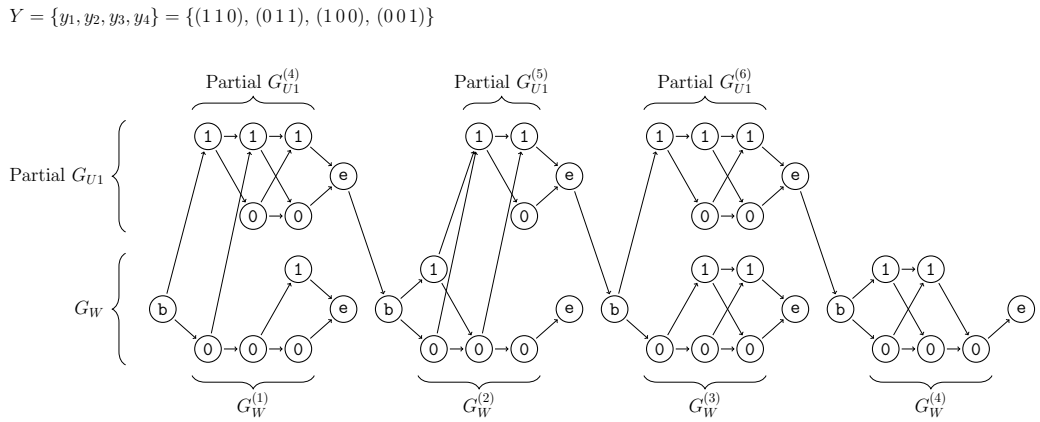


Figure 4 New gadget G_{U1W} obtained by merging G_W with part of gadget G_{U1}. The pattern can "escape" to the G_{U1} part to match a prefix until it finds a suitable $G_W^{(j)}$ where to match a subpattern encoding an orthogonal vector. Figure adapted from [12].

Nevertheless, there is one major non-trivial limitation to the reduction scheme proposed in Section 4. Even after making the edges directed, there are nodes that have two out-neightbours with the same label. For instance, nodes labelled with `e` in the top row might have two out-neighbours labelled with `e`. This might suggest that forcing every out-neighbour to have different labels could make the graph close enough to a DFA so that the problem becomes easier. Perhaps surprisingly, there is a way to modify the reduction so that it covers even this case. The construction is shown in Figure 4. Intuitively, we merge the top and middle row of G together, with the idea being that the pattern tries to match every sub-pattern in the underlying G_W as much as possible. When the match cannot be continued, the pattern will momentarily escape to a partial universal gadget for the remaining of the current sub-pattern. The non-determinism is removed by the fact that the nodes labelled `b` in the top and middle row are now merged together.

One last concern is the degree of the graph. By looking at the graph that we obtain after merging the top and middle row, we notice that every node is never connected to more than three other nodes. In a DAG, we can capture this concept by saying that the sum of outdegree and indegree of any node is at most three.

We are now ready to state the conditional lower bound for SMLG in its strongest form.

▶ **Theorem 10.** *The String Matching in Labeled Graphs (SMLG) problem on pattern string P and labelled* deterministic *directed acyclic graph (DAG) $G = (V, E)$ cannot be solved neither in $O(|E|^{1-\varepsilon}\,|P|)$ nor in $O(|E|\,|P|^{1-\varepsilon})$ time unless SETH fails, for any constant $\varepsilon > 0$. This holds even if it is restricted to a binary alphabet and to graphs in which the sum of outdegree and indegree of any node is at most three.*

In this statement, we are claiming that the theorem holds also for a binary alphabet. Indeed, there exists an binary encoding that allows to cover also this case. This requires to take care of some technicalities that we do not find fitting for this work, and hence we refer the reader to the literature [12].

5.2 Indexing Lower Bound

Given the results of previous sections, there seems to be no hope of solving SMLG in less than quadratic time, as long as SETH holds. But rules are made to be broken, or at least circumvented. In the lower bounds shown so far, we always analyzed the setting in which we

have to solve the problem from ground zero, without building any data structure to help us in performing queries. Hence, one question arises: can we break through the quadratic complexity if we build some kind of index on the graph? In other words, can we pay a quadratic (or even greater) cost upfront during preprocessing, so that the complexity reduces at query time? Although tantalizing, this option is also ruled out [11].

▶ **Theorem 11.** *For any $\alpha, \beta, \delta > 0$ such that $\beta + \delta < 2$, there is no algorithm preprocessing a labeled graph $G = (V, E, \ell)$ in time $O(|E|^\alpha)$ such that for any pattern string P we can solve the SMLG problem on G and P in time $O(|P| + |E|^\delta |P|^\beta)$, unless OVC is false. This holds even if restricted to a binary alphabet, and to deterministic DAGs in which the sum of out-degree and in-degree of any node is at most three.*

For $\delta = 1$ and $\beta = 1$ this lower bound is tight because there exists a matching online algorithm [3, 4]. However, this bound does not disprove a hypothetical polynomial indexing algorithm with query time $O(|P| + |E|^\delta |P|^2)$, for some $0 < \delta < 1$. Since in practical applications graphs are much larger than the pattern, such an algorithm would be quite significant for small enough δ. However, when the graph is allowed to have cycles, we also show that this is impossible under OVC.

▶ **Theorem 12.** *For any $\alpha, \beta, \delta > 0$, with either $\beta < 1$ or $\delta < 1$, there is no algorithm preprocessing a labeled graph $G = (V, E, \ell)$ in time $O(|E|^\alpha)$ such that for any pattern string P we can solve the SMLG problem on G and P in time $O(|P| + |E|^\delta |P|^\beta)$, unless OVC is false.*

Theorem 12 is obtained by slightly modifying the reduction of [10] with the introduction of certain cycles that allow querying patterns of length longer than the graph size. As for the online SMLG lower bound, it can be proven that this results holds also when restricted to a binary alphabet, and for graphs in which the sum of out-degree and in-degree of any node is at most three.

Without diving into technical details, we remark that this result is given as an application of a more general technique involving *linear independent-components* reductions [11], combined with some improvements on folklore knowledge about OV indexability. Intuitively, a linear independent-components reduction is a reduction from OV performed so that the instance of the output problem can be separated in two parts, each one depending only on one of the two sets of vectors. For instance, in the case of SMLG, we observe that that the reduction builds the pattern using only the first set of vectors, and the graph using only the second.

This independence property of the two components is crucial when combined with the following fact about OV. Suppose that in a OV instance we partition sets X and Y of n vectors into many subsets $X_1, \ldots, X_{\frac{n}{N}}$ and $Y_1, \ldots, Y_{\frac{n}{N}}$ where $N = |X_i| = |Y_j|$, so that each (X_i, Y_j) is a smaller OV instance and, moreover, solving all such instances also solves OV for the original X and Y. If we claim that we can index all the X_i in polynomial time to answer queries to the Y_j in subquadratic time, then we contradict OVC. This is because it is always possible to choose a small enough N such that both the total indexing time and the total query time fall below quadratic. For example, assuming we can index X_i in time $O(N^\alpha)$, we can take $N = O(n^{\frac{1}{\alpha}})$. Then, given that we have $\frac{n}{N}$ instances (X_i, Y_j), the total indexing time is $O(\frac{n}{N} N^\alpha) = O(n^{2-\frac{1}{\alpha}})$, and the total query time is bound to be subquadratic in n if every Y_j can be queried in time subquadratic in N.

Using the properties of linear independent-components reductions, we can use indexes for SMLG to answer queries for OV, and thus we can transfer the indexing lower bounds from OV to SMLG as in Theorem 11 and 12. Moreover, the consequences of this reduction scheme can be further strengthened if one considers a generalized version of OV where X and Y can have different cardinalities [11].

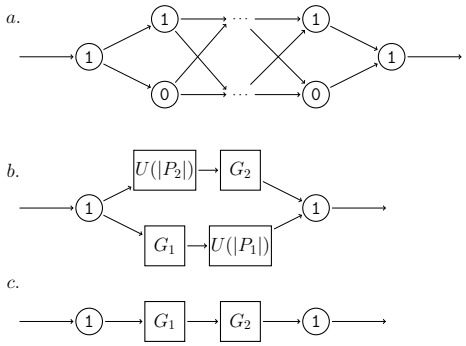


Figure 5 The universal gadget of variable length (a), the gadget encoding gate $\vee$ (b) and the gadget encoding gate $\wedge$ (c). Subgraphs G_1 and G_2 represent two gadgets encoding two subtree of the formula, respectively, that are the left and right children of an $\vee$ (b) gate or an $\wedge$ (c) gate. Figure adapted from Gibney et al. [16].

5.3 Lower Bounds from Formula-SAT for shaving Logarithmic Factors

All lower bounds shown for SMLG are conditional, and the strength of a conditional lower bound comes from the reliability of its hypothesis. It is then desirable to always look for more believable hypotheses on which we can base our conditional lower bounds. This is what was done in the work by Abboud et al. [1], where the starting problems of the proposed reductions are more general versions of the SAT problem, rather than CNF-SAT. This allows not only to base their conditional lower bounds on more reliable hypotheses, but also to achieve better complexity bounds. Later, Gibney et al.[16] showed how to use such techniques to obtain better lower bounds for problems like SMLG.

▶ **Theorem 13.** *If SMLG can be solved in time* $O\left(\frac{|E||P|}{\log^c |E|}\right)$ *or* $O\left(\frac{|E||P|}{\log^c |P|}\right)$ *for all* $c > 0$, *then* $\mathit{NTIME}[2^{O(n)}]$ *does not have non-uniform polynomial-size log-depth circuits.*

This result is achieved by showing a subexponential-time reduction from the Formula-SAT problem to SMLG. To define the Formula-SAT problem, let us first define a deMorgan formula. A deMorgan formula is a Boolean formula that can be represented as a binary tree where the leaves represents a variable or its negation, while internal nodes represent either one of the logical operators $\{\wedge, \vee\}$. Notice that negation is allowed only at the leaf level. The Formula-SAT problem is then the satisfiability problem over a deMorgan formula.

We now show the key ideas of the reduction that differ from the one based on OV. First, here we reduce from a SAT problem over n variables and thus, as for showing SETH $\Rightarrow$ OVC, we consider two sets of partial truth assignments, each one defined on $\frac{n}{2}$ variables. Next, since the formula is structured as a tree, the operators $\wedge$ and $\vee$ can be nested, and hence the reduction should have a recursive structure. The base case of the recursion are the gadgets that encode the input gates. Here, some care is needed to avoid having to deal with negation, but we defer the details to the work of Gibney et al. [16], where they use an intermediate problem to solve this issue. For building the recursive gadgets, an universal gadget of variable length $U(\mu)$ is needed. This has exactly the same structure of the universal gadgets in the reduction from OV, but with the two rows of 0- and 1-nodes elongated to accommodate matches for subpatterns of length μ. Then, the gadgets encoding gates $g = (g_1 * g_2)$, $* \in \{\wedge, \vee\}$, can be built by combining previously constructed gadgets for gates g_1 and g_2 as in Figure 5.

References

1 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 375–388. ACM, 2016. doi:10.1145/2897518.2897653.

2 Tatsuya Akutsu. A linear time pattern matching algorithm between a string and a tree. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium, CPM 93, Padova, Italy, June 2-4, 1993, Proceedings*, volume 684 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1993. doi:10.1007/BFb0029792.

3 Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. In Frank K. H. A. Dehne, Andrew Rau-Chaplin, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures, 5th International Workshop, WADS '97, Halifax, Nova Scotia, Canada, August 6-8, 1997, Proceedings*, volume 1272 of *Lecture Notes in Computer Science*, pages 160–173. Springer, 1997. doi:10.1007/3-540-63307-3_56.

4 Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *J. Algorithms*, 35(1):82–99, 2000. doi:10.1006/jagm.1999.1063.

5 Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008. doi:10.1145/1322432.1322433.

6 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

7 Manuel Cáceres. Parameterized algorithms for string matching to dags: Funnels and beyond. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France*, volume 259 of *LIPIcs*, pages 7:1–7:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CPM.2023.7.

8 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part I. *J. ACM*, 70(4):27:1–27:73, 2023. doi:10.1145/3607471.

9 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.

10 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPIcs*, pages 55:1–55:15, 2019. doi:10.4230/LIPICS.ICALP.2019.55.

11 Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. *Theor. Comput. Sci.*, 975:114128, 2023. doi:10.1016/J.TCS.2023.114128.

12 Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Trans. Algorithms*, 19(3):21:1–21:25, 2023. doi:10.1145/3588334.

13 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing elastic founder graphs. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ISAAC.2021.20.

14 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theor. Comput. Sci.*, 698:67–78, 2017. doi:10.1016/J.TCS.2017.06.016.

15 Garrison Erik, Sirén Jouni, Novak Adam M, Hickey Glenn, Eizenga Jordan M, Dawson Eric T, Jones William, Garg Shilpa, Markello Charles, Lin Michael F, Paten Benedict, and Durbin Richard. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36:875, August 2018. doi:10.1038/nbt.422710.1038/nbt.4227.

16 Daniel Gibney, Gary Hoppenworth, and Sharma V. Thankachan. Simple reductions from formula-sat to pattern matching on labeled graphs and subtree isomorphism. In Hung Viet Le and Valerie King, editors, *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, pages 232–242. SIAM, 2021. doi:10.1137/1.9781611976496.26.

17 Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

18 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/JCSS.2001.1774.

19 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

20 Udi Manber and Sun Wu. Approximate string matching with arbitrary costs for text and hypertext. In *Advances In Structural And Syntactic Pattern Recognition, Bern, Switzerland, 26-28 August 1992*, pages 22–33. World Scientific, 1992. doi:10.1142/9789812797919_0002.

21 Gonzalo Navarro. Improved approximate pattern matching on hypertext. *Theor. Comput. Sci.*, 237(1-2):455–463, 2000. doi:10.1016/S0304-3975(99)00333-3.

22 Kunsoo Park and Dong Kyue Kim. String matching in hypertext. In Zvi Galil and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, Espoo, Finland, July 5-7, 1995, Proceedings*, volume 937 of *Lecture Notes in Computer Science*, pages 318–329. Springer, 1995. doi:10.1007/3-540-60044-2_51.

23 Nicola Rizzo, Massimo Equi, Tuukka Norri, and Veli Mäkinen. Elastic founder graphs improved and enhanced. *Theor. Comput. Sci.*, 982:114269, 2024. doi:10.1016/J.TCS.2023.114269.

24 Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and Philip S. Yu. A survey of heterogeneous information network analysis. *IEEE Trans. Knowl. Data Eng.*, 29(1):17–37, 2017. doi:10.1109/TKDE.2016.2598561.

25 Mikko Rautiainen Tobias and Marschall. Aligning sequences to general graphs in $O(V + mE)$ time. *bioRxiv*, 2017. doi:10.1101/216127.

26 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/J.TCS.2005.09.023.

27 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians (ICM 2018)*, pages 3447–3487, 2018. doi:10.1142/9789813272880_0188.

Enumeration of Ordered Trees with Leaf Restrictions

Yasuaki Kobayashi
Faculty of Information Science and Technology, Hokkaido University, Sapporo, Japan

Dominik Köppl[1]
Department of Informatics, Yamanashi University, Japan

Yasuko Matsui
Department of Mathematical Sciences, Tokai University, Japan

Hirotaka Ono[1]
Department of Mathematical Informatics, Nagoya University, Japan

Toshiki Saitoh
School of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka, Japan

Yushi Uno
Graduate School of Informatics, Osaka Metropolitan University, Japan

Abstract

An α-ary tree for a constant $\alpha \geq 2$ is a rooted tree in which each node has at most α children. A node having no children is called a leaf. For a given rooted tree and a node v, the number of edges from the root to v is called the depth of v. We call a vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_d)$ of nonnegative integers an (α-ary) distribution if there is an α-ary tree T such that the number of leaves at each depth $i \in [1..d]$ in T is w_i. Although not every vector of nonnegative integers is a distribution, a distribution can be associated with many α-ary trees. In this paper, we present an algorithm to enumerate all α-ary trees for a given distribution. Our algorithm reports the first tree in $O(d + \sum_{i=1}^{d} w_i)$ time, and then each subsequent α-ary tree in $O(\max_{i=1}^{d} w_i)$ time by representing each tree as the difference from the previous one. The algorithm can be restricted to computing all trees that are *full*, i.e., trees whose nodes have exactly α or no children.

2012 ACM Subject Classification Mathematics of computing → Enumeration; Mathematics of computing → Trees; Theory of computation

Keywords and phrases binary trees, ordered trees, rooted trees, enumeration algorithm, constant-time delay

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.8

Category Research

Funding *Yasuaki Kobayashi*: JSPS KAKENHI Grant Numbers JP23K28034, JP24H00686, and JP24H00697.
Dominik Köppl: JSPS KAKENHI Grant Numbers JP23H04378 and JP25K21150.
Yasuko Matsui: JSPS KAKENHI Grant Numbers JP20H05964 and JP20K04973.
Hirotaka Ono: JSPS KAKENHI Grant Numbers JP20H05967, JP22H00513, JP24K02898, and JP25K03077, and JST CRONOS Grant Number JPMJCS24K2.
Toshiki Saitoh: JSPS KAKENHI Grant Number JP21H05857.
Yushi Uno: JSPS KAKENHI Grant Numbers JP20H05964 and JP21K11757.

[1] Corresponding author

1 Introduction

It is our pleasure to dedicate this work to Roberto Grossi, whose broad enthusiasm for algorithms and data structure has significantly shaped the landscape of enumeration algorithms [9]. Over the past decades, his contributions made significant impact on structural and algorithmic aspects of trees, strings, and graphs, inspiring both theoretical advances and practical applications.

In this spirit, we turn our attention to the enumeration of ordered trees, a topic rooted in combinatorics with strong connections to data structures, formal languages, and algorithmic analysis. Ordered trees play a fundamental role in various domains, hierarchical data representations, and the analysis of recursive algorithms are major examples. Enumerating ordered trees uncovers rich combinatorial patterns, with which we hope to offer a tiny tribute to Roberto's enduring influence, exploring a topic in line with his passion for enumerating combinatorial structures of mathematical beauty.

We here tackle a problem whose restricted version on binary trees has been posed in 2023 as an open problem presented at IWOCA 2023: The question is to efficiently enumerate all rooted full binary trees whose number of leaves at each depth is given as an input. This problem spawned from the research of prefix-free codes in coding theory.

Here, we tackle this enumeration problem in even more general settings: binary (not necessarily full) trees and α-ary trees of a fixed depth d for a constant $\alpha \geq 2$. For any of these tree types, our algorithm outputs the first solution in $O(d+n)$ time. Subsequently, it outputs the difference between two successive solutions in $O(m)$ time with constant delay, where n is the number of leaves and m is the maximum number of leaves at any depth. We here define *delay* as the time between the end of the previous output and the start of the subsequent output.

2 Related Work

With respect to the enumeration of rooted ordered trees, we are aware of an algorithm with constant delay to output all trees [3], or with restrictions on a specific diameter [12] or on a fixed number of internal nodes and leaves [17]. Given the number m of internal nodes, Zaks [18] gave an enumeration algorithm for all α-ary trees having m internal nodes. This algorithm reports the trees in a lexicographical order based on a bit encoding of the trees. Recently, an algorithm [5] enumerating AVL trees has been proposed. In addition, closed formulas have been studied for counting various properties in rooted ordered trees of unbounded degree [6, 7].

For the enumeration of full binary trees with n leaves, we propose a $2n$-bit encoding of a full binary tree. We use this encoding in the same way as Ruskey and Proskurowski [13] for the enumeration of binary trees by using Gray codes for enumerating bit encodings. Another bit encoding variant has been proposed by Baba et al. [1], who however used a depth-first encoding based on DFUDS [2], while our encoding is level-wise in the spirit of LOUDS [10].

3 Preliminaries

Given a fixed integer $\alpha \geq 2$, an *α-ary tree* is an ordered *rooted tree* whose nodes each have at most α children, which are ordered. In this paper, we only consider trees that have at least two nodes. Except for the root, the parent of each node is uniquely determined. There is a left-to-right ordering of the child nodes with respect to the parent. A node is called a *leaf* if it has no child, otherwise it is called an *internal* node. For a given rooted tree and a node v, the number of edges from the root to v is called the *depth* of v, which we denote by $\mathsf{depth}(v)$.

In particular, the depth of the root is zero. The *height* of the tree is the depth of the leaf with the largest depth. A *full α-ary tree* is an α-ary tree whose nodes are either leaves or internal nodes with α children. A special case is $\alpha = 2$, for which we name a 2-ary tree a *binary tree*.

In what follows, we start with the enumeration of full binary trees, which is the easiest case. Then we extend our techniques to arbitrary binary trees, and finally consider α-ary trees.

4 Properties of Binary Trees

Our input is a so-called *binary (tree leaf) distribution*, a term coined by Buro [4, Lemma 2.3]. It is defined as follows. We call an integer vector $\boldsymbol{w} = (w_1, \ldots, w_d)$ a *binary distribution* if there is a binary tree T with depth d such that T has w_i leaves with depth i for every $i \in [1..d]$. We say that the *binary (tree leaf) distribution* of T is $\boldsymbol{w}$ – there can be many trees that have the same binary distribution. A binary distribution is also called a *full-binary distribution* if there is a *full* binary tree with the same property. Buro [4, Thm. 2.4] showed that the full-binary distribution of a full binary tree with maximum path length is unique. Here we study the connection of arbitrary binary distributions and their corresponding trees. In detail, the first problem we want to tackle in this paper is as follows.

▶ **Problem 1.** *Given a (full-)binary distribution $\boldsymbol{w}$, enumerate all (full) binary trees whose (full-)binary distribution is $\boldsymbol{w}$.*

We first characterize the necessary conditions for an integer vector to be a (full) binary distribution. For that, we make use of Kraft's inequality [11].

▶ **Lemma 2** (Kraft's inequality). *For a binary tree with leaves $v_1, \ldots, v_n$, it holds that $\sum_{i=1}^{n} 1/2^{\mathrm{depth}(v_i)} \leq 1$.*

Let $f(\boldsymbol{w}) = \sum_{i=1}^{d} w_i/2^i$ be the *weight* of a vector $\boldsymbol{w}$ of nonnegative integers. Then $f(\boldsymbol{w}) \leq 1$ is a necessary condition according to Kraft's inequality. We show that it is also sufficient.

▶ **Lemma 3.** *A vector of nonnegative integers $\boldsymbol{w}$ is a binary distribution if and only if $f(\boldsymbol{w}) \leq 1$.*

Proof by induction on the depth d. For $d = 1$, the binary tree can have only one or two leaves, i.e., $\boldsymbol{w} = (1)$ or $\boldsymbol{w} = (2)$, and in any case $w_1 \leq 2$ holds. For the induction hypothesis, assume that our claim holds for a depth d. Given a vector of nonnegative integers $\boldsymbol{w} = (w_1, \ldots, w_{d+1})$ with $f(\boldsymbol{w}) \leq 1$, we show that there exists a binary tree whose binary distribution is $\boldsymbol{w}$. For that, we define a vector $\boldsymbol{w}' = (w_1, \ldots, w_{d-1}, w'_d)$ and distinguish the following two cases.

- If w_{d+1} is even, we set $w'_d = w_d + w_{d+1}/2$, and thus $f(\boldsymbol{w}) = f(\boldsymbol{w}')$. By the induction hypothesis, there is a binary tree T' whose binary distribution is $\boldsymbol{w}'$. If we expand $w_{d+1}/2$ leaves with depth d in T to internal nodes with two leaves, we obtain a binary tree whose binary distribution in $\boldsymbol{w}$.
- If w_{d+1} is odd, we rewrite the inequality $1 \geq f(\boldsymbol{w}) = \sum_{i=1}^{d+1} w_i 2^{-i}$ as $2^{d+1} \geq \sum_{i=1}^{d+1} w_i 2^{d+1-i}$. Observing that the right summation only consists of even terms except for w_{d+1}, the left-hand side is even while the right-hand side is odd. This allows us to increment the right-hand side by one while still retaining validity, i.e., as $2^{d+1} \geq \sum_{i=1}^{d} w_i 2^{d+1-i} + w_{d+1} + 1$. We therefore can apply the even case analysis for the vector $(w_1, \ldots, w_d, w_{d+1} + 1)$. ◀

▶ **Lemma 4.** *A vector* $\boldsymbol{w} = (w_1, \ldots, w_d)$ *with nonnegative integers and* $w_d > 0$ *is the full-binary distribution of a full binary tree if and only if* $f(\boldsymbol{w}) = 1$.

Proof. We prove both directions by induction, for which we use $d = 1$ as the starting point. For depth 1, on the one hand, there is only one full binary tree with the root having two leaves as children. On the other hand, there is only one vector $\boldsymbol{w} = (w_1) = (2)$ with $f(\boldsymbol{w}) = 1$, and therefore the claim holds. For the induction hypothesis, let us assume that the claim holds for d.

Direction $\Rightarrow$. Consider a full binary tree T whose full-binary distribution is $\boldsymbol{w} = (w_1, \ldots, w_{d+1})$. We can group the w_{d+1} leaves into $w_{d+1}/2$ pairs, each pair of leaves sharing the same parent at depth w_d. We therefore can contract these pairs with their parents to a contracted tree T' having $w_d + w_{d+1}/2$ leaves at depth w_d. Since T' is a full binary tree and $w_d + w_{d+1}/2 > 0$, according to the induction hypothesis, the full-binary distribution $\boldsymbol{w}' = (w_1, \ldots, w_d + w_{d+1}/2)$ of T' satisfies $f(\boldsymbol{w}') = 1$. This yields $f(\boldsymbol{w}') = f(\boldsymbol{w})$ because

$$1 = \sum_{i=1}^{d-1} \frac{w_i}{2^i} + \frac{w_d + w_{d+1}/2}{2^d} = \sum_{i=1}^{d+1} \frac{w_i}{2^i}\,.$$

Direction $\Leftarrow$. Given a vector $\boldsymbol{w} = (w_1, \ldots, w_{d+1})$ with nonnegative integers such that $f(\boldsymbol{w}) = 1$, i.e., $\sum_{i=1}^{d+1} \frac{w_i}{2^i} = 1$. Multiplying this equation on both sides by 2^{d+1} gives $2^d w_1 + 2^{d-1} w_2 + \cdots + 2 w_d + w_{d+1} = 2^{d+1}$, and hence w_{d+1} must be even since all other terms are even. Therefore, $w'_d = w_d + w_{d+1}/2$ is integer and $\boldsymbol{w}' = (w_1, \ldots, w_{d-1}, w'_d)$ is a vector with $f(\boldsymbol{w}') = 1$ because

$$\sum_{i=1}^{d-1} \frac{w_i}{2^i} + \frac{w'_d}{2^d} = \sum_{i=1}^{d-1} \frac{w_i}{2^i} + \frac{w_d}{2^d} + \frac{w_{d+1}}{2^{d+1}} = 1.$$

By the induction hypothesis, $\boldsymbol{w}'$ is a full-binary distribution of a full binary tree T'. We now expand $w_{d+1}/2$ leaves of T' at depth d to internal nodes, each having two leaf children. This gives us a full binary tree whose full-binary distribution is $\boldsymbol{w}$. ◀

Let $g(i, \boldsymbol{w}) = \sum_{j=i}^{d} w_j / 2^{j-i}$ for $i \in [1..d]$.

▶ **Corollary 5.** *Given a full-binary tree distribution* $\boldsymbol{w} = (w_1, \ldots, w_d)$*, then* $g(i, \boldsymbol{w})$ *is the number of nodes with depth* j*, which is even, for every* $i \in [1..d]$.

Proof. Following the proof of Lemma 4, we iteratively merge the leaves on depth j with their parent node for j from d to $i + 1$. At each step, we obtain a full binary tree such that the number of leaves with the highest depth is always even. ◀

▶ **Theorem 6.** *The number of full binary trees having a full-binary distribution* $\boldsymbol{w} = (w_1, \ldots, w_d)$ *with* $w_d > 0$ *and* $f(\boldsymbol{w}) = 1$ *is*

$$\prod_{i=1}^{d-1} \binom{g(i, \boldsymbol{w})}{w_i}\,.$$

Proof. We show the claim by induction on d. The base case is $d = 1$, for which we have seen that $\boldsymbol{w} = (2)$ is uniquely defined. For the induction hypothesis, let us assume that the claim holds for d.

Consider a full-binary distribution $\boldsymbol{w} = (w_1, \ldots, w_{d+1})$ with $w_{d+1} > 0$ and $f(\boldsymbol{w}) = 1$. Then $g(i, \boldsymbol{w})$ is even for all i according to Corollary 5. In particular, the vector $\boldsymbol{w}' := (w'_1, \ldots, w'_d)$ defined by $w'_i = w_i$ for $i \in [1..d-1]$ and $w'_d = w_d + w_{d+1}/2$ is an integer vector with $f(\boldsymbol{w}') = 1$. With the induction hypothesis for $\boldsymbol{w}'$ we obtain

$$\prod_{i=1}^{d-1} \binom{\sum_{j=i}^{d} \frac{w'_j}{2^{j-i}}}{w'_i} = \prod_{i=1}^{d-1} \binom{w_i + \frac{w_{i+1}}{2} + \cdots + \frac{w_{d-1}}{2^{(d-1)-i}} + \frac{w_d + w_{d+1}/2}{2^{d-i}}}{w_i} = \prod_{i=1}^{d-1} \binom{\sum_{j=i}^{d+1} \frac{w_j}{2^{j-i}}}{w_i}.$$

Each of these trees has $w_d + w_{d+1}/2$ leaves with depth d. Fix one of them, which we modify to obtain a tree whose full-binary distribution is $\boldsymbol{w}$. For that, we select $w_{d+1}/2$ of those leaves, for which we have $\binom{w_d + w_{d+1}/2}{w_{d+1}/2} = \binom{w_d + w_{d+1}/2}{w_d}$ possibilities. By modifying each of the trees whose full-binary distribution is $\boldsymbol{w}'$ in all possible ways, we obtain

$$\prod_{i=1}^{d-1} \binom{\sum_{j=i}^{d+1} \frac{w_j}{2^{j-i}}}{w_i} \cdot \binom{w_d + w_{d+1}/2}{w_d} = \prod_{i=1}^{d} \binom{\sum_{j=i}^{d+1} \frac{w_j}{2^{j-i}}}{w_i}.$$

By construction, all modifications are distinct by using only expansions. If two expansions are equal, they must have originated from the same tree we expanded. ◀

▶ **Example 7.** For $d = 2$, there are two full-binary distributions $\boldsymbol{w}_1 = (0, 4)$ and $\boldsymbol{w}_2 = (1, 2)$. For $\boldsymbol{w}_1$, $\binom{0+\frac{4}{2}}{0} = 1$, and there is only the perfect full binary tree having $\boldsymbol{w}_1$ as full-binary distribution. For $\boldsymbol{w}_2$, $\binom{1+\frac{2}{2}}{1} = 2$, there are two full binary trees, as shown in Figure 1.

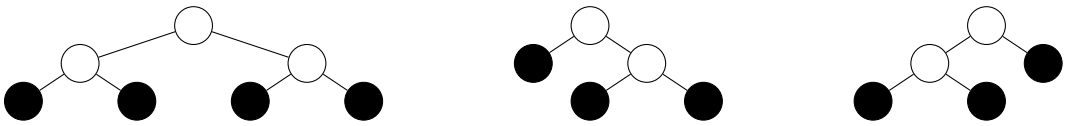


Figure 1 Illustration of Example 7, where the left tree has the full-binary distribution $(0, 4)$ and the other trees $(1, 2)$. Black nodes are leaves.

5 Enumeration of Full Binary Trees

In what follows, we present an enumeration algorithm for Problem 1 in the case of full binary trees. Our main idea is to process the trees in a linear order, which we define by a binary encoding of the trees.

For that, we recall the result of Corollary 5 that $g(i, \boldsymbol{w})$ gives the number of nodes with depth $i \in [1..d]$, which is common to all full binary trees having the full-binary distribution $\boldsymbol{w}$. In short, we define the vector $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_d)$ with $\eta_i := g(i, \boldsymbol{w})$ for $i \in [1..d]$. Then $\sum_{i=1}^{d} \eta_i$ is the number of all nodes. In particular, we can use an η_i-length bit vector to specify which nodes on depth i are internal nodes or leaves. The list of bit vectors on all depths is sufficient to represent a full binary tree. Then $\sum_{i=1}^{d} \eta_i = \sum_{i=1}^{d} \sum_{j=i}^{d} \frac{w_j}{2^{j-i}} \leq 2 \sum_{i=1}^{d} w_i$. For $n = \sum_{i=1}^{d} w_i$, this means that we can represent each full binary tree whose full-binary distribution is $\boldsymbol{w}$ in $2n$ bits. We use this bit representation to enumerate these trees. For that, we build on the algorithm of [14].

▶ **Lemma 8** ([14]). *There is an algorithm that enumerates all subsets with $m \in [1..n]$ integers of the integer range $[1..n]$ in ascending lexicographic order with constant delay.*

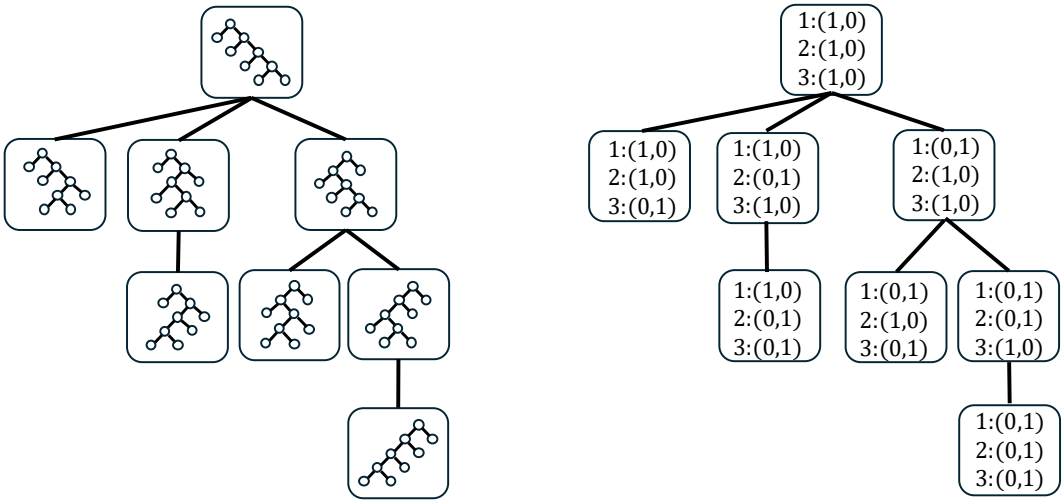


Figure 2 Enumerating all full binary trees whose distribution is $\boldsymbol{w} = (1, 1, 1, 2)$ with $d = 4$. Both trees on the left and right depict the same enumeration tree of $\boldsymbol{w}$ explained in Section 5, but with a different representation. A node on the left depicts a full binary tree T whose encoding $\boldsymbol{B}_T$ is given by a node on the right at the same position in the respective tree. We partition each $\boldsymbol{B}_T$ by the depths $[1..d-1]$ (depth d contains only zeros). For instance, the root node of the right enumeration tree E specifies with 10 that there is a left leaf and a right internal node on each depth in $[1..3]$, while the deepest leaf of E specifies a tree with opposite characteristics.

The connection to our problem is that a bit vector of length n with m "1"s exactly represents such a subset. Since we can impose the natural lexicographic order on the $2n$-length bit vectors, the missing step is to find, given a $2n$-length bit vector representing a full binary tree, the lexicographic succeeding bit vector that represents a tree we want to output.

To facilitate our starting, we assume that $w_i \geq 1$ for every $i \in [1..d]$. Suppose that we have a full-binary tree T whose distribution is $\boldsymbol{w}$. We deduce from T a bit vector $\boldsymbol{B}_T$ representing T. For explanation, we assume that the least significant bits are on the left. We start with a bit vector $\boldsymbol{B}_T$ of length $\sum_{i=1}^{d} \eta_i \leq 2n$ to represent the order of the internal nodes and leaves on each depth. To do this, we partition $\boldsymbol{B}_T$ by depths $\boldsymbol{B}_T = (\boldsymbol{b}^{(1)} \cdots \boldsymbol{b}^{(d)})$ such that $\boldsymbol{b}^{(i)}$ is a bit vector of length η_i with $b^{(i)}_j = 1$ if and only if the j-th node with depth i is a leaf. In particular, $\boldsymbol{b}^{(i)}$ has w_i zeros. As an example, the values of $\boldsymbol{B}$ for the trees depicted in Figure 1 are `001111` (left), `1011` (middle), and `0111` (right). (Recall that $\boldsymbol{B}_T$ does not encode the root node, for which we assume it is always an internal node.) As a practical optimization, we can omit the highest depth d since it always contains leaves, i.e., $\boldsymbol{b}^{(d)}$ contains only zeros. Since we can reconstruct T from $\boldsymbol{B}_T$ uniquely, there is a one-to-one mapping from full binary trees with at least two nodes and a subset of bit strings defined by the above tree encoding. Consequently, each full binary tree has exactly one specific bit vector representation.

In what follows, we represent each full binary tree T having the distribution $\boldsymbol{w}$ with the bit vector $\boldsymbol{B}_T$, which is stored as a node in a so-called enumeration tree E. The root of E represents the full binary tree T obtained by grouping all internal nodes to the left side, which we expressed by the lexicographic least bit vector, obtained by shifting all "1"s in every $\boldsymbol{b}^{(i)}$ of $\boldsymbol{B}_T$ to the left. An E-node is augmented by a bit vector $\boldsymbol{B}_T$ representing a full binary tree T and an integer, which we call the *change level*. The change level of an E-node v states the depth at which the bit vector of v differs from its parent; as an exception, we

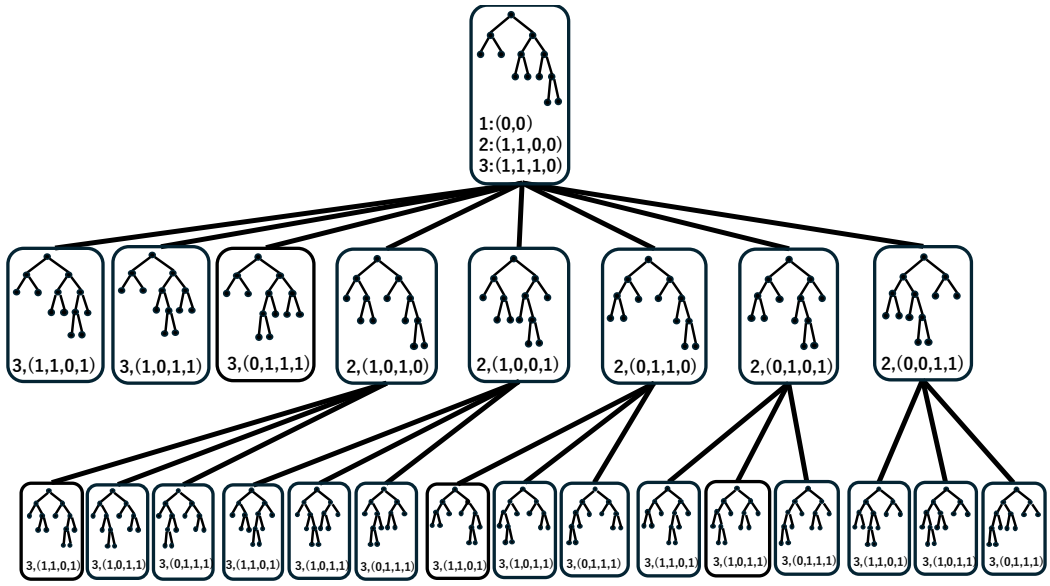


Figure 3 An enumeration tree E on an extended example compared to Figure 2, with distribution $\boldsymbol{w} = (0, 2, 3, 2)$. We here provide a combined view of both trees as illustrated in Figure 2. While the root node shows its full bit vector, we only depict the bit vector at the change level of each other node.

stipulate that the change level of the root node is 0. The change level is an invariant we impose on E in the sense that, when traversing from a node u to of its children v, the tree represented by u and by v can and must differ only at the change level of v. A consequence is that E-nodes with change level d are leaves.

We organize the children of an E-node as follows. Fix one E-node v with bit vector $\boldsymbol{B}_T = (\boldsymbol{b}^{(1)} \cdots \boldsymbol{b}^{(d)})$ and change level $\delta \in [0..d-1]$. Then the children of v correspond to all possible rearrangements of the bit patterns in one of the vectors $\boldsymbol{b}^{(\delta+1)}, \ldots, \boldsymbol{b}^{(d)}$. We start with the first children of v that enumerate all possible bit patterns of $\boldsymbol{b}^{(d)}$ (resulting by permutations). So they have change level d with a bit vector that differs from v's bit vector only at depth d. (In particular, these nodes are leaves by the definition of the change level.) Next, we continue with the group of children for $\boldsymbol{b}^{(d-1)}$ and change level $d-1$, and so on, up until $\boldsymbol{b}^{(\delta+1)}$ and change level $\delta+1$. In particular, we arrange each group of children with the same change level i in lexicographic order of their corresponding bit vector $\boldsymbol{b}^{(i)}$. This order of the children and the order of their groups allows us to enumerate all full binary trees by a pre-order traversal of E in lexicographic order. (In particular, the lexicographic order imposes the invariant that we never enumerate a full binary tree twice.) This structure of E warrants the following – see Figure 2 for an example and Figure 3 for a more elaborated example involving a non-binary choice at one level:

- Any subtree rooted at a node of E is an enumeration tree itself. That is because an E-node with change level $\delta \in [1..d]$ representing a tree T is the root node of an enumeration tree that considers only the suffix $\boldsymbol{b}^{(\delta+1)} \cdots \boldsymbol{b}^{(d)}$ of $\boldsymbol{B}_T$ for the enumeration. In detail, for an E-node with change level δ it holds that $\boldsymbol{b}^{(i)} = (\overbrace{1 \cdots 1}^{w_i} \overbrace{0 \cdots 0}^{\eta_i - w_i})$ for every $i \in [\delta+1..d-1]$ by the construction E – recall that a node with change level δ only makes a change on $\boldsymbol{b}^{(\delta+1)}$ compared to its parent node, and reading the change levels upwards to the root

gives a monotone decreasing sequence of change levels (so there is no way that any deeper level has changed with respect to the initial bit vector of the root.) In particular, an E-node is the lexicographically least among all its descendants.

- Since deeper recursion levels only touch shorter suffixes of the bit vectors, the rightmost leaf of an E-node v is lexicographically smaller than v's right sibling (if it exists). Therefore, all E-nodes represent different vectors, and the sum of all E-nodes is the size of the output we want to compute.

If we now run the algorithm of Lemma 8 on d nested loops we can traverse the constructed enumeration tree E in a pre-order traversal and output each node upon visiting it. The maximum delay happens when moving to the next value in the lowest $d-1$ loops. This delay is $O(d)$, but $O(1)$ amortized because to reach a depth of d' we have already output d' nodes.

It is left to argue that we can transform the sequence of outputted E-nodes into the full binary trees we want to output. For that, we initially build the full binary tree represented by the root of E from scratch, taking $O(n)$ time. Subsequently, for each E-node visit, it suffices to change the order of the nodes of the previously computed full binary tree at a specific depth, costing $O(\max_i g(i, \boldsymbol{w}))$ time.

▶ **Lemma 9.** *Given a full-binary distribution $\boldsymbol{w}$ with $w_i \geq 1$ for all $i \in [1..d]$, we can enumerate all full binary trees whose full-binary distribution is $\boldsymbol{w}$ in lexicographic order with constant amortized delay or $O(d)$ worst-case delay. The precomputation takes $O(n)$ time.*

We can generalize this approach to full-binary distributions having zero values. The obstacle is that we cannot guarantee constant delay since we might traverse long paths with zero weights. As a remedy, we want to skip each depth i with $w_i = 0$. To do that, let us denote by $I = \{i \in [1..d] \mid w_i > 0\}$ all depths with leaves. Then it suffices to restrict the values of a change level of a node to values of $I \cup \{0\}$. By construction, the root node has change level 0, and a node with change level δ has children whose augmented value is the successor of δ in I.

▶ **Lemma 10.** *Given a full-binary distribution $\boldsymbol{w}$ with $w_d \geq 1$, we can enumerate all full binary trees whose full-binary distribution is $\boldsymbol{w}$ in lexicographic order with constant amortized delay or $O(d)$ worst-case delay. The precomputation takes $O(n)$ time.*

To obtain worst-case constant delay time, we can drop the requirement to output in lexicographic order by using a folklore technique, such as the alternating output technique [15]. The idea is to output nodes on odd depths when visiting them the first time, but postpone the output of nodes on even depths until we backtrack and move outside their respective subtrees.

▶ **Theorem 11.** *Given a full-binary distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$, the corresponding full binary tree can be output in $O(d+n)$ time for the first tree and then output all trees in the form of differences with a size of $O(m)$ with a delay of $O(1)$ in $O(m)$ time, where $n = \sum_{i=1}^{d} w_i$ and $m = \max\{w_i \mid i \in [1..d]\}$.*

6 Enumeration of Binary Trees

In what follows, we show how to generalize the enumeration of full binary trees to general binary trees, where internal nodes are allowed to have one or two children.

From Lemmas 3 and 4, the binary distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$ of a non-full binary tree also satisfies $f(\boldsymbol{w}) = \sum_{i=1}^{d} w_i/2^i < 1$. We make the connection to full binary trees by adding dummy leaves to all internal nodes that have only one child. To this end, we denote the

number of dummy leaves added at depth i by x_i, and construct a vector $\boldsymbol{x} = (x_1, \ldots, x_d)$. The sum $\boldsymbol{x} + \boldsymbol{w}$ gives a vector that is the binary distribution of a full binary tree because $\sum_{i=1}^{d}(w_i + x_i)/2^i = 1$. In what follows, we call $\boldsymbol{x}$ the *complement vector* of $\boldsymbol{w}$, and $\boldsymbol{w} + \boldsymbol{x}$ the *completion* of $\boldsymbol{w}$. For such $\boldsymbol{x}$, a nonnegative integer vector $\boldsymbol{y}$ that satisfies $\boldsymbol{y} \leq \boldsymbol{x}$ is called the *subcomplement vector* of $\boldsymbol{w}$.

▶ **Lemma 12.** *For a complement vector $\boldsymbol{x}$ of $\boldsymbol{w}$ we have that $x_i \leq g(i, \boldsymbol{w})$.*

Proof. We can only attach a dummy leaf at depth i to internal nodes that have exactly one child on depth i. By the pigeonhole principle, there are at most $g(i, \boldsymbol{w})$ many such children on depth i. ◀

From Lemmas 3 and 4, the following immediately holds.

▶ **Lemma 13.** *If the binary distribution $\boldsymbol{w}$ satisfies $f(\boldsymbol{w}) < 1$, then there exists a complement vector for $\boldsymbol{w}$.*

However, the complement vector is not unique, in general. For example, $\boldsymbol{w} := (0, 2, 2)$ is not a full-binary distribution since $2/4 + 2/8 = 3/4$, while $(0, 1, 0)$ and $(0, 0, 2)$ are both complement vectors of $\boldsymbol{w}$. See Figure 4 for an example.

Figure 4 Examples for Lemma 13 with the binary distribution $\boldsymbol{w} = (0, 2, 2)$, where two binary trees exist, which can be complemented by adding dummy leaves. The left tree and the right tree can be complemented by the vectors $(0, 0, 2)$ and $(0, 1, 0)$ to full binary trees, respectively. Black nodes are leaves, and dummy nodes are rectangles.

Also, even if $\boldsymbol{x}$ satisfies $f(\boldsymbol{w} + \boldsymbol{x}) = 1$, it does not necessarily mean that $\boldsymbol{x}$ is the complement vector of $\boldsymbol{w}$. For example, $\boldsymbol{w} := (0, 2, 0, 2)$ has $(0, 0, 2, 2)$ as a complement vector, but $\boldsymbol{x} = (0, 0, 0, 6)$ is not a complement vector: A tree having the binary distribution $\boldsymbol{w}$ cannot have six dummy leaves on depth 4. However, a necessary condition is given below.

▶ **Lemma 14.** *A nonnegative integer vector $\boldsymbol{x} = (0, \ldots, 0, y_d)$ is a subcomplement vector of a complement vector $\boldsymbol{y} = (y_1, \ldots, y_d)$ of a binary distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$ if and only if $y_d \equiv w_d \pmod 2$, $y_d \in [0, w_d]$, and $f(\boldsymbol{w}) + y_d/2^d \leq 1$.*

Proof. Let $\boldsymbol{x} = (0, \ldots, 0, y_d)$.

Direction $\Rightarrow$. In order for $\boldsymbol{x}$ to be a subcomplement vector of $\boldsymbol{w}$, a binary tree with $y_d + w_d$ leaves, including dummy leaves, must exist. Therefore, it follows from Corollary 5 that $y_d + w_d$ is even, so $y_d \equiv w_d \pmod 2$. Also, since dummy leaves can only be attached to internal nodes that have exactly one child (in this case, a leaf), $y_d \leq w_d$ by Lemma 12. Furthermore, in order for $\boldsymbol{x} + \boldsymbol{w}$ to be a binary tree (with dummy leaves), $f(\boldsymbol{w} + \boldsymbol{x}) = \sum_{i=1}^{d}(w_i + x_i)/2^i = \sum_{i=1}^{d} w_i/2^i + x_d/2^d \leq 1$ must hold.

Direction $\Leftarrow$. We show that there exists a complement vector $\boldsymbol{y}$ for $\boldsymbol{w}$ such that $\boldsymbol{x} \leq \boldsymbol{y}$.

For depth $d = 1$, $\boldsymbol{w}$ must be either (1) or (2). In the case of $\boldsymbol{w} = (2)$, $\boldsymbol{w}$ represents a full binary tree, so the complement vector of $\boldsymbol{w}$ is $\boldsymbol{y} = (0)$, and thus $\boldsymbol{x} = (0)$. In the case of $\boldsymbol{w} = (1)$, the only complement vector is $\boldsymbol{y} = (1)$, and hence $\boldsymbol{x} = (1)$ is the unique subcomplement vector of $\boldsymbol{y} = (1)$ satisfying $x_d = y_d$. Thus, in both cases the subcomplement vectors are uniquely defined by the right-hand side conditions.

In what follows, we assume that $d \geq 2$. Given a binary distribution $\boldsymbol{w}$ of a binary tree T, by assumption $w_d + y_d$ and $w_d - y_d$ are even numbers. Hence, we can apply the contraction trick like in previous proofs. For that, consider the vector $\boldsymbol{w'} := (w_1, \ldots, w_{d-2}, w_{d-1} + (w_d + y_d)/2)$. The vector $\boldsymbol{w'}$ is well-defined: Since $w_d \equiv y_d \mod 2$ and $y_d \leq w_d$, $(w_d + y_d)/2 = (w_d - y_d)/2 + y_d$ is a positive integer. Since

$$f(\boldsymbol{w'}) = \sum_{i=1}^{d-2} w_i/2^i + (w_{d-1} + (w_d + y_d)/2)/2^{d-1} = f(\boldsymbol{w}) + y_d/2^d \leq 1,$$

by Lemmas 3 and 13, $\boldsymbol{w'}$ is a binary distribution of a binary tree T'. The normal leaves (i.e., excluding dummy leaves) of depth $d-1$ in T' are $w_{d-1} + (w_d + y_d)/2$ in total, of which

- w_{d-1} of them are former leaves of T unaffected by the contraction,
- $(w_d + y_d)/2$ internal nodes of T are changed into leaves on depth $d-1$ by contracting them with their two child leaves (where one may be a dummy leaf).

The resulting binary tree T' therefore has $w_{d-1} + (w_d + y_d)/2$ leaves on depth $d-1$ in total. The contraction of T to T' by removing depth d does not affect the number of leaves at depth $d-2$ or less in T. By definition, there is a complement vector $\boldsymbol{z} = (z_1, \ldots, z_{d-1})$ of $\boldsymbol{w'}$ that can turn T' into a full binary tree. Finally, we turn y_d leaves of T' on depth d into internal nodes, each having one child. This restores T and induces the complement vector $\boldsymbol{z} = (z_1, \ldots, z_{d-1}, y_d)$ of $\boldsymbol{w}$. ◀

For every binary tree, there is only one way to add dummy leaves to internal nodes to make it a full binary tree. Hence, the complement vector for each binary tree, whose completion is the binary distribution of the full binary tree, is uniquely determined. Conversely, given a full binary tree and a configuration classifying leaves in dummy leaves and non-dummy leaves, we can obtain the original binary tree by removing dummy leaves according to the configuration to obtain the original binary tree. Based on this observation, we first find the completion, i.e., the corresponding full binary tree, for $\boldsymbol{w}$, and then enumerate the binary trees by considering all ways of adding dummy leaves while enumerating the corresponding full binary completions. That is, by enumerating all possible complement vectors for $\boldsymbol{w}$, and then considering the process of enumerating the different ways of adding dummy leaves while considering the corresponding full binary completion, we can enumerate all binary trees. In summarizing the above, the enumeration algorithm is designed in a three-layer structure:

1. enumeration of complement vectors (Section 6.1),
2. enumeration of the way of adding dummy leaves to each complement vector (Section 6.2), and
3. enumeration of binary trees for each way of adding dummy leaves (Section 6.3).

6.1 Enumeration of Complement Vectors

We enumerate the complement vectors with the insights of Lemmas 13 and 14. In detail, Lemma 14 gives us a strategy to enumerate the complement vectors by enumerating subcomplement vectors. We show the algorithmic steps in Algorithm 1, which is a recursive algorithm

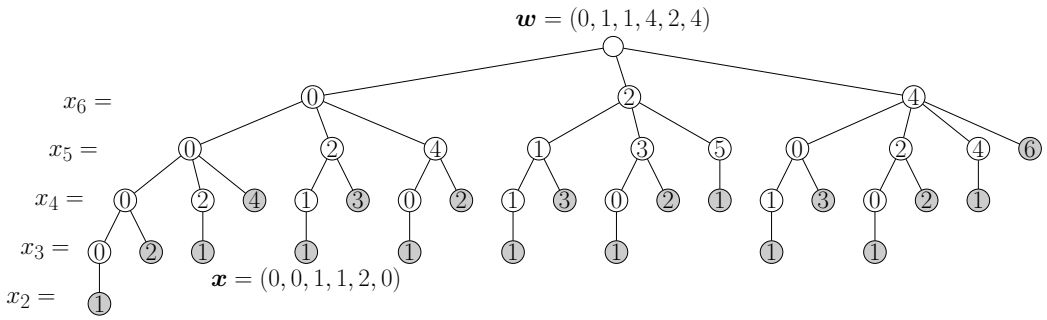


Figure 5 Enumeration tree of complement vectors for the binary distribution $\boldsymbol{w} = (0, 1, 1, 4, 2, 4)$, where leaves correspond to the complement vectors. A node on depth i determines the value $x[d - i + 1]$ of the complement vector. The details are described in Section 6.1.

Algorithm 1 Subroutine for enumerating complement vectors for $\boldsymbol{w}$, cf. Section 6.1.

```
1: call EnumCompVector with W ← f(w), x = (0, ..., 0), i = d and c = 0
2: procedure EnumCompVector(W, x, i, c)      ▷ x: subcomplement vector, i ∈ [1..d]:
   depth, c: number of conceptional leaves on depth i.
3:                ▷ determine all dummy nodes on depth i and recursively on lower depths.
4:     x_i ← (w_i + c) mod 2      ▷ initialization. By Lemma 14 the parity must be the same.
5:     while x_i ≤ w_i + c  and  W + x_i/2^i ≤ 1 do
6:         y ← x + (0, ..., 0, x_i, 0, ...)          ▷ y meets the requirements of Lemma 14.
7:         if W + x_i/2^i = 1 then
8:             Output y                                  ▷ output a complement vector
9:             return
10:        else
11:            EnumCompVector(W + x_i/2^i, y, i − 1, (x_i + w_i)/2) ▷ recurse on depth i − 1
12:        end if
13:        x_i ← x_i + 2
14:    end while
15:    return
16: end procedure
```

that determines the values of x_i in descending order $x_d, x_{d-1}, \ldots$ in a depth-first manner. We initially call EnumCompVector with the arguments $(f(\boldsymbol{w}), \boldsymbol{0}, d, 0)$, for the weight $f(\boldsymbol{w})$ of the binary distribution $\boldsymbol{w}$, $\boldsymbol{0}$ the d-dimensional vector with 0 entries, the depth d and the number of conceptional leaves on depth d. The last argument needs a definition: This algorithm recursively contracts leaves and dummy leaves at depth $i + 1$ to the so-called *conceptional* leaves on depth i. If we have x_{i+1} dummy leaves and w_{i+1} ordinary leaves on depth $i+1$, then the number of conceptional leaves at depth i is $c_i = (x_{i+1} + w_{i+1})/2$, and we consider from then on the binary distribution $(w_1, w_2, \ldots, w_{i-1}, w_i + c_i)$ for a tree of depth i. Now, in the function EnumCompVector, we enumerate all x_i satisfying the conditions of Lemma 14 at Line 5. We proceed with each such x_i at Line 6 with a subcomplement vector $\boldsymbol{y} = \boldsymbol{x} + (0, \ldots, 0, x_i)$. At Line 6, we check if we have obtained a completion with $\boldsymbol{y}$ at that point. If $\boldsymbol{y}$ is a complement vector of $\boldsymbol{w}$, we output $\boldsymbol{y}$ at Line 7 and return. If we can still increase a value in $\boldsymbol{y}$ to add dummy leaves (Line 10), we recursively call the procedure for the preceding depth $i - 1$. By Lemma 14, if we recurse, we either continue recursing or

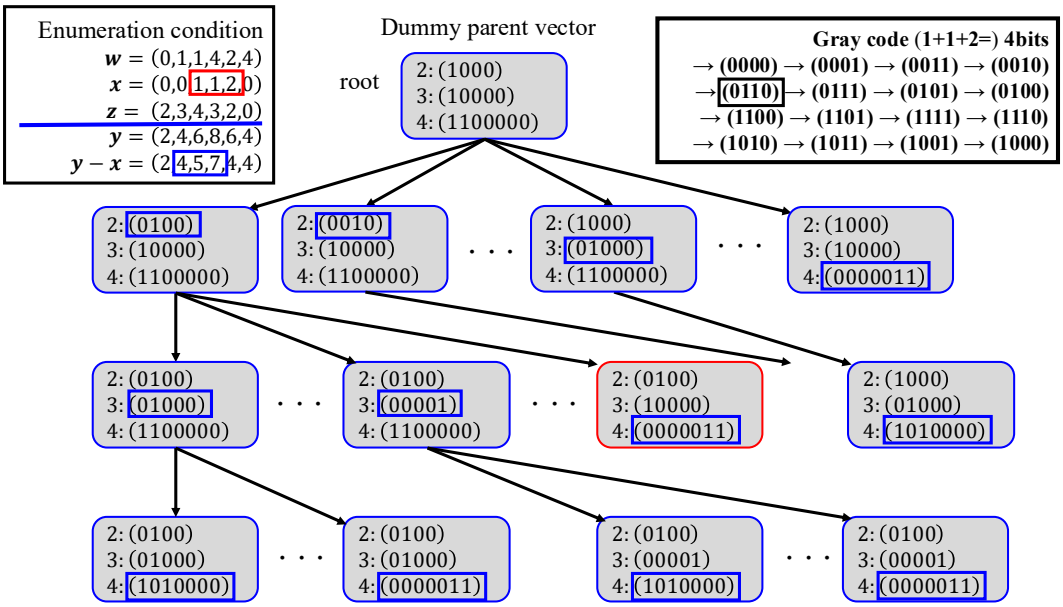


Figure 6 Enumeration of dummy parent configurations and Gray code, cf. Section 6.2. We start with $\boldsymbol{w}$ and $\boldsymbol{x}$ defined in Figure 5. The third row in the left upper box is a vector $\boldsymbol{z}$ with $z_i = \sum_{j=i+1}^{d} (w_j + x_j)/2^{j-i}$ such that $\nu_i = w_i + x_i + z_i$. We only have dummy leaves on the depths $3, 4, 5$ (those depths i for which $x_i > 0$ marked by a red rectangle around the entries of $\boldsymbol{x}$). Therefore, our dummy parent configuration considers only the depths $2, 3, 4$, for which we generate bit vectors for each node in the depicted enumeration tree. The lengths of the bit vectors are determined by $\vec{\nu} - \vec{x}$ by the entries i with $x_{i+1} > 0$, which are 4,5,7 in our case (blue rectangle in the upper left box). For each node in the enumeration tree we visit, we enumerate all Gray codes on the upper right box by visiting linearly all depicted nodes in this linked list. The Gray code has length $\sum_{i=1}^{d} x_i$ and assigns a dummy parent the role whether it has a left or a right dummy leaf.

turn a subcomplement vector into a complement vector. We can write each recursive call in the form of a tree, where each node is a call of EnumCompVector: the root is the initial call, and each leaf is a call that returns a complement vector. See Figure 5 for an example. Finally, since we can evaluate the checks in the pseudocode in $O(1)$ time, the delay between the output of two complement vectors is $O(d)$ time.

6.2 Enumeration of Dummy Leaves

Given the complement vector $\boldsymbol{x}$ for $\boldsymbol{w}$, there can be multiple ways to add the dummy leaves specified by $\boldsymbol{x}$ to a binary tree whose binary distribution is $\boldsymbol{w}$. Our task is to enumerate all these ways. For that, we follow the strategy of Section 5, where we enumerate the leaf/internal node configurations at depth i for full binary trees. However, unlike normal leaves, the sibling of a dummy leaf must be a normal leaf or an internal node. To facilitate checking this condition, we enumerate the internal node one level above the dummy leaf itself, rather than at the configuration of the dummy leaf. We call such an internal node the *dummy parent* of a dummy leaf. For each dummy parent, it suffices to enumerate the two possibilities of having its dummy leaf as its left or right child.

First, as in Section 5, suppose that we have a full-binary distribution $\boldsymbol{w}+\boldsymbol{x}$. From the proof of Corollary 5, the number of nodes at depth i is $\nu_i := g(i, \boldsymbol{w} + \boldsymbol{x}) = \sum_{j=i}^{d} (w_j + x_j)/2^{j-i}$.

Suppose δ is the smallest depth at which a dummy leaf appears for the first time. At depth δ, there are x_δ dummy leaves. By definition, there are x_δ dummy parents at depth $\delta - 1$. We specify these dummy parents by a bit vector of length $\nu_{\delta-1}$ with "1"s at x_δ positions.

In doing so, at depth $\delta-1$, we have already determined the dummy parents of the dummy leaves at depth δ, regardless of whether they have a dummy leaf as their left or right child, so the number of nodes that we can freely choose at depth δ is $\nu_\delta - x_\delta$. We again specify the dummy parents at depth δ and recurse. Therefore, we can express our selection of dummy parents by a $(\nu_\delta - x_\delta)$-length bit vector with "1"s in $x_{\delta+1}$ positions. By recursing on deeper levels, we can apply this technique on all depths, generalizing from δ to any $i \in [\delta..d]$. In what follows, we want to express the configuration of dummy parents on each level by a bit vector, for which we want to bound the size of ν_i.

▶ **Lemma 15.** *Each ν_i is at most $2\sum_{j=i}^{d} w_j$.*

Proof. From Lemma 14, $\nu_d = w_d + x_d \le 2w_d$. So there is an $i \in [0..d-1]$ such that the induction hypothesis $\nu_{d-k} \le 2\sum_{j=d-k}^{d} w_j$ holds for every $k \in [0..i]$. Then we obtain

$$\nu_{d-(i+1)} = w_{d-(i+1)} + \nu_{d-i}/2 + x_{d-(i+1)} \le 2w_{d-(i+1)} + \nu_{d-i} \le 2\sum_{j=d-(i+1)}^{d} w_j\,,$$

where we used Lemma 12 for $x_{d-(i+1)} \le \nu_{d-i}/2 + w_{d-(i+1)}$. ◀

Therefore, we can specify dummy parents with an $O(\sum_{j=i}^{d} w_j)$-length bit vector at each depth. We denote the sequence of bit vectors for specifying the dummy parent configuration by $\boldsymbol{D}$.

Furthermore, we express whether a dummy leaf at depth i is the left or right child of its parent by a bit vector of length x_i (for each configuration of the dummy parents at depth $i-1$, there are 2^{x_i} ways for the x_i dummy leaves at depth i to be attached). In total, we can express the dummy leaf configuration by a $(\sum_{i=1}^{d} x_i)$-length bit vector. Since $x_i \le \nu_i \le 2\sum_{j=i}^{d} w_j$ by the above analysis, this bit vector has length $O(d\sum_{i=1}^{d} w_i)$.

An enumeration of bit vectors are Gray codes [8], which have $O(1)$ delay because two subsequently output bit vectors differ by one bit. Here we use Gray codes to enumerate the dummy leaf configuration for each dummy parent configuration with constant delay.

To conclude, we can enumerate the configuration of dummy leaves in the following manner:

- Enumerate the configuration of the dummy parents with constant time delay, and output this configuration in $O(\sum_{i=1}^{d} w_i)$.
- Generate a Gray code of $\sum_{i=1}^{d} x_i = O(d\sum_{i=1}^{d} w_i)$ bits for each configuration of dummy parents, and enumerate the configuration of dummy leaves.

Figure 6 shows an example of a dummy parent enumeration tree and a Gray code for a binary distribution $\boldsymbol{w} = (0, 1, 1, 4, 2, 4)$ and its complement vector $\boldsymbol{x} = (0, 0, 1, 1, 2, 0)$.

6.3 Enumeration of the Leaf Configuration

Based on the previous discussions, at this point, we have determined the configuration of dummy parents $\boldsymbol{D}$ with x_i dummy leaves and x_{i+1} dummy parents of ν_i nodes at each depth i of the binary tree. Here, we assign roles to the remaining $\nu_i - x_i - x_{i+1}$ nodes as leaves and internal nodes having no dummy leaves as children. As before, we can specify the roles in

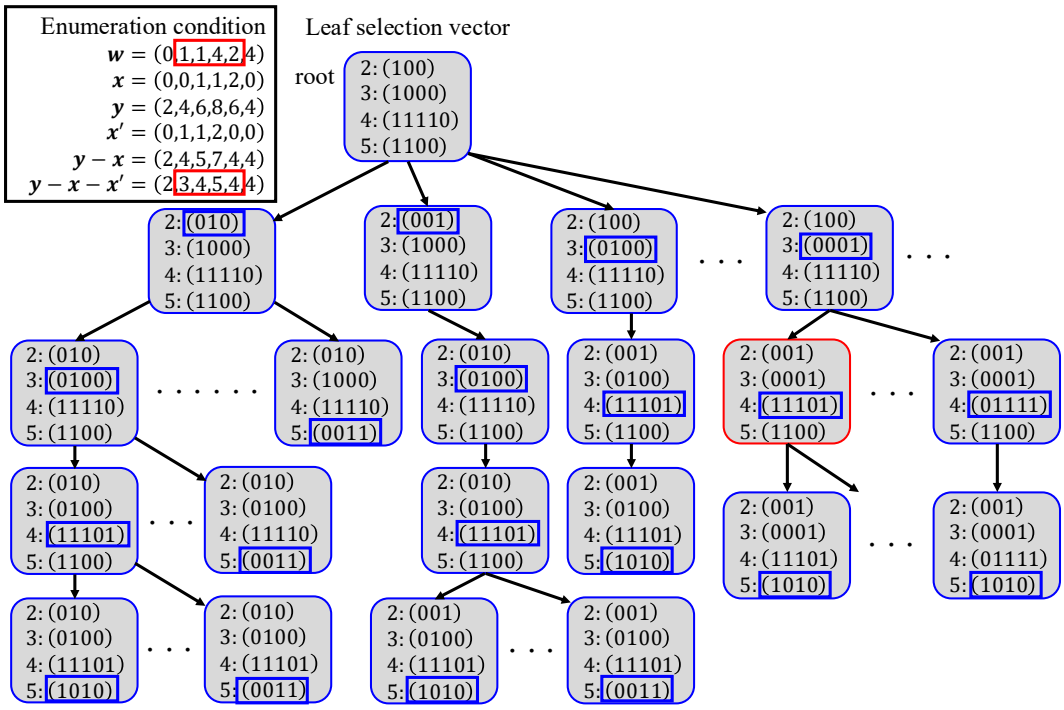


Figure 7 Enumeration of all leaf configurations, cf. Section 6.3. We follow the example of Figure 6. Here, $\boldsymbol{x}'$ denotes the vector $x'_i = x_{i+1}$. The number of leaves and internal nodes having no dummy leaves at depth i is $\nu_i - x_i - x_{i+1} = \nu_i - x_i - x'_i$. Since all nodes on depth 1 are internal nodes ($w_1 = 0$) and all nodes on depth d are leaves ($\nu_d = w_i$), the leaf selection vector only addresses the depths 2,3,4 (red rectangles in the left upper box).

the form of a bit vector of length $\nu_i - x_i - x_{i+1}$ with w_i "1"s. We call such a vector sequence a *leaf configuration vector sequence* and denote it by $\boldsymbol{L}$. Fixing one configuration of dummy leaves, we enumerate $\prod_{i=1}^{d} \binom{\nu_i - x_i - x_{i+1}}{w_i}$ combinations in the same way as in Section 5 to obtain all binary trees having that configuration of dummy leaves. We proceed in this order here despite that once $\boldsymbol{x}$ is determined, the number of dummy parents and leaves at each depth is fixed, so we can determine the leaf configuration vector independently of the dummy parent configuration. Figure 7 gives an example of a leaf configuration enumeration tree for a binary distribution $\boldsymbol{w} = (0, 1, 1, 4, 2, 4)$ and its complement vector $\boldsymbol{x} = (0, 0, 1, 1, 2, 0)$.

6.4 Overview of the Algorithm

We combine the above to enumerate binary trees using enumeration trees and Gray codes. For the sake of explanation, we enumerate the configuration of dummy leaves not directly after enumerating the configuration of dummy parents – we can take care of the dummy leaves at any time after we have determined the dummy parents. This also allows us to postpone the computation of the configuration of dummy leaves to the end.

We visualize our pipeline that ends at the generation of Gray codes following pointers from three nested enumeration tree traversals in Figure 8. There, we start with a top-down traversal of the leftmost tree T. Each leaf of T determines a complement vector $\boldsymbol{x}$. We perform the actual traversal by executing EnumCompVector. Fixing $\boldsymbol{x}$, we move to the neighboring trees $T^D(\boldsymbol{x})$ and $T^L(\boldsymbol{x})$ to enumerate all dummy parent configurations and all leaf configurations $\boldsymbol{D}$ and $\boldsymbol{L}$, respectively. First, we enter the enumeration tree $T^D(\boldsymbol{x})$. Each time we visit a new node in $T^D(\boldsymbol{x})$, we jump into $T^L(\boldsymbol{x})$ and traverse this tree. For each

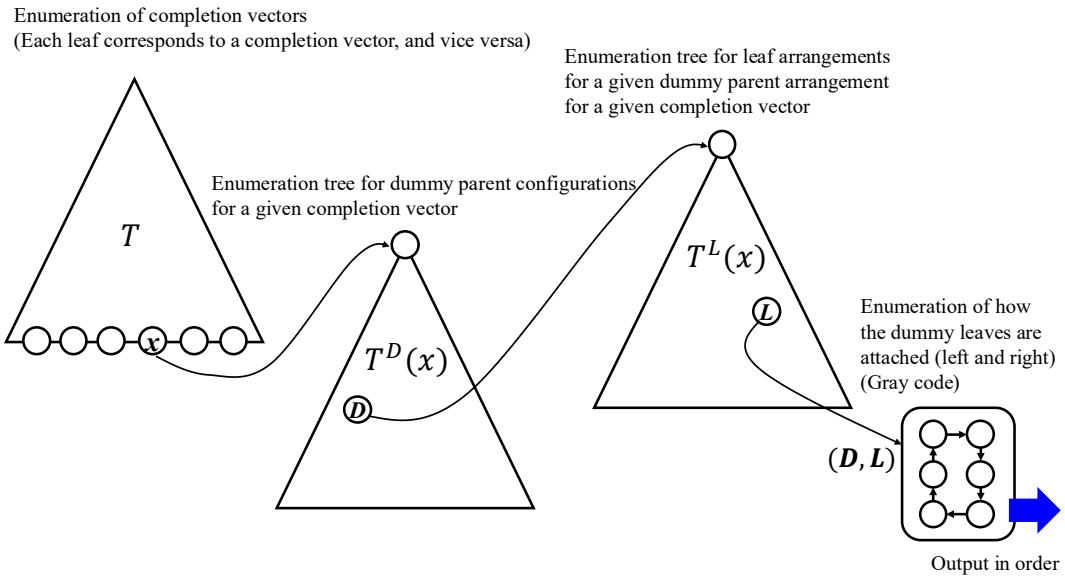


Figure 8 Layout of the algorithm enumerating all binary trees for a given binary distribution, cf. Section 6.4. An example of the leftmost tree T is given by Figure 5, and for the subsequent trees T^D and T^L by Figure 6 and Figure 7, respectively.

$T^L(\boldsymbol{x})$ node we visit, we have determined $(\boldsymbol{D}, \boldsymbol{L})$. Given $|x| = \sum_{i=1}^{d} x_i$, for each Gray code $\boldsymbol{g} \in \{0,1\}^{|x|}$ we enumerate, we finally output $(\boldsymbol{D}, \boldsymbol{L}, \boldsymbol{g})$. From the discussion in Section 5, we can determine and output the pair $(\boldsymbol{D}, \boldsymbol{L})$ with constant delay. Since we can also determine $\boldsymbol{g}$ with constant delay, we can output $(\boldsymbol{D}, \boldsymbol{L}, \boldsymbol{g})$ for a fixed $\boldsymbol{x}$ with constant delay. However, the delay in outputting $\boldsymbol{x}$ itself is $O(d)$, but this time bound is dominated by the time bound to output the first $(\boldsymbol{D}, \boldsymbol{L}, \boldsymbol{g})$ for $\boldsymbol{x}$. This leads to the following complexities.

▶ **Theorem 16.** *Given a binary distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$, the corresponding binary tree can be output in $O(d+n)$ time for the first tree and then output all trees in the form of differences with a size of $O(\sum_{i=1}^{d} w_i)$ with a delay of $O(1)$.*

While the complexities stated in Theorem 16 are in terms of $\boldsymbol{w}$, we create a full-binary tree whose binary distribution is $\boldsymbol{w} + \boldsymbol{x}$ for each completion vector $\boldsymbol{x}$. Nevertheless, $x_i \leq w_i$ for all $i \in [1..d]$, and therefore the number of nodes of this full-binary tree is asymptotically equal to the number of leaves of the tree we output.

Finally, an example of the binary tree corresponding our running example is given in Figure 9.

7 Extension to α-ary trees

In this section, we extend the results of the previous section to general α-ary trees for a constant $\alpha \geq 2$. Like for binary trees ($\alpha = 2$), we first define the notion of an α-ary distribution.

7.1 Properties of full α-ary trees and their enumeration

We call a vector of integers $\boldsymbol{w} = (w_1, \ldots, w_d)$ a *(full) α-ary distribution* if there is a (full) α-ary tree that has w_i leaves on depth i for every $i \in [1..d]$. For a nonnegative integer vector $\boldsymbol{w} = (w_1, \ldots, w_d)$, we define the function $f_\alpha(\boldsymbol{w}) = \sum_{i=1}^{d} w_i/\alpha^i$, which is a generalization of f with $f_2 = f$. The problem we want to tackle in this section can be stated as follows.

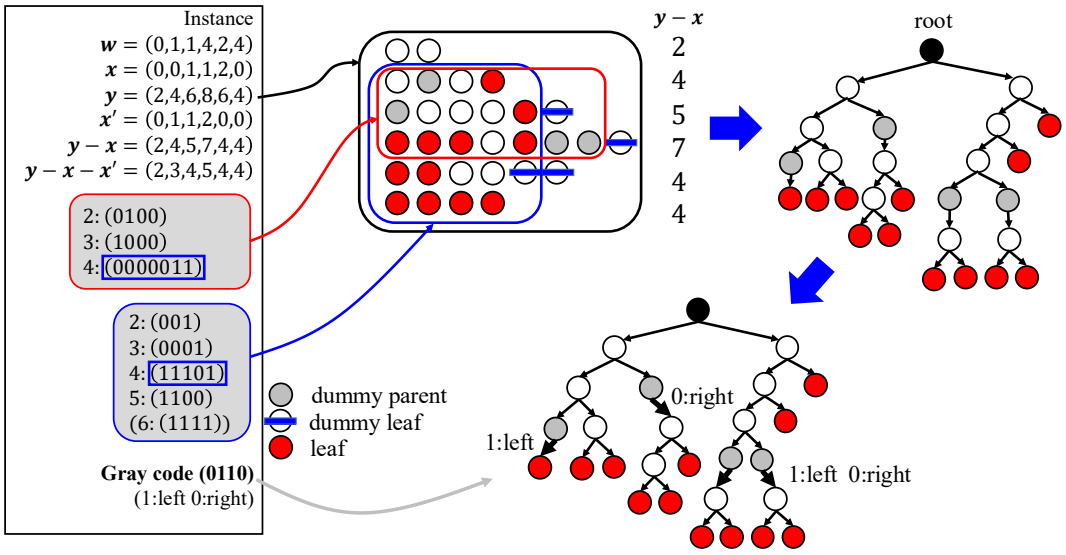


Figure 9 Binary tree corresponding to a vector representation returned by the algorithm whose complexities are stated in Theorem 16. The left box depicts a node of $T^D(\boldsymbol{x})$ (red, cf. Figure 6) and $T^L(\boldsymbol{x})$ (blue, cf. Figure 7). Informally, the red node specifies with a "1" the placement of a dummy parent, and the blue node with a "1" the placement of a (normal) leaf. Finally, a Gray code determines for each dummy parent whether its normal leaf is a left child or a right child.

▶ **Problem 17.** *Given a (full) α-ary distribution $\boldsymbol{w}$, enumerate all (full) α-ary trees whose α-ary distribution is $\boldsymbol{w}$.*

Like in the case of binary trees, the following holds.

▶ **Lemma 18** (Kraft's inequality)**.** *Let $\alpha \geq 2$. A nonnegative integer vector $\boldsymbol{w} = (w_1, \ldots, w_d)$ that satisfies $w_d > 0$ is a α-ary distribution if and only if $f_\alpha(\boldsymbol{w}) \leq 1$.*

▶ **Corollary 19.** *Let $\boldsymbol{w} = (w_1, \ldots, w_d)$ be a nonnegative integer vector satisfying $w_d > 0$ and is a full α-ary distribution. Then, $g_\alpha(j, \boldsymbol{w}) := \sum_{i=j}^{d} w_i/\alpha^{i-j}$ is a multiple of α, and represents the number of nodes at depth j of the α-ary tree having $\boldsymbol{w}$ as its α-ary distribution.*

▶ **Theorem 20.** *Let $\alpha \geq 2$. Given a full α-ary distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$ satisfying $w_d > 0$, the number of all full α-ary trees that have $\boldsymbol{w}$ as their α-ary distribution is*

$$\prod_{i=1}^{d-1} \binom{\sum_{j=i}^{d} \frac{w_j}{\alpha^{j-i}}}{w_i}.$$

From these properties, the enumeration of full binary trees can be extended to the enumeration of full α-ary trees. For that, we observe that the approach for enumerating full binary trees analyzed in Section 5 is independent of the degree of the tree, since the representation only considers the configuration of internal nodes and leaves per depth. Therefore, we can generalize this approach in a straightforward manner by replacing f and g with f_α and g_α, respectively, and updating the values of $\eta_j := g_\alpha(j, \boldsymbol{w})$. We obtain the following result based on an extension of Theorem 11:

▶ **Theorem 21.** *Given a full α-ary distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$, the corresponding full α-ary tree can be output in $O(n)$ time for the first tree and then output all trees in the form of differences with a size of $O(m)$ with a delay of $O(1)$ in $O(m)$ time, where $n = \sum_{i=1}^{d} w_i$ and $m = \max\{w_i \mid i \in [1..d]\}$.*

7.2 Enumeration of α-ary trees

We can carry out the enumeration of α-ary trees in the same way as binary trees, for which we had three levels of enumeration: for the complement vectors, the dummy leaves, and the actual leaves. Here, we proceed in the same way.

7.2.1 Enumeration of full α-ary tree complement vectors

Suppose we have an α-ary distribution $\boldsymbol{w}$ with $f_\alpha(\boldsymbol{w}) < 1$. We fix one α-ary tree corresponding to $\boldsymbol{w}$ and add dummy leaves to all internal nodes with fewer than α children to make it a full α-ary tree. In this case, let the number of dummy leaves added at depth i be x_i, and consider the vector $\boldsymbol{x} = (x_1, \ldots, x_d)$. By construction, we obtain that $\sum_{i=1}^{d}(w_i + x_i)/\alpha^i = 1$. We call the vector $\boldsymbol{x}$ constructed in this way the *complement vector* of $\boldsymbol{w}$, and we call $\boldsymbol{w} + \boldsymbol{x}$ the *full α-ary tree complement* of $\boldsymbol{w}$. For such $\boldsymbol{x}$, a nonnegative integer vector $\boldsymbol{x}'$ is called a *subcomplement vector* of $\boldsymbol{w}$ if $\boldsymbol{x} \geq \boldsymbol{x}'$. Similarly to binary trees in Lemma 13, the following holds.

▶ **Lemma 22.** *Given an α-ary distribution $\boldsymbol{w}$ with $f_\alpha(\boldsymbol{w}) < 1$, there exists a complement vector of $\boldsymbol{w}$.*

Generalizing Lemma 14, the following is true.

▶ **Lemma 23.** *A nonnegative integer vector $(0, \ldots, 0, y_d)$ is a subcomplement vector of the complement vector $\boldsymbol{y} = (y_1, \ldots, y_d)$ of an α-ary distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$ if and only if $y_d \equiv w_d \pmod{\alpha}$, $y_d \in [0..(\alpha-1)w_d]$, and $f_\alpha(\boldsymbol{w}) + y_d/\alpha^d \leq 1$.*

Using these lemmas, we can generalize Algorithm 1 to Algorithm 2 to enumerate the complement vector of an α-ary distribution.

Algorithm 2 Subroutine for enumerating full α-ary tree complement vectors for $\boldsymbol{w}$.

```
1:  procedure EnumCompVector-α(W, x, i, c)      ▷ W: value of f_α, x: subcomplement vector, i: depth i
2:          ▷ Determine the number of dummy leaves to add below depth i (w is a global variable)
3:      x_i ← w_i + c mod α                      ▷ Initialization. Align with α according to Lemma 23
4:      while x_i ≤ w_i + c  and  W + x_i/α^i ≤ 1 do      ▷ For all x_i satisfying x_i ≤ w_i + c
5:          y ← x + (0, ..., 0, x_i, 0, ...)             ▷ x meets the requirements of Lemma 23
6:          if W + x_i/α^i = 1 then
7:              Output y                                 ▷ Output as a full α-ary tree complement
8:              return
9:          else
10:             EnumCompVector(W + x_i/α^i, y, i − 1, (x_i + w_i)/α)   ▷ recurse on depth i − 1
11:         end if
12:         x_i ← x_i + α
13:     end while
14:     return
15: end procedure
```

7.2.2 Enumeration of the configuration of dummy leaves

Suppose that we have determined the complement vector $\boldsymbol{x}$ of $\boldsymbol{w}$. The next step is to determine the configuration of dummy leaves, which is (like for binary trees) not unique, in general. However, in the case of α-ary trees, among all siblings of a dummy leaf there must be at least one *normal node*, i.e., an internal node or a leaf that is not dummy. Therefore, we cannot infer from a dummy parent its number of dummy leaves, and thus specifying the configuration of dummy nodes with a Gray code like for binary trees seems not possible. Instead, we enumerate the number of dummy leaves that each dummy parent has from 0 to $\alpha - 1$, and then enumerate the configuration of dummy leaves corresponding to that number. This strategy is slightly more complicated than in the case of binary trees because of this additional intermediate step, but the calculation time is asymptotically absorbed by the time spent for determining one complement vector. Having computed the complement vector, the following steps are analogous to the case of binary trees, for which we obtain constant time delay per output.

We enumerate all possible ways to place the dummy leaves at each depth $\delta \in [1..d]$ by choosing a combination of the configuration of the dummy leaves at depth δ. For that, we first specify how many dummy leaves each parent has at depth $\delta - 1$. According to Corollary 19, in the complemented tree corresponding to the binary distribution $\boldsymbol{w} + \boldsymbol{x}$ with the added dummy leaves, there are $\eta_\delta = g_\alpha(\delta, \boldsymbol{w})$ nodes at depth δ (i.e., there are η_δ/α parents at depth $\delta - 1$). We assign each parent a rank from $[1..\eta_\delta/\alpha]$ in order from left to right, and we say that $p_i^{(\delta)}$ is the number of dummy leaves that the i-th parent has. Since each parent must contain at least one normal node, $p_i^{(\delta)} \in [0..\alpha - 1]$, and the number of dummy leaves at depth δ is x_δ, so $\boldsymbol{p}^{(\delta)} = (p_1^{(\delta)}, \ldots, p_{\eta_\delta/\alpha}^{(\delta)})$ satisfies

$$\sum_{i=1}^{\eta_\delta/\alpha} p_i^{(\delta)} = x_i \,. \tag{1}$$

We use a known result for enumerating all distinct integer tuples $(p_1^{(\delta)}, \ldots, p_{\eta_\delta/\alpha}^{(\delta)})$ with $p_i^{(\delta)} \in [0..\alpha - 1]$ satisfying Equation (1) with constant delay [16]. For each such integer tuple, we enumerate the configuration of dummy leaves corresponding to each $\boldsymbol{P}$. After that, all α-ary trees can be enumerated with constant time delay as in the case of binary trees.

8 Conclusion

We have addressed the problem to efficiently enumerate trees whose numbers of leaves on each depth are given by a query vector. Specifically, for any leaf distribution $\boldsymbol{w} = (w_1, \ldots, w_d)$ of (full) binary or (full) α-ary trees, we obtain the following time complexities. First, we can output the first tree in $O(d + n)$ time. We can output every subsequent tree, after constant delay, in $O(m)$ time by encoding this tree as a difference to a previous one using $O(m)$ words. Here, $n = \sum_{i=1}^{d} w_i$ is the total number of nodes and $m = \max\{w_i \mid i \in [1..d]\}$ is the maximal number of nodes at any depth. These results hold for both full binary trees and full α-ary trees with $\alpha \geq 2$, and extend to general binary trees with analogous enumeration guarantees.

As future work, we want to determine the exact number of full binary trees and full α-ary trees with distribution $\boldsymbol{w}$ by a closed-form product formulas involving binomial coefficients. We further strive to give practical implementations of our proposed enumeration algorithms.

References

1 Masahiro Baba, Hirotaka Ono, Kunihiko Sadakane, and Masafumi Yamashita. A succinct representation of a full binary tree. Technical Report 0, Record of Joint Conference of Electrical and Electronics Engineers in Kyushu, 2009. doi:10.11527/jceeek.2009.0.59.0.

2 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. doi:10.1007/s00453-004-1146-6.

3 Terry Beyer and Sandra Mitchell Hedetniemi. Constant time generation of rooted trees. *SIAM J. Comput.*, 9(4):706–712, 1980. doi:10.1137/0209055.

4 Michael Buro. On the maximum length of Huffman codes. *Inf. Process. Lett.*, 45(5):219–223, 1993. doi:10.1016/0020-0190(93)90207-P.

5 Jeremy Chizewer, Stephen Melczer, J. Ian Munro, and Ava Pun. Enumeration and succinct encoding of AVL trees. In *Proc. AofA*, volume 302 of *LIPIcs*, pages 2:1–2:12, 2024. doi:10.4230/LIPICS.AOFA.2024.2.

6 Nachum Dershowitz and Shmuel Zaks. Enumerations of ordered trees. *Discret. Math.*, 31(1):9–28, 1980. doi:10.1016/0012-365X(80)90168-5.

7 Sen-Peng Eu, Seunghyun Seo, and Heesung Shin. Enumerations of vertices among all rooted ordered trees with levels and degrees. *Discret. Math.*, 340(9):2123–2129, 2017. doi:10.1016/J.DISC.2017.04.007.

8 Frank Gray. Pulse code communication. US Patent 2632058, 1953.

9 Roberto Grossi. Enumeration of paths, cycles, and spanning trees. In *Encyclopedia of Algorithms*, pages 640–645. Springer, 2016. doi:10.1007/978-1-4939-2864-4_728.

10 Guy Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

11 Leon Gordon Kraft. A device for quantizing, grouping, and coding amplitude-modulated pulses. Master's thesis, Massachusetts Institute of Technology, 1949.

12 Shin-Ichi Nakano and Takeaki Uno. Constant time generation of trees with specified diameter. In *Proc. WG*, volume 3353 of *LNCS*, pages 33–45, 2004. doi:10.1007/978-3-540-30559-0_3.

13 Frank Ruskey and Andrzej Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11(1):68–84, 1990. doi:10.1016/0196-6774(90)90030-I.

14 Ivan Stojmenovic. A simple systolic algorithm for generating combinations in lexicographic order. *Computers & Mathematics with Applications*, 24(4):61–64, 1992.

15 Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms. Technical Report NII-2003-004E, NII Technical Report, 2003. doi:10.20736/0000000385.

16 Timothy R. Walsh. Loop-free sequencing of bounded integer compositions. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 33:323–345, 2000.

17 Katsuhisa Yamanaka, Yota Otachi, and Shin-Ichi Nakano. Efficient enumeration of ordered trees with k leaves. *Theor. Comput. Sci.*, 442:22–27, 2012. doi:10.1016/j.tcs.2011.01.017.

18 Shmuel Zaks. Lexicographic generation of ordered trees. *Theor. Comput. Sci.*, 10:63–82, 1980. doi:10.1016/0304-3975(80)90073-0.

Appendix

Table 1 Symbols used in this article.

symbol	meaning
$\boldsymbol{w}$	distribution
d	maximal depth and dimension of $\boldsymbol{w}$
n	number of leaves
m	maximum value in $\boldsymbol{w}$
$\boldsymbol{x}$	complement vector
$\boldsymbol{y}$	subcomplement vector
$f(\boldsymbol{w})$	$\sum_{i=1}^{d} w_i/2^i$ weight function
$\boldsymbol{\eta}$	$\eta_i := g(i, \boldsymbol{w})$
ν_i	$g(i, \boldsymbol{w} + \boldsymbol{x})$
$g(i, \boldsymbol{w})$	$\sum_{j=i}^{d} w_i/2^{j-i}$ for $i \in [1..d]$

On String and Graph Sanitization

Giulia Bernardini
University of Milan, Italy

Huiping Chen
University of Birmingham, UK

Grigorios Loukides
King's College London, UK

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Data sanitization is a process that conceals sensitive patterns from a given dataset. A secondary goal is to not severely harm the utility of the underlying data along this process. We survey some recent advancements on two related data sanitization topics: string and graph sanitization. In particular, we highlight the important contributions of our friend Prof. Roberto Grossi along this journey.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases data privacy, data sanitization, string algorithms, graph algorithms

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.9

Category Research

Funding *Giulia Bernardini*: Member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).
Solon P. Pissis: Supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

Acknowledgements We want to thank Prof. Roberto Grossi for this journey (and more). We would also like to thank the editors for providing us with the opportunity and pleasure to write this paper.

1 Introduction

Disseminating string data is often performed to support applications such as location-based service provision or DNA sequence analysis. However, this dissemination may reveal sensitive patterns that represent some form of private or confidential knowledge (e.g., trips to STD clinics from a string representing a user's visited location or a genetic disorder from a string representing an individual's DNA sequence). To prevent sensitive patterns from being exposed, string data must often undergo *sanitization* [3, 5], a process that conceals sensitive patterns from the data, without, however, severely harming the utility of the data.

Another fundamental data type is graphs. Graphs represent data in domains such as social networks, communication networks, or the Web. In these domains, users form dense or cohesive groups, referred to as *communities*. For example, many of the users who belong to the same political group may also be connected with each other on a social network, creating a community in the graph representing this social network. There is much work in the literature on mining communities from a graph [10]. Our work in [8] studied how to break such communities from a given graph without, however, severely harming the utility of the graph. This process can also be viewed as sanitization, in the sense that communities can convey private or confidential information, such as membership in a political group.

The authors feel extremely privileged to have worked with Prof. Grossi on string and graph sanitization. Our collaboration with Prof. Grossi on these two areas started in December 2018, when the first author was a Ph.D. student at the University of Milano-Bicocca, the second author was a Ph.D. student supervised by the third author at King's College London, and the last author had recently moved from the latter institution to CWI.

Prof. Grossi has made key contributions in the topics of string and graph sanitization. In addition, he shaped not only the definitions of these problems, the hardness results, and the algorithms to solve them but also the practical implementations of the algorithms. In the following, we summarize the most important of these contributions.

In the first step of string sanitization (sanitizing sensitive patterns), Prof. Grossi had the idea of using a special letter to conceal the occurrences of the sensitive patterns in the input string. In particular, this led to a novel string transformation that preserves the order and frequencies of all k-mers of the string. This transformation is very different from existing approaches that deleted [12] or permuted [13] letters of the string, and led to the first approach that provided utility guarantees. Prof. Grossi also played a key role in developing optimal algorithms that employ the aforementioned transformation. In the second step of string sanitization (replacing the occurrences of the special letter), Prof. Grossi came up with an ingenious NP-hardness proof of the problem. This constituted an important step in our work on string sanitization as it opened the way to design efficient fixed-parameter tractable algorithms to tackle the problem; see Section 2.

In graph sanitization, Prof. Grossi had an innovative idea that helped us evaluate the effectiveness of our heuristic algorithms on larger datasets compared to those that our exact algorithm could handle. Specifically, he suggested using a lower bound on the value of the optimal solution that could also be computed efficiently and then comparing the lower bound value with the value output by a heuristic. He also devised an algorithm to compute the lower bound. If the lower bound and the value output by a heuristic are reasonably close, then the heuristic is good because the optimal value lies between them; see Section 3.

Beyond his inestimable contributions to research, Prof. Grossi set an example with how he interacts with others; always with respect, kindness, and true interest and willingness to help. Also, the way Prof. Grossi cheered us up when the results of the work were not as expected was crucial to continue and improve not only the results, but also ourselves.

2 String Sanitization

2.1 The Model: Combinatorial String Dissemination

The Combinatorial String Dissemination (CSD) model was introduced by Bernardini et al [3]. In this model, we have a string W that we would like to disseminate for analysis. Toward effective dissemination, a dissemination that achieves a *good trade-off* between *privacy* and *utility*, we need to specify a set of privacy-related constraints and a set of utility-related properties, to then determine the best possible sequence of edit operations to be applied to W so that the utility-related properties are satisfied subject to the privacy-related constraints.

A specific CSD setting that has been considered by Bernardini et al [3] is the following. The set of constraints (C1) is defined using an integer $k > 0$ and a set of length-k strings, known as the set of *sensitive patterns*; in particular, it consists in strictly forbidding the occurrence of any sensitive pattern in W', the string to be constructed from W. The set of properties is defined using two widely used string properties. The first one (P1) says that the *order* of occurrence of the length-k non-sensitive patterns is preserved in W'; the second (P2) says that the *frequency* of the length-k non-sensitive patterns is also preserved in W'.

This model leads to interesting combinatorial problems on strings [4, 3, 15, 6, 9, 5, 16, 2]. We define a few such problems in the following subsections and briefly describe optimal algorithms for solving them or solid evidence as to why they are unlikely to be solved exactly.

2.2 Sanitizing Sensitive Patterns

In the TFS (Total order, Frequency, Sanitization) problem, we are given a string $W = W[1] \dots W[n]$ of length n over an alphabet Σ, and we are asked to output a *shortest* string $X := W'$ that satisfies C1, P1, and P2. The following example shows a TFS instance.

▶ **Example 1.** Let $W = \texttt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\{\texttt{aaaa}, \texttt{baaa}, \texttt{bbaa}\}$. Then, $X = \texttt{aabaa\#aaababbba\#baab}$, for some letter $\texttt{\#} \notin \Sigma$.

(Note that since $\texttt{\#} \notin \Sigma$, one could use the occurrences of # in X to learn about potential occurrences of sensitive patterns in W; see Section 2.3.) The following result is known.

▶ **Theorem 2** ([3]). *The length of X is in $\Theta(kn)$. Given the set of occurrences of sensitive patterns in W, there exists an algorithm that solves TFS in the optimal $\mathcal{O}(kn)$ time.*

The algorithm proceeds by reading W from left to right and constructs X, which is initially empty. When the length-k substring S read is non-sensitive, it merges it with X. Otherwise, it applies two rules. In the first rule (R1), given S, it constructs the string:

$$S' = S[1] \dots S[k-1]\texttt{\#}S[2] \dots S[k].$$

Note that the letter # is a special letter that does not belong to the alphabet Σ of W.

▶ **Example 3.** For $S = \texttt{baaa}$ and $k = 4$, R1 constructs string $S' = \texttt{baa\#aaa}$.

In the second rule (R2), given S' of length $2(k-1)+1$ obtained from R1, the algorithm tries to check if a shorter string S' is possible. In particular, when the $k-1$ letters before # are the same as the $k-1$ letters after it, we remove the # and the $k-1$ subsequent letters. One can easily notice that the above edit operations on S' will violate neither P1 nor P2.

▶ **Example 4.** Let $S = \texttt{aaaa}$ and $k = 4$. R1 will construct string $S' = \texttt{aaa\#aaa}$. By applying R2, we get $S' = \texttt{aaa}$, which is a string of length $k - 1 = 3$.

Instead of S, the string S', produced by rules R1 and R2, is merged with X.

▶ **Example 5.** Consider the fragment $F = \texttt{baaaa}$ of $W = \texttt{aa}\underline{\texttt{baaaa}}\texttt{babbbaab}$, with $k = 4$, and the set of sensitive patterns $\{\texttt{aaaa}, \texttt{baaa}, \texttt{bbaa}\}$. For $S = \texttt{baaa}$, R1 will construct string $S' = \texttt{baa\#aaa}$ resulting in fragment $F' = \texttt{baa\#aaaa}$. We will then need to deal with $S = \texttt{aaaa}$. R1 will construct string $S' = \texttt{aaa\#aaa}$ to replace it. However, applying R2, we get $S' = \texttt{aaa}$, which is shorter. Thus $F = \texttt{baaaa}$ will be replaced by fragment $F'' = \texttt{baa\#aaa}$.

By carefully implementing R1 and R2, we can construct string X in $\mathcal{O}(kn)$ time. As for the length of X in the worst case, it suffices to construct the de Bruijn sequence of order $k-1$ as the input string W, and assign every other consecutive length-k substring to be a sensitive pattern. Recall that a *de Bruijn sequence* of order k' over an alphabet Σ is a string in which every possible length-k' string over Σ occurs exactly once as a substring. Thus, for such an input string W, X must be of length $\Omega(kn)$ because R2 cannot be applied.

The question that now arises naturally is whether we can hope for a shorter string W' by relaxing the constraints of the TFS problem. In response, Bernardini et al. [3] introduced the following related problem. In the PFS (Partial order, Frequency, Sanitization) problem,

we are given a string W of length n, and we are asked to output a *shortest* string $Y := W'$ that satisfies C1, Π1, and P2, where the Π1 property is a relaxation of the P1 property: it replaces the total order by a *partial* order saying that the order of occurrence of the length-k non-sensitive patterns in maximal fragments with no # occurring remains the same.

▶ **Example 6.** Let $W =$ aabaaaababbbaab, $k = 4$, and let the set of sensitive patterns be {aaaa, baaa, bbaa}. Then, $Y =$ aaababbba#aabaab, for some letter # $\neq$ a and # $\neq$ b.

The following result is known.

▶ **Theorem 7** ([3]). *Given the set of occurrences of sensitive patterns in W, there exists an algorithm that solves PFS in the optimal $\mathcal{O}(n + |Y|)$ time.*

(Note that the $|Y|$ term in Theorem 7 accounts for the fact that Y can be longer than W.) We briefly explain how we can arrive at Theorem 7. Consider that we have (a representation of) string X obtained via TFS. If any two maximal #-free fragments of X, called *blocks*, overlap by $k - 1$ letters, then we can further apply R2 while still satisfying Π1 and P2.

▶ **Example 8.** Recall that for $W =$ aabaaaababbbaab, $k = 4$, and the set of sensitive patterns {aaaa, baaa, bbaa}, the TFS problem outputs $X =$ aa<u>baa</u>#aaababbba#<u>baa</u>b. Observe that blocks aa<u>baa</u> and <u>baa</u>b overlap by $k - 1$ letters, so we can apply R2 further.

Now, it might seem that we must solve the famously NP-hard Shortest Common Superstring (SCS) problem [11] on the set of blocks originating from X to construct Y. Fortunately, this is not the case as the *allowed overlaps are of fixed length $k-1$*. To solve this problem, we map the prefix and suffix of length $k - 1$ of every block to an integer identifier. We can then ignore the middle part of each block, thus transforming every block to a string of length 2. After this reduction, we can solve SCS on a collection of two-letter strings in linear time [3].

2.3 Hide and Mine

Say we have completed the task in Section 2.2: sanitizing the occurrences of all sensitive patterns by means of a special letter that we represent by #. Although sensitive patterns are no longer present in the sanitized sequence, the occurrences of # reveal the locations where they used to occur. It is not hard to imagine that a malicious adversary could use this information to try and reconstruct the concealed information from the context surrounding the #s. This is because, although the adversary cannot know precisely which special letter has been used in the sanitization process, they know that it is (by definition) not part of the original alphabet, and thus it can be relatively easily identified. Imagine that the dataset consists of a sequence of locations: # could be a non-existing location or a location far away from those in the original sequence. In a sequence of online purchases, # could be an item not sold in a certain online store; in a genomic sequence, # is a letter that does not correspond to any nucleotide base. In all such examples, it is not hard for an attacker to identify # with a simple scan of the database and thus partially retrieve the concealed information.

Therefore, a complete sanitization pipeline should eventually replace the occurrences of # with letters of the original alphabet. This immediately poses a problem: each replacement introduces k new "spurious" occurrences of length-k patterns over the original alphabet, which might influence downstream analysis results based on the frequency of length-k substrings. This kind of analysis extracts length-k substrings that appear at least τ times, for some fixed integer τ, assuming that these τ-frequent patterns carry important information.

The natural problem arising is to decide whether it is possible to replace every occurrence of # with some letter from the original alphabet such that: (i) no sensitive pattern is reintroduced; and (ii) the set of τ-frequent non-sensitive patterns before and after sanitization is the same. We term this problem *Hide and Mine* (HM). Intuitively, HM is hard because replacements are not independent of one another: making a choice at one position could prevent feasible choices from being available at other positions, while this would not have been the case when making a different choice in the first place. But how does one prove its hardness?

The answer came from a brilliant idea by Prof. Grossi: we should reduce from the famous *Bin Packing* [14] problem! More precisely, we should reduce from a variant of the problem called *Unique-Weights Bin Packing* (UWBP). The reduction is far from being trivial: we will provide a friendly, informal description of the proof in the rest of this section.

Reducing UWBP to Hide and Mine

The UWBP problem asks us to decide whether N items can be packed into M bins, each bin with a fixed capacity B that cannot be exceeded: each item i has a given weight w_i, and no two items have the same weight.

▶ **Example 9.** Consider the instance $\mathcal{I}^+$ of UWBP consisting of $M = 3$ bins, each with capacity $B = 7$, and the $N = 5$ items with weights $w_1 = 2, w_2 = 3, w_3 = 4, w_4 = 5, w_5 = 6$. The answer to $\mathcal{I}^+$ is positive: it suffices, for instance, to pack item 5 in one bin, items 2 and 3 in another bin, and items 1 and 4 in a third bin. Now consider a second instance $\mathcal{I}^-$ with the same bins and the same number of items as $\mathcal{I}^+$, but this time the weights are $w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6, w_5 = 7$. The answer to $\mathcal{I}^-$ is negative: there is no way to distribute all the 5 items into the 3 bins without exceeding the capacity of any bin.

The problem UWBP is *strongly* NP-complete, that is, it remains NP-complete even when all of its parameter values (i.e., the bin capacity and the item weights) are polynomially bounded in N, the number of items.

Prof. Grossi's intuition was that bins could be modelled by unique alphabet letters, string gadgets could be constructed to model the event "item j is packed into bin i" and to force each item to be packed in at least one bin, the frequency threshold τ could be tuned to encode the bin capacity B, and the set of sensitive patterns could be chosen to avoid situations in the HM instance that do not correspond to any scenario in the original UWBP instance. Let us now present this idea in detail (but informally) using instance $\mathcal{I}^+$ of Example 9. The alphabet of the HM instance we construct from $\mathcal{I}^+$ consists of the three letters x, y, \$, and a unique letter for each bin: we will denote b_1, b_2 and b_3 the letters corresponding to bin number 1, 2, and 3, respectively, and use # to denote the special character to be replaced in the input string. The value of k (length of patterns) of HM is chosen to be the maximum weight of the input items plus 3: in the case of $\mathcal{I}^+$, we have $k = 6 + 3 = 9$.

Item-in-bin Gadgets

The first class of gadgets consists of a string t_{ij} for each bin i and each item j, containing a single occurrence of #. Each of these string gadgets is paired with a set of sensitive patterns designed in such a way that only two replacements are possible: the first one models the event "pack item j into bin i" by introducing w_j occurrences of a length-k substring specific to bin i; the second one naturally corresponds to item j not being packed in bin i. These string gadgets each have a length of $2k - 1$ and are of the following form:

$$t_{ij} = b_i \underbrace{x \dots x}_{k-1-w_j} \underbrace{b_i \dots b_i}_{w_j-1} \# \underbrace{b_i \dots b_i}_{k-1}$$

▶ **Example 10.** Consider instance $\mathcal{I}^+$ of Example 9. The gadget t_{13}, modelling the possibility of packing item 3 in the first bin, is the string $t_{13} = b_1xxxxb_1b_1b_1\#b_1b_1b_1b_1b_1b_1b_1b_1$. Four length-$k$ strings, with $k = 9$, are added to the set of sensitive patterns to rule out the possibility of replacing # with $b_2, b_3, \$$ or y in t_{13}; namely, $b_2b_1b_1b_1b_1b_1b_1b_1b_1$, $b_3b_1b_1b_1b_1b_1b_1b_1b_1$, $b_1\$b_1b_1b_1b_1b_1b_1b_1$, and $b_1yb_1b_1b_1b_1b_1b_1b_1$. It should be clear that there are only two replacement options: replacing # by x, corresponding to the choice not to pack the item 3 in the first bin, and replacing it by b_1, corresponding to the opposite choice. Replacement of # with b_1 introduces $w_3 = 4$ occurrence of the string $b_1b_1b_1b_1b_1b_1b_1b_1b_1$, specific to bin 1. The gadget t_{11}, modelling the possibility of packing item 1 in the first bin, is the string $t_{11} = b_1xxxxxxb_1\#b_1b_1b_1b_1b_1b_1b_1b_1$; the associated sensitive patterns are the same as t_{13}. Replacing # with b_1 introduces $w_1 = 2$ occurrences of $b_1b_1b_1b_1b_1b_1b_1b_1b_1$. The gadget t_{23}, modelling the possibility of packing item 3 in the second bin, has the same form as t_{13}, except b_1 is replaced by b_2; the associated sensitive patterns are $b_1b_2b_2b_2b_2b_2b_2b_2b_2$, $b_3b_2b_2b_2b_2b_2b_2b_2b_2$, $b_2\$b_2b_2b_2b_2b_2b_2b_2$ and $b_2yb_2b_2b_2b_2b_2b_2b_2$, leaving b_2 and x as the only possible replacements.

The other gadgets are obtained similarly to those in Example 10; see [5] for the formal definitions.

Each-item-in-some-bin Gadgets

The second class of gadgets consists of a string u_{ij} for each bin i and each item j: these string gadgets also contain an occurrence of # each and are paired with some sensitive patterns. The role of these gadgets is to ensure that each item $\bar{j}$ is packed in some bin, corresponding to replacing # with b_i in at least one of the gadgets $t_{i\bar{j}}$. The latter is enforced by designing $u_{i\bar{j}}$ and the corresponding sensitive patterns so that: (i) # can only be replaced by either x or y; (ii) replacing # by x in $u_{i\bar{j}}$ introduces an occurrence of a length-k string that is also introduced when replacing # by x in $t_{i\bar{j}}$; and choosing the frequency threshold τ in such a way that (iii) if # is replaced by x in $t_{i\bar{j}}$, then # in $u_{i\bar{j}}$ must be replaced by y; and (iv) # cannot be replaced by y in $u_{i\bar{j}}$ for all i, as otherwise, the frequency threshold is exceeded. These string gadgets each have a length of $2k - 1$ and are of the following form:

$$u_{ij} = b_i \underbrace{x \dots x}_{k-1-w_j} \underbrace{b_i \dots b_i}_{w_j-1} \# \underbrace{y \dots y}_{w_j} \underbrace{x \dots x}_{k-w_j-2} y$$

▶ **Example 11.** Consider again instance $\mathcal{I}^+$ of Example 9. For bin 1 and item 1 we have $u_{11} = b_1xxxxxxb_1\#yyxxxxxy$ and four associated sensitive patterns: $b_1yyxxxxxy$, $b_2yyxxxxxy$, $b_3yyxxxxxy$, and $b_1\$yyxxxxx$, forbidding replacing # with any letter but letters x or y. Note that replacing # by x in u_{11} introduces an occurrence of the length-k string $b_1xxxxxxb_1x$, which is also introduced when replacing # by x in t_{11} (thus not packing item 1 in bin 1: see Example 10). For bin 2 and item 3 we have $u_{23} = b_2xxxxb_2b_2b_2\#yyyyxxxy$ and the associated sensitive patterns: $b_1yyyyxxxy$, $b_2yyyyxxxy$, $b_3yyyyxxxy$, and $b_2\$yyyyxxx$.

The other gadgets u_{ij} and their associated sensitive patterns can be obtained similarly to those in Example 11; see [5] for the formal definitions.

Frequency Threshold and Final Construction

It would be tempting to set the frequency threshold τ to the bin capacity B, since when # is replaced by b_i in a gadget t_{ij} it creates exactly w_j occurrences of $b_i \cdots b_i$; exceeding the capacity of bin i should correspond to introducing more than B occurrences of $b_i \cdots b_i$. However, to ensure that the only feasible solutions to HM correspond to packing each item in some bin, we need to bound the number of allowed occurrences of other non-sensitive patterns like those introduced by replacing # in gadgets u_{ij}, which is linked to the number M of bins rather than to their capacity B. For this reason, in the reduction, we set $\tau = \max\{B, M\}+1$: to maintain the correspondence between the number of occurrences of length-k strings $b_i \cdots b_i$ and bin capacity, we append to the final string constructed by the reduction $\tau - B - 1$ occurrences of $b_i \cdots b_i$ for each i.

The final string is obtained by concatenating all gadgets t_{ij} and u_{ij} interleaved with \$\$, followed by an appropriate number of occurrences of some of the strings introduced by replacements in the gadgets. For a formal proof of this reduction, we refer the reader to [5].

▶ **Theorem 12** ([5]). *The HM problem is strongly NP-complete.*

3 Graph Sanitization

3.1 The Community Breaking Problem

The community structure is a fundamental property of a graph, and understanding how this structure can be maintained or disrupted is crucial. In our graph sanitization work, we introduced the following general *Community Breaking* (CB) problem: Given an undirected graph $G(V, E)$, a set of nodes $U \subseteq V$, and a notion of community, identify a smallest subset E' of E, so that no community in $G' = G(V, E \setminus E')$ contains a node in U.

In [8], we considered a specific community notion, namely the k-truss. A *k-truss* is a subgraph of a graph in which every edge is part of at least $k-2$ triangles formed entirely within the subgraph; see Fig. 1a for an example.

Building on the CB problem and the k-truss notion, we defined *MIN-k-TBS* (Minimum k-Truss Breaking Set): Given an undirected graph $G(V, E)$ and a parameter k, find a smallest subset E' of E such that $G(V, E \setminus E')$ contains no k-truss. MIN-k-TBS is obtained from the CB problem by considering communities based on the notion of k-truss and $U = V$.

(a) 4-truss. **(b)** 4-truss-free graph.

Figure 1 (a) The subgraph induced by the solid edges is a 4-truss because every edge of the subgraph is contained in at least $4-2=2$ triangles of the subgraph. (b) The graph obtained after deleting the set $\{(0,1),(3,4),(5,6)\}$ of (dashed) edges contains no 4-truss.

▶ **Example 13.** An optimal solution to MIN-k-TBS with $k=4$ in Fig. 1b is the set of (dashed) edges $E' = \{(0,1),(3,4),(5,6)\}$: deleting these edges leads to a graph with no 4-truss and deleting any fewer edges leads to a graph that contains a 4-truss.

The MIN-k-TBS problem is *intuitively* challenging: there are up to $2^{|E|}$ edge subsets that one may consider; and k-trusses have a hierarchical structure. For example, a $(k+i)$ truss, for any k and $i \geq 0$, is also a k-truss and contains at least $\binom{k+i}{k}$ smaller k-trusses. Finding a minimum set of edges to delete to make a graph triangle-free is known to be NP-hard [17], and that is exactly the MIN-3-TBS problem. Assuming the Exponential Time Hypothesis, we cannot even solve this problem in $2^{o(|E'|)} \cdot n^{O(1)}$ time [1]. We proved that MIN-k-TBS is also NP-hard for $k > 3$ using a reduction from MIN-3-TBS.

▶ **Theorem 14** ([8]). *For every $k \geq 3$, MIN-k-TBS is NP-hard.*

Here is a brief sketch of our proof. We prove that for every $k \geq 3$, the MIN-k-TBS problem is NP-hard by reducing from the known NP-hard MIN-3-TBS problem, which involves making a graph triangle-free. In our reduction, for each triangle in the original graph, we add $k-3$ extra nodes to form a clique with the triangle's vertices. This construction ensures that a k-truss exists in the new graph G_k if and only if a triangle exists in the original graph. It follows that solving MIN-3-TBS for G is equivalent to solving MIN-k-TBS for G_k. Therefore, the problem MIN-k-TBS is NP-hard for every $k \geq 3$.

We also presented an exact exponential-time algorithm to solve the MIN-k-TBS problem. The algorithm first enumerates all minimal k-trusses (i.e., a k-truss for which removing any single edge would destroy the k-truss) and then computes a minimum transversal of the corresponding hypergraph. Consequently, its overall time complexity is exponential in the number $|E|$ of edges. To practically improve on the efficiency of this algorithm, we developed three heuristics, MBH_S, MBH_C, and SNH, which run in polynomial time; see [8] for details.

3.2 Lower Bound on the Size of OPT

Due to the exponential-time complexity of our exact algorithm, computing the optimal solution is a heavy task even for small graphs with a few hundred nodes. Prof. Grossi designed an algorithm to compute a lower bound on the size of the optimal solution, which facilitates the evaluation of our heuristic algorithms.

The main idea is to use cliques as a "proxy" for trusses. Since a k-clique (i.e., a complete subgraph consisting of k vertices, where every pair of vertices is directly connected by an edge) is a k-truss, we must at the very least make the input graph G free from k-cliques to solve MIN-k-TBS. The algorithm works in three phases:

1. Computes an *edge clique partition* of G. We partition the graph into edge-disjoint cliques. This partitioning ensures that each edge belongs to exactly one clique, allowing us to analyse dense substructures individually.
2. Applies the best available lower bound on each clique. For each clique, we estimate the minimum number of edges that must be removed to eliminate all k-trusses within it. We employ two complementary approaches: (1) a bound derived from Turan's theorem [7], which limits the maximum number of edges in a graph that avoids a clique of size k; and (2) a triangle-based approach, as each edge in a k-truss must belong to at least k -2 triangles – counting the number of triangles gives an estimate of the minimum deletions required.
3. Outputs a lower bound on the size of the optimal solution by summing the bounds of the cliques. This is possible because the cliques are all edge-wise disjoint.

The overall time complexity of our LB algorithm is $O(q\Delta^2|E|)$, where q, Δ, and $|E|$ are the size of the largest clique, the highest degree, and the number of edges in G, respectively.

Interestingly, the lower bound produced by the above algorithm helped us demonstrate that our heuristics were very competitive to the exact algorithm on large instances in terms of solution quality, even if we were unable to run the exact algorithm on these instances!

References

1. N. R. Aravind, R. B. Sandeep, and Naveen Sivadasan. Dichotomy results on the hardness of H-free edge modification problems. *SIAM Journal on Discrete Mathematics*, 31(1):542–561, 2017. doi:10.1137/16M1055797.
2. Giulia Bernardini, Philip Bille, Inge Li Gørtz, and Teresa Anna Steiner. Differentially private substring and document counting. *Proc. ACM Manag. Data*, 3(2):95:1–95:27, 2025. doi:10.1145/3725232.
3. Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giovanna Rosone, and Michelle Sweering. Combinatorial algorithms for string sanitization. *ACM Trans. Knowl. Discov. Data*, 15(1):8:1–8:34, 2021. doi:10.1145/3418683.
4. Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String sanitization under edit distance. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 161 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CPM.2020.7.
5. Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering. Hide and mine in strings: Hardness, algorithms, and experiments. *IEEE Trans. Knowl. Data Eng.*, 35(6):5948–5963, 2023. doi:10.1109/TKDE.2022.3158063.
6. Giulia Bernardini, Alberto Marchetti-Spaccamela, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Constructing strings avoiding forbidden substrings. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 191 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.9.
7. B. Bollobás. *Extremal graph theory*. Courier Corporation, 2004.
8. Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. On breaking truss-based and core-based communities. *ACM Trans. Knowl. Discov. Data*, 18(6):135:1–135:43, 2024. doi:10.1145/3644077.
9. Huiping Chen, Changyu Dong, Liyue Fan, Grigorios Loukides, Solon P. Pissis, and Leen Stougie. Differentially private string sanitization for frequency-based mining tasks. In *IEEE International Conference on Data Mining (ICDM)*, pages 41–50. IEEE, 2021. doi:10.1109/ICDM51629.2021.00014.
10. Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. A survey of community search over big graphs. *VLDB J.*, 29(1):353–392, 2020. doi:10.1007/S00778-019-00556-X.
11. John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980. doi:10.1016/0022-0000(80)90004-5.
12. Aris Gkoulalas-Divanis and Grigorios Loukides. Revisiting sequential pattern hiding to enhance utility. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1316–1324, 2011. doi:10.1145/2020408.2020605.
13. Robert Gwadera, Aris Gkoulalas-Divanis, and Grigorios Loukides. Permutation-based sequential pattern hiding. In *IEEE 13th International Conference on Data Mining*, pages 241–250, 2013. doi:10.1109/ICDM.2013.57.
14. Edward G. Coffman Jr., Gábor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: Combinatorial analysis. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 151–207. Springer, 1999. doi:10.1007/978-1-4757-3023-4_3.

15 Takuya Mieno, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String sanitization under edit distance: Improved and generalized. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 191 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.19.

16 Teresa Anna Steiner. Differentially private approximate pattern matching. In *15th Innovations in Theoretical Computer Science Conference, (ITCS)*, volume 287 of *LIPIcs*, pages 94:1–94:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ITCS.2024.94.

17 M. Yannakakis. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981. doi:10.1137/0210021.

Faster Run-Length Compressed Suffix Arrays

Nathaniel K. Brown
Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Travis Gagie[1]
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Giovanni Manzini
Department of Computer Science, University of Pisa, Italy

Gonzalo Navarro
Department of Computer Science, University of Chile, Santiago, Chile

Marinella Sciortino
Department of Mathematics and Computer Science, University of Palermo, Italy

Abstract

We first review how we can store a run-length compressed suffix array (RLCSA) for a text T of length n over an alphabet of size σ whose Burrows-Wheeler Transform (BWT) consists of r runs in $O\left(r \log(n/r) + r \log \sigma + \sigma\right)$ bits such that later, given character a and the suffix-array (SA) interval for P, we can find the SA interval for aP in $O(\log r_a + \log \log n)$ time, where r_a is the number of runs of copies of a in the BWT. We then show how to modify the RLCSA such that we find the SA interval for aP in only $O(\log r_a)$ time, without increasing its asymptotic space bound. Our key idea is applying a result by Nishimoto and Tabei (ICALP 2021) and then replacing rank queries on sparse bitvectors by a constant number of select queries. We also review two-level indexing and discuss how our faster RLCSA may be useful in improving it. Finally, we briefly discuss how two-level indexing may speed up a recent heuristic for finding maximal exact matches of a pattern with respect to an indexed text.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Run-length compressed suffix arrays, interpolative coding, two-level indexing

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.10

Category Research

Funding *Nathaniel K. Brown*: Supported in part by a Johns Hopkins University Computer Science PhD Fellowship and NIH grants R21HG013433 and R01HG011392.
Travis Gagie: Funded in part by NSERC grant RGPIN-07185-2020.
Giovanni Manzini: Partially supported by the NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem (Spoke 6, CUP: I53C22000780001) and by the project PAN-HUB funded by the Italian Ministry of Health (ID: T4-AN-07, CUP: I53C22001300001).
Gonzalo Navarro: Funded in part by CeBiB under Basal Funds FB0001 and AFB240001, ANID, Chile.
Marinella Sciortino: Partially supported by the project "ACoMPA" (CUP B73C24001050001) funded by the NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem (Spoke 6) and by MUR PRIN project "PINC" (no. 2022YRB97K).

Acknowledgements Many thanks to the first author's CSCI 4419 / 6106 students Anas Alhadi, Nour Allam, Dove Begleiter, Nithin Bharathi Kabilan Karpagavalli, Suchith Sridhar Khajjayam and Hamza Wahed, and to Christina Boucher, Ben Langmead, Jouni Sirén and Mohsen Zakeri.

[1] Corresponding author.

1 Introduction

Grossi and Vitter's compressed suffix arrays (CSAs) [11] and Ferragina and Manzini's FM-indexes [8] are sometimes treated as almost interchangeable, but their query-time bounds are quite different. With a CSA for a text T of length n over an alphabet of size σ, when given a character a and the suffix-array (SA) interval for a pattern P we can find the SA interval for aP in $O(\log n_a)$ time, where n_a is the number of occurrences of a in the text; with an FM-index we use $O(\log \sigma)$ time. This difference carries over to run-length compressed suffix arrays (RLCSAs) [18, 24] and run-length compressed FM-indexes (RLFM-indexes) [10, 17], with both taking space proportional to the number r of runs in the Burrows-Wheeler Transform (BWT) of the text but the former being generally faster for texts over large alphabets with relatively few runs of each character, and the latter being faster for texts over smaller alphabets.

In Section 2 we review (with some artistic license) CSAs and RLCSAs. In Subsection 3.1 we show how to use interpolative coding to build an RLCSA for T that takes $O\left(r \log(n/r) + r \log \sigma + \sigma\right)$ bits and allows us to find the SA interval for aP from that of P in $O(\log r_a + \log \log n)$ time, where r_a is the number of those runs in the BWT containing copies of a. In Subsection 3.2 we review a result by Nishimoto and Tabei [20] about splitting the runs in the BWT so that we can evaluate LF in constant time, without increasing the number of runs by more than a constant factor. In Subsection 3.3 we present our main result: how to modify the RLCSA from Section 2 such that finding the SA interval for aP takes only $O(\log r_a)$ time, without increasing the asymptotic space bound. In Section 4 we discuss two-level indexing, for which we build one index for the text and another for the parse of the text, and how our faster RLCSA may be more suitable for indexing parses than current options. Finally, in Section 5 we briefly discuss how two-level indexing may speed up a recent heuristic for finding long maximal exact matches (MEMs) of a pattern with respect to an indexed text.

2 Preliminaries

Suppose we are given a text $T[0..n-1]$ over an alphabet of size σ and asked to index it such that, given a pattern $P[0..m-1]$, we can quickly count the number of occurrences of P in T. More specifically, we want to find the interval in the suffix array (SA) of T containing the starting positions of occurrences of P. Consider the matrix whose rows are the lexicographically sorted cyclic shifts of T and let F and L be the first and last column of that matrix, respectively; this means F contains the characters in T in lexicographic order and L is the BWT of T.

2.1 Compressed suffix arrays

The key idea behind compressed suffix arrays (CSAs) is to store $\Psi[0..n-1]$ compactly while supporting certain searches on it quickly, where $\Psi[0..n-1]$ is the permutation of $\{0, \ldots, n-1\}$ such that $\Psi[i]$ is the position of SA entry $(\mathrm{SA}[i]+1) \bmod n$ in $\mathrm{SA}[0..n-1]$ or, equivalently, the position in L of $F[i]$. (This means Ψ is the inverse of the LF mapping used in FM-indexes.) By the definition, Ψ consists of at most σ increasing intervals – one for each distinct character that occurs in the text, corresponding to the interval of suffixes starting with that character – and if we can support fast binary searches on these intervals then we can support fast pattern matching.

For example, consider the text

$$T = \texttt{CCTGGGCGAT\$CTTACACGAT\$GTTACCAGCT\$CTTACGCGCT\$CTGACGAATT\$CTTACGCGAT\#}\,,$$

for which SA, Ψ, F and L are shown on the left in Figure 1. If we know SA[22..28] is the SA interval for `CG` (in the green rectangle) and we want the SA interval for `GCG`, then we can search in the increasing interval

$$\Psi[36..48] = 6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$$

for `G` (in the red rectangle, with Ψ values between 22 and 28 shown as orange arrows and the others shown as black arrows) for the successor $\Psi[41] = 23$ of 22 and the predecessor $\Psi[43] = 28$ of 28. We thus learn that the SA interval for `GCG` is SA[41..43] (in the blue rectangle). Knowing this, we can continue backward stepping.

2.2 Run-length compressed suffix arrays revisited

Run-length compressed suffix array (RLCSA) were introduced in [24] for indexing highly repetitive collections. In this section we present an alternative, but functionally equivalent, description of RLCSAs which is more suitable for describing our improvements.

▶ **Definition 1.** *For a text $T[0..n-1]$, the array $L'[0..r-1]$ stores the sequence of r characters in the runs of the run-length encoding of L.*

▶ **Definition 2.** *For a text $T[0..n-1]$, the array $F'[0..r-1]$ stores the r characters in L' in lexicographic order.*

▶ **Definition 3.** *For a text $T[0..n-1]$, the array $\Psi'[0..r-1]$ is the permutation of $\{0, \ldots, r-1\}$ such that $\Psi'[i]$ is the position of $F'[i]$ in L'.*

In this paper we view a RLCSA as a data structure storing $\Psi'[0..r-1]$ compactly while supporting certain searches on it quickly. By the definition of Ψ', it still consists of at most σ increasing intervals – one for each distinct character that occurs in T, corresponding to the interval of suffixes starting with that character – and if we can still support fast binary searches on these intervals then we can still support fast pattern matching.

For example, consider

$$T = \texttt{CCTGGGCGAT\$CTTACACGAT\$GTTACCAGCT\$CTTACGCGCT\$CTGACGAATT\$CTTACGCGAT\#}$$

again, for which Ψ', F' and L' are shown on the right in Figure 1. If we know the SA interval SA[22..28] for `CG` starts at offset 0 in the L run of character $L'[12]$ and ends at offset 1 in the L run of character $L'[15]$ (in the green rectangle) and we want the SA interval for `GCG`, then we can search in the increasing interval

$$\Psi'[25..33] = 1, 3, 7, 13, 15, 22, 25, 39$$

for `G` (in the red rectangle, with Ψ' values between 12 and 15 shown as orange arrows and the others shown as black arrows) for the successor $\Psi'[28] = 13$ of 12 and the predecessor $\Psi'[29] = 15$ of 15.

Because L' and F' do not have the predecessor-successor relationship of L and F, we cannot deduce that the SA interval for `GCG` starts in the L run of character $L'[28]$ and ends in the L run of character $L'[29]$ (and, in fact, in this example it does not). Instead, we store two n-bit SD-bitvectors [21], B_L and B_F, with r copies of 1 each. The 1s in B_L mark the starting positions of runs in L and the 1s in B_F mark the positions in F of the marked characters in L. In our example

$$B_F = 111001101110011111111111000101110110111001\mathbf{01}0011110000010101111000$$
$$B_L = 100000110111011001011111010010011100111001\mathbf{01}1011111111110001011110.$$

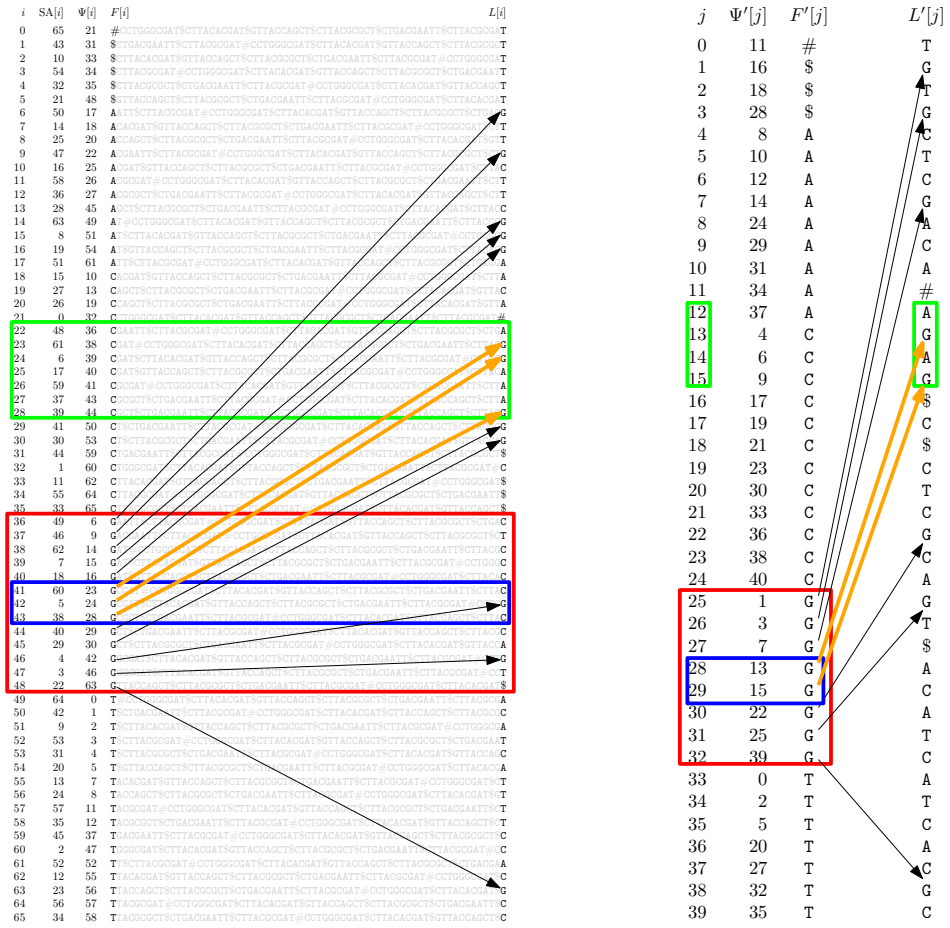


Figure 1 For

$T =$ `CCTGGGCGAT$CTTACACGAT$GTTACCAGCT$CTTACGCGCT$CTGACGAATT$CTTACGCGAT#`

we show SA, Ψ, F and L on the left and the Ψ', F' and L' on the right. If we know SA[22..28] is the SA interval for `CG` (in the green rectangle on the left) and we want the SA interval for `GCG`, then we can search in the increasing interval

$$\Psi[36..48] = 6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$$

for `G` (in the red rectangle on the left, with Ψ values between 22 and 28 shown as orange arrows and the others shown as black arrows) for the successor $\Psi[41] = 23$ of 22 and the predecessor $\Psi[43] = 28$ of 28. We thus learn that the SA interval for `GCG` is SA[41..43] (in the blue rectangle on the left).

On the other hand, if we know SA[22..28] starts at offset 0 in the L run of character $L'[12]$ – that is, at offset 0 in the 13th run, counting from 1 – and ends at offset 1 in the L run of character $L'[15]$ (in the green rectangle on the right), then we can search in the increasing interval

$$\Psi'[25..32] = 1, 3, 7, 13, 15, 22, 25, 39$$

for `G` (in the red rectangle, with Ψ' values between 12 and 15 shown as orange arrows and the others shown as black arrows) for the successor $\Psi'[28] = 13$ of 12 and the predecessor $\Psi'[29] = 15$ of 15 (in the blue rectangle on the right). We then use select and rank queries on two n-bit sparse vectors to find the SA interval for `GCG`, the L runs containing that interval's starting and ending positions, and those positions' offsets in those runs.

The interval $B_F[41..43]$ in B_F starting immediately before the bit with offset 0 in the block whose starting position is marked with the 29th copy of 1 and ending immediately before the bit with offset 1 in the block whose starting position is marked with the 30th copy of 1, is shown in blue. (We are interested in the blocks marked with the 29th and 30th copies of 1 because we count from 0 in the j column in Figure 1, so those blocks correspond to $\Psi'[28]$ and $\Psi'[29]$.) We can find this interval with 2 select_1 queries on B_F, which take constant time.

The corresponding interval $B_L[41..43]$ in B_L is also shown in blue, starting immediately before the bit with offset 3 in the block whose starting position is marked with the 22nd copy of 1 and ending immediately before the bit with offset 1 in the block whose starting position is marked with the 24th copy of 1. We can find the 2 indices 22 and 24 with 2 rank_1 queries on B_L, which take $O(\log\log n)$ time. This means the SA interval for `GCG` is SA[41..43] and it starts at offset 3 in the L run of character $L'[21]$ and ends at offset 1 in the L run of character $L'[23]$. Knowing this we can continue backward stepping.

The RLCSA in Sirén's PhD thesis [24] for a text $T[0..n-1]$ with r BWT runs takes $O\left(r\log(n/r) + r\log\sigma + \sigma\log n\right)$ bits. Given a character a and the SA interval for P, it can find the SA interval for aP in $O(\log n)$ time.

3 Faster RLCSAs

3.1 Searchable Interpolative coding

Suppose we are given an increasing list $\ell_1, \ldots, \ell_k$ of k integers in the range $[0..n-1]$. To encode them with *interpolative coding* [19], we first write $\ell_{\lceil k/2\rceil}$ using $\lfloor \lg(n-1)\rfloor + 1$ bits (except that we write 0 using 1 bit). All the numbers $\ell_1, \ldots, \ell_{\lceil k/2\rceil - 1}$ are in the range $[0..\ell_{\lceil k/2\rceil} - 1]$, so we can encode them recursively. All the numbers $\ell_{\lceil k/2\rceil+1}, \ldots, \ell_k$ are in the range $[\ell_{\lceil k/2\rceil}+1..n-1]$, so we can encode them recursively as $\ell_{\lceil k/2\rceil+1} - \ell_{\lceil k/2\rceil} - 1, \ldots, \ell_k - \ell_{\lceil k/2\rceil} - 1$. Each encoding has $O(\log n)$ bits, so we can read them in $O(1)$ time. If we imagine the list stored as keys in a balanced binary search tree then we encode the keys according to a pre-order traversal: when we reach each key ℓ_i, we know ℓ_i lies between the numbers shown to the left and right of ℓ_i and we encode ℓ_i using the maximum number of bits we would need for any key in that range.

For example, if $n = 66$, $k = 13$ and the list is $6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$ then, as illustrated in Figure 2, we start by encoding $\ell_7 = 24$ using $\lfloor \lg 65\rfloor + 1 = 7$ bits as `0011000`. We then encode $\ell_3 = 14$ using $\lfloor \lg 23\rfloor + 1 = 5$ bits as `01110`. We then encode $\ell_1, \ell_2, \ell_5, \ell_4, \ell_6 = 6, 9, 16, 15, 23$ as `0110`, `010`, `0001`, `0`, `110`, and $\ell_{10}, \ell_8, \ell_9, \ell_{12}, \ell_{11}, \ell_{13}$ as `000101`, `011`, `0`, `001111`, `1011`, `10000`. When we reach 46, say, in a pre-order traversal of the tree in Figure 2, we know it lies between 31 and 65, so we encode it using $\lfloor \lg(65-31)\rfloor + 1 = 6$ bits as $(46-31)_2 =$ `001111`.

The binary search tree has height $\lfloor \lg k\rfloor$ and the bottom level contains at most k keys. By Jensen's Inequality, we encode those keys using $O(k\log(n/k) + k)$ bits. Similarly, there at most $k/2^h$ keys at height h and we encode those keys using $O\left(\frac{k}{2^h}\log\frac{n}{k/2^h} + \frac{k}{2^h}\right) = O\left(\frac{k}{2^h}\log(n/k) + \frac{k(h+1)}{2^h}\right)$ bits. Since

$$\sum_{h=0}^{\lfloor \lg k\rfloor} \frac{k(h+1)}{2^h} = O(k)\,,$$

we use $O(k\log(n/k) + k)$ bits in total.

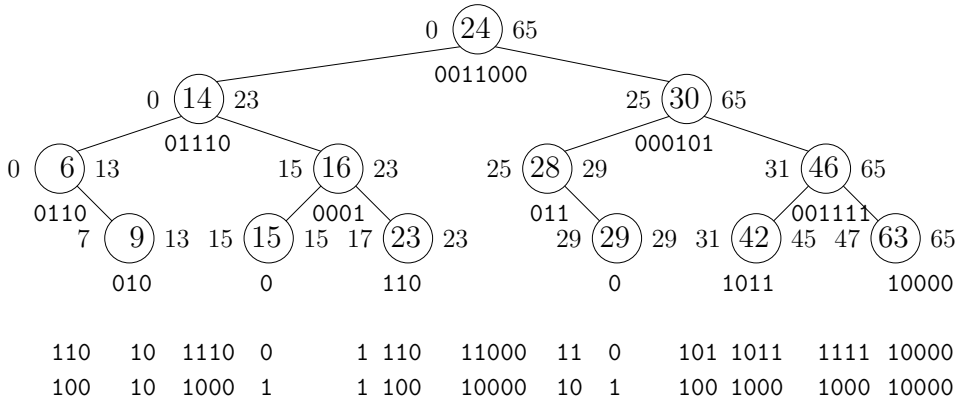


Figure 2 A balanced binary search tree storing the $k = 13$ keys from the increasing list $6, 9, 14, 15, 16, 23, 24, 28, 29, 30, 42, 46, 63$ with each key in the range $[0..n-1 = 65]$. When we reach each key in a pre-order traversal or binary search, we know it lies between the two values shown to its left and right, so we can encode it as the binary number shown below it, using a total of $O(k \log(n/k) + k)$ bits. If we store a bitvector marking the start of each encoding as visited in an in-order traversal, as shown below the tree, then we can omit the leading 0s from the encodings and support binary search in time $O(\log k)$ without changing our asymptotic space bound.

In this paper we want to perform binary search on the list – the reader may have noticed that our example is $\Psi[36..48]$ from Figure 1 – so we want fast access to the encodings of the numbers in it in the order we check them in a binary search. We can store the encodings according to an in-order traversal instead of a pre-order traversal, and store an uncompressed bitvector with as many bits as there are in the concatenation of the encodings and 1s marking where the encodings start. Since the bitvector delimits the encodings, however, we can delete the leading 0s from each encoding before concatenating them and building the bitvector. The in-order encodings for our example are shown below the tree in Figure 2, with the leading 0s removed, and the bitvector is shown below them. Since the bitvector uses at most as many bits as the encodings, we still use $O(k \log(n/k) + k)$ bits in total and – although random access still takes $O(\log k)$ time – we can perform binary search in $O(\log k)$ total time. This scheme is similar to Teuhola's [25] and Claude, Nicholson and Seco's [5].

To find the successor of 22 in the list, we start at the root knowing $n = 66$ and $k = 13$ and perform $\text{select}_1(7)$ and $\text{select}_1(8)$ queries on the bitvector to find the starting and ending positions of the encoding `0011000` of $\ell_7 = 24$ in the range $[0..65]$. Since $22 < 24$, we then perform $\text{select}_1(3)$ and $\text{select}_1(4)$ queries to find the starting and ending positions of the encoding `01110` of $\ell_3 = 14$ in the range $[0..23]$. Since $22 > 14$, we then perform $\text{select}_1(5)$ and $\text{select}_1(6)$ queries to find the starting and ending positions of the encoding `0001` of $\ell_5 = 16$ in the range $[15..23]$. Since $22 > 16$, we then perform $\text{select}_1(6)$ and $\text{select}_1(7)$ queries to find the starting and ending positions of the encoding `110` of $\ell_6 = 23$ in the range $[17..23]$. Since $22 < 23$, we know the successor of 22 in L is 23. We can find the predecessor of 28 in $O(\log k)$ time symmetrically.

If we apply interpolative coding with fast binary search to the increasing interval of Ψ for a character a in a text T of length n, then we use $O(n_a \log(n/n_a) + n_a)$ bits and can support binary search in $O(\log n_a)$ time, where n_a is the frequency of a in T. If we do this for all the characters then we use $O(n(H_0(T) + 1))$ bits, where H is the 0th-order empirical entropy of T. If we encode the increasing interval of Ψ' for a with interpolative coding, then we use $O(r \log(r/r_a) + r_a)$ bits and can support binary search in $O(\log r_a)$ time, where r_a is

the number of runs of copies of a in the BWT of T (and, equivalently, in L). If we do this for all the characters then we use $O(r(H_0(L')+1))$ bits, where L' is again the sequence of r characters in the runs of the run-length encoding of T. To be able to find the increasing interval for a in Ψ', we store an r-bit uncompressed bitvector with 1s marking where the intervals start.

▶ **Theorem 4.** *We can store Ψ' for T in $O(r(H_0(L')+1)) \subseteq O(r \log \sigma)$ bits and support binary search in the increasing interval for a character a in $O(\log r_a)$ time, where r_a is the number of runs of copies of a in the BWT of T.*

To use Theorem 4 in an RLCSA, we store

- an uncompressed bitvector marking which distinct characters occur in T, in $O(\sigma)$ bits;
- the SD-vectors B_F and B_L in $O(r \log(n/r))$ bits;
- an uncompressed bitvector with 1s marking where the intervals for the characters start in Ψ', in $O(r)$ bits;
- Ψ' in $O(r \log \sigma)$ bits.

If we are given a and the SA interval for P then we can find the SA interval for aP by

- using a rank query on the first uncompressed bitvector to find a's rank among the distinct characters that occur in T, in $O(1)$ time;
- using rank queries on B_L to find the runs in L overlapping the SA interval for P, in $O(\log \log n)$ time;
- using select queries on the second uncompressed bitvector to find the interval for a in Ψ', in $O(1)$ time;
- using binary search in the interval for a in Ψ' to find the successor and predecessor of the ranks of the first and last runs in L overlapping the SA interval for P, in $O(\log r_a)$ time;
- using select queries on B_F and arithmetic to find the SA interval for aP in $O(1)$ time.

We store $O(r \log(n/r) + r \log \sigma + \sigma)$ bits in total and find the SA interval for aP in $O(\log r_a + \log \log n)$ total time. Notice that the $O(\log \log n)$ term in the query time comes only from the rank query on B_L.

▶ **Corollary 5.** *We can store an RLCSA for T in $O\left(r \log(n/r) + r \log \sigma + \sigma\right)$ bits such that, given character a and the SA interval for P, we can find the SA interval for aP in $O(\log r_a + \log \log n)$ time.*

3.2 Splitting Theorem for RLCSAs

Nishimoto and Tabei [20] showed how we can split the runs in L such that no block in B_F overlaps more than a constant number of blocks in B_L without increasing the number of runs by more than a constant factor, and then store LF in $O(r \log n)$ bits and evaluate it in constant time. Brown, Gagie and Rossi [4] slightly generalized their key theorem:

▶ **Theorem 6** (Nishimoto and Tabei [20]; Brown, Gagie and Rossi [4])**.** *Let π be a permutation on $\{0, \ldots, n-1\}$,*

$$P = \{0\} \cup \{i \ : \ 0 < i \leq n-1, \pi(i) \neq \pi(i-1)+1\},$$

and $Q = \{\pi(i) \ : \ i \in P\}$. For any integer $d \geq 2$, we can construct P' with $P \subseteq P' \subseteq \{0, \ldots, n-1\}$ and $Q' = \{\pi(i) \ : \ i \in P'\}$ such that

- *if $q, q' \in Q'$ and q is the predecessor of q' in Q', then $|[q, q') \cap P'| < 2d$,*
- *$|P'| \leq \frac{d|P|}{d-1}$.*

If $L[i] = L[i-1]$ then $\mathrm{LF}(i) = \mathrm{LF}(i-1) + 1$, so

$$\{0\} \cup \{i \ : \ 0 < i \leq n-1, \mathrm{LF}(i) \neq \mathrm{LF}(i-1) + 1\}$$

has cardinality r. If $\mathrm{LF}(i) = \mathrm{LF}(i-1) + 1$ then, since Ψ and LF are inverse permutations, $\Psi[j] = \Psi[j-1] + 1$ where $j = \mathrm{LF}(i)$. Therefore,

$$\{0\} \cup \{j \ : \ 0 < j \leq n-1, \Psi[j] \neq \Psi[j-1] + 1\}$$

also has cardinality r and applying Theorem 6 with $d = 2$ to Ψ splits the runs in the BWT such that no block in B_F overlaps more than 3 blocks in B_L, without increasing the number of runs by more than a factor of 3/2. In our example, the number of runs increases by only 1, from 40 to 41, as shown below with the split block – corresponding to the first run of 6 copies of `T` in L – in red:

$$\begin{aligned} B_F &= \texttt{11100110111001111111111100010111011011100101001111}\mathbf{\texttt{100100}}\texttt{10101111000} \\ B_L &= \mathbf{\texttt{100100}}\texttt{11011101100101111101001001110011100011011111111111000101111}\texttt{10}\,. \end{aligned}$$

Suppose we apply Theorem 6 with $d = 2$ to Ψ and then store, for $0 \leq b < r$, the index of the block in B_L containing $\mathrm{LF}(i_b)$ and $\mathrm{LF}(i_b)$'s offset in that block, where i_b is the starting position of block b in B_L. Nishimoto and Tabei called this the *move table* for LF (see also [4, 26]) and it takes a total of $O(r \log n)$ bits. If we know $B_L[j]$ is in block b in B_L with offset $j - i_b$ then, since the block in B_F to which LF maps block b in B_L now overlaps at most the block containing $B_L[\mathrm{LF}(i_b)]$ and the next 2 blocks in B_L, we can find the index of the block in B_L containing $B_L[\mathrm{LF}(j)] = B_L[\mathrm{LF}(i_b)] + j - i_b$ and $B_L[\mathrm{LF}(j)]$'s offset in that block with at most 2 constant-time select queries on B_L. We could use at most 2 constant-time lookups instead if we have the starting positions of the blocks in B_L stored explicitly in another $O(r \log n)$ bits.

3.3 A faster RLCSA without rank queries

Recall that the $O(\log \log n)$ term in the query-time bound in Corollary 5 comes only from the use of rank queries on an SD-vector. Since rank and select queries can be combined to support predecessor queries and select queries on sparse bitvectors can easily be supported in constant time and space polynomial in the number of 1s, rank queries on compact sparse bitvectors inherit lower bounds from predecessor queries [3] – so they cannot be implemented in constant time. Therefore, to get rid of that $O(\log \log n)$ term, we must somehow avoid rank queries.

We could replace the rank queries with a move table, but that would result in an $O(r \log n)$ term in our space bound. Instead, we introduce an uncompressed $2r$-bit bitvector B_{FL} indicating how the starting positions of the blocks in F and L are interleaved. Specifically, we scan B_F and B_L simultaneously – assuming we have already applied Theorem 6 to them so that no block in F overlaps more than 3 blocks in L (so r is a constant factor larger than it was before the application of the theorem) – and

- if we see 0s in both bitvectors in position i then we write nothing;
- if we see a 1 in B_F and a 0 in B_L then we write a 1 (indicating that a block starts in F);
- if we see a 0 in B_F and a 1 in B_L then we write a 0 (indicating that a block starts in L);
- if we see 1s in both bitvectors then we write a 0 and then a 1 (indicating that blocks start in both L and F).

This way, $B_{FL}.\text{select}_1(j)$ tells us which at most 3 blocks in L – those corresponding to the 0 preceding the jth copy of 1 in B_{FL} and possibly to the next 2 copies of 0 – could overlap block j in F (counting from 1). We can then find the starting positions of those blocks in L using at most 3 select queries on B_L.

For our example, taking B_F and B_L to be as shown just after Theorem 6,

$$\begin{array}{rcl} & & \texttt{0123456789012345678901234567890123456789012345 6789012345678901234 5} \\ B_F & = & \texttt{11100110111001111111111100010111011011100}\mathbf{\texttt{10100}}\texttt{1111001001010111 1000} \\ B_L & = & \texttt{10010011011101100101111101001001110011}\mathbf{\texttt{10001101}}\texttt{1111111110001011110}\,, \end{array}$$

(with the grey numbers only to show positions) we have

$$\begin{array}{rcl} & & \texttt{012345678901234567890123456789012345678 9} \\ B_{FL} & = & \texttt{01110101010101001011101101010101010110}\ldots \\ & & \texttt{0123456789012 34 5678901234567890123456789 01} \\ & & \ldots\texttt{10011010}\mathbf{\texttt{10}}\texttt{1}\mathbf{\texttt{0}}\texttt{1}\mathbf{\texttt{0}}\texttt{0100101010100010001101101010 0}\,. \end{array}$$

In position 38 we see 1s in both B_F and B_L, so we write 01 in B_{FL} (in positions 49 and 50, respectively); in positions 39 and 40 we see 0s in both in B_F and B_L, so we write nothing; in position 41 we see a 1 in B_F and a 0 in B_L, so we write a 1 in B_{FL}; in position 42 we see a 0 in B_F and a 1 in B_L, so we write a 0 in B_{FL}; in position 43 we see 1s in both B_F and B_L, so we write 01 in B_{FL}; in position 44 we see 0s in both B_F and B_L, so we write nothing; and in position 45 we see a 0 in B_F and a 1 in B_L, so we write a 0 in B_{FL} (in position 55). Admittedly, when $n = 66$ and after applying Theorem 6 $2r = 82$, it seems foolish to store a $2r$-bit uncompressed bitvector instead of simply storing B_L uncompressed. This is due to the small size of our example, however; for massive and highly repetitive datasets, r can easily be hundreds of times smaller than n.

Suppose we know the SA interval SA[41..43] for aP starts at offset 0 in block 28 in F and ends at offset 1 in block 29 in F and we want to find which blocks contain its starting and ending positions in L and the offsets of those positions. In Section 2, we performed 2 rank queries on B_L, but now we perform queries $B_{FL}.\text{select}_1(29) = 51$ and $B_{FL}.\text{select}_1(30) = 54$ (with arguments 29 and 30 instead of 28 and 29 because we mark with a 1 the starting of the first block in F, which we index with 0; the results 51 and 54 are indexed from 0 as well). Since the 29th and 30th copies of 1 are $B_{FL}[51]$ and $B_{FL}[54]$ (shown in red above), they are preceded by the $51 - 29 + 1 = 23$rd and $54 - 30 + 1 = 25$th copies of 0, respectively.

Because we applied Theorem 6, this means the 29th and 30th blocks in F (shown in red in B_F above) overlap the 23rd block in L and possibly the 24th and 25th blocks (shown in blue in B_L), and the 25th block and possibly the 26th and 27th blocks (also shown in blue in B_L). Notice that, because we split the 34th block in F but the first block in L for Theorem 6, the block numbers we find in F are the same as in Section 2 but the block numbers we find in L will be incremented. Although in general we need 6 select queries on B_L, in this case we can use only 5 – $B_L.\text{select}_1(23), \ldots, B_L.\text{select}_1(27)$ – to find where these blocks begin in constant time, and determine which contain the starting and ending positions of the SA interval SA[41..43]: the 23rd and the 25th, respectively.

In short, we replace a rank query on SD-bitvector B_L by queries on uncompressed bitvector B_{FL} and constant-time select queries on B_L. This gives us the following theorem:

▶ **Theorem 7.** *We can store an RLCSA for T in $O\left(r \log(n/r) + r \log \sigma + \sigma\right)$ bits such that, given character a and the SA interval for P, we can find the SA interval for aP in $O(\log r_a)$ time, where r_a is the number of runs of copies of a in the BWT of T.*

Instead of viewing B_{FL} as replacing slow rank queries while using the overall same space, we can also view it (and B_F and B_L) as replacing an $O(r \log n)$-bit move table while using the same overall query time. Brown, Gagie and Rossi [4] implemented a similar approach to speeding up LF computations in an RLFM-index, but only alluded to it briefly in their paper – the path to `Bitvector` in their Figure 3 – and gave no analysis nor bounds. We conjecture that a similar approach can also be applied to reduce the size of fast move tables for ϕ and ϕ^{-1} [13], which return $\mathrm{SA}[i-1]$ and $\mathrm{SA}[i+1]$ when given $\mathrm{SA}[i]$.

4 Two-level indexing

Corollary 5 and Theorem 7 suggest that RLCSAs should perform well compared to FM-indexes and RLFM-indexes when the BWT is over a fairly large alphabet and the number of runs of each character is fairly small; Ordóñez, Navarro and Brisaboa [23] have confirmed this experimentally. When indexing a highly repetitive text over a small alphabet, we can make RLCSAs more practical by storing a table of k-tuples that tells us in which range of Ψ' to search based on which character we are trying to match and which $k-1$ characters we have just matched. (This table can be represented with a bitvector to save space.) The table for our example from Figure 1 and $k = 2$ is shown below:

`#C`	0	`AG`	8	`CT`	20..24	`T#`	33
`$C`	1..2	`AT`	9..12	`GA`	25..27	`T$`	33
`$G`	3	`CA`	13..14	`GC`	28..29	`TA`	34..35
`AA`	4	`CC`	15..16	`GG`	30..31	`TG`	36..37
`AC`	4..7	`CG`	17..19	`GT`	32	`TT`	38..39

This says that if we want the SA interval for `GCG` and we have just matched the suffix `CG`, then we should search in the range $\Psi'[28..29]$. On the other hand, notice that the largest range of Ψ' in which we will ever search is now $\Psi'[20..24]$ – of length 5 – when we are trying to match a `C` after just matching a `T`; without such a table, the largest range we search is $\Psi'[13..24]$ – of length 12 – when trying to match a `C`.

There are interesting cases in which we want to index highly repetitive texts over large alphabets, however. For example, consider indexing a minimizer digest of a pangenome – considering minimizers as meta-characters from a large alphabet instead of tuples of characters from a small alphabet [1, 2, 7, 27] – or *two-level indexing* such a text. For two-level indexing we build one index for the text and another for a parse of the text; the alphabet of the parse is the dictionary of distinct phrases, which is usually large, but the parse itself is usually much smaller than the text and its BWT is usually still run-length compressible (albeit less than the BWT of the text) when the text is highly repetitive.

Something like two-level indexing was proposed by Deng, Hon, Köppl and Sadakane [6] but they did not use an index for the text and its absence made their implementation quite slow for all but very long patterns. Hong, Oliva, Köppl, Bannai, Boucher and Gagie [12] described another approach, which we will review here, but they used standard FM-indexes for the text and the parse instead of RLFM-indexes, so their two-level index was noticeably faster but hundreds of times larger than its competitors.

Consider the 50 similar toy genomes of length 50 each in Figure 3. Suppose we parse their concatenation similarly to `rsync`, by inserting a phrase break whenever we see a trigger string – `ACA`, `ACG`, `CGC`, `CGG`, `GAC`, `GAG`, `GAT`, `GTG`, `GTT`, `TCG` or `TCT` – or when we reach the `TA#` at the end of the text. (Considering $\# = \$ = 0$, $\mathtt{A} = 1$, $\mathtt{C} = 2$, $\mathtt{G} = 3$ and $\mathtt{T} = 4$ and viewing triples as 3-digit numbers in base 5, the trigger strings are the triples in the concatenation whose values are congruent to 0 modulo 6.) If we replace each phrase in the parse by its

```
CTTCCGCGGTGATAAAGGGGGCGGTAATGTCGCGAAACAGTCTTTTCTA$    CTTACGCGGTGATTCAGGGGGCGGTAATTTCGCGGATCAGTCTTTTCTA$
CTTACGCGGTGATACAGGGGGCCGTAATTTCGCGGAACAGTCTTTTCTA$    CTTACGCGGTTATCCAGGGGGTGGTACTTTCGGTGAACAGTCTGTTCTA$
CTTACGCGACGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$    CTTACGCGGTGATCCAGGGGGCAGTAATTTCGCGGAACAGTCTTTTCTA$
CTTATGCGATGATCCTGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$    CTTACGCGATGATCCAGGGGGCGGTAATTTCGCTGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACACTCTTCTCTA$    CTTACGCGATGATCCATTGGGCGGTAATTCCGCGGAACAGTCTTTTCTA$
CTTACGCGATGATCCAGTGGGCGGTCTTTTCGCGGAACAGTCTTTTCGA$    CTTACGCGATGATCCATGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGCAACAGTCTTTTCTA$    CTTACGCGATGATCCAGGGGGCTGTATTTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCTCGGAACAGTCTTTTCTA$    CTTACGCGCTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTATGCGGTGATCCACGGGGCGGAAGTATCGCGGAACAGTCTTTTTTA$    CTTACGAGATGAGCTAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGATGATCCAGGGGGCGGTAACTTCGCGGAACAGTCTTTTCTA$    CTTACGCGATGCTCCAGGGGGCGGTGATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGACGATCCAGGGGGCAGTAATTTCGCGGAACAGTCTTTTCTA$    CTTTCGCGATGATCCAGGGGGCGGTCATTTCGCGGAACAGTCTTTTCTA$
CATACGCGGGGATCCAAGGGGCGGTAATTTCGCGGAACAGTCTTTGACA$    CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGCACAGTCCTTTCTA$
CTTTCACGGTGATCCAGGGGTGGGTAATTTCGCGGAACAGTCTTTTCTA$    CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAATCTTTTCTA$    CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGATGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$    CTTACTCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTT$
CTTACGCGGTGATCCAGGGGGCGCTAATTTCGCGAAACAGTCTTTTCTA$    CTTACGCGATGATCCAGAGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGTGGTGATCCAGGGGGCGGTAATTTCGAGGAACAGTCTTTAATA$    CTTACGCGGTGATCCAGGGGGCGGTAATTCCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGCGCGGTAATTTCGCGGAACAATCTTTTCTA$    CTTACGCGGGGATCCAGGGGGCGGTAATTTCGCGGAACAATCTTTTCTA$
CTTACGCGATGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTAATCTA$    CTTACGCGGTGATCCAGGGGTCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$    CATACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$    CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAGTCTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGCGGAACAATCTTTTCTA$    CCTACGCGATGATGCAGGGGGCGGTAATTTCGCGGAACAGTCTTTCCTA$
CTTACGCGATGATCCAGGGGGCGGTAATTGCGCGGAACAGTCTTTTCTA$    CTTACGCGATGTTCCAGGGGGCGGTAATTTCGCGGAATAGTTTTTTCTA$
CTTACGCGGTGATCCAGGGGGCGGTAATTTCGGGGAACAGTCTTTTCTA$    CTTACGCGGTGATCCAGCGGGCGGTAATTTCGCGGAATAGTCTTTTCTA$
CTTACGCGATCTTCCAGGGGGCCGAAATTTCGCGTAACAGTCTTTTCTA$    CTTACGCGGTAATCCAGGGGGCGGTAATGTCGCGGAACAGTCTTTTCTA#
```

Figure 3 A set of 50 similar toy genomes of length 50 each, with the first 49 separated by copies of `$` and the last one terminated by `#`.

lexicographic rank in the dictionary of distinct phrases, counting from 1, and terminate the sequence with a 0, we get the 562-number sequence shown in Figure 4. With a larger example, of course, we obtain longer phrases on average and better compression from the parsing.

Run-length compression naturally works better on the BWT of the concatenation of the genomes than on the BWT of the parse, as shown in Figures 5 and 6. Again, with a larger example we would achieve better compression, also from the run-length compressed BWT (RLBWT) of the parse. Even this small example, however, gives some intuition how the dictionary of distinct phrases in the parse is usually large, but the parse is usually much smaller than the text and its BWT is usually still run-length compressible. In this case there are 90 distinct phrases in the dictionary, the parse is less than a quarter as long as the text, and the average run length in its RLBWT is slightly more than 2. An FM-index based on the RLBWT of the parse would generally use at least about $\lceil \lg 90 \rceil = 7$ rank queries on bitvectors for each backward step. The most common value in the runs, 19, occurs in only 16 runs, so we should spend at most about $\lg 16 = 4$ steps in each binary search.

To search for a pattern, we start by backward stepping in the index for the text until we reach the left end of the rightmost trigger string in the pattern. We keep count of how often each trigger string occurs in the text and an n-bit sparse bitvector with 1s marking the lexicographic ranks of the lexicographically least suffixes starting with each trigger string. This way, when we reach the left end of the rightmost trigger string, with a rank query on that bitvector we can compute the lexicographic ranks of the suffixes starting with the suffix of the pattern we have processed so far among all the suffixes starting with trigger strings, and map from the index of the text into the index for the parse. The width of the BWT interval stays the same and may span several lexicographically consecutive phrases in the dictionary – all those starting with the suffix of the pattern we have processed so far – but it is possible to start a backward search in the index for the parse with a lexicographic range of phrases rather than with a single phrase.

When we reach the left end of the leftmost trigger string in the pattern, we can use the same bitvector to map back into the index for the text and match the remaining prefix of the pattern with that. While matching the pattern phrase by phrase against the index

```
44, 55, 79, 19, 11, 70, 22, 46, 64, 88, 6, 22, 55, 79, 19, 17, 59, 22, 55, 12,
64, 88, 6, 22, 48, 45, 19, 32, 73, 22, 55, 12, 64, 88, 8, 50, 39, 73, 22, 55,
12, 64, 88, 6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 43, 78, 41, 6, 622, 50, 50,
36, 58, 78, 87, 22, 55, 12, 64, 87, 6, 22, 55, 79, 19, 32, 73, 22, 51, 12, 64,
88, 6, 22, 55, 79, 19, 32, 74, 40, 45, 12, 64, 88, 9, 79, 19, 26, 45, 58, 13,
22, 55, 12, 64, 90, 22, 50, 50, 32, 68, 22, 55, 12, 64, 88, 6, 22, 48, 45, 19,
30, 22, 55, 12, 64, 88, 4, 22, 55, 57, 25, 73, 22, 55, 12, 64, 86, 2, 1, 45,
79, 19, 35, 60, 22, 55, 12, 64, 88, 6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 21,
88, 6, 22, 50, 50, 32, 73, 22, 55, 12, 64, 88, 6, 22, 55, 79, 19, 31, 73, 22,
46, 64, 88, 6, 79, 65, 19, 32, 73, 18, 47, 64, 83, 22, 55, 79, 19, 29, 55, 73,
22, 55, 12, 21, 88, 6, 22, 50, 50, 32, 73, 22, 55, 12, 64, 16, 6, 22, 55, 79,
19, 32, 73, 22, 55, 12, 64, 88, 6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 64, 88,
6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 21, 88, 6, 22, 50, 50, 32, 72, 55, 12,
64, 88, 6, 22, 55, 79, 19, 32, 73, 45, 56, 64, 88, 6, 22, 50, 41, 77, 22, 61,
64, 88, 6, 22, 55, 79, 19, 75, 73, 22, 55, 19, 24, 88, 6, 22, 55, 82, 20, 62,
45, 79, 12, 64, 66, 41, 6, 22, 55, 79, 19, 30, 22, 55, 12, 64, 88, 6, 22, 50,
50, 32, 73, 22, 80, 64, 88, 6, 22, 50, 50, 38, 71, 55, 12, 64, 88, 6, 22, 50,
50, 37, 73, 22, 55, 12, 64, 88, 6, 22, 50, 50, 33, 22, 55, 12, 64, 88, 6, 22,
51, 81, 32, 73, 22, 55, 12, 64, 88, 6, 18, 19, 49, 42, 73, 22, 55, 12, 64, 88,
6, 22, 50, 54, 79, 19, 85, 22, 55, 12, 64, 88, 10, 22, 50, 50, 32, 76, 22, 55,
12, 64, 88, 6, 22, 55, 79, 19, 32, 73, 22, 55, 23, 63, 6, 22, 55, 79, 19, 32,
73, 22, 55, 12, 64, 88, 6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 64, 88, 7, 45,
79, 19, 32, 73, 22, 55, 12, 64, 88, 67, 22, 50, 50, 27, 58, 73, 22, 55, 12, 64,
88, 6, 22, 55, 79, 19, 32, 71, 55, 12, 64, 88, 6, 22, 55, 57, 32, 73, 22, 55,
12, 21, 88, 6, 22, 55, 79, 19, 34, 45, 73, 22, 55, 12, 64, 88, 4, 22, 55, 79,
19, 32, 73, 22, 55, 12, 64, 88, 6, 22, 55, 79, 19, 32, 73, 22, 55, 12, 64, 88,
5, 22, 50, 50, 52, 73, 22, 55, 12, 64, 84, 22, 50, 66, 32, 73, 22, 55, 15, 89,
6, 22, 55, 79, 19, 28, 53, 73, 22, 55, 14, 88, 6, 22, 55, 69, 70, 22, 55, 12,
64, 88, 3, 0
```

Figure 4 The 563-number sequence (20 numbers per line) over the alphabet $\{0, \ldots, 90\}$ we get from the concatenation of the toy genomes in Figure 3 by parsing, replacing each phrase by its rank in the dictionary (counting from 1) and appending a 0.

for the parse, we can either compare against phrases in the stored dictionary or just use Karp-Rabin hashes (allowing some probability of false-positive matches). We still have to parse the pattern, but that requires a single sequential pass, while FM-indexes in particular are known for poor memory locality. They key idea is that, ideally, we match most of the pattern phrase by phrase instead of character by character, reducing the number of cache misses.

We plan to reimplement two-level indexes for collections of similar genomes with RLFM-indexes for the collections themselves and CSAs, standard RLCSAs and our sped-up RLCSAs for the parses from Theorem 7 of those collections, and compare them experimentally. We also plan to try indexing minimizer digests with CSAs and RLCSAs.

5 Boyer-Moore-Li with two-level indexing

Olbrich, Büchler and Ohlebusch [22] recently showed how working with `rsync`-like parses of genomes instead of the genomes themselves can speed up multiple alignment. More specfically, they find and use as anchors finding maximal substrings (call multi-MUMs) of the parses that occur exactly once in each parse. In this section we speculate about how two-level indexing may similarly speed up searches for long maximal exact matches (MEMs). A MEM of a pattern $P[0..m-1]$ with respect to a text T is a substring $P[i..j]$ of P such that

- $P[i..j]$ occurs in T,
- $i = 0$ or $P[i-1..j]$ does not occur in T,
- $j = m-1$ or $P[i..j+1]$ does not occur in T.

Finding long MEMs is an important task in bioinformatics and there are many tools for it.

A^{8} T^{1} A^{41} T^{28} G^{1} T^{18} C^{1} T^{1} G^{2} T^{1} G^{27} A^{1} G^{10} C^{1} T^{1} A^{1} G^{6} T^{1} C^{1}
A^{1} G^{1} T^{1} G^{2} T^{2} C^{4} T^{39} A^{1} T^{3} G^{1} A^{5} T^{1} C^{1} A^{41} T^{1} G^{2} T^{42} C^{2} T^{2} C^{1}
A^{1} T^{1} C^{1} G^{1} C^{1} G^{2} C^{1} G^{1} A^{1} C^{6} A^{1} C^{13} T^{1} C^{22} A^{1} C^{22} T^{1} C^{22} T^{1} A^{1}
G^{2} C^{2} A^{2} G^{10} A^{1} G^{25} T^{1} G^{6} T^{1} A^{1} G^{1} A^{4} G^{15} T^{2} G^{1} C^{1} A^{2} G^{2} A^{3} C^{1}
A^{27} C^{1} A^{2} G^{1} A^{9} T^{1} A^{1} G^{1} C^{1} A^{4} G^{1} C^{1} T^{1} A^{1} C^{6} A^{1} C^{12} G^{1} C^{11} T^{1}
C^{10} G^{2} A^{35} T^{1} A^{7} C^{1} $\$^{2}$ C^{2} T^{45} G^{1} T^{3} G^{1} T^{1} $\$^{1}$ T^{3} G^{2} C^{1} G^{2} A^{1} T^{1}
A^{2} G^{17} T^{2} A^{9} T^{1} A^{7} T^{1} A^{1} T^{18} C^{1} T^{2} C^{1} T^{9} G^{1} T^{9} A^{3} G^{1} C^{1} A^{22} T^{1}
G^{1} T^{1} G^{35} T^{1} G^{9} T^{1} G^{2} A^{1} G^{17} T^{1} G^{23} T^{1} G^{18} T^{1} G^{4} A^{1} G^{4} C^{1} A^{1} T^{17}
C^{1} T^{28} G^{1} C^{1} G^{2} T^{1} A^{1} T^{1} A^{1} G^{2} C^{1} G^{1} T^{2} $\$^{44}$ T^{1} $\#^{1}$ A^{1} T^{2} $\$^{1}$ T^{1}
$\$^{1}$ A^{1} C^{1} T^{44} C^{4} G^{6} T^{1} G^{15} T^{1} G^{22} T^{1} C^{3} T^{1} C^{1} A^{1} T^{2} G^{2} T^{4} C^{1} T^{9}
G^{1} T^{10} C^{1} T^{13} C^{1} A^{1} C^{13} T^{1} C^{2} T^{2} C^{1} G^{1} T^{1} G^{4} C^{16} T^{1} C^{4} T^{1} G^{2} C^{38}
G^{1} C^{4} A^{1} C^{2} G^{1} C^{1} G^{8} C^{1} G^{29} C^{2} T^{1} C^{20} G^{1} C^{3} A^{1} T^{1} C^{2} G^{1} C^{6} A^{1}
C^{10} G^{1} C^{26} G^{2} C^{1} G^{54} A^{1} G^{5} T^{1} G^{16} A^{1} G^{8} C^{1} G^{7} T^{1} G^{2} C^{3} G^{5} C^{1} G^{13}
A^{1} G^{2} T^{1} G^{19} A^{23} T^{1} A^{18} G^{1} T^{1} G^{3} C^{25} G^{1} C^{14} T^{1} C^{1} G^{1} C^{10} T^{1} C^{18} G^{2}
C^{2} G^{17} A^{1} G^{3} C^{1} A^{1} G^{21} A^{1} T^{1} G^{1} A^{1} T^{2} G^{1} A^{6} G^{1} A^{37} G^{28} A^{1} G^{2} C^{1}
G^{1} T^{2} A^{1} T^{1} C^{8} T^{1} C^{15} A^{1} C^{22} A^{1} G^{2} T^{1} G^{1} C^{1} G^{6} C^{1} G^{35} A^{1} T^{14} C^{1}
T^{3} A^{1} T^{14} A^{1} T^{11} G^{1} C^{1} A^{2} T^{1} G^{1} T^{2} G^{1} T^{2} A^{1} G^{1} A^{7} T^{1} A^{22} T^{1} A^{5}
C^{1} A^{7} T^{4} A^{1} G^{1} T^{2} G^{1} T^{12} G^{1} T^{23} A^{1} T^{6} C^{1} T^{1} G^{1} T^{1} C^{1} T^{13} C^{1} T^{16}
A^{1} T^{13} G^{1} T^{2} G^{1} T^{1} A^{1} C^{1} G^{13} A^{3} G^{20} A^{1} G^{10} C^{1} T^{1} A^{1} G^{3} A^{1} G^{4} A^{1}
G^{1} A^{1} G^{5} A^{1} G^{1} C^{1} A^{1} G^{7} A^{1} G^{2} A^{2} G^{2} A^{5} G^{2} A^{2} T^{1} A^{2} T^{1} G^{1} A^{1}
C^{1} G^{3} C^{1} A^{3} C^{2} T^{2} C^{42} G^{1} C^{2} T^{1} A^{1} C^{1} G^{1} A^{2} C^{1} T^{19} C^{1} T^{47} G^{1} T^{21}
C^{1} T^{2} A^{2} T^{1} C^{3} T^{1} A^{2} C^{1} A^{9} T^{1} A^{8} T^{1} A^{20} C^{1} T^{28} C^{1} T^{12} A^{1} T^{1} C^{1}
T^{1} C^{2} A^{1} C^{20} T^{1} C^{20} T^{2} C^{1} G^{1}

Figure 5 The RLBWT of the concatenation of the toy genomes shown in Figure 3, consisting of 449 runs (20 runs per line).

3, 2, 86, 88^{12}, 41, 88^{6}, 89, 41, 88, 87, 88^{4}, 16, 63, 88^{11}, 19,
55^{5}, 79, 55^{25}, 51, 55, 45, 55^{2}, 58, 55, 64, 19, 6, 73, 79, 55,
79^{2}, 45, 79^{2}, 65, 79^{9}, 45, 79^{5}, 18, 79, 82, 12^{3}, 70, 73, 6^{2}, 67,
90, 6^{3}, 10, 6^{3}, 5, 6, 84, 73, 6, 73^{7}, 87, 70, 30, 73^{2}, 68,
59, 30, 73, 33, 73^{2}, 60, 73^{3}, 76, 73, 85, 73, 13, 73^{3}, 4, 6^{3},
83, 6^{8}, 4, 6^{8}, 77, 73, 55, 19, 57, 19, 50, 19^{4}, 50, 19, 50,
19^{3}, 57, 19, 50, 19, 81, 50, 19^{6}, 66, 19, 50, 19, 50, 19, 50^{3},
74, 78, 66, 50, 49, 12, 0, 40, 48, 73, 26, 34, 62, 7, 1,
22, 18, 22, 19, 50^{10}, 8, 22^{12}, 50, 22^{3}, 50, 28, 50, 22^{17}, 71, 22,
71, 22^{8}, 72, 22^{11}, 29, 44, 22^{21}, 45, 55, 45, 27, 36, 17, 35, 22,
20, 23, 12, 47, 12^{9}, 56, 12^{2}, 80, 12^{2}, 46, 12^{10}, 61, 46, 12^{5}, 79,
50, 64, 88, 32, 55, 11, 69, 38, 32^{2}, 31, 32, 55, 32^{3}, 52, 25,
45, 32, 37, 42, 32, 58, 32, 39, 32^{5}, 53, 32, 75, 32^{3}, 19, 32,
41, 43, 58, 45, 55, 9, 55^{14}, 45, 55^{3}, 45, 55, 54, 6, 22, 51,
55, 64, 19, 64, 78, 64^{7}, 21^{2}, 64^{6}, 14, 64^{11}, 21, 64, 24, 64^{5}, 15,
64

Figure 6 The RLBWT of the sequence shown in Figure 6, consisting of 226 runs (15 runs per line).

Li [14] gave a practical algorithm, called forward-backward, for finding all the MEMs of P with respect to T using FM- or RLFM-indexes for T and its reverse T^{rev}. Assume all the distinct characters in P occur in T; otherwise, we split P into maximal substrings consisting only of copies of characters occurring in T and find the MEMs of those with respect to T. We first use the index for T^{rev} to find the longest prefix $P[0..e_1]$ of P that occurs in T, which is the leftmost MEM. If $e_1 = m - 1$ then we are done; otherwise, $P[e_1 + 1]$ is in the next MEM, so we use the index for T to find the longest suffix $P[s_2..e_1 + 1]$ of $P[0..e_1 + 1]$ that occurs in T. The next MEM starts at s_2, so conceptually we recurse on $P[s_2..m - 1]$. The total number of backward steps in the two indexes is proportional to the total length of all the MEMs.

Gagie [9] proposed a heuristic for speeding up forward-backward when we are interested only in MEMs of length at least L. We call this heuristic Boyer-Moore-Li, following a suggestion from Finlay Maguire [16]. Since any MEM of length at least L starting in $P[0..L-1]$ includes $P[L-1]$, we first use the index for T to find the longest suffix $P[s..L-1]$ of $P[0..L-1]$ that occurs in T. If $s = 0$ then we fall back on forward-backward to find the leftmost MEM and the starting position of the next MEM. Otherwise, since we know there are no MEMs of length at least L starting in $P[0..s-1]$, conceptually we recurse on $P[s..m-1]$. Li [15] tested Boyer-Moore-Li and found it practical enough that he incorporated it into his tool `ropebwt3`.

Suppose we build an `rsync`-like parse of $T[0..n-1]$ and two-level indexes for T and T^{rev} based on that parse and parse P when we get it. With a naïve two-level version of Boyer-Moore-Li, we would simply use the two-level indexes in place of the normal FM- or RLFM-indexes for T and T^{rev}. We conjecture, however, that we can do better in practice.

Let $P[k]$ be the last character of the last phrase that ends strictly before $P[L]$, let $P[j]$ be the first character of the first phrase such that $P[j..k]$ occurs in T, and let $P[i]$ be the second character of the phrase preceding the one containing $P[j]$. Notice we can find i, j and k by matching phrase by phrase using only the top level (for the parse) of the two-level index for T. If $i > 0$ then we can immediately discard $P[0..i-1]$ and conceptually recurse on $P[i..m-1]$; otherwise, we proceed normally.

Of course, the value i is at most the value s found by regular Boyer-Moore-Li and could be much smaller, in which case discarding $P[0..i-1]$ benefits us much less than discarding $P[0..s-1]$. We hope this is usually not the case and we look forward to testing Boyer-Moore-Li with two-level indexing.

References

1 Omar Y Ahmed, Massimiliano Rossi, Travis Gagie, Christina Boucher, and Ben Langmead. SPUMONI 2: improved classification using a pangenome index of minimizer digests. *Genome Biology*, 24:122, 2023.

2 Lorraine AK Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, and Solon P Pissis. U-index: A universal indexing framework for matching long patterns. *arXiv*, 2025. `arXiv:2502.14488`.

3 Paul Beame and Faith E Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65:38–72, 2002. `doi:10.1006/JCSS.2002.1822`.

4 Nathaniel K Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In *Proceedings of the 20th International Symposium on Experimental Algorithms (SEA)*, 2022.

5 Francisco Claude, Patrick K Nicholson, and Diego Seco. On the compression of search trees. *Information Processing & Management*, 50:272–283, 2014. `doi:10.1016/J.IPM.2013.11.002`.

6 Jin-Jie Deng, Wing-Kai Hon, Dominik Köppl, and Kunihiko Sadakane. FM-indexing grammars induced by suffix sorting for long patterns. In *Proceedings of the Data Compression Conference (DCC)*, 2022.

7 Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12:958–968, 2021.

8 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52:552–581, 2005. doi:10.1145/1082036.1082039.

9 Travis Gagie. How to find long maximal exact matches and ignore short ones. In *Proceedings of the 28th Conference on Developments in Language Theory (DLT)*, 2024.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67:1–54, 2020. doi:10.1145/3375890.

11 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35:378–407, 2005. doi:10.1137/S0097539702402354.

12 Aaron Hong, Marco Oliva, Dominik Köppl, Hideo Bannai, Christina Boucher, and Travis Gagie. PFP-FM: an accelerated FM-index. *Algorithms for Molecular Biology*, 19:15, 2024. doi:10.1186/S13015-024-00260-8.

13 Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Symposium on Combinatorial Pattern Matching (CPM)*, 2009.

14 Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28:1838–1844, 2012. doi:10.1093/BIOINFORMATICS/BTS280.

15 Heng Li. BWT construction and search at the terabase scale. *Bioinformatics*, 40:btae717, 2024.

16 Finlay Maguire. Personal communication, 2024.

17 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12:40–66, 2005.

18 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17:281–308, 2010. doi:10.1089/CMB.2009.0169.

19 Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3:25–47, 2000. doi:10.1023/A:1013002601898.

20 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2021.

21 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

22 Jannik Olbrich, Thomas Büchler, and Enno Ohlebusch. Generating multiple alignments on a pangenomic scale. *Bioinformatics*, 41(3):btaf104, 2025.

23 Alberto Ordóñez, Gonzalo Navarro, and Nieves R Brisaboa. Grammar compressed sequences with rank/select support. *Journal of Discrete Algorithms*, 43:54–71, 2017. doi:10.1016/J.JDA.2016.10.001.

24 Jouni Sirén. *Compressed Full-Text Indexes for Highly Repetitive Collections*. PhD thesis, University of Helsinki, 2012.

25 Jukka Teuhola. Interpolative coding of integer sequences supporting log-time random access. *Information Processing & Management*, 47:742–761, 2011. doi:10.1016/J.IPM.2010.11.006.

26 Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *iScience*, 27, 2024.

27 Alan Zheng, Ishmeal Lee, Vikram S. Shivakumar, Omar Y. Ahmed, and Ben Langmead. Fast and flexible minimizer digestion with digest. *bioRxiv*, 2025. URL: https://www.biorxiv.org/content/early/2025/01/08/2025.01.02.631161.

Turing Arena Light: Enhancing Programming Education Through Competitive Environments

Giorgio Audrito
University of Turin, Italy

Luigi Laura
International Telematic University Uninettuno, Rome, Italy

Alessio Orlandi
Google, Zürich, Switzerland

Dario Ostuni
Università degli Studi di Milano, Italy

Romeo Rizzi
Università di Verona, Italy

Luca Versari
Google, Zürich, Switzerland

Abstract

Turing Arena light, the spiritual successor of Turing Arena, is a contest management system that is designed to be more geared towards the needs of classroom teaching, rather than competitive programming contests. It strives to be as simple as possible, while being very flexible and extensible.

The fundamental idea behind Turing Arena light is to have two programs that talk to each other through the standard input and output channels. One of the two programs is the problem manager, which is a program that interacts with a solution to give it the input and evaluate its output, and eventually give a verdict. The other program is the solution, which is the program written by the contestant that is meant to solve the problem.

In this paper we describe the architecture and the design of Turing Arena light.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Software and its engineering → Development frameworks and environments; Social and professional topics → Computing education

Keywords and phrases Competitive Programming, Contest Management Systems, Online Judges

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.11

Category Education

Supplementary Material *Software (Source Code)*: `https://github.com/romeorizzi/TALight` [32] archived at `swh:1:dir:7e06febf4432bbcf7ce6301cd4de80837fa094d4`

Acknowledgements All the authors of this contribution first met thanks to Roberto Grossi and his dedicated work with the Italian (and International) Olympiads in Informatics. TALight is also a successor of the Fully Integrated Contest Analyzer that Roberto and some of the authors were planning a few years ago that evolved into the CMS [19]. This paper, like the entire volume, is dedicated to Roberto.

1 Introduction

Programming contest management systems are the backbone of competitive programming events, handling everything from problem distribution and solution submission to automated judging and live scoreboarding [28, 19, 20]. Over the years, these systems have evolved from ad-hoc scripts and manual procedures into sophisticated platforms that emphasize

security, scalability, and fairness [18]. Traditional contest systems like the **Programming Contest Control System (PC²)**, used in ACM ICPC since the 1990s, enabled basic contest operations (login, submissions, judging interface) and were reliable for on-site contests. However, many early systems required judges to manually run solutions or provided limited automation.

Turing Arena light (TALight) is a new contest management system that distinguishes itself by focusing on *simplicity, interactivity, and flexibility*. It was conceived as a lightweight platform geared toward educational use and practice environments rather than large-scale contests [31]. TALight's design philosophy is to keep the core system minimal and conceptually simple, delegating most functionality to problem-specific modules. Uniquely, all problems in TALight are treated as interactive by default, meaning a contestant's solution interacts in real-time with a problem manager program that provides inputs and checks outputs.

2 Turing Arena light

In this chapter we introduce *Turing Arena light*, the successor of *Turing Arena* [31]. *Turing Arena light* is a contest management system that is designed to be more geared towards the needs of classroom teaching, rather than competitive programming contests. It strives to be as simple[1] as possible, while being very flexible and extensible.

While we will discuss each point in more detail later, as an overview the design of *Turing Arena light* focuses on the following aspects:

- *Simplicity*: the design of *Turing Arena light* tries to keep things as simple as possible, while achieving the desired functionalities. While a meaningful objective metric for simplicity is hard to define, the current implementation of *Turing Arena light* consists of only 2197 lines of code, with an average of 39 chars per line and an overall of 85666 bytes [32].
- *Interactivity*: in *Turing Arena light* all problems are interactive by default. This means the contestant's solution for a problem always interacts in real-time with the problem. In particular, a problem in *Turing Arena light* is defined by the problem *manager*, which is a program that interacts with the contestant's solution and gives a verdict at the end of the interaction. By being interactive by default, *Turing Arena light* allows a wider range of problems to be implemented with less effort, while not causing much overhead for non-interactive problems.
- *Flexibility*: *Turing Arena light* is designed to be able to run on all major operating systems, and allow solutions and problem managers to be written in any programming language, while still being able to guarantee a certain level of security. To achieve this, *Turing Arena light* only consists of a small core written in Rust [21], whose main purpose is to spawn the process of the problem manager on the server, to spawn the process of the contestant's solution on its own machine, and to connect the standard input and output of the two processes. Thus, the contestants' code is never run on the server, and the problem manager can run without a sandbox, being trusted code written by the problem setter.
- *Extensibility*: as stated in the previous point, *Turing Arena light* only consists of a small core that has the fundamental role of spawning to processes and connecting their standard input and output. All the other functionalities are implemented by the problem manager itself, possibly using a common library of utilities. This allows the problem setter to implement any kind of problem, while still being able to use the same contest management system.

[1] Simple might mean very different things, in this context it is conceptual simplicity.

3 Related Work

Competitive programming systems can be classified into three main categories, i.e. Contest Management Systems (CMS), Online Judges and Classroom Ad-hoc Tools, each serving distinct purposes while sharing some overlapping features.

Contest Management Systems (CMS) are sophisticated platforms specifically designed for formal competitions like the International Olympiad in Informatics (IOI), ACM-ICPC, or national olympiads. Systems like DOMjudge [26], CMS [19, 20], and Kattis [6] provide robust infrastructure for high-stakes, in-person events. They focus on security, reliability, and scalability to handle numerous concurrent submissions while maintaining fair evaluation conditions. These systems typically include features like real-time scoreboards, detailed analytics for judges, and stringent sandboxing mechanisms to ensure solution integrity. CMS platforms prioritize standardized evaluation environments where all participants compete under identical conditions with controlled resource limitations.

Online Judges serve a broader educational purpose by providing continuous access to problem-solving opportunities outside formal competitions. Platforms like Codeforces, LeetCode, and SPOJ host extensive problem libraries that users can attempt at their own pace. Unlike Contest Management Systems, they emphasize learning progression through difficulty-ranked challenges, detailed performance statistics, and community engagement via discussion forums and editorials. While they can host virtual contests, their primary value lies in self-directed practice. Many online judges incorporate gamification elements like ratings, badges, and streaks to motivate continued participation. Furthermore, since these systems have, in some cases, order of thousands of different tasks, there is a vast literature related to the development of recommender systems able to suggest a suitable task depending on the learner's abilities [1, 7, 9, 8]; also the problem of plagiarism is addressed [17]. We refer the interested reader to the surveys of Wasik et al. [37] and Watanobe et al. [38].

Classroom Ad-hoc Tools like Turing Arena Light are specifically tailored for educational settings where pedagogical considerations outweigh competitive rigor. These systems prioritize ease of use, interactive problem types, and flexibility to accommodate diverse learning objectives. Unlike the standardized environments of CMS platforms, classroom tools often allow students to work in familiar development environments on their own machines. They typically feature simplified interfaces, immediate feedback mechanisms, and support for interactive problems that engage students through real-time interactions. While less suited for large-scale competitions, these tools excel at reinforcing classroom concepts and providing instructors with meaningful insights into student progress.

4 Architecture and design

This section explores the technical architecture and design principles that form the foundation of Turing Arena Light. We begin by explaining the core interaction model between problem managers and solutions, which differentiates TALight from traditional contest management systems. Then, we examine each primary component in detail: the problem manager that defines and evaluates tasks, the server that orchestrates communication, the client that runs on contestants' machines, and the user interface that contestants interact with. Throughout this section, we highlight how TALight's design choices support its goals of simplicity, interactivity, flexibility, and extensibility while maintaining a lightweight yet powerful infrastructure for educational programming environments.

The fundamental idea behind *Turing Arena light* is to have two programs that talk to each other through the standard input and output channels. One of the two programs is the problem *manager*, which is a program that interacts with a solution to give it the input and evaluate its output, and eventually give a verdict. The other program is the *solution*, which is the program written by the contestant that is meant to solve the problem.

While this is not too far off from what other contest management systems do, the two main differences are that in *Turing Arena light* these two programs run on different machines, and the interaction between them is done in real-time. This is unlike mainstream contest management systems, where the two programs run on the same machine (like in DOMjudge [5], CMS [19] and Codeforces [3]), or where the interaction is not done in real-time (like in the old Google Code Jam [12] and Meta Hacker Cup [23]).

In the following subsections we will discuss the components of *Turing Arena light* and how they interact with each other. We will start from the problem manager, going through the server and the client, and finally discussing the user interface.

4.1 Problem manager

```
%YAML 1.2
---
public_folder: public
services:
  free_sum:
    evaluator: [python, free_sum_manager.py]
    args:
      numbers:
        regex: ^(onedigit|twodigits|big)$
        default: twodigits
      obj:
        regex: ^(any|max_product)$
        default: any
      num_questions:
        regex: ^([1-9]|[1-2][0-9]|30)$
        default: 10
      lang:
        regex: ^(hardcoded|hardcoded_ext|en|it)$
        default: it
  help:
    evaluator: [python, help.py]
    args:
      page:
        regex: ^(free_sum|help)$
        default: help
      lang:
        regex: ^(en|it)$
        default: it
```

Figure 1 Description file for a problem in *Turing Arena light*.

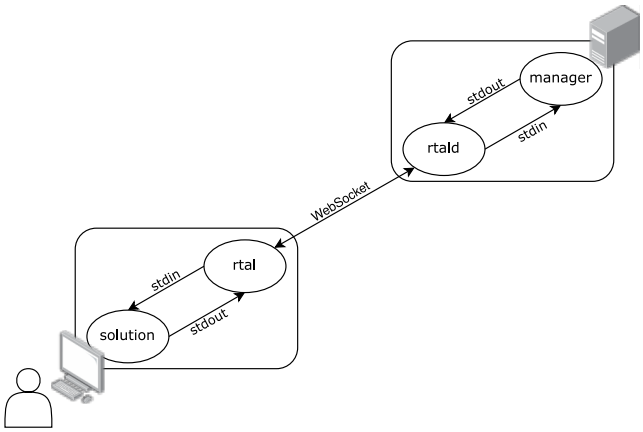


Figure 2 Architecture of *Turing Arena light*.

A problem in *Turing Arena light* is defined as a set of *services* and a set of *attachments*. A service is a program that can be spawned with a set of well-defined parameters, and that will ultimately interact with the solution. An attachment is a generic file that can be attached to the problem and downloaded by the contestant, such as the statement of the problem, or a library that the contestant can use in their solution.

A service defines which parameters it accepts, and the accepted values for each parameter. Parameters can be either strings or files. Each string parameter has a regular expression that defines the set of accepted values and a default value. Furthermore, a service defines which program will be invoked with the given parameters: the problem *manager* (also called the *evaluator*). The attachments are just regular files in a folder on the file system.

The description of a problem is contained in a file called `meta.yaml`, which is a YAML [2] file. The file contains the description of all the services, and their parameters, and the directory of the attachments. An example of a `meta.yaml` file is shown in Figure 1. Thus, a problem in *Turing Arena light* is represented by a folder containing a `meta.yaml` file, and all the files and subdirectories needed for services and attachments.

4.2 Server

After the problem manager, there is the server. The server is the beating heart of *Turing Arena light*: its role is to accept incoming connections from the clients, spawn the problem manager corresponding to the client requested problem and service, passing to it the parameters specified by the client, and finally connect the standard input and output of the problem manager to the client.

Note that up to this point, *Turing Arena light* is merely a specification of how the problem is defined and how the interaction between the problem manager and the solution should happen. This opens up the possibility of having multiple implementations of the *Turing Arena light* framework, since the specification is very simple and does not require any particular technology, such as sandboxing.

Currently, there is only one implementation of the *Turing Arena light* framework, which is `rtal` (*Rust Turing Arena light*). It is written in Rust [21], and it is the reference implementation of *Turing Arena light*. The server component, `rtald`, is a small program that, given a folder containing problems, listens for incoming connections from the clients, and spawns the correct problem manager, and relays the standard input and output of the problem manager to the client via a protocol based on WebSockets [10].

4.3 Client

On the other side of the network[2] there is the client. The client is the program that the contestant runs on their machine to connect to the server and interact with the problem manager. Its role is to connect to the server, send the request for a problem and a service, send the string and file parameters for the service, and finally spawn and attach itself to the standard input and output of the solution running on the local machine of the contestant.

Once everything is up and running, the client will send the standard output of the solution to the server, which will relay it to the problem manager, and forward on the standard input of the solution all the incoming data from the server. Basically, the client is a proxy that connects the standard input and output of the solution to the server.

Like the server, there is also a `rtal` component for the client, also called `rtal`. This client component is a command line program that takes as parameters the address of the server, the problem and the service, and the parameters for the service. It also takes the command to run the solution. The client will then connect to the server, send the request for the problem and service, and spawn the solution with the given command, proxying the data between the solution and the server.

4.4 User interface

As far as the contestant is concerned, what they must do is to write a solution to the problem in their favourite programming language. The only requirement is that it reads from the standard input and writes to the standard output. To read the problem statement, the contestant can download the attachments of the problem using the client. The client will download the attachments and save them on the local machine of the contestant.

Once the solution is ready, the contestant can run the client passing the right parameters, including the command to run their solution. The client will then connect to the server, send the request for the problem and service, and spawn the solution with the given command. Note that the solution is spawned and run on the local machine of the contestant, which means that the contestant has full freedom on which files it can read and write, which resources it can use, and so on. This is unlike other contest management systems that support real-time interaction, where the solution is run on a sandboxed environment on the server.

The ability to run the solution on the local machine opens to many possibilities. For example, the contestant can precompute some large set of data, save it on their machine, and then use it during the interaction with the problem manager to speed up the computation. Another example is the potential to use external libraries, multithreading, or even GPU computation. All of this is possible because the solution is run on the local machine of the contestant, where they have full control, and not on the server.

5 Implementation details

As mentioned in the previous section, *Turing Arena light* currently has only one full implementation, which is *Rust Turing Arena light* (`rtal`). Like the name suggests, it is written in Rust [21]. The choice of language was motivated by the fact that Rust is a systems programming language, and thus it is well suited for writing low-level programs that need

[2] Which might even be on the same machine, if both the server and the client are running on the same machine.

```rust
pub const META: &str = "meta.yaml";

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Problem {
    pub name: String,
    pub root: PathBuf,
    pub meta: Meta,
}

#[derive(Debug, Default, Serialize, Deserialize, Clone)]
pub struct Meta {
    pub public_folder: PathBuf,
    pub services: HashMap<String, Service>,
}

#[derive(Debug, Default, Serialize, Deserialize, Clone)]
pub struct Service {
    pub evaluator: Vec<String>,
    pub args: Option<HashMap<String, Arg>>,
    pub files: Option<Vec<String>>,
}

#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct Arg {
    #[serde(with = "serde_regex")]
    pub regex: Regex,
    pub default: Option<String>,
}
```

Figure 3 Problem description definition in *Rust Turing Arena light*.

to interact with the operating system and other programs. Furthermore, one key factor is portability: Rust is a compiled language whose compiled binaries require only minimal external dependencies to run, which makes it ideal to produce distributable binaries. This is important because *Turing Arena light* is meant to be used by students, which might not have the technical knowledge to install and configure a complex system. Having a single binary that can be downloaded and run without any configuration is a big advantage.

The implementation of *Turing Arena light* is split into three components: the server (`rtald`), the client (`rtal`), and the checker (`rtalc`). All three components share some common parts. The main one is the problem description definition, also known as the `meta.yaml` file. The definition can be found in Figure 3. The definition is written using Rust structures which are then serialized to and deserialized from YAML using *serde* [29], a serialization framework for Rust. `rtalc` is a small independent command-line program that takes as input a directory containing the problem description, and checks that the description is valid and matches the content of the directory. This is useful to check that the problem description is correct before uploading it to the server.

The two main jobs of the client and the server are process spawning and networking. For both of these tasks, `rtal` and `rtald` use the *tokio* [30] library, which is a framework for writing asynchronous programs in Rust. For the process spawning part, there is nothing particularly interesting: the server spawns the problem manager, and the client spawns the

```rust
pub const MAGIC: &str = "rtal";
pub const VERSION: u64 = 4;

#[derive(Serialize, Deserialize, Debug)]
pub enum Request {
    Handshake {
        magic: String,
        version: u64,
    },
    MetaList {},
    Attachment {
        problem: String,
    },
    ConnectBegin {
        problem: String,
        service: String,
        args: HashMap<String, String>,
        tty: bool,
        token: Option<String>,
        files: Vec<String>,
    },
    ConnectStop {},
}

#[derive(Serialize, Deserialize, Debug)]
pub enum Reply {
    Handshake { magic: String, version: u64 },
    MetaList { meta: HashMap<String, Meta> },
    Attachment { status: Result<(), String> },
    ConnectBegin { status: Result<Vec<String>, String> },
    ConnectStart { status: Result<(), String> },
    ConnectStop { status: Result<Vec<String>, String> },
}
```

Figure 4 Network protocol definition in *Rust Turing Arena light*.

solution. They then, through *tokio*, manage the channels of the standard input and output of the spawned processes. All the internal communication within the server and the client is done using the actor threading model [15, 16].

For the networking part, the communication protocol between the server and the client is based on WebSockets [10]. The protocol definition is shown in Figure 4. The protocol is based on JSON [4] messages, which are serialized and deserialized using *serde*. These messages are then exchanged between the server and the client using WebSockets. The interaction between the server and the client is shown in Figure 2. Using WebSockets enables a client of *Turing Arena light* to be implemented as a web application.

Both `rtal` and `rtald` run their spawned processes in an unsandboxed environment. This is done to avoid the complexity of sandboxing, but we argue that it does not pose a major security risk. The reason is that, for the client, the program being run is the contestant's own written solution, which is run on their local machine. Thus, the contestant has full control over the program, and can do whatever they want with it. For the server, the program being run is the problem manager, which is written by the problem setter. Thus, as long as the

```python
class TC:
  def __init__(self, data, time_limit=1):
    self.data = data
    self.tl = time_limit

  def run(self, gen_tc, check_tc):
    output = open(join(environ["TAL_META_OUTPUT_FILES"], "result.txt"
        ), "w")
    total_tc = sum(map(lambda x: x[0], self.data))
    print(total_tc, flush=True)
    tc_ok = 0
    tcn = 1
    for subtask in range(len(self.data)):
      for tc in range(self.data[subtask][0]):
        tc_data = gen_tc(*self.data[subtask][1])
        stdout.flush()
        start = time()
        try:
          ret = check_tc(*tc_data)
          msg = None
          if isinstance(ret, tuple):
            result = ret[0]
            msg = ret[1]
          else:
            result = ret
          if time() - start > self.tl:
            print(f"Case #{tcn:03}: TLE", file=output)
          elif result:
            print(f"Case #{tcn:03}: AC", file=output)
            tc_ok += 1
          else:
            print(f"Case #{tcn:03}: WA", file=output)
          if msg is not None:
            print(file=output)
            print(msg, file=output)
            print(file=output)
        except Exception as e:
          print(f"Case #{tcn:03}: RE", file=output)
          print(file=stderr)
          print("".join(traceback.format_tb(e.__traceback__)), e,
              file=stderr)
        tcn += 1
    print(file=output)
    print(f"Score: {tc_ok}/{total_tc}", file=output)
    output.close()
```

Figure 5 Snippet of the python version of the competitive-programming like problem manager library for *Turing Arena light*.

problem setter is trusted, there is no need to sandbox the problem manager. This is usually the case, as the problem setter is the one who also is responsible for the server where the `rtald` program is running. If this is not the case, then `rtald` can be run in a virtualized environment, such as a Docker container [22], to mitigate the risk of a bug in the problem manager that could cause unauthorized access to the server.

5.1 Problem manager libraries

So far we have discussed the architecture, the design and the implementation of *Turing Arena light*. However, we have not yet discussed how the problem manager is implemented. As mentioned in the previous sections, the problem manager is a program that interacts with the solution, and gives a verdict at the end of the interaction. The problem manager, just like the solution, has to communicate with its counterpart, which is the solution, using the standard input and output channels. Thus, the problem manager has full freedom on how to interact with the solution, as long as it does so using the aforementioned channels.

While this grants the problem maker a great deal of freedom, it also means that the problem maker has to potentially write a lot of boilerplate code each time they want to implement a new problem. To mitigate this problem, a problem maker can create a library of utilities that can be used to implement the problem manager. This library can be based on a particular style of problems, so that the problem maker can offer a consistent experience to the contestants.

In our case, we wrote a library called `tc.py`. A snippet of the library is shown in Figure 5. This library allows to write a *old-Google-Code-Jam* like problem by only writing the code essential to the problem, and leaving all the boilerplate code to the library. What the manager has to implement is a function that generates a test case, and a function that evaluates the solution given by the contestant on a test case. The library will then take care of the rest, including enforcing the time limit, generating the right number of test cases, and assigning and storing the score for the solution. Note that with the *Turing Arena light* there is no way to enforce the memory limit, as the solution is run on the local machine of the contestant. However, the time limit can be enforced by measuring how much time passes between the sending of the input and the receiving of the output. While this is not a very precise measurement, it is good enough for distinguishing between solutions that have very different computational complexities.

```
CREATE TABLE users (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    other TEXT
);

CREATE TABLE problems (
    name TEXT PRIMARY KEY
);

CREATE TABLE submissions (
    id INTEGER PRIMARY KEY,
    user_id TEXT NOT NULL,
    problem TEXT NOT NULL,
    score INTEGER NOT NULL,
    source BLOB NOT NULL,
    address TEXT,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (problem) REFERENCES problems(name)
);
```

Figure 6 SQLite schema for database used by `tc.py` and `tc.rs`.

As the name suggests, the `tc.py` library is written in Python [36], and it is meant to be used with problem managers written in Python. This works great for problems where the optimal solution plays well with Python, however in problems where the performance of the solution is critical, having the problem manager written in Python may make the evaluation of the contestant's output too slow. To mitigate this problem, we ported the `tc.py` library to Rust, thus creating the `tc.rs` library [34]. By using Rust as the programming language for the problem manager, the whole execution of the problem manager is much faster. The functionality of the two libraries is the same, and they are interoperable with each other. This means that in a single contest, the problem maker can use both Python and Rust problem managers.

Turing Arena light has no built-in support for saving the results of the contest, as this job is left to the problem manager. This is done to allow the problem maker to have full control over how the results are saved. In `tc.py` and `tc.rs` we implemented a simple database that saves the results of the contest in a SQLite [25] database. The schema of the database is shown in Figure 6. The database provides a way to save the results of the contest, and it enables contestants to see their position in the ranking during the contest, using a service defined in *Turing Arena light*.

6 Graphical user interface

The Rust implementation of *Turing Arena light* only comes with a command line interface for the client. While this is enough to run the contest, it is not very user friendly. Contestants have to remember the right parameters to pass to the client, and the less experienced ones might have trouble working with a terminal. To mitigate this problem, a graphical user interface for the client was developed.

A web application was developed as a new client for *Turing Arena light* [33]. A screenshot of the application is shown in Figure 7. It was developed using the *Angular* framework [14], and it is written in TypeScript [35]. The peculiar thing about this application is that aside

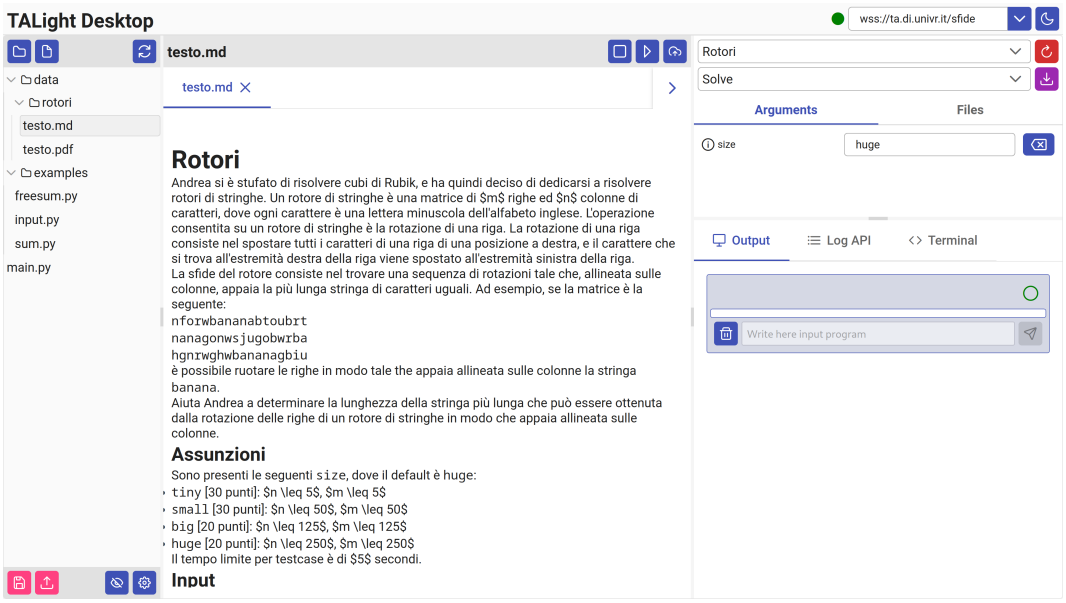


Figure 7 Graphical user interface of *Turing Arena light*.

from offering all the functionalities of the command line client, it also offers a way to write the solution directly in the browser. Not only that, but the solution is run directly in the browser, without the need to install any additional software. This functionality is currently only available for Python solutions, but it could be extended to other languages as well. To do this, the Python interpreter has been compiled to JavaScript, using *Pyodide* [27]. This allows to run Python code directly in the browser. Thus, the contestant can do everything from an integrated environment in its browser.

Aside from running the solution in the browser, the web application also implements an emulated file system within the browser. This allows the contestant to send file parameters and receive file attachments and file outputs, all from the browser. Another useful feature that derives from having a file system is the ability to save and restore the working environment. This is useful for example when the contestant is working on a problem, and they want to save their progress and continue working on it later. Another scenario is when a template is provided to the contestant, and they can start working directly on it. The file system can be exported as a tar archive, or can be stored in the cloud using either GitHub [11], Google Drive [13], or OneDrive [24]. They can be later imported back from a tar archive or from the cloud, specifically from GitHub.

7 Future directions

Turing Arena light has been developed enough to be used in a real-world classroom setting, and it has been used in the course of *Competitive Programming* at the University of Verona. It has been used for both the laboratory lessons and the exams, and it has been well received by the students. However, there is still a debate to be had in which direction *Turing Arena light* should move forward.

While the extreme flexibility of *Turing Arena light* made it possible to experiment a lot with different kinds of problems, it also made it difficult to find a common ground on which to standardize some common features, without having all of the problem manager libraries reimplement them. One such feature is the ability to save the results of the contest. While *Turing Arena light* does not have any built-in support for saving the results of the contest, it is possible to implement it in the problem manager. However, this means that each problem manager has to reimplement the same functionality, which is not ideal.

Moreover, some feature are implementable only by standardizing them at the core of *Turing Arena light*. One such feature is the ability of accurately measuring the time consumed by the solution. Right now, the time used by the solution is measured by measuring the time between the sending of the input and the receiving of the output. However, this is not a very accurate measurement, as it does not take into account the time spent sending and receiving the packets over the network. This is not a problem when the server and the client are on the same local network, as it happened in the course of *Competitive Programming*, but it becomes a problem when the server and the client are on different networks, such as when the server is on the Internet.

There is a solution to mitigate this problem, which is to encrypt the data, send it, then start the clock and send the decryption key. Doing it this way, one can eliminate the time spent sending the data, which can be a significant amount of time when the input is big. However, to implement such a solution, it would require to have some mechanism to make the problem manager and the core communicate on a meta-level to require this functionality from the core. However, such mechanism could cause a narrowing of the flexibility of *Turing Arena light*.

We currently offer client implementations in Rust and a web-based interface, educators and users may prefer clients in other programming languages. Implementing new RTAL clients is relatively straightforward through two approaches:

- Binding Generation: The recommended approach involves generating language bindings from the core Rust library. This method ensures automatic compatibility with future protocol updates and requires minimal maintenance. Our Python implementation already follows this pattern, with the complete binding implementation available in our repository (`py.rs`[3]).
- Protocol Reimplementation: Alternatively, developers can independently implement the WebSocket-based communication protocol used between server and client. This is feasible due to the protocol's simplicity – it consists of only 11 distinct message types as defined in our protocol specification (`proto.rs`[4]). However, this approach requires manual updates to each client implementation whenever the protocol evolves.

Finally, while the command-line interface has worked great for the course of *Competitive Programming*, it is not very probable that it would be fine for other courses with less *programming-focused* students. Thus, the development of the graphical user interface continues, and it is planned to be tested in the next iteration of the course of *Competitive Programming*, and possibly in other courses with more *less specialized* students.

References

1 Giorgio Audrito, Tania Di Mascio, Paolo Fantozzi, Luigi Laura, Gemma Martini, Umberto Nanni, and Marco Temperini. Recommending tasks in online judges. In *Advances in Intelligent Systems and Computing*, pages 129–136. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-23990-9_16.

2 Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009.

3 Codeforces. URL: https://codeforces.com/.

4 Douglas Crockford. The application/json media type for javascript object notation (json). Technical report, IETF, 2006. doi:10.17487/RFC4627.

5 Jaap Eldering, Thijs Kinkhorst, and Peter van de Warken. Dom judge–programming contest jury system. 2020, 2010.

6 Emma Enstrom, Gunnar Kreitz, Fredrik Niemela, Pehr Soderman, and Viggo Kann. Five years with kattis — using an automated assessment system in teaching. In *2011 Frontiers in Education Conference (FIE)*, pages T3J–1–T3J–6. IEEE, October 2011. doi:10.1109/FIE.2011.6142931.

7 P Fantozzi and L Laura. Recommending tasks in online judges using autoencoder neural networks. *Olymp. Inform.*, December 2020. doi:10.15388/ioi.2020.05.

8 Paolo Fantozzi and Luigi Laura. Collaborative recommendations in online judges using autoencoder neural networks. In *Advances in Intelligent Systems and Computing*, pages 113–123. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-53036-5_12.

9 Paolo Fantozzi and Luigi Laura. A dynamic recommender system for online judges based on autoencoder neural networks. In *Methodologies and Intelligent Systems for Technology Enhanced Learning, 10th International Conference. Workshops*, pages 197–205. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-52287-2_20.

10 Ian Fette and Alexey Melnikov. The websocket protocol, 2011. doi:10.17487/RFC6455.

11 Github. URL: https://github.com/.

[3] https://github.com/romeorizzi/TALight/blob/v0.2.5/rtal/src/py.rs
[4] https://github.com/romeorizzi/TALight/blob/v0.2.5/rtal/src/proto.rs

12 Google code jam. URL: https://codingcompetitionsonair.withgoogle.com/#code-jam.
13 Google drive. URL: https://drive.google.com/.
14 Brad Green and Shyam Seshadri. *AngularJS*. O'Reilly Media, Inc., 2013.
15 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, 1973. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.
16 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.
17 Fariha Iffath, A. S. M. Kayes, Md. Tahsin Rahman, Jannatul Ferdows, Mohammad Shamsul Arefin, and Md. Sabir Hossain. Online judging platform utilizing dynamic plagiarism detection facilities. *Comput.*, 10(4):47, April 2021. doi:10.3390/computers10040047.
18 José Paulo Leal and Fernando M. A. Silva. Mooshak: a web-based multi-site programming contest system. *Softw. Pract. Exp.*, 33(6):567–581, 2003. doi:10.1002/spe.522.
19 Stefano Maggiolo and Giovanni Mascellani. Introducing CMS: A contest management system. *Olympiads in Informatics*, 6, 2012.
20 Stefano Maggiolo, Giovanni Mascellani, and Luca Wehrstedt. Cms: a growing grading system. *Olympiads in Informatics*, page 123, 2014.
21 Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014. doi:10.1145/2663171.2663188.
22 Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.
23 Meta hacker cup. URL: https://www.facebook.com/codingcompetitions/hacker-cup.
24 Onedrive. URL: https://onedrive.live.com/.
25 Michael Owens. *The definitive guide to SQLite*. Springer, 2006.
26 Minh Tuan Pham and Tan Bao Nguyen. The domjudge based online judge system with plagiarism detection. In *2019 IEEE-RIVF International Conference on Computing and Communication Technologies, RIVF 2019, Danang, Vietnam, March 20-22, 2019*, pages 1–6. IEEE, March 2019. doi:10.1109/RIVF.2019.8713763.
27 Pyodide. URL: https://pyodide.org/.
28 Miguel A Revilla, Shahriar Manzoor, and Rujia Liu. Competitive learning in informatics: The uva online judge experience. *Olympiads in Informatics*, 2:131–148, 2008.
29 Serde. URL: https://serde.rs/.
30 Tokio. URL: https://tokio.rs/.
31 Turing arena. URL: https://github.com/turingarena/turingarena.
32 Turing arena light. URL: https://github.com/romeorizzi/TALight.
33 Turing arena light desktop. URL: https://talco-team.github.io/TALightDesktop/.
34 Turing arena light rust utilities. URL: https://github.com/dariost/tal-utils-rs.
35 Typescript. URL: https://www.typescriptlang.org/.
36 Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36. Santa Clara, CA, 2007.
37 Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. A survey on online judge systems and their applications. *ACM Comput. Surv.*, 51(1):3:1–3:34, January 2018. doi:10.1145/3143560.
38 Yutaka Watanobe, Md. Mostafizer Rahman, Taku Matsumoto, Rage Uday Kiran, and Penugonda Ravikumar. Online judge system: Requirements, architecture, and experiences. *Int. J. Softw. Eng. Knowl. Eng.*, 32(6):917–946, June 2022. doi:10.1142/S0218194022500346.

Encoding Data Structures for Range Queries on Arrays

Seungbum Jo
Chungnam National University, Daejeon, South Korea

Srinivasa Rao Satti
Norwegian University of Science and Technology, Trondheim, Norway

Abstract

Efficiently processing range queries on arrays is a fundamental problem in computer science, with applications spanning diverse domains such as database management, computational biology, and geographic information systems. A range query retrieves information about a specific segment of an array, such as the sum, minimum, maximum, or median of elements within a given range. The challenge lies in designing data structures that allow such queries to be answered quickly, often in constant or logarithmic time, while keeping space overhead (and preprocessing time) small. Encoding data structures for range queries has emerged as a pivotal area of research due to the increasing demand for high-performance systems handling massive datasets. These structures consider the data together with the queries and aim to store only as much information about the data as is needed to answer the queries. The data structure does not need to access the original data to answer the queries. Encoding-based solutions often leverage techniques from succinct data structures, bit manipulation, and combinatorial optimization to achieve both space and time efficiency. By encoding the array in a manner that preserves critical information, these methods strike a balance between query time and space usage. In this survey article, we explore the landscape of encoding data structures for range queries on arrays, providing a comprehensive overview of some important results on space-efficient encodings for various types of range query.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases range queries, RMQ, Cartesian tree, top-k queries, range median, range mode

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.12

Category Research

1 Introduction

Efficiently processing range queries on arrays is a fundamental problem in computer science, with applications spanning diverse domains such as database management, computational biology, geographic information systems, and data analytics. A range query retrieves specific information about a contiguous segment of an array, such as the sum, minimum, maximum, or median of elements within a given range. Designing data structures to process such queries efficiently is critical, especially as datasets grow in size and complexity. The challenge lies in enabling fast query responses – often aiming for constant or logarithmic time – while minimizing the space overhead and preprocessing time required by the data structures.

Encoding data structures for range queries have emerged as a crucial area of research to address these challenges. Unlike traditional approaches that rely on augmenting the array with auxiliary structures or preprocessing, encoding-based methods focus on storing a compact representation of the data tailored specifically to the types of queries to be answered. These structures are designed to store only the information necessary for query resolution, eliminating the need to access the original array during query processing. The encoding data structures typically combine techniques from succinct/compressed data structures, combinatorial optimization and algorithmic design.

This survey aims to provide a comprehensive and structured overview of the key developments in the field of encoding data structures for range queries on arrays. We categorize the encoding solutions based on the specific types of range queries they support, such as range minimum/maximum queries, range top-k queries, range mode queries, and range majority/minority queries. The survey article by Skala [51] from 2013 gives a detailed summary of various results for "range queries on arrays". Thus we mainly focus on the new results since 2013, briefly summarizing the previous results.

Data structures can be classified into two categories: *indexing* and *encoding* data structures. For an indexing data structure, we preprocess the data and build an *index* so that subsequent queries can be answered efficiently by probing the index and the input data. On the other hand, for an encoding data structure, we preprocess the input data and build an *encoding* of the input so that subsequent queries can be answered by probing only the encoding (i.e., with no access to the input at query time). In this survey, we mainly focus on the results on encoding data structures, but occasionally mention a few results on indexing data structures for comparison. For many problems, the size of an encoding can be smaller than the size of the input, which is in fact the case for many range queries that we consider.

For example, given a query range, a range minimum query returns the minimum element within the query range. Range median, range mode, range selection and top-k, range mode and majority/minority queries are defined analogously. With this definition of range queries, one can reconstruct the input array by asking point queries with range that contains a single element, and hence the size of an encoding is at least the size of the input, and storing the input explicitly gives an optimal space encoding. To avoid this, we define the range queries as follows. For a query range, the range minimum query returns a position of the minimum element within the query range (and analogously for other range queries), instead of the value at the position. With this definition, one can obtain an encoding of size linear in bits for range minimum queries, which is asymptotically less than the space required to store the input array in the word RAM model.

In this article, we consider the encoding data structures for a 1D array $A[1, n]$, and a 2D array $A[1, m][1, n]$, where $m \leq n$. We assume that the array indices start from 1. For 2D arrays, we use the term *range* to mean an rectangular range which is defined as a Cartesian product of a given set of intervals in each dimension. We assume a word RAM model with word-size $\Theta(\log n)$ bits. For the encoding structures where we do not mention the query times, queries can be supported in polynomial time by essentially decoding the entire structure.

2 Range minimum queries

Given an input array and a query range, a range minimum query (RMQ) returns the position of the minimum element within the query range. From its wide applications (e.g., constructing an indexing structure on a string [15]), designing a data structure to answer RMQ has intensive attention from the community. For the 1D case, any encoding for RMQ requires at least $2n - o(n)$ bits due to its bijective relationship with the Cartesian tree of the input [53]. The best-known result for a worst-case input is the $(2n + o(n))$-bit data structure by Fisher and Heun that supports $O(1)$ query time[1]. For earlier results, see [51].

When the input is highly compressible, there are some results whose space usages are parameterized based on the compressed size of the input while still supporting efficient query time. Here, compressibility is considered in two ways: (1) The compressibility of the

[1] considering the $o(n)$-term, the current best result is Navarro and Sadakane's $(2n + O(n/(\frac{\log n}{t})^t))$-bit data structure that supports queries in $O(t)$ time [45].

input [2, 18, 21, 49], and (2) the compressibility of the Cartesian tree of the input [21, 43]. Note that data structures in (1) can access the input, as they maintain the input array in a compressed form. In contrast, the results in (2) cannot access to the input as they maintain the compressed Cartesian tree with some auxiliary structures for the query support.

There also has been progress on space-query time trade-off lower bounds for the problem under the cell-probe model, which measures query time by counting the number of memory cell accesses only. Liu and Yu showed that any data structure answering RMQ in $O(t)$ time requires at least $2n + O(n/(\log n)^{O(t^2 \log^2 t)})$ bits of space [42]. Later, Liu improved the space lower bound to $2n + O(n/(\log n)^{O(t \log^2 t)})$ bits [41].

2.1 Range minimum and maximum queries

To address range minimum and maximum queries on 1D array simultaneously, a straightforward approach is to use separate data structures for each query. Since any data structure designed for range minimum queries can be easily adapted for range maximum queries, the data structure of Fischer and Heun [17] provide a $(4n + o(n))$-bit data structure that can answer both queries in $O(1)$ time. Gawrychowski and Nicolson [26] showed that if an array contains no consecutive equal values, there exists a data structure that uses $3n + o(n)$ bits and supports both queries in $O(1)$ time. Furthermore, they proved that any encoding data structure for answering both queries requires at least $3n - \Theta(\log n)$ bits, as any Baxter permutation can be fully reconstructed using only range minimum and maximum queries.

When the input array contains consecutive equal values, a straightforward solution is to use two separate data structures proposed by Fischer [14] for range minimum and maximum queries. This approach uses $5.08n + o(n)$ bits of space and supports both queries in $O(1)$ time. Additionally, for any given p, the data structure also can answer the position of the p-th leftmost minimum or maximum value within a query range in $O(1)$ time [35]. Jo and Satti [35] improved the space usage of the data structure to $4.585n + o(n)$ bits while maintaining the same $O(1)$ query time. Subsequently, Tsur [52] further reduced the space to $3.701n$ bits with $O(n)$ time for queries. Recently, Jo and Kim [33] proposed a data structure that uses $3.701n + o(n)$ bits while supporting all queries in $O(\log^{(\ell)} n)$ time for any positive integer ℓ (here $\log^{(\ell)}$ denotes logarithm iterated ℓ times for any constant $\ell \geq 1$). They also showed that any data structure for answering these queries requires at least $3.16n - \Theta(\log n)$ bits when the input contains consecutive equal values.

2.2 Range minimum queries on non-permutation input

Consider a 1D array with duplicate entries. In this case, some ranges may have multiple answers for RMQ. The data structures discussed in Skala's survey [51] return either the leftmost or rightmost position among the possible answers. Motivated by the efficient construction of compressed suffix trees [50], Fischer and Heun [16] considered the problem of finding the position of an approximated median among all positions of minimums within a given range. Specifically, for any constant $1 < c < 1/2$, they proposed a $(\log(3 + 2\sqrt{2})n + o(n/c) \approx 2.54n + o(n/c))$-bit data structure that can answer the position of the r-th leftmost minimum in $O(1/c)$ time, where $r \in [\frac{1}{2}(1/2 - c), \frac{1}{2}(1/2 + c)]$. The data structure uses a super Cartesian tree, a Cartesian tree in which each edge is colored either red or blue.

2.3 Range minimum queries on 2D array

When the input is an $m \times n$ 2D array with $m \leq n$, Demaine et al. [11] showed that there is no Cartesian tree-like structure for the 2D case. Specifically, no structure exists that fully encodes the answer to all queries and can be constructed in linear time. They further

showed that when $m = n$, any encoding data structure for answering RMQ queries requires $\Omega(n^2 \log n)$ bits. Since the input array can be stored using $O(n^2 \log n)$ bits, this implies that any encoding data structure uses asymptotically the same space as indexing data structures [1, 5, 54] when $m = \Theta(n)$. In this survey, we focus on the case where $m = o(n)$.

Brodal et al. [5] proposed an $O(nm \cdot \min(m, \log n))$-bit data structure with $O(1)$ query time. The $O(nm^2)$-bit data structure is achieved by maintaining $O(m^2)$ 1D RMQ structures to support queries over the range $[i, j] \times [1, n]$ for all $1 \leq i \leq j \leq m$ along with the 1D RMQ structures on the columns. Additionally, they proved that any encoding for answering RMQ requires at least $\Omega(nm \log m)$ bits. When m is 2 or 3, Golin et al. [29] improved the encoding space of the result in [5] (see Table 1) by introducing the joint Cartesian tree of the input. This structure is used to compare two minimum elements in ranges where the row ranges are $[1, m-1]$ and m, respectively. Furthermore, when $m = 2$, they proposed a data structure that answers queries in $O(t)$ time using $5n + O(n \log t/t)$ bits of space, for any $t = (\log n)^{O(1)}$.

Table 1 Encoding space for RMQ queries on $m \times n$ 2D array with $m = 2$ or 3.

Input	Space (in bits)	ref
$2 \times n$	$7n - O(\log n)$	[5]
	$5n - O(\log n)$	[29]
$3 \times n$	$(12 + \log 5)n - O(\log n)$	[5]
	$(6 + \log 5)n + o(n)$	[29]

Also for the 2D array case with $1 \leq i \leq m$ and $1 \leq j \leq n$, the query range can be restricted as follows: (1) 1-sided: $[1, m] \times [1, j]$, (2) 2-sided: $[1, i] \times [1, j]$, (3) 3-sided: $[1, i] \times [j_1, j_2]$ for $1 \leq j_1 \leq j_2 \leq n$, and (4) 4-sided: any rectangular range. Golin et al.[29] provided upper bounds and matching lower bounds for the encoding space required to answer RMQ under these four restricted query cases (see Table 2). Here the encoding space is expressed as an expected value, assuming that the input is arranged in row- or column-major order, uniformly chosen from all permutations of size mn.

Table 2 Expected encoding space (in bits) for RMQ with restricted query ranges [29].

1-sided	2-sided	3-sided	4-sided
$\Theta(\log^2 n)$	$\Theta(\log^2 n \log m)$	$\Theta(n \log^2 m)$	$\Theta(mn)$

For general m and query ranges, Brodal et al. [6] proposed an $O(nm \log m)$-bit encoding for answering RMQ. Based on the their lower bound result of [6], this encoding is asymptotically optimal. For $m = o(n)$, the problem of designing a $o(nm \log n)$-bit data structure for a 2D array that answers queries in sublinear time remains an open problem.

RMQ on (partial) Monge Matrices. A 2D matrix (array) M is called Monge if $M[i_1, j_1] + M[i_2, j_2] \geq M[i_1, j_2] + M[i_2, j_1]$ for any $1 \leq i_1 \leq i_2 \leq m$ and $1 \leq j_1 \leq j_2 \leq n$ (it is more common to define a Monge matrix with $\leq$ rather than $\geq$. In this case, a matrix defined with $\geq$ is referred to as an inverse Monge matrix. All the results presented in this survey can be easily adapted for inverse Monge matrices [38]). Solving RMQ on Monge matrices is of interest due to their various applications in combinatorial optimization and computational geometry [38]. Due to the property of Monge matrices, it is possible to design encoding data structures for RMQ that are more space-efficient than those for general 2D arrays.

Table 3 Data structures on $n \times n$ Monge and partial Monge matrices. Here $\alpha(n)$ denotes the inverse Ackermann function.

Input	Space (in bits)	Query time	ref
Monge	$O(n \log^2 n)$	$O(\log^2 n)$	[37]
	$O(n \log n)$	$O(\log n)$	[22]
	$O(n^{1+\epsilon})$	$O(1)$	
	$O(n \log n)$	$O(\log \log n)$	[23]
Partial Monge	$O(n \log^2 n \cdot \alpha(n))$	$O(\log^2 n)$	[37]
	$O(n \log n)$	$O(\log n \cdot \alpha(n))$	[22]
	$O(n \log n)$	$O(\log \log n)$	[23]

The first non-trivial result was proposed by Kaplan et al. [37] (the journal version of the paper published in 2017 [38]). They showed that for an $n \times n$ Monge matrix, there exists an $O(n \log^2 n)$-bit data structure that can answer RMQ in $O(\log^2 n)$ time.

The result was later improved by Gawrychowski et al. [22] who proposed a data structure using $O(n \log n)$ bits while supporting the query in $O(\log n)$ time. Furthermore, they showed that with $O(n^{1+\epsilon})$ bits of space for any $0 < \epsilon < 1$, the query can be answered in $O(1)$ time. This was further improved in a subsequent work by Gawrychowski et al. [23], where they presented a data structure using $O(n \log n)$ bits of space while supporting $O(\log \log n)$ query time. Additionally, they provided a lower bound of the data structure by showing that any data structure of size $O(n \cdot polylog(n))$ bits requires $\Omega(\log \log n)$ time to answer RMQ on an $n \times n$ Monge matrix. This lower bound was derived by reducing the predecessor problem to RMQ, implying that their data structure is asymptotically optimal (the journal version of the results in [22] and [23] was published in 2020 [24]).

Monge matrices can be generalized to partial Monge matrices, which are Monge matrices with some undefined entries, and the defined entries in each row and column form a contiguous interval. Solving RMQ on partial Monge matrices has applications in areas such as algorithms for maximum flow in planar graphs [38]. In [37], as well as in subsequent works [22] and [23], data structures for partial Monge matrices were proposed by extending those designed for Monge matrices. A summary of these results is presented in Table 3.

3 Range top-k queries

Range selection and range top-k queries are natural extensions of the RMQ, defined as follows: Given a positive integer k and a query range, a range selection query (denoted as sel-k) returns the position of the k-th largest value within the range of the input. Similarly, a range top-k query (denoted as top-k) returns the positions of the k'-largest values within the range for all $k' \leq k$. From these definitions, RMQ can be considered a special case of sel-k or top-k with $k = 1$. There are two variations of top-k: (1) sorted top-k reports the answers in sorted order, based on the corresponding values in the input, and (2) unsorted top-k reports the answers in an arbitrary order. For 1D array, Gawrychowski and Nicholson [25] showed that the space lower bound for answering sorted and unsorted top-k are the same within additive lower order terms when $k = o(n)$. Throughout this survey, we use top-k to refer to the sorted one.

For $k = 2$, Davoodi et al. [10] proposed a $(3.272n + o(n))$-bit data structure that can answer top-2 in $O(1)$ time. Their solution is based on the Cartesian tree of the input, combined with additional information to support the queries. Furthermore, they showed that at least $2.656n - O(\log n)$ bits are required to answer top-2.

Table 4 Data structures for top-k on a 1D array.

Query	Space (in bits)	Query time	ref
	Upper bounds		
top-2	$3.272n + o(n)$	$O(1)$	[10]
	$2.755n + o(n)$		[26]
top-k	$O(n \log k)$	$O(k)$	[31]
	$n \log k + n(k+1)\log(1+1/k) + o(n \log k)$	$O(\log n)$	[26]
	$1.5n \log k - \Theta(n)$	$poly(k \log n)$	[25]
	Lower bounds		
top-2	$2.656n - O(\log n)$		[10]
	$2.755n - o(n)$		[26]
top-k	$n \log k - O(n + k \log k)$		[31]
	$n \log k + n(k+1)\log(1+1/k) - o(n \log k)$		[26]

For general k, the first non-trivial result was introduced by Grossi et al.[31]. This work is an extended journal version of two earlier conference papers published in 2013 [32] and 2014 [44]. They first gave a space lower bound result that at least $n \log k - O(n + k \log k)$ bits are necessary to answer sel-k or top-k, even when the query range is restricted to being 1-sided (i.e., a prefix of the input). Then using the concept of shallow cuttings [9], they design two $O(n \log k)$-bit data structures. These structures support: (1) sel-k in $O(1 + \log k' / \log \log n)$ time, and (2) top-k in $O(k')$ time, for any $1 \leq k' \leq k$. Hence, both data structures use asymptotically optimal space. Moreover, the query time for (1) is also optimal for any data structures using $O(n \cdot polylog(n))$ space, as shown by the lower bound result of Jørgensen and Larsen [36]. However, when the query range is 1-sided, they show that the time lower bound on range selection queries can be circumvented. Specifically, for 1-sided sel-k, they proposed two data structures that (1) uses $n \log k + o(n \log k) + n$ bits and supports queries in any $\omega(1)$ time, or (2) uses $(1+\epsilon)n \log k$ bits and supports queries in $O(1/\epsilon)$ time, for any constant $0 < \epsilon < 1$.

The space upper and lower bounds for answering top-k from [31] were later improved by Gawrychowski and Nicholson [23]. They showed that at least $n \log k + n(k+1)\log(1+1/k)$ bits are necessary to answer top-k and proposed an encoding scheme whose space usage is optimal up to lower-order additive terms. For instance, when $k = 2$, their encoding uses $2.755n + o(n)$ bits of space, improving upon the result of Davoodi et al.[10]. In the extended version of their paper [25], they also presented a $(1.5n \log k - \Theta(n))$-bit data structure that can answer top-k in $poly(k \log n)$ time. See Table 4 for a summary of the results on data structures for top-k in a 1D array.

The approximated selection query is to find the position of an element whose rank lies between $k - \alpha s$ and $k + \alpha s$ for a constant $0 < \alpha < 1/2$, where s denotes the length of the query range. In the special case where $k = 1/2$, the query is referred to as an approximate range median query. Bose et al. [4] introduced a data structure that uses $O(n \log n/\alpha)$ bits of space, which can answer approximate range median queries in $O(1)$ time. For general k, El-Zein et al. [13] proposed a data structure with size $O(n/\alpha^3)$ bits that also supports $O(1)$ query time. When k is fixed, they showed that the size of the data structure can be reduced to $O(n/\alpha^2)$ bits while maintaining the same query time. Therefore, the result improves the space usage of [4] for approximated range median queries. Additionally, they showed that both data structures use asymptotically optimal space for constant α by proving an $\Omega(n)$-bit encoding lower bound for approximate range median queries.

Jo et al. [34] studied the top-k on an $m \times n$ 2D array with $m \leq n$. For queries restricted to the range $[1, m] \times [1, j]$, they proposed an $O(\min nk \lg m, nm \lg k)$-bit encoding for sorted top-k and an $O(\min nk \lg(m/k), nm \lg(k/m))$-bit encoding for unsorted top-k. This result implies a space gap between the encodings for sorted and unsorted top-k in 2D, unlike the 1D case, even when $k = o(mn)$. For arbitrary rectangular query ranges, they presented an $(m \log \binom{(k+1)n}{n} + 2nm(m-1) + o(n))$-bit encoding to answer top-k. Compared to the $O(nm \log n)$-bit trivial encoding, which explicitly stores the input, their encoding uses less space when $m = o(\log n)$.

4 Range mode

A *mode* of a multiset S is an element of S that occurs at least as frequently as any other element in S. In the range mode problem, we are given an array A of n elements which we can preprocess so as to answer *range mode queries* efficiently. Given a query range (i, j), the range mode query returns any position, between i and j, of a mode of the multiset $\{A[i], A[i+1], \cdots, A[j]\}$. Krizanc et al. [40] were the first to consider the data structure version of the range mode problem. They proposed two structures achieving different time-space tradeoffs: (i) a data structure that takes $O(n^{2-2\epsilon})$ words and supports queries in $O(n^{\epsilon} \log n)$ time, for any $0 < \epsilon \leq 1/2$, and (ii) a data structures that takes $O(n^2 \log\log n / \log n)$ words and supports range mode queries in $O(1)$ time. The space bound of the second structure was improved to $O(n^2/\log n)$ words by Petersen [47]. Subsequently, Petersen and Grabowski [48] improved both the tradeoffs to shave-off a log factor, to obtain the following results: (i) an $O(n^{2-2\epsilon})$ space structure that supports queries in $O(n^{\epsilon})$ time, for any $0 < \epsilon < 1/2$, and (ii) an $O(n^2 \log\log n / \log^2 n)$ space structure that supports the queries in $O(1)$ time.

Greve et al. [30] showed that any data structure that uses S memory cells of w bits needs $\Omega(\frac{\log n}{\log(Sw/n)})$ time to answer range mode queries. Chan et al. [7] designed a data structure that uses $O(n)$ words of space and answers range mode queries in $O(\sqrt{n/\log n})$ time. Also, by reducing the Boolean matrix multiplication problem to the range mode problem, they showed that any data structure for range mode must have either $\Omega(n^{\omega/2})$ preprocessing time or $\Omega(n^{\omega/2-1})$ query time in the worst case, where ω denotes the matrix multiplication exponent.

As all the above data structures use at least linear space, they can store the input as part of the data structure. One can improve the space usage significantly by considering approximate versions of the query as described in the next subsection.

4.1 Approximate range mode

In the approximate range mode problem, given a query range (i, j) and a parameter $c \geq 1$, we are interested in returning a position k such that the element $A[k]$ occurs at least $1/c$ times the number of occurrences of the mode of the query range.

Bose et al. [4] were the first to consider this problem whose proposed data structures achieve constant query time for $c = 2, 3$ and 4, using storage space of $O(n \log n)$, $O(n \log\log n)$ and $O(n)$ words, respectively. They also give another data structure that takes $O(n/\epsilon)$ words and answers $(1+\epsilon)$-approximate range mode queries in $O(\log\log_{1+\epsilon} n)$ time. This gives a linear space data structure that answers c-approximate range mode queries in $O(\log\log n)$ time, for constant c. Greve et al. [30] propose an improved data structure that uses linear space and answers 3-approximate range mode queries in $O(1)$ time. Using this data structure,

they design another data structure that takes $O(n/\epsilon)$ words and answers $(1+\epsilon)$-approximate range mode queries in $O(\log(1/\epsilon))$ time. This gives a linear space data structure that answers c-approximate range mode queries in $O(1)$ time, for constant c.

Finally, El-Zein et al. [13] designed an encoding data structure for approximate range mode queries that occupies $O(n/\epsilon)$ bits of space and answers $(1+\epsilon)$-approximate range mode queries in $O(\log(1/\epsilon))$ time. This improves the space usage of Greve et al. [30] by a factor of $\log n$ while maintaining the query time. They also show that the space usage of their structure is asymptotically optimal for constant ϵ by proving a matching lower bound.

5 Range majority and minority

Range majority and range minority queries are fundamental problems in data mining and theoretical computer science. They involve preprocessing a sequence such that, given a range (i, j), one can efficiently determine elements that occur frequently (majority) or infrequently (minority) within the range. These problems are closely related to range mode queries, which aim to find the most frequent element but are computationally harder.

Range majority. Range majority problems are mainly studied under the assumption that one can access either the original or a compressed version of the input array. For the case when τ is fixed at preprocessing time, Karpinski and Nekrich [39] gave a data structure that takes $O(n/\tau)$ words and supports τ-majority queries in $O((\log\log n)^2/\tau)$ time. Durocher et al. [12] independently considered the same problem and obtained an improved result which takes $O(n\log(1/\tau))$ words and supports queries in optimal $O(1/\tau)$ time. For the case when τ is not fixed at preprocessing time, Chan et al. [8] gave a structure that uses $O(n\log n)$ words and supports queries in optimal $O(1/\tau)$ time. Gagie et al. [19] gave another structure for this case that takes $O(n(H_0+1))$ words while supporting queries in optimal time (here H_k denotes the k-th order empirical entropy of the input). Belazzougui et al. [3] designed two improved structures: one that takes $nH_0 + o(n)(H_0+1)$ bits and supports queries in $(1/\tau)\cdot\omega(1)$ time, for any slowly growing function, and another structure that takes $(1+\epsilon)nH_0 + o(n)$ bits, for any constant $\epsilon > 0$ and supports queries in $O(1/\tau)$ time. For the case when the alphabet size σ satisfies $\log\sigma = O(\log w)$, they also gave another structure that uses $nH_0 + o(n)$ bits and supports queries in $O(1/\tau)$ time.

For encoding data structures, Navarro and Thankachan [46] were the first to consider the encoding version of the τ-majority problem They obtained an encoding for range τ-majority queries that takes $O(n\lceil\log(1/\tau)\rceil$ bits and supports range τ'-majority queries, for any $\tau < \tau' < 1$, in time $O((1/\tau)\log\log_w(1/\tau)\log n)$, where $w = \Omega(\log n)$ is the word size. Moreover, they showed that the space usage is optimal by showing that any encoding for range τ-majority queries must use $\Omega(n\lceil\log(1/\tau)\rceil)$ bits. They also propose another structure that takes $O(n\lceil\log(1/\tau)\rceil + n\log\log n)$ bits and answers range τ'-majority queries in $O((1/\tau)\log\log_w(1/\tau))$ time. Finally, Gawrychowski and Nicholson [27] improved the query time of the first structure above of Navarro and Thankachan to obtain a structure that uses $O(n\log(1/\tau))$ bits and supports queries in $O(1/\tau)$ time. Moreover they showed that the space bound is optimal even for a weaker query in which one must decide whether the query range contains at least one τ-majority element. Gawrychowski and Nicholson [28] also showed that for an array of size $2n\log^c n$, for a large constant c, any data structure for checking an existence of element with $1/\log^c n$-majority either needs $\Omega(n^2)$ space or $\Omega(\log^{c-1} n)$ query time, through a reduction from the set intersection problem. This implies that it is unlikely that one can improve the query time to the output sensitive bound of $O(occ+1)$ when returning $occ = o(1/\tau)$ positions for range τ-majority queries.

Range minority. Parameterized range minority problem was introduced by Chan et al. [8]. In this problem, we need to preprocess a given array such that given a parameter τ and a range (i, j), we need to return an element within the range that is not one of its τ-majorities, if there exists one. Currently, there are no encoding results for range minority queries. Also encoding data structure for minority queries are likely harder than the encoding data structures for majority queries. This is because at most $1/\tau$ elements can be candidates for τ-majority, whereas such a lower bound doesn't exist for minority queries. Here, we mention some indexing data structures for range τ-minority queries.

Chan et al. gave a structure that takes $O(n)$ words and supports queries in $O(1/\tau)$ time. By exploiting the duality of this problem with range τ-majorities problem, Belazzougui et al. [3] obtain exactly the same tradeoffs they obtained for the τ-majority problem, mentioned above. Also, analogous to their range majority structure, Gagie et al. [20] propose a data structure that takes $nH_k + 2n + o(n \log \sigma)$ bits for any $k = o(\log_\sigma n)$, and answers range τ-minority queries in $O((\log \log_w \sigma)/\tau)$ time, where $w = \Omega(\log n)$ is the word size.

References

1 Amihood Amir, Johannes Fischer, and Moshe Lewenstein. Two-dimensional range minimum queries. In *CPM*, volume 4580 of *LNCS*, pages 286–294. Springer, 2007. doi:10.1007/978-3-540-73437-6_29.

2 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012. doi:10.1016/J.TCS.2012.08.010.

3 Djamal Belazzougui, Travis Gagie, J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Range majorities and minorities in arrays. *Algorithmica*, 83(6):1707–1733, 2021. doi:10.1007/S00453-021-00799-7.

4 Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In *STACS*, volume 3404 of *LNCS*, pages 377–388. Springer, 2005. doi:10.1007/978-3-540-31856-9_31.

5 Gerth Stølting Brodal, Pooya Davoodi, Moshe Lewenstein, Rajeev Raman, and Srinivasa Rao Satti. Two dimensional range minimum queries and fibonacci lattices. *Theor. Comput. Sci.*, 638:33–43, 2016. doi:10.1016/J.TCS.2016.02.016.

6 Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012. doi:10.1007/S00453-011-9499-0.

7 Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory Comput. Syst.*, 55(4):719–741, 2014. doi:10.1007/S00224-013-9455-2.

8 Timothy M. Chan, Stephane Durocher, Matthew Skala, and Bryan T. Wilkinson. Linear-space data structures for range minority query in arrays. *Algorithmica*, 72(4):901–913, 2015. doi:10.1007/S00453-014-9881-9.

9 Timothy M. Chan and Bryan T. Wilkinson. Adaptive and approximate orthogonal range counting. *ACM Trans. Algorithms*, 12(4):45:1–45:15, 2016. doi:10.1145/2830567.

10 Pooya Davoodi, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Encoding range minima and range top-2 queries. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2016):20130131, 2014. doi:10.1098/rsta.2013.0131.

11 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. In *Automata, Languages and Programming, 36th International Colloquium, ICALP Proceedings, Part I*, volume 5555 of *LNCS*, pages 341–353. Springer, 2009. doi:10.1007/978-3-642-02927-1_29.

12 Stephane Durocher, Meng He, J. Ian Munro, Patrick K. Nicholson, and Matthew Skala. Range majority in constant time and linear space. *Inf. Comput.*, 222:169–179, 2013. doi:10.1016/J.IC.2012.10.011.

13 Hicham El-Zein, Meng He, J. Ian Munro, Yakov Nekrich, and Bryce Sandlund. On approximate range mode and range selection. In *ISAAC*, volume 149 of *LIPIcs*, pages 57:1–57:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ISAAC.2019.57.

14 Johannes Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011. doi:10.1016/J.TCS.2011.01.036.

15 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.

16 Johannes Fischer and Volker Heun. Finding range minima in the middle: Approximations and applications. *Math. Comput. Sci.*, 3(1):17–30, 2010. doi:10.1007/S11786-009-0007-8.

17 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

18 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. An(other) entropy-bounded compressed suffix tree. In *CPM*, volume 5029 of *LNCS*, pages 152–165. Springer, 2008. doi:10.1007/978-3-540-69068-9_16.

19 Travis Gagie, Meng He, J. Ian Munro, and Patrick K. Nicholson. Finding frequent elements in compressed 2d arrays and strings. In *SPIRE*, volume 7024 of *LNCS*, pages 295–300. Springer, 2011. doi:10.1007/978-3-642-24583-1_29.

20 Travis Gagie, Meng He, and Gonzalo Navarro. Compressed dynamic range majority and minority data structures. *Algorithmica*, 82(7):2063–2086, 2020. doi:10.1007/S00453-020-00687-6.

21 Pawel Gawrychowski, Seungbum Jo, Shay Mozes, and Oren Weimann. Compressed range minimum queries. *Theor. Comput. Sci.*, 812:39–48, 2020. doi:10.1016/J.TCS.2019.07.002.

22 Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Improved submatrix maximum queries in monge matrices. In *ICALP (1)*, volume 8572 of *LNCS*, pages 525–537. Springer, 2014. doi:10.1007/978-3-662-43948-7_44.

23 Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Submatrix maximum queries in monge matrices are equivalent to predecessor search. In *Automata, Languages, and Programming – 42nd International Colloquium, ICALP Proceedings, Part I*, volume 9134 of *LNCS*, pages 580–592. Springer, 2015. doi:10.1007/978-3-662-47672-7_47.

24 Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Submatrix maximum queries in monge and partial monge matrices are equivalent to predecessor search. *ACM Trans. Algorithms*, 16(2):16:1–16:24, 2020. doi:10.1145/3381416.

25 Pawel Gawrychowski and Patrick K. Nicholson. Optimal encodings for range min-max and top-k. *CoRR*, abs/1411.6581, 2014. arXiv:1411.6581.

26 Pawel Gawrychowski and Patrick K. Nicholson. Optimal encodings for range top-k , selection, and min-max. In *ICALP (1)*, volume 9134 of *LNCS*, pages 593–604. Springer, 2015. doi:10.1007/978-3-662-47672-7_48.

27 Pawel Gawrychowski and Patrick K. Nicholson. Optimal query time for encoding range majority. In *Algorithms and Data Structures – 15th International Symposium, WADS Proceedings*, volume 10389 of *LNCS*, pages 409–420. Springer, 2017. doi:10.1007/978-3-319-62127-2_35.

28 Pawel Gawrychowski and Patrick K. Nicholson. Optimal query time for encoding range majority. *CoRR*, abs/1704.06149, 2017. arXiv:1704.06149.

29 Mordecai J. Golin, John Iacono, Danny Krizanc, Rajeev Raman, Srinivasa Rao Satti, and Sunil M. Shende. Encoding 2d range maximum queries. *Theor. Comput. Sci.*, 609:316–327, 2016. doi:10.1016/J.TCS.2015.10.012.

30 Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. In *ICALP (1)*, volume 6198 of *LNCS*, pages 605–616. Springer, 2010. doi:10.1007/978-3-642-14165-2_51.

31 Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Asymptotically optimal encodings of range data structures for selection and top-k queries. *ACM Trans. Algorithms*, 13(2):28:1–28:31, 2017. doi:10.1145/3012939.

32 Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti. Encodings for range selection and top-k queries. In *Algorithms – ESA 2013 – 21st Annual European Symposium, Proceedings*, volume 8125 of *LNCS*, pages 553–564. Springer, 2013. doi:10.1007/978-3-642-40450-4_47.

33 Seungbum Jo and Geunho Kim. Space-efficient data structure for next/previous larger-/smaller value queries. In *LATIN 2022: Theoretical Informatics – 15th Latin American Symposium, Proceedings*, volume 13568 of *LNCS*, pages 71–87. Springer, 2022. doi:10.1007/978-3-031-20624-5_5.

34 Seungbum Jo, Rahul Lingala, and Srinivasa Rao Satti. Encoding two-dimensional range top-k queries. *Algorithmica*, 83(11):3379–3402, 2021. doi:10.1007/S00453-021-00856-1.

35 Seungbum Jo and Srinivasa Rao Satti. Simultaneous encodings for range and next/previous larger/smaller value queries. *Theor. Comput. Sci.*, 654:80–91, 2016. doi:10.1016/J.TCS.2016.01.043.

36 Allan Grønlund Jørgensen and Kasper Green Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 805–813. SIAM, 2011. doi:10.1137/1.9781611973082.63.

37 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in monge matrices and monge partial matrices, and their applications. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 338–355. SIAM, 2012. doi:10.1137/1.9781611973099.31.

38 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in monge matrices and partial monge matrices, and their applications. *ACM Trans. Algorithms*, 13(2):26:1–26:42, 2017. doi:10.1145/3039873.

39 Marek Karpinski and Yakov Nekrich. Searching for frequent colors in rectangles. In *CCCG*, 2008.

40 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

41 Mingmou Liu. Nearly tight lower bounds for succinct range minimum query. *CoRR*, abs/2111.02318, 2021. arXiv:2111.02318.

42 Mingmou Liu and Huacheng Yu. Lower bound for succinct range minimum query. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 1402–1415. ACM, 2020. doi:10.1145/3357713.3384260.

43 J. Ian Munro, Patrick K. Nicholson, Louisa Seelbach Benkner, and Sebastian Wild. Hypersuccinct trees – New universal tree source codes for optimal compressed tree data structures and range minima. In *ESA*, volume 204 of *LIPIcs*, pages 70:1–70:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ESA.2021.70.

44 Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti. Asymptotically optimal encodings for range selection. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS*, volume 29 of *LIPIcs*, pages 291–301. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPICS.FSTTCS.2014.291.

45 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. doi:10.1145/2601073.

46 Gonzalo Navarro and Sharma V. Thankachan. Optimal encodings for range majority queries. *Algorithmica*, 74(3):1082–1098, 2016. doi:10.1007/S00453-015-9987-8.

47 Holger Petersen. Improved bounds for range mode and range median queries. In *SOFSEM 2008: 34th Conference on Current Trends in Theory and Practice of Computer Science, Proceedings*, volume 4910 of *LNCS*, pages 418–423. Springer, 2008. doi:10.1007/978-3-540-77566-9_36.

48 Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2009. doi:10.1016/J.IPL.2008.10.007.

49 Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Trans. Algorithms*, 7(4):53:1–53:34, 2011. doi:10.1145/2000807.2000821.

50 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

51 Matthew Skala. Array range queries. In *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *LNCS*, pages 333–350. Springer, 2013. doi:10.1007/978-3-642-40273-9_21.

52 Dekel Tsur. The effective entropy of next/previous larger/smaller value queries. *Inf. Process. Lett.*, 145:39–43, 2019. doi:10.1016/J.IPL.2019.01.011.

53 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980. doi:10.1145/358841.358852.

54 Hao Yuan and Mikhail J. Atallah. Data structures for range minimum queries in multidimensional arrays. In *SODA*, pages 150–160. SIAM, 2010. doi:10.1137/1.9781611973075.14.

Secure Compressed Suffix Arrays

Kunihiko Sadakane
The University of Tokyo, Japan

Abstract

This paper proposes a *secure compressed suffix array*, which is a data oblivious and compressed version of the suffix array used for finding substrings of a large string. Secure compressed suffix arrays can be used for indexing a large collection of strings containing personal information such as DNA data.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases suffix array, compression, encryption, oblivious algorithm, secure computation

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.13

Category Research

1 Introduction

As the amount of data increases, the following points are becoming more problematic.

- Processing time for analyzing the data increases. We need efficient algorithms and data structures.
- We need to care about the use of sensitive data such as personal information.

For the former problem, many researches on algorithms for compressed data have been conducted. Seminal results on this topic are succinct bit-vectors [6, 11], succinct ordered trees [9], and compressed suffix arrays [4, 2]. For the latter problem, *data anonymization* [16] and *secure computation* [17] have been proposed. Because data anonymization modifies input data, some information will be lost. We focus on secure computation, which is a technique to process encrypted data without decryption.

There are two main schemes for secure computation: secret sharing [13] and fully homomorphic encryption [3].

Assume that there are a client and a server. There are two main scenarios.

1. A client has private data and wants to use a cloud service to process the data. Data are stored in the cloud server in an encrypted form. The client runs a program on itself. When some data are necessary, the client asks the server to obtain the data. Then the client decrypts the data, does some computation, encrypts the data and sends back to the server. In this scenario, the task of the server is to store the data and give accesses to a part of the data to the client. The data must be stored as an encrypted form on the server, but computation on encrypted data is not necessary. Therefore it is enough to hide the access pattern to the data on the server.
2. The server stores encrypted data and the secret key is not known to the server. The client asks the server to run a program. The server does some computation and returns the answer to the client. The client decrypts the answer using the secret key. In this scenario, algorithms executed on the server must be data oblivious and the computation must be done on encrypted data.

The second scenario is preferable because the client does not require computation power for analyzing big data. However, we need to design special algorithms for the server which are data oblivious and which run on encrypted data. We call such algorithms *secure algorithms*.

This paper proposes *secure compressed suffix array*, which is a data oblivious and encrypted version of the compressed suffix array [4].

From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday.
Editors: Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter; Article No. 13; pp. 13:1–13:8

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Preliminaries

2.1 Suffix arrays

Let T be a string of length n on alphabet $\mathcal{A}$ of size σ. The j-th character of T is denoted by $T[j]$ $(j = 0, 1, \ldots, n-1)$ and it belongs to $\mathcal{A}$. We assume that a unique terminator \$, which is smaller than any character in $\mathcal{A}$, is appended to T. That is, $T[n] = \$$. A substring $T[i]T[i+1]\cdots T[j]$ of T is denoted by $T[i, j]$. The substring $T[j, n]$ is called the j-th suffix of T and denoted by T_j.

The suffix array [8] of the string T is an integer array $\mathrm{SA}[0, n]$ defined as $\mathrm{SA}[i] = j$ $(i = 0, 1, \ldots, n)$ if T_j is the lexicographically the i-th suffix of T. It always holds that $\mathrm{SA}[0] = n$, which corresponds to the shortest suffix consisting of only the terminator.

If we have T and SA, we can support the following queries:

- $\mathrm{Count}(P, T)$: returns the number of occurrences of P in T in $\mathrm{O}(|P| \log n)$ time
- $\mathrm{Locate}(P, T)$: returns the positions of occurrences of P in T in $\mathrm{O}(|P| \log n + occ)$ time where $occ = \mathrm{Count}(P, T)$

To support these queries, we use the following query:

- $\mathrm{Range}(P, T)$: returns the maximal range $[s, e] \subset [0, n]$ so that for any $i \in [s, e]$ the substring $T[SA[i], SA[i] + |P| - 1]$ is equal to P.

Let $[s, e] = \mathrm{Range}(P, T)$. Then it holds $\mathrm{Count}(P, T) = e - s + 1$ and $\mathrm{Locate}(P, T) = \{\mathrm{SA}[s], \mathrm{SA}[s+1], \ldots, \mathrm{SA}[e]\}$.

The size of the suffix array SA is $n \log n$ bits. We also need to store the string T itself, which occupies $n \log \sigma$ bits. The suffix array requires a huge space compared with the string itself, especially for strings on small alphabets, such as DNA strings. For human DNA, $\sigma = 4$, whereas $n > 2^{31}$. Then $\log n$ is more than 15 times larger than $\log \sigma$.

2.2 Succinct bit-vectors

Succinct data structures are data structures storing objects in minimum number of bits. More precisely, consider storing an element x of a set S. Then the *information-theoretic lower bound* of the number of bits to represent x is defined as $\lceil \log_2 x \rceil$. Hereafter we omit the base 2 of logarithm. Let Z denote this value. Then a data structure for storing x is called *succinct* if it uses $Z + \mathrm{o}(Z)$ bits, and *compact* if it uses $\mathrm{O}(Z)$ bits.

The most fundamental succinct data structure is a bit-vector supporting rank and select queries. A bit-vector is a string $B[1, n]$ on the binary alphabet $\{0, 1\}$ and rank and select are defined as follows.

- $\mathrm{rank}_c(B, i)$ returns the number of c's in $B[0, i]$. We define $\mathrm{rank}_c(B, 0) = 0$ and $\mathrm{rank}_c(B, i) = \mathrm{rank}_c(B, n)$ if $i > n$.
- $\mathrm{select}_c(B, j)$ returns the position of the j-th c in B. We define $\mathrm{select}_c(B, 0) = 0$ and $\mathrm{select}_c(B, j) = n + 1$ if $j > \mathrm{rank}_c(B, n)$.

For a bit-vector of length n, rankand selectare computed in constant time using the bit-vector itself and an $\mathrm{O}(n \log \log n / \log n)$ bit auxiliary data structure [11].

We briefly review the rank data structure. The bit-vector B is partitioned into super-blocks of length L_1 each, and each super-block is further partitioned into blocks of length L_2 each. We store rank values for all super-block boundaries in array R_1 using $\mathrm{O}((n \log n)/L_1)$ bits, and rank values for all block boundaries in array R_2 using $\mathrm{O}((n \log L_1)/L_2)$ bits. Inside a block, we count the number of c's using table lookups for every $1/2 \log n$ bits. The size of the table is $\mathrm{O}(\sqrt{n} \log n \log \log n) = \mathrm{o}(n)$ and the time complexity is $\mathrm{O}(L_2 / \log n)$. If we set $L_1 = \Theta(\log^2 n)$ and $L_2 = \frac{1}{2} \log n$, we obtain the desired bound.

Table 1 Oblivious RAM data structures for storing n bits. Bits are grouped into blocks of $b \geq \log n$ bits each. Bandwidh blowup is the number of blocks to be accessed to obtain one block obliviously.

Server space (#bits)	Bandwidth blowup	User space (#blocks)	Reference
$O(n)$	$O(\log^2 n / \log\log n)$	$O(1)$	Kushilevitz et al. [7]
$O(n)$	$O(\log^2 n)$	$\omega(\log n)$	Stefanov et al. [15]
$n(1+\Theta(1/\log n))$	$O(\log^2 n)$	$O(\log n \log\log n)$	Onodera, Shibuya [10]

2.3 Secure algorithms

If we forget about time efficiency, any computation can be done on encrypted data if we can support additions and multiplications on two encrypted integers. There exist two such schemes.

- Secret Sharing [13]. Data are distributed into two or more servers and for each server the stored data look like random values and no information is leaked. Additions can be done locally in each server, whereas for multiplications the servers must comminicate each other.
- Fully Homomorphic Encryption (FHE) [3, 1]. Any number of additions and multiplications on encrypted numbers can be done.

Both schemes have a drawback that the computation is much slower than plain (unencrypted) data. In Secret Sharing schemes the servers communicate each other for computing multiplications. This takes much more time than the computation on plain data in a single server. In FHE schemes, there are no communication because there is only one server. However multiplications are extremely slow. Therefore it is important to develop efficient secure algorithms.

In this paper, we assume that in both schemes, integer addition, multiplication, division, less-than comparison are done efficiently in unit time. Then our algorithm runs in both schemes.

2.4 Oblivious RAM

Oblivious RAMs [7, 15, 10], or ORAMs for short, are data structures supporting oblivious read and write to an array. Without loss of generality we can assume the array stores a binary string S of length n. S can be regarded as an array of length n/w storing w-bit integers. An ORAM has a parameter b called the block size. S is partitioned into blocks of length b each, and a block is accessed as a unit. To achieve oblivious access, more than one blocks are accessed to obtain one block. The ratio is called bandwidth blowup.

Obliviousness is defined as follows. Let $\vec{y} = ((\mathrm{op}_1, a_1, d_1), (\mathrm{op}_1, a_1, d_1), \ldots, (\mathrm{op}_M, a_M, d_M))$ be a sequence of accesses to an ORAM where op_i is either read or write, a_i is the address of the i-th access, and d_i is the content to be written in the a_i-th block when op_i is write. Let $A(\vec{y})$ be denote the sequence of accesses to the server given $\vec{y}$. Then the ORAM is said to be oblivious if two any access sequences $\vec{y}$ and $\vec{z}$ of the same length, $A(\vec{y})$ and $A(\vec{z})$ are computationally indistinguishable by anyone but the client, and if the failure probability is negligible [15].

Table 1 shows existing oblivious RAM data structures. Onodera and Shibuya [10] give a succinct oblivious RAM, that is, the server space is $n + o(n)$ bits. The other two data structures are compact, that is, the server space is $O(n)$ bits.

3 Compressed Suffix Arrays

3.1 The original structure

Grossi and Vitter [5, 4] have proposed *compressed suffix arrays*, which are a compressed version of suffix arrays. Let SA be the suffix array of string T of length n on alphabet $\mathcal{A}$ of size σ. The compressed suffix array is a data structure which efficiently supports the following operation:

- Lookup(i): returns SA[i].
- Inverse(j): returns i such that $j = \mathrm{SA}[i]$ (the inverse suffix array).

The core component of the compressed suffix arrays is the Ψ function, defined as follows.

$$\Psi[i] = \mathrm{SA}^{-1}[SA[i] + 1]$$

if $i > 0$, and $\Psi[0] = \mathrm{SA}^{-1}[1]$. The Ψ function has a good property that it is piecewise monotone. Precisely, let $[s_c, e_c]$ be the range of the suffix array such that $T[SA[i]] = c$ for any $i \in [s_c, e_c]$ where $c \in \mathcal{A}$. Then if $i, j \in [s_c, e_c]$ and $i < j$, it holds $\Psi[i] < \Psi[j]$. Then the following function is strictly increasing.

$$\Psi'[i] = T[\mathrm{SA}[i]] \cdot (n+1) + \Psi[i]$$

We can compress Ψ' in $n(2 + \log\sigma) + \mathrm{o}(n)$ bits so that any $\Psi'[i]$ is computed in constant time. From $\Psi'[i]$, we can obtain $T[\mathrm{SA}[i]]$ and $\Psi[i]$ easily in constant time.

The encoding of Ψ' is as follows. each entry of Ψ' is a $(\log\sigma + \log n)$-bit integer. We partition it into higher part and lower part. The higher part has $\log n$ bits and the lower part has the rest ($\log\sigma$ bits). The lower parts for all entries are stored in an integer array $L[0, n]$ in $n \log\sigma$ bits. The upper parts for all entries are represented by a bit-vector $H[0, 2n]$ as follows. Let $d_i = (\Psi'[i] \div \sigma) - (\Psi'[i-1] \div \sigma)$ for $i = 0, 1, \ldots, n$ (we assume $\Psi'[-1] = 0$). Because Ψ' is increasing, $d_i \geq 0$ for any i. We encode d_i's by unary codes. That is, we write d_i many zeros followed by a one, to H. Then, $H[i]$ can be computed in constant time as

$$H[i] = \mathrm{select}_1(H, i) - i$$

From the definition of Ψ, if $\mathrm{SA}[i] = j$, it holds $\mathrm{SA}[\Psi[i]] = j + 1$. That is, if we know the lexicographic order i of a suffix T_j, we can obtain that of the next suffix T_{j+1} by computing $\Psi[i]$. We sample the values of the suffix array for every L_3 entries and stores them in an array A. We also store a bit-vector $F[0, n]$ such that $F[i] = 1$ iff $SA[i]$ is sampled. Then Lookup(i) is computed as follows.

$$\mathrm{Lookup}(i) = \begin{cases} A[\mathrm{rank}_1(F, i)] & \text{if } F[i] = 1 \\ \mathrm{Lookup}(\Psi[i]) - 1 & \text{if } F[i] = 0 \end{cases}$$

The array A uses $\mathrm{O}((n \log n)/L_3)$ bits and F uses $n + \mathrm{o}(n)$ bits. If we set $L_3 = \Theta(\log n)$, the space for A is $\mathrm{O}(n)$ bits and Lookup(i) is computed in $\mathrm{O}(\log n)$ time. See Figure 1 for an example of

3.2 Self-indexes

The original compressed suffix array is a compressed representation of the suffix array and used together with the string itself. We can change it to a self-index, that is, a data structure for string matching which does not use the string. It is enough to add $n + \mathrm{o}(n) + \mathrm{O}(\sigma \log n)$ bits. We use a bit-vector $D[0, n]$ showing that if $D[i] = 1$ then $i = 0$ or $T[SA[i]] \neq T[SA[i-1]]$.

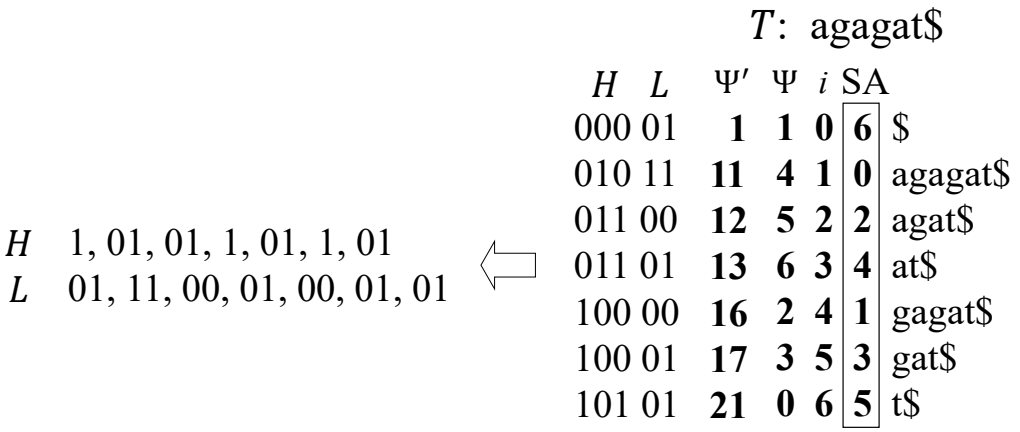


Figure 1 An example of compressed suffix arrays.

That is, $D[i] = 1$ iff lexicographically the i-th suffix has a different head character from the $(i-1)$-st suffix. We use an array C of length σ such that $C[\mathrm{rank}_1(D,i)] = T[SA[i]]$. We define $\mathrm{Head}(i) = C[\mathrm{rank}_1(D,i)]$. The array uses $\sigma \log \sigma = \mathrm{O}(\sigma \log n)$ bits (we assume $\sigma \le n$). We use another array $C^{-1}[1,\sigma]$ storing in $C[c]$ the range $[s_c, e_c] \subset [0,n]$ such that for any $i \in [s_c, e_c]$ it holds $T[SA[i]] = c$. This array uses $\mathrm{O}(\sigma \log n)$ bits.

Using C, we can recover a substring of T. Assume the lexicographic order i of a suffix T_j is known ($\mathrm{SA}[i] = j$). Then the character $T[j+k]$ is equal to $\mathrm{Head}(\Psi^k[i])$. The substring $T[j, j+\ell-1]$ is computed in $\mathrm{O}(\ell)$ time by iteratively computing $i := \Psi[i]$. Computing the lexicographic order i of T_j is equivalent to computing $\mathrm{Inverse}(j)$, and it is done similarly to computing an entry of the suffix array using Ψ and the sampled suffix array [12].

To support $\mathrm{Range}(P,T)$, we use what we call *backward search*. Let m be the length of P. First we obtain the range for the shortest suffix $P[m,m]$ by $[s,e] := C^{-1}[P[m]]$. Then, given the range $[s,e]$ for a suffix $P[j+1,m]$, we compute the range $[s',e']$ for the suffix $P[j,m]$. This is done as follows.

$$\begin{aligned} s' &:= \mathrm{argmin}_{i \in C^{-1}[P[j]]} \{\Psi[i] \ge s\} \\ e' &:= \mathrm{argmax}_{i \in C^{-1}[P[j]]} \{\Psi[i] \le e\} \end{aligned}$$

By a simple binary search, the new range is obtained in $\mathrm{O}(\log n)$ time, and therefore $\mathrm{Range}(P,T)$ is done in $\mathrm{O}(|P| \log n)$ time, which is the same as the suffix array [12].

We can simplify the algorithm as follows.

$$\begin{aligned} s' &:= \mathrm{argmin}_{i \in [0,n]} \{\Psi'[i] \ge P[j] \cdot (n+1) + s\} \\ e' &:= \mathrm{argmax}_{i \in [0,n]} \{\Psi'[i] \le P[j] \cdot (n+1) + e\} \end{aligned}$$

Now we do not need the array C^{-1}. This has another merit that it is easier to make it oblivious, which will be shown in the next section.

4 Secure Compressed Suffix Arrays

We show that the compressed suffix arrays can be modified so that all the operations Ψ, Range, Lookup, and Inverse. First we show that Range, Lookup, and Inverse can be done obliviously if Ψ' is computed obliviously. We assume that the lengths of the string T and the pattern P, and the alphabet size σ are public.

4.1 Computing Range

As shown in Section 3.2, Range(P,T) is done by $|P|$ many binary searches on Ψ'. Given the range $[s,e]$ for $P[j+1,m]$, we compute the range $[s',e']$ for $P[j,m]$. To do so, we first compute $P[j]\cdot(n+1)+s$ and $P[j]\cdot(n+1)+e$, which can be done obliviously. Then using a binary search on Ψ' we compute s'. The initial range is $[0,n]$, and in each step we update the range based on the result of a less-than comparison. Using Ψ' is easier than using Ψ because we do not need the array C^{-1} that must support oblivious accesses. We can compute Range(P,T) using $\mathrm{O}(|P|\log n)$ many oblivious accesses to Ψ'.

4.2 Computing Lookup and Inverse

Lookup(i) can be done using the sampled array A, the bit-vector F, and Ψ. The original algorithm repeats computing $i := \Psi[i]$ until $F[i]=1$. This is not oblivious because the number k of iteration depends on i. Precisely, it holds Lookup$(i) = A[\mathrm{rank}_1(F,\Psi^k[i])]-k$ where $k\geq 0$ is the smallest number such that $F[\Psi^k[i]]=1$.

To change the algorithm oblivious, we fix the number of iterations to L_3. Because the suffix array entries are sampled every L_3 entries in text order, for exactly one entry it holds $F[\Psi^k[i]]=1$ among those for $k=0,1,\ldots,L_3-1$. Therefore we change the algorithm as follows.

1. $x := 0$
2. for $k=0,1,\ldots,L_3-1$
3. $\quad x := x + F[i]\cdot(A[\mathrm{rank}_1(F,i)]-k)$
4. $\quad i := \Psi[i]$
5. return x

Here we need oblivious accesses to F and A, and oblivious computation of $\mathrm{rank}_1(F,i)$.

For A, we use the Path ORAM [15] with block size $b=\log^2 n$. Then the space usage is $\mathrm{O}(n)$ bits and an entry of A is obtained in $\mathrm{O}(\log^2 n)$ many accesses to blocks, which can be done in $\mathrm{O}(\log^3 n)$ time.

For F, we use the succinct ORAM [10] with block size $b=\log^2 n$. Then a block of F is obtained in in $\mathrm{O}(\log^2 n)$ many accesses to blocks, which can be also done in $\mathrm{O}(\log^3 n)$ time. For computing a rank on F, we use the array R_1 defined in Section 2.2. This is stored using the Path ORAM. The space usage is $\mathrm{O}(n/\log n)$ bits. The rank inside a block is computed by logical operations in $\mathrm{O}(\log n)$ time. Therefore a rank value is computed in $\mathrm{O}(\log^3 n)$ time.

The computation of Inverse(j) is similar.

4.3 Computing Ψ

Finally, we show how to compute $\Psi[i]$. As shown in Section 3.1, Ψ' is represented by a bit-vector $H[0,2n]$ and an array $L[0,n]$ of $\log\sigma$-bit integers. The array L is stored using the succinct ORAMs. The bit-vector H is stored similarly to F, but here we also need to store the auxiliary data structure for select_1. This can be stored using the Path ORAM.

To sum up, $\Psi[i]$ is computed in $O(\log^3 n)$ time and the space usage is $n(2+\log\sigma)+o(n)\log\sigma$ bits.

4.4 Summary

$\Psi[i]$ takes $O(\log^3 n)$ time. A step of a backward search is a binary search on Ψ', and therefore it is done in $O(\log^4 n)$ time. Then Range(P,T) takes $O(|P|\log^4 n)$ time. For Lookup(i), we set $L_3 = O(\log n)$. Then it takes $O(\log^4 n)$ time. The space usage is $(n+o(n))\log\sigma + O(n)$ bits in total.

5 Concluding Remarks

We have proposed secure compressed suffix arrays. For a string of length n on an alphabet of size σ, It uses $(n+o(n))\log\sigma + O(n)$ bits of space and the Count(P,T) query is done in $O(|P|\log^4 n)$ time, and the Lookup(i) query is done in $O(\log^4 n)$ time. Therefore there is an $O(\log^3 n)$ multiplicative overhead compared with the original compressed suffix arrays [4]. To improve the running time, we need more efficient succinct ORAM [10] and standard ORAMs [14]. Future work will be developing such oblivious RAM data structures and giving efficient implementions.

References

1 Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.

2 P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

3 Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1536414.1536440.

4 R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

5 Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on the Theory of Computing*, pages 397–406, Portland, OR, 2000.

6 G. Jacobson. Space-efficient Static Trees and Graphs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

7 Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 143–156, 2012. doi:10.1137/1.9781611973099.13.

8 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

9 J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

10 Taku Onodera and Tetsuo Shibuya. Succinct Oblivious RAM. In Rolf Niedermeier and Brigitte Vallée, editors, *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)*, volume 96 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2018.52.

11 R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007. doi:10.1145/1290672.1290680.

12 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003. doi:10.1016/S0196-6774(03)00087-7.

13 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. doi:10.1145/359168.359176.

14 Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25385-0_11.

15 Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2508859.2516660.

16 Latanya Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002. doi:10.1142/S0218488502001648.

17 Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986. doi:10.1109/SFCS.1986.25.

Specific Patterns Against Reference Sequences

Marie-Pierre Béal[1]
Univ. Gustave Eiffel, CNRS, LIGM, Champs-sur-Marne, France

Maxime Crochemore
Univ. Gustave Eiffel, CNRS, LIGM, Champs-sur-Marne, France
King's College London, UK

Abstract

We design alignment-free techniques for comparing a set of sequences or just a word, called a target, against another set of words, called a reference. This is done with the detection of factor patterns that distinguish the target from the reference. A target-specific factor of a target T against a reference R is then a factor w of a word in T that is not a factor of a word in R but whose proper factors of w are factors of a word in R. The strategy is based on the notion of minimal absent/forbidden words.

We first address the computation of the set of target-specific factors of a target T against a reference R, where T and R are finite sets of sequences. The result is the construction of an automaton accepting the set of all considered target-specific factors. The construction algorithm runs in linear time according to the size of $T \cup R$.

The second result is the design of an algorithm to compute all the occurrences in a single sequence T of its target-specific factors against a reference R. The algorithm runs in real-time on the target sequence, independently of the number of occurrences of target-specific factors.

2012 ACM Subject Classification Theory of computation → Regular languages; Theory of computation → Pattern matching

Keywords and phrases Specific pattern, Minimal absent word, Minimal forbidden word, Directed Acyclic Word Graph (DAWG), Suffix automaton

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.14

Category Research

1 Introduction

The goal of this article is to design an alignment-free technique for comparing a set of sequences or words, called a target, against a set of words, called a reference.

The motivation comes from the analysis of genomic sequences as done, for example, by Khorsand et al. in [23] in which the authors introduce the notion of sample-specific strings. To avoid alignments but to extract interesting elements that differentiate the target from the reference, or in general two words, the chosen specific fragments are minimal absent words, also called minimal forbidden words. Target-specific words are factors of the target that are minimal absent words of the reference.

These types of factors associated with absent patterns are rather commonly used to efficiently compare sequences by avoiding complete alignments of full sequences, see, for example, [11] and references therein. In bioinformatics, target-specific words serve as signatures for newly sequenced biological molecules, helping to identify their characteristics. In the domain of molecular biology they allow the discovery of remarkable patterns in some genomic sequences, such as persitent patterns in the analysis of SARS-CoV-2 genomes that are absent in the human genome [31] and minimal sequences in Ebola virus also absent in the human genome [33]), or to build phylogenies of biological molecular sequences using a

[1] Corresponding Author

distance based on absent words (see [10, 32]). These patterns are also helpful to improve both pattern matching methods (see [15]) and text compression with the concept of antidictionaries (see for example [17, 1]). They are also central in some approaches related to the sanitisation of data, that is, the process of hiding confidential information [5, 6], to quote only a few applications.

The notion of a minimal absent word was introduced by Mignosi et al. [27] (see also [4]) in relation to combinatorial aspects of certain sequences. The first linear-time computation of the minimal absent words of the factors of a single sequence is described in [16] (see also [14]). Its time complexity is $O(n)$ on a fixed-size alphabet for a sequence of size n. The algorithm uses the computation of the directed acyclic word graph (DAWG), also called suffix automaton (see [8, 14]), of a single sequence. The same time complexity holds for an integer alphabet of polynomial size (see [19], [20]). This result is obtained using the $O(n)$ time complexity for computing the DAWG of a single sequence, in the case of an integer alphabet of polynomial size. For a general ordered alphabet A, the running time becomes $O(n \log |A|)$.

These algorithms extend to the computation of the minimal absent words of the set of factors of a finite set of sequences of total size n. This is done in [3] in $O(n)$ for a fixed-size alphabet. It becomes $O(n \log |A|)$ for a general ordered alphabet. It is done in $O(n)$ in [29] for an integer alphabet of polynomial size, extending the linear time computation of the DAWG in [19] and [20] to a finite set of sequences. It is also mentioned in [2] that the DAWG of a finite set of sequences can be computed in $O(n)$ time using sparse matrices for an integer alphabet of polynomial size.

Due to the significant role of the notion, the efficient computation of minimal absent words has attracted considerable attention (see, for example, [30] and references therein).

In this article, we continue exploring the approach of target-specific words as in [23] by introducing new algorithmic techniques to detect them. A more general view of the usefulness of formal languages in analyzing a series of genomes using pangenomic graphs is described by Bonizzoni et al. in [9].

A preliminary version of this paper was presented at the DLT 2023 conference [2]. In the present article, the motivation is strengthened due to additional references, the presentation of algorithms has been improved and complete proofs have been added. In addition, Section 5 reveals a surprising phenomenon on the tight link between DAWG matching and the search for minimal absent words, a technique at the core of our solution in Section 4.

The results

We design two algorithms that use intensively the notion of suffix links of indexing data structures, such as suffix trees (see [21, 14]) and DAWGs. The links can also be simulated using suffix arrays [26] and their implementations, such as the FM-index [18]. The algorithm in [23] uses the FMD index of Li [25]. All these data structures can accommodate the sequences and their reverse complements.

First, we address the computation of the set of target-specific factors of a target T against a reference R, where T and R are finite sets of sequences over an alphabet A. The result is the construction of an automaton that accepts the set of all the target-specific factors considered. This automaton is a digital tree whose leaves correspond to the specific words. The construction algorithm runs in linear time according to the size n of $T \cup R$, when A has a fixed size. The time complexity is $O(n \log |A|)$ when A is an ordered alphabet. Further, using the result of [29], the running time becomes $O(n)$ when A is an integer alphabet of

polynomial size according to n. Our algorithm uses a marking technique of the DAWG of a finite set of sequences, very close to the skip links used in [29], to compute minimal absent words using a DAWG.

The second result shows the design of an algorithm to compute all the occurrences in a single sequence T of its target-specific factors against a reference R. The algorithm runs in real-time on the target sequence over a fixed-size alphabet and independently of the number of occurrences of target-specific factors. This is obtained after standard processing of the DAWG of the reference, similarly as above. This improves on the result in [23], where the running time of the main algorithm depends on the number of occurrences of sought factors.

Definitions

Let A be a finite alphabet and A^* be the set of finite words drawn from the alphabet A, including the empty word ε. A *language* is a set of finite words. The concatenation of two words u, v is denoted by uv, and, if $x = uv$, v is also denoted by $u^{-1}x$. A *factor* of a word $x \in A^*$ is a word $v \in A^*$ that satisfies $x = uvw$ for some words $u, w \in A^*$. A *proper factor* of x is a factor distinct from the whole word. If P is a set of words, we denote by $\mathrm{Fact}(P)$ the set of factors of words in P, and, if P is finite, $\mathrm{size}(P)$ denotes the sum of lengths of the words in P. A language L is *factorial* is each factor of a word of L belongs to L.

A *minimal absent word* (also called a minimal forbidden word) for a given factorial language $L \subseteq A^*$ with respect to a given alphabet B containing A is a word of B^* that does not belong to L but whose all proper factors do.

Let R, T be two sets of finite words. A *T-specific word with respect to R* is a word u such that: u is a factor of a word in T, u is not a factor of a word in R and any proper factor of u is a factor of a word in R. The set R is called the *reference* and T the *target* of the problem.

Note that a word is a T-specific word with respect to R if and only if it is a minimal absent word of $\mathrm{Fact}(R)$, with respect to the alphabet of letters occurring in $R \cup T$, and is also in $\mathrm{Fact}(T)$. As a consequence, the set of T-specific words with respect to R is both prefix-free (i.e., no word of the set is a prefix of another word of the set) and suffix-free (i.e., no word of the set is a suffix of another word of the set).

It follows from the definition that the set S of T-specific words with respect to R is:

$$\mathrm{Fact}(T) \cap (A^* - \mathrm{Fact}(R)) \cap A\,\mathrm{Fact}(R) \cap \mathrm{Fact}(R)A,$$

where A is the alphabet of letters of words R and T. It is thus a regular language when R and T are regular, in particular, when R and T are finite.

A *finite deterministic automaton* is denoted by $\mathcal{A} = (Q, A, i, F, \delta)$, where A is a finite alphabet, Q is its finite set of states, $i \in Q$ is the unique initial state, $F \subseteq Q$ is the set of final states and δ is the partial function from $Q \times A$ to Q representing the transitions of the automaton. The partial function δ extends naturally to $Q \times A^*$ and a word u is accepted by $\mathcal{A}$ if and only if $\delta(i, u)$ is defined and belongs to F.

2 Background: directed acyclic word graph

In this section, we recall the definition and sketch the construction of the directed acyclic word graph of a finite set of words. This description already appears in [3].

Let $P = \{x_1, x_2, \ldots, x_r\}$ be a finite set of r finite words. A linear-time construction of a deterministic finite state automaton recognizing $\mathrm{Fact}(P)$ has been obtained by Blumer *et al.* in [8, 7] (see also [28]). Their construction is an extension of the well-known incremental

construction of the suffix automaton of a single word (see for instance [12, 14]). The words are added one by one to the automaton. In the sequel, we call this algorithm the DAWG algorithm since it outputs a deterministic automaton called a *directed acyclic word graph*. Let us denote it by $DAWG(P) = (Q, A, i, Q, \delta)$. Let $Suff(v)$ denote the set of suffixes of a word v and $Suff(P)$ the union of all $Suff(v)$ for $v \in P$.

The states of $DAWG(P)$ are the equivalence classes of the right invariant equivalence $\equiv_{Suff(P)}$ defined as follows. For $u, v \in \mathrm{Fact}(P)$,

$$u \equiv_{Suff(P)} v \text{ iff } \forall i \in [1, r] \quad u^{-1}Suff(x_i) = v^{-1}Suff(x_i).$$

There is a transition labelled by a from the class of a word u to the class of ua. The automaton $DAWG(P)$ has a unique initial state, which is the class of the empty word, and all its states are final. Note that the (syntactic) congruence $\sim$ defining the minimal automaton of the language is

$$u \sim v \text{ iff } \bigcup_{i=1}^{r} u^{-1}Suff(x_i) = \bigcup_{i=1}^{r} v^{-1}Suff(x_i),$$

and is not the same as the below equivalence. In other words, $DAWG(P)$ is not always a minimal automaton.

The construction of $DAWG(P)$ is performed in time $O(\mathrm{size}(P) \times \log |A|)$ in [8]. A time complexity of $O(\mathrm{size}(P))$ can be obtained for an integer alphabet of polynomial size in [19] and [20]. It can also be obtained with an implementation of automata with sparse matrices (see [14, Exercise 1.15]).

An essential element of the efficient construction is the notion of suffix links between states, denoted by s. We first define the function s' from $\mathrm{Fact}(P) \setminus \{\varepsilon\}$ to $\mathrm{Fact}(P)$ as follows: for any $v \in \mathrm{Fact}(P) \setminus \{\varepsilon\}$,

$$s'(v) \text{ is the longest suffix of } v \text{ that satisfies } u \not\equiv_{Suff(P)} v.$$

Then, we define the partial function s from Q to Q as follows. When $p = \delta(i, v)$ for $v \neq \varepsilon$, $s(p)$ is the state $\delta(i, s'(v))$. The function s is not defined on the initial state i.

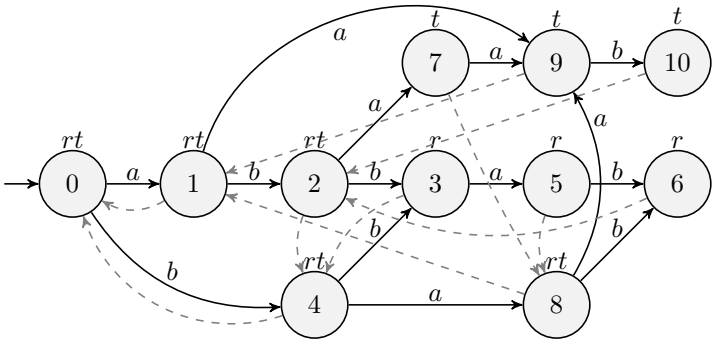


Figure 1 Automaton $DAWG(P)$ for $P = \{abbab, abaab\}$.

▶ **Example 1.** The DAWG obtained with the DAWG algorithm applied to $P = \{abbab, abaab\}$ is displayed in Figure 1. Dashed edges represent the suffix links and marks r, t on states relate to the reference word *abbab* and the target *abaab*. Note that this deterministic automaton is not minimal, as the states 6 and 10, 5 and 9, and 3 and 7, respectively, can be merged pairwise.

3 Computing the set of T-specific words

In this section, we assume that the reference R and the target T are two finite sets of words and our goal is to compute the set of T-specific factors of T against R. To do so, we first compute the directed acyclic word graph $DAWG(R \cup T) = (Q, A, i, Q, \delta)$ of $R \cup T$. Further, we compute a table, called mark, indexed by the set of states Q and that satisfies: for each state p in Q, mark$[p]$ is one of the three values r, t or both r, t according to the fact that each word labeling a path from i to p is a factor of some word in R, in T or in both. This information can be obtained during the construction of the directed acyclic word graph without increasing the running time.

The following algorithm outputs the set of T-specific words occurring in T with respect to R in the form of a trie (digital tree, see [14]).

SPECIFIC-TRIE((Q, A, i, Q, δ) DAWG of $(R \cup T)$, s its suffix link)

```
1  for each p ∈ Q with mark[p] = r, t in width-first search from i do
2      for each a ∈ A do
3          if δ(p, a) = q with mark[q] = r, t not reached yet then
4              δ'(p, a) ← q
5          elseif (δ(p, a) defined with mark[δ(p, a)] = t) and
               (p = i or δ(s(p), a) defined with mark[δ(s(p), a)] = r or r, t) then
6              δ'(p, a) ← new sink
7  return (Q, A, i, {sinks}, δ')
```

▶ **Proposition 2.** *Let $DAWG(R \cup T)$ be the output of Algorithm* DAWG *on the finite set of words $R \cup T$, let s be its suffix function, and let* mark *be the table defined as above. Algorithm* SPECIFIC-TRIE *builds the trie recognizing the set of T-specific words with respect to R.*

Proof. Let S be the set of T-specific words with respect to R.

Consider a word ua ($a \in A$) accepted by the automaton $\mathcal{A} = (Q, A, i, \{\text{sinks}\}, \delta')$ returned by the algorithm. Note that $\mathcal{A}$ accepts only nonempty words. Let $p = \delta'(i, u)$. Since the *DAWG* automaton is processed with a width-first search, u is the shortest word for which $\delta(i, u) = p$. Therefore, if $u = bv$ with $b \in A$, we have $\delta(i, v) = s(p)$ by definition of the suffix function s. When the test "($\delta(p, a)$ defined and mark$[\delta(p, a)] = t$) and ($\delta(s(p), a)$ defined and mark$[\delta(s(p), a)] = r$ or r, t)" is satisfied, this implies that $va \in \text{Fact}(R)$. Thus, $bva \notin \text{Fact}(R)$, while $bv, va \in \text{Fact}(R)$ and $bva \in \text{Fact}(T)$. So, ua is a T-specific word with respect to R. If u is the empty word, then $p = i$. The transition from i to the sink labeled by a is created under the condition "$\delta(p, a)$ defined and mark$[\delta(p, a)] = t$", which means that $a \in \text{Fact}(T)$. The word a is again a T-specific word with respect to R. Thus the words accepted by $\mathcal{A}$ are T-specific words with respect to R.

Conversely, let $ua \in S$. If u is the empty word, this means that a does not occur in $\text{Fact}(R)$ and occurs in $\text{Fact}(T)$ therefore there is a transition labeled by a from i in $DAWG(R \cup T)$ to a state marked t. Thus, a transition from i to a sink state in $\mathcal{A}$ is created in line 6, and a is accepted by $\mathcal{A}$. Now assume that $u = bv$. The word u is in $\text{Fact}(R)$. So let $p = \delta(i, u)$. Note that u is the shortest word for which $p = \delta(i, u)$, because all such words are suffixes of each other in the *DAWG* automaton. The word ua is not in $\text{Fact}(R)$ and is in $\text{Fact}(T)$, so the condition "$\delta(p, a)$ defined and mark$[p, a] = t$" is satisfied. Let $q = s(p)$. We have $q = \delta(i, v)$ because of the minimality of the length of u and the definition of s. Since va is in $\text{Fact}(R)$, the condition "$\delta(s(p), a)$ defined and mark$[\delta(s(p), a)] = r$ or r, t" at line 5 is satisfied. This results in the creation of a transition at line 6, enabling $\mathcal{A}$ to accept ua as desired. ◀

▶ **Example 3.** The automaton $DAWG(R \cup T)$, where $R = \{abbab\}$ and $T = \{abaab\}$ is shown in Figure 1. The output of Algorithm Specific-trie applied to it is shown in Figure 2, where the black squares are the accepting sink states of the trie. The set of T-specific words with respect to R is $\{aa, aba\}$.

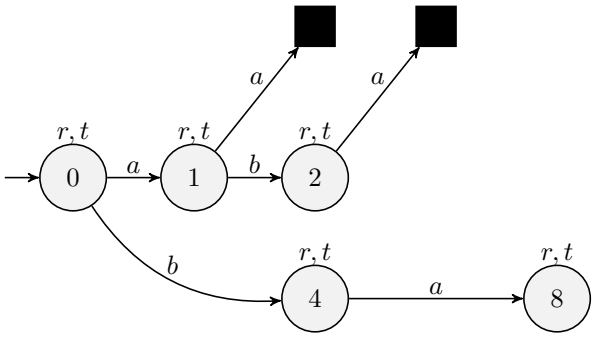


Figure 2 The trie of T-specific words with respect to R.

A main point in algorithm Specific-trie is that it uses the function s defined on states of the input DAWG. It is not possible to proceed similarly when considering the minimal factor automaton of $\mathrm{Fact}(R \cup T)$ because there is no analogue function s. However, it is possible to reduce the automaton $DAWG(R \cup T)$ by merging states having the same future (right context) and the same image by s. For example, on the DAWG of Figure 1, states 6 and 10 can be merged because $s(6) = s(10) = 2$. States 3 and 7, nor states 5 and 9 cannot be merged with the same argument.

▶ **Proposition 4.** *Algorithms* Dawg *and* Specific-trie *together run in time* $O(\mathrm{size}(R \cup T) \times |A|)$ *when applied to reference* R *and target* T*, two finite sets of words, if the transition functions are implemented by transition matrices. This complexity is* $O(\mathrm{size}(R \cup T) \times \log |A|)$ *for an ordered alphabet, and* $O(\mathrm{size}(R \cup T))$ *for an integer alphabet of polynomial size.*

Proof. The running time depends on the time for computing the DAWG of $R \cup T$. The relies on the time for computing the transitions of the DAWG, that is $\delta(q, a)$, which is constant for an integer alphabet of polynomial size, due to [29]. ◀

For P a set of words, we denote by A_P the set of letters occurring in P.

▶ **Proposition 5.** *Let* R, T *be two finite sets of words. The number of* T*-specific words with respect to* R *is no more than* $(2\,\mathrm{size}(R) - 2)(|A_R| - 1) + |A_T \setminus A_R| - |A_R| + m$*, if* $\mathrm{size}(R) > 1$*, where* m *the number of words in* R*. The bound becomes* $|A_T \setminus A_R|$ *when* $\mathrm{size}(R) \leq 1$*.*

Proof. We let S denote the set of T-specific words with respect to R. Since S is included in the set of minimal absent words of $\mathrm{Fact}(R)$ with respect to the alphabet $A = A_R \cup A_T$, the bound comes from [3, Corollary 4.1]. ◀

In conclusion, the algorithm generates a trie of minimal absent words, which represent potentially interesting patterns. This trie can be used to explore the set of patterns that align with the application's objectives.

4 Computing occurrences of target-specific factors: the T-specific table

In this section, we consider that T is a single word and R is a finite set of words as before. The goal of the section is to design an algorithm that computes all the occurrences in T of words that are T-specific with respect to R. To do so, we define the T-specific table associated with the pair R, T of words of the problem.

A letter of T at position k is denoted by $T[k]$ and $T[i\,..\,j]$ denotes the factor $T[i]T[i+1]\cdots T[j]$ of T. Then, the T-specific table Ts is defined, for $i = 0, \ldots, |T|-1$, by

$$Ts[i] = \begin{cases} j, & \text{if } T[i\,..\,j] \text{ is } T\text{-specific}, i \le j, \\ -1, & \text{else.} \end{cases}$$

Note that the set of T-specific factors is prefix-free, that is, no proper prefix of an element of the set is also in the set. (The set of T-specific factors is also suffix-free.) Therefore, for each position k on T, there is at most one T-specific factor of T starting at k (and for each position j on T there is at most one T-specific factor of T ending at j).

Instead of computing the T-specific table Ts, in a straightforward way, the algorithm below can be transformed to compute the list of pairs (i, j) of positions on T for which $Ts[i] = j$ and $j \neq -1$.

To compute the table we use $\mathcal{R}$, the Suffix automaton or rather the DAWG of R. The former is the minimal automaton accepting the suffixes of R (see [14, Section[5.4]) and the latter has the same structure but with all states as terminal states instead (see [8]). As such, it accepts $\mathrm{Fact}(R)$ the set of factors of R.

The automaton is given with its transition function δ, its initial state i and and is equipped with both the suffix link s (used here as a failure link) and the length function ℓ defined on states. The function ℓ is defined by: $\ell[p] = \max\{|z| \in A^* \mid \delta(i, z) = p\}$. The functions s and ℓ transform the automaton into a search machine (see [14, Section 6.6]).

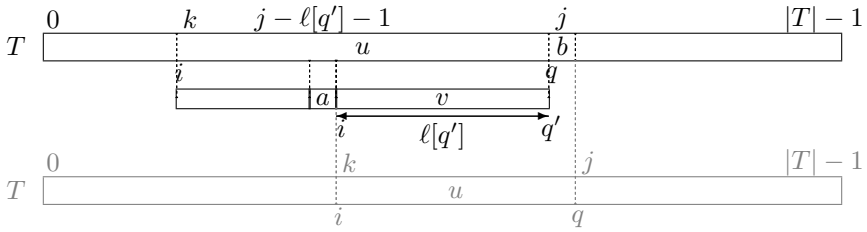


Figure 3 A T-specific word found: when $u \in \mathrm{Fact}(R)$ and $ub \notin \mathrm{Fact}(R)$, either avb or b is a T-specific factor with respect to R (a, b are letters). The gray bottom displays the situation, and specifically the variables q and j used in Algorithm TsTable, after processing letter b.

Figure 3 illustrates the principle of Algorithm TsTable. Let us assume that the factor $u = T[k\,..\,j-1]$ is a factor of R but ub is not for some letter b. Then, let v be the longest suffix of u for which vb is a factor of R. If it exists, then clearly avb, with the letter a preceding v, is T-specific. Indeed, $av, vb \in \mathrm{Fact}(R)$ and $avb \notin \mathrm{Fact}(R)$, which means that avb is a minimal absent word of R while occurring in T. Therefore, setting $q' = \delta(i, v)$ and since $|v| = \ell[q']$, the minimal absent word avb is identified by setting $Ts[j - \ell[q'] - 1] = j$. If there is no suffix of u satisfying the condition, the letter b alone is T-specific and is identified by setting $Ts[j] = j$.

```
TsTable(T target word, R DAWG(R), i initial(R))
 1  (q, j) ← (i, 0)
 2  while j < |T| do
 3        Ts[j] ← −1
 4        if δ(q, T[j]) undefined then
 5              while q ≠ i and δ(q, T[j]) undefined do
 6                    q ← s[q]
 7              if δ(q, T[j]) undefined then      ▷ q = i
 8                    Ts[j] ← j
 9              else Ts[j − ℓ[q] − 1] ← j
10                    q ← δ(q, T[j])
11        else q ← δ(q, T[j])
12        j ← j + 1
13  return Ts
```

▶ **Theorem 6.** *Algorithm* TsTable *computes the T-specific table with respect to R and runs in linear time, that is, $O(|T|)$ on a fixed-size alphabet.*

Proof. The algorithm implements the ideas detailed above and illustrated by Figure 3. The formal proof relies on the following invariant condition of the main while loop: let u be a factor of R and q be a state of $\mathcal{R}$ that satisfies $q = \delta(i, u)$, and let j be the current position on T, then u is the longest factor of R ending at position $j - 1$ on T. Before the first pass in the main while loop starts, u is the empty word, $q = i$, $j = 0$, and the condition is satisfied.

During each pass in the main while loop, by default $Ts[j]$ is first set to -1 (line 3) to cover the possibility that no minimal absent word ends at position j. Let us examine what is done during a pass in the while loop after the instruction at line 3.

- If $\delta(q, T[j])$ is defined at line 4, the next value of q is set at line 11, followed by the increment of j at line 12. Therefore, $uT[j]$ is the longest factor of R ending at $j - 1$, as required for the invariant to hold.
- The case where $\delta(q, T[j])$ is undefined at line 4 is illustrated on Figure 3. Then, the loop at lines 5-6 uses the suffix link s to finds the longest suffix v of u for which $vb = vT[j]$ is a factor of R by the definition of s in the DAWG.
  - If the execution of the loop terminates with $\delta(q, T[j])$ still undefined, this indicates that q is the initial state and that v is the empty word. Consequently, the letter $T[j]$ is the minimal absent word at position j.
  - Otherwise, first note that v is shorter than u because ub is not a factor of R. Thus, there exists a letter a such that av is a suffix of u. Therefore, avb is the minimal absent word ending at position j because $av, vb \in \mathrm{Fact}(R)$ but $avb \notin \mathrm{Fact}(R)$. This is established at line 9 because the position of letter a is $j - \ell[q] - 1$. The subsequent execution of lines 9 and 10 ensures that the invariant condition holds in this case as well.

As for the running time, note that the instructions at lines 3 and 7-12 execute in constant time for each value of j. All the executions of the instruction at line 6 execute in time $O(|T|)$ because the link s reduces strictly the potential length of the T-specific word ending at j, incrementing the starting position $j - \ell[q] - 1$ of v in the picture.

Moreover, each computation of a transition $\delta(q, T[j])$ executes in constant time on a fixed-size alphabet if, for example, the DAWG is implemented with a sparse matrix technique.

Thus, the entire execution is completed in time $O(|T|)$. ◀

Algorithm TsTable can be improved to run in real-time on a fixed-size alphabet. This is done by optimising the suffix link s defined on the automaton $\mathcal{R}$. To do so, let us define, for each state q of $\mathcal{R}$,

$$\mathsf{out}(q) = \{a \mid \delta(q, a) \text{ defined for letter } a\}.$$

Then, the optimised suffix link G is defined by $G[\mathit{initial}(\mathcal{R})] = \mathbf{nil}$ and, for any other state q of $\mathcal{R}$, by

$$G[q] = \begin{cases} s[q], & \text{if } \mathsf{out}(q) \subset \mathsf{out}(s[q]), \\ G[s[q]], & \text{else.} \end{cases}$$

Note that, since we always have $\mathsf{out}(q) \subseteq \mathsf{out}(s[q])$, the definition of G can be reformulated as

$$G[q] = \begin{cases} s[q], & \text{if } \mathsf{deg}(q) < \mathsf{deg}(s[q]), \\ G[s[q]], & \text{else,} \end{cases}$$

where deg is the outgoing degree of a state. Therefore, its computation can be performed in linear time with respect to the number of states of $\mathcal{R}$. After substituting G for s in Algorithm TsTable, when the alphabet is of size α the instruction at line 6 executes no more than α times for each value of q. So the time to process a given state q is constant. This is summarized in the next corollary.

▶ **Corollary 7.** *When using the optimised suffix link, Algorithm TsTable runs in real time on a fixed-size alphabet.*

On a more general alphabet of size α, processing a given state of the automaton can be done in time $O(\log \alpha)$.

5 Absent word searching vs string matching

Searching for absent factor occurrences as done in the previous section is based on the DAWG matching technique [13] (see also [14, Section 6.6]) in the domain of string matching algorithms. As such, Algorithm TsTable looks like a side product of string matching.

The goal of string matching algorithms is to locate a given pattern p in a longer text T. Figure 3 displays the generic situation when a part u of the pattern has been found in T and is about to be appended with the letter b. The extension of u is successful if b matches its aligned letter of the pattern, which eventually can lead to the detection of an occurrence of the whole pattern.

The situation in Figure 3 is also the generic situation when searching for minimal absent words. However, in contrast, this is an unsuccessful match of the letter b that immediately yields the detection in T of a minimal absent word of p.

The role of the DAWG matching technique is essential here to detect minimal absent words of p occurring in T because the DAWG of p stores and provides direct access to all factors of p. Instead, if an online string matching algorithm is used, like KMP [24] or Simon-Hancart [34, 22] algorithms (see also [14, Chapter 2]), the algorithm will detect only some absent words. That is, those of the form aub in which au is a factor of p but ub is only a prefix of it. This does not produce all the minimal absent words. However, other types of string matching based on text indexes can certainly be used for the same purpose.

As a conclusion, the algorithm designed in the previous section reveals a surprising phenomenon: the very tight link between the existence of minimal absent words that pop up naturally in the analysis of pattern searching based on a Suffix automaton or a DAWG.

References

1 Lorraine A. K. Ayad, Golnaz Badkobeh, Gabriele Fici, Alice Héliou, and Solon P. Pissis. Constructing antidictionaries of long texts in output-sensitive space. *Theory Comput. Syst.*, 65(5):777–797, 2021. doi:10.1007/S00224-020-10018-5.

2 Marie-Pierre Béal and Maxime Crochemore. Fast detection of specific fragments against a set of sequences. In *27th International Conference on Developments in Language Theory, DLT 2023*, volume 13911 of *Lecture Notes in Computer Science*, pages 51–60, 2023. doi:10.1007/978-3-031-33264-7_5.

3 Marie-Pierre Béal, Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Computing forbidden words of regular languages. *Fundam. Informaticae*, 56(1-2):121–135, 2003. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi56-1-2-08.

4 Marie-Pierre Béal, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Forbidden words in symbolic dynamics. *Adv. Appl. Math.*, 25(2):163–193, 2000. doi:10.1006/aama.2000.0682.

5 G. Bernardini, C. Liu, G. Loukides, A. Marchetti-Spaccamela, S.P. Pissis, L. Stougie, and M. Sweering. Missing value replacement in strings and applications. *Data Mining and Knowledge Discovery*, 39(12), 2025. doi:10.1007/s10618-024-01074-3.

6 Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering. Hide and Mine in Strings: Hardness, Algorithms, and Experiments . *IEEE Transactions on Knowledge & Data Engineering*, 35(06):5948–5963, June 2023. doi:10.1109/TKDE.2022.3158063.

7 A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.

8 Anselm Blumer, J. Blumer, Andrzej Ehrenfeucht, David Haussler, and Ross M. McConnell. Building the minimal DFA for the set of all subwords of a word on-line in linear time. In Jan Paredaens, editor, *Automata, Languages and Programming, 11th Colloquium, Antwerp, Belgium, July 16-20, 1984, Proceedings*, volume 172 of *Lecture Notes in Computer Science*, pages 109–118. Springer, 1984. doi:10.1007/3-540-13345-3_9.

9 Paola Bonizzoni, Clelia De Felice, Yuri Pirola, Raffaella Rizzi, Rocco Zaccagnino, and Rosalba Zizza. Can formal languages help pangenomics to represent and analyze multiple genomes? In Volker Diekert and Mikhail V. Volkov, editors, *Developments in Language Theory - 26th International Conference, DLT 2022, Tampa, FL, USA, May 9-13, 2022, Proceedings*, volume 13257 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2022. doi:10.1007/978-3-031-05578-2_1.

10 Supaporn Chairungsee and Maxime Crochemore. Using minimal absent words to build phylogeny. *Theor. Comput. Sci.*, 450:109–116, 2012. doi:10.1016/J.TCS.2012.04.031.

11 Panagiotis Charalampopoulos, Maxime Crochemore, Gabriele Fici, Robert Mercas, and Solon P. Pissis. Alignment-free sequence comparison using absent words. *Inf. Comput.*, 262:57–68, 2018. doi:10.1016/j.ic.2018.06.002.

12 Maxime Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986. doi:10.1016/0304-3975(86)90041-1.

13 Maxime Crochemore. Longest common factor of two words. In Ehrig, Kowalski, Levi, and Montanari, editors, *TAPSOFT'87 (Pisa, 1987)*, number 249 in LNCS, pages 26–36. Springer-Verlag, 1987. doi:10.1007/3-540-17660-8_45.

14 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.

15 Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Absent words in a sliding window with applications. *Inf. Comput.*, 270, 2020. doi:10.1016/j.ic.2019.104461.

16 Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Inf. Process. Lett.*, 67(3):111–117, 1998. doi:10.1016/S0020-0190(98)00104-5.

17 Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Sergio Salemi. Data compression using antidictionaries. *Proc. IEEE*, 88(11):1756–1768, 2000. doi:10.1109/5.892711.

18 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

19 Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing dawgs and minimal absent words in linear time for integer alphabets. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*, volume 58 of *LIPIcs*, pages 38:1–38:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.MFCS.2016.38.

20 Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Linear-time computation of dawgs, symmetric indexing structures, and maws for integer alphabets. *Theor. Comput. Sci.*, 973:114093, 2023. doi:10.1016/J.TCS.2023.114093.

21 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.

22 Christophe Hancart. On simon's string searching algorithm. *Inf. Process. Lett.*, 47(2):95–99, 1993. doi:10.1016/0020-0190(93)90231-W.

23 Parsoa Khorsand, Luca Denti, Human Genome Structural Variant Consortium, Paola Bonizzoni, Rayan Chikhi, and Fereydoun Hormozdiari. Comparative genome analysis using sample-specific string detection in accurate long reads. *Bioinformatics Advances*, 1(1), May 2021. doi:10.1093/bioadv/vbab005.

24 Donald E. Knuth, J. H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

25 Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, May 2012. doi:10.1093/bioinformatics/bts280.

26 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

27 Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Forbidden factors in finite and infinite words. In Juhani Karhumäki, Hermann A. Maurer, Gheorghe Paun, and Grzegorz Rozenberg, editors, *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 339–350. Springer, 1999.

28 Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings—practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. 232 pages. doi:10.1017/CBO9781316135228.

29 Kouta Okabe, Takuya Mieno, Yuto Nakashima, Shunsuke Inenaga, and Hideo Bannai. Linear-time computation of generalized minimal absent words for multiple strings. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 331–344. Springer, 2023. doi:10.1007/978-3-031-43980-3_27.

30 Armando J. Pinho, Paulo Jorge S. G. Ferreira, Sara P. Garcia, and João M. O. S. Rodrigues. On finding minimal absent words. *BMC Bioinform.*, 10, 2009. doi:10.1186/1471-2105-10-137.

31 Diogo Pratas and Jorge M Silva. Persistent minimal sequences of sars-cov-2. *Bioinformatics*, 36(21):5129–5132, July 2020. doi:10.1093/bioinformatics/btaa686.

32 Mohammad Saifur Rahman, Ali Alatabbi, Tanver Athar, Maxime Crochemore, and M. Sohel Rahman. Absent words and the (dis)similarity analysis of dna sequences: an experimental study. *BMC Research Notes*, 9(186):1–8, 2016. doi:10.1186/s13104-016-1972-z.

33 Raquel M. Silva, Diogo Pratas, Luísa Castro, Armando J. Pinho, and Paulo Jorge S. G. Ferreira. Three minimal sequences found in ebola virus genomes and absent from human DNA. *Bioinform.*, 31(15):2421–2425, 2015. doi:10.1093/bioinformatics/btv189.

34 Imre Simon. String matching algorithms and automata. In Juhani Karhumäki, Hermann A. Maurer, and Grzegorz Rozenberg, editors, *Results and Trends in Theoretical Computer Science, Colloquium in Honor of Arto Salomaa, Graz, Austria, June 10-11, 1994, Proceedings*, volume 812 of *Lecture Notes in Computer Science*, pages 386–395. Springer, 1994. doi:10.1007/3-540-58131-6_61.

Wavelet Tree, Part I: A Brief History

Paolo Ferragina
Department L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy
Department of Computer Science, University of Pisa, Italy

Raffaele Giancarlo
Department of Mathematics and Computer Science, University of Palermo, Italy

Giovanni Manzini
Department of Computer Science, University of Pisa, Italy

Giovanna Rosone
Department of Computer Science, University of Pisa, Italy

Rossano Venturini
Department of Computer Science, University of Pisa, Italy

Jeffrey Scott Vitter
Department of Computer Science, Tulane University, New Orleans, LA, USA
Department of Computer and Information Science, The University of Mississippi, MS, USA

Abstract

The *Wavelet Tree* data structure introduced in Grossi, Gupta, and Vitter [16] is a space-efficient technique for rank and select queries that generalizes from binary symbols to an arbitrary multisymbol alphabet. Over the last two decades, it has become a pivotal tool in modern full-text indexing and data compression because of its properties and capabilities in compressing and indexing data, with many applications to information retrieval, genome analysis, data mining, and web search. In this paper, we survey the fascinating history and impact of Wavelet Trees; no doubt many more developments are yet to come. Our survey borrows some content from the authors' earlier works.

This paper is divided into two parts: one (this one) giving a brief history of Wavelet Trees, including its varieties and practical implementations, dedicated to this Festschrift's honoree Roberto Grossi; the second part deals with Wavelet Tree-based text indexing and is included in the Festschrift dedicated to Giovanni Manzini [10].

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Data compression

Keywords and phrases Wavelet tree, data compression, text indexing, compressed suffix array, Burrows-Wheeler transform, rank and select

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.15

Category Research

Acknowledgements The authors would like to thank Hongwei Huo for helpful comments and figures.

1 Introduction

The field of compressed full-text indexing [29] involves the design of data structures (aka, indexes) that support fast substring matching using small amounts of space. For a text string $\mathcal{T}[1, n]$ over an arbitrary alphabet Σ of size σ and a pattern $\mathcal{P}[1, m]$, the goal of text indexing is to preprocess $\mathcal{T}$ using succinct space so that queries like the following can be quickly answered: (1) count the number *occ* of occurrences of $\mathcal{P}$ in $\mathcal{T}$; (2) locate the *occ* positions in $\mathcal{T}$ where $\mathcal{P}$ occurs; and (3) starting at text position *start*, extract the length-ℓ substring $\mathcal{T}[start, start + \ell - 1]$.

A main goal is to create an index whose size is roughly equal to the size of the text in compressed format, with search performance comparable to the well-known indexes on uncompressed text, such as suffix trees and suffix arrays. Some compressed data structures are in addition *self-indexes* in that the data structure encapsulates the original text and, thus, can quickly recreate any portion of it. As a result, the original text is not needed, it can be discarded and replaced by only the (self-)index.

Most compressed indexing techniques developed in the last two decades make use of the powerful *Wavelet Tree* data structure, developed by Grossi, Gupta, and Vitter [16], with many applications to information retrieval, genome analysis, data mining, and web search. A Wavelet Tree supports fast rank and select queries on any string from a general multisymbol alphabet Σ. As such, it provides an elegant extension of rank-select data structures for binary sequences [31, 29] to the general case of alphabets of arbitrary size.

The Wavelet Tree represents a string of symbols from an arbitrary alphabet Σ as a hierarchical set (or tree) of binary strings. It has the following key compression property: If each such binary string is compressed according to its information-theoretic minimum size, then the original string is compressed to its information-theoretic minimum size.

The Wavelet Tree tree structure can be thought of as a generalization in the context of computational geometry of the two-dimensional range search data structure of Chazelle [4] in which the two-dimensional points stored in a binary tree node are initially sorted in one dimension and then recursively partitioned into children based upon the second dimension. Kärkkäinen [21] used a similar data structure in a very different way and purpose in the context of highly repetitive texts.

Why the name "Wavelet Tree"? One of the codevelopers suggested the name based upon his research interests in the fields of image compression and database query optimizaton, where he used wavelet transforms and their hierarchical decompositions to approximate multiresolution data [33].

In this paper (`PART I`), we give a brief history of Wavelet Trees, including its varieties and practical implementations. We refer the reader to `PART II`, included in the Festschrift dedicated to Giovanni Manzini, for a discussion about the important role of Wavelet Trees in text indexing, particularly for compressed suffix arrays and the FM-index, which can produce self-indexes with size related to the higher-order entropy of the text.

2 Preliminaries

Let $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ be a finite ordered alphabet of size σ. Let $\mathcal{T} = \mathcal{T}[0, n-1]$ be a text string consisting of n symbols from alphabet Σ. For simplicity, we assume that $\mathcal{T}[n-1]$ is a special end-of-text symbol # that is the lexicographically smallest symbol in Σ. We use $\mathcal{T}[i, j]$ to denote the substring of $\mathcal{T}$ consisting of the symbols $\mathcal{T}[i]\mathcal{T}[i+1]\ldots\mathcal{T}[j]$. A prefix of $\mathcal{T}$ is a substring of the form $\mathcal{T}[0, i]$, and a suffix is a substring of the form $\mathcal{T}[j, n-1]$.

Two key operations we will need for accessing and traversing Wavelet Trees are *rank* and *select*:

▶ **Definition 1** (rank and select). *Let $\mathcal{B}[1, m]$ be a bit array of length m and let bit $b \in \{0, 1\}$. We define the rank and select operations as follows:*

- *$rank_b(\mathcal{B}, i)$: return the number of occurrences of bit b in $\mathcal{B}[1, i]$ for any $1 \leq i \leq m$;*
- *$select_b(\mathcal{B}, j)$: return the position in $\mathcal{B}$ of the jth occurrence of bit b for any $1 \leq j \leq \mathsf{rank}_b(\mathcal{B}, m)$.*

▶ **Definition 2** ((0th-order) empirical entropy). *Let $\mathcal{T}$ be a text string of length n over an alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ of size σ. The 0th-order empirical entropy of $\mathcal{T}$ is defined as*

$$\mathcal{H}_0 = \mathcal{H}_0(\mathcal{T}) = \frac{1}{n} \sum_{\omega \in \Sigma} n_\omega \log \frac{n}{n_\omega}$$

where n_ω is the number of occurrences in $\mathcal{T}$ of symbol $\omega \in \Sigma$.

A more stringent notion of entropy is the *information-theoretic minimum* (or *0th-order finite-set entropy*), given by

$$\frac{1}{n} \log \binom{n}{n_1, n_2, \ldots, n_\sigma}.$$

It is always less than or equal to the 0th-order empirical entropy.

Higher orders of entropy can be defined readily if we encode symbols based upon their frequencies of occurrence when grouped into common contexts.

▶ **Definition 3** (kth-order empirical entropy). *Let $\mathcal{T}$ be a text string of length n over an alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ of size σ. The kth-order empirical entropy of $\mathcal{T}$ is defined as*

$$\mathcal{H}_k = \mathcal{H}_k(\mathcal{T}) = \frac{1}{n} \sum_{\mathsf{x} \in \Sigma^k} n_\mathsf{x}\, \mathcal{H}_0(\mathcal{T}_\mathsf{x})$$

where $\mathsf{x} \in \Sigma^k$ designates a string (context) of length k and $\mathcal{T}_\mathsf{x}$ denotes the string of length n_x formed by taking the symbol immediately preceding each occurrence of x in $\mathcal{T}$ and concatenating all these symbols together.

In a like manner, we can define the *kth-order finite-set entropy* by

$$\frac{1}{n} \sum_{\mathsf{x} \in \Sigma^k} \log \binom{n_\mathsf{x}}{n_{1,\mathsf{x}}, n_{2,\mathsf{x}}, \ldots, n_{\sigma,\mathsf{x}}},$$

where $n_{y,\mathsf{x}}$ for $y \in \Sigma$ is the number of occurrences of $y\mathsf{x}$ as a substring in $\mathcal{T}$. The kth-order finite-set entropy is always less than or equal to the kth-order empirical entropy.

In other words, the key point of the kth-order entropy definition is that we can achieve it by encoding the relevant parts of $\mathcal{T}$ using 0th-order entropy. (This property was later appropriately dubbed "boosting.") In particular, if we form the substrings $\mathcal{T}_\mathsf{x}$ for each relevant k-context x and compress each $\mathcal{T}_\mathsf{x}$ to achieve space of n_x times its 0th-order empirical entropy $\mathcal{H}_0(\mathcal{T}_\mathsf{x})$, then the resulting compression for the full text string $\mathcal{T}$ will be n times the kth-order empirical entropy $\mathcal{H}_k(\mathcal{T})$. Wavelet Trees are a natural and elegant way to compress each substring $\mathcal{T}_\mathsf{x}$ to $n_\mathsf{x}\, \mathcal{H}_0(\mathcal{T}_\mathsf{x})$ bits plus lower-order terms, and thus the resulting cumulative encoding of $\mathcal{T}$ has size in bits bounded by $n\, \mathcal{H}_k(\mathcal{T})$ plus lower-order terms. This fact is exploited in `PART II` to show how Wavelet Trees can be used to create text indexes that use space related to the high-order entropy of $\mathcal{T}$.

For two strings S and T, we define the *lexicographic order* $S \prec T$ (and say that S is smaller than T) if S is a proper prefix of T, or if there exists an index $1 \le i \le \min\{|S|, |T|\}$ such that $S[i] < T[i]$ and for all $j < i$, $S[j] = T[j]$.

We use $SA[0, n-1]$ to denote the suffix array (SA) of $\mathcal{T}$. It consists of the positions of all the suffixes of $\mathcal{T}$ in lexicographical order. For example, in the text string $\mathcal{T} =$ `tcaaaatatatgcaacatatagtattagattgtat#` in Table 1, the smallest suffix begins at position $SA[0] = 35$, and the next smallest suffix begins at position $SA[1] = 2$. The ordered suffixes

Table 1 For text $\mathcal{T}$ = `tcaaaatatatgcaacatatagtattagattgtat#` containing symbols from the alphabet Σ = {`#,a,c,g,t`} shown in the first column, the subsequent columns show the suffix array SA, its neighbor function Ψ, and the LF function of the BWT. The (wide) second-to-last column shows the suffixes in sorted order; each suffix starts at the symbol under F (first) and ends at the symbol under L (last). The BWT of $\mathcal{T}$ is the string of symbols L = `tcacaattttcatttgtgaattaatagaaag#ataa`. The last column $\mathcal{B}$ is the bit array for the root node of the Wavelet Tree of L; symbols `#`, `a`, and `c` (for the left subtree) are designated by a 0, and symbols `g` and `t` (for the right subtree) are designated by a 1. The Wavelet Tree for L is pictured in Figure 1.

i	$\mathcal{T}$	SA	Ψ	LF	F	L	$\mathcal{B}$
0	t	35	31	23	# t c a ...a	t	1
1	c	2	2	16	a a a a ...t	c	0
2	a	3	4	1	a a a t ...c	a	0
3	a	13	5	17	a a c a ...g	c	0
4	a	4	11	2	a a t a ...a	a	0
5	a	14	18	3	a c a t ...c	a	0
6	t	26	19	24	a g a t ...t	t	1
7	a	20	22	25	a g t a ...a	t	1
8	t	33	23	26	a t # t ...g	t	1
9	a	18	25	27	a t a g ...a	t	1
10	t	16	27	18	a t a t ...a	c	0
11	g	5	28	4	a t a t ...a	a	0
12	c	7	29	28	a t a t ...a	t	1
13	a	9	32	29	a t g c ...a	t	1
14	a	23	34	30	a t t a ...g	t	1
15	c	28	35	19	a t t g ...a	g	1
16	a	1	1	31	c a a a ...#	t	1
17	t	12	3	20	c a a c ...t	g	1
18	a	15	10	5	c a t a ...a	a	0
19	t	27	15	6	g a t t ...t	a	0
20	a	11	17	32	g c a a ...a	t	1
21	g	31	26	33	g t a t ...t	t	1
22	t	21	30	7	g t a t ...t	a	0
23	a	34	0	8	t # t c ...t	a	0
24	t	25	6	34	t a g a ...a	t	1
25	t	19	7	9	t a g t ...t	a	0
26	a	32	8	21	t a t # ...t	g	1
27	g	17	9	10	t a t a ...c	a	0
28	a	6	12	11	t a t a ...a	a	0
29	t	8	13	12	t a t g ...t	a	0
30	t	22	14	22	t a t t ...a	g	1
31	g	0	16	0	t c a a ...t	#	0
32	t	10	20	13	t g c a ...t	a	0
33	a	30	21	35	t g t a ...a	t	1
34	t	24	24	14	t t a g ...t	a	0
35	#	29	33	15	t t g t ...g	a	0

are pictured in the (wide) second-to-last column of the table. For convenience, we regard text $\mathcal{T}$ as a circular string and implicitly use modular arithmetic base n for the indices of $\mathcal{T}$. In other words, references to a symbol $\mathcal{T}[i]$ actually refer to $\mathcal{T}[i \bmod n]$.

Further development of suffix arrays that leads to the notion of compressed suffix arrays is discussed in `PART II`. Compressed suffix arrays are self-indexes that use space proportional to the high-order entropy of the text and provide fast pattern-matching queries. The key

notion that leads to the compression of suffix arrays is the important *neighbor function*, which for each $i \in [0, n-1]$ maps i to the lexicographic order of the suffix formed by taking the ith smallest suffix of $\mathcal{T}$ and rotating it to the left by one symbol.

The Burrows-Wheeler Transform (BWT) is another reversible transformation of a text string $\mathcal{T}$ that rearranges the symbols of $\mathcal{T}$ in a way that groups together the same symbols from similar contexts, thus resulting in high compressibility [2, 26]. In terms of Table 1, the BWT is represented by the last subcolumn (designated by L). `PART II` provides further development of the BWT that leads to the FM-index, which like the compressed suffix array, uses space proportional to the high-order entropy of the text and provides fast pattern-matching queries. The key notion behind the FM-index and its compression is the LF function, which is the inverse of the neighbor function for compressed suffix arrays mentioned above. That is, the LF function maps each i to the lexicographic order of the suffix formed by taking the ith smallest suffix of $\mathcal{T}$ and rotating it to the right (rather than to the left) by one symbol. The symbol under L in row i is the same symbol under F in row $LF(i)$.

3 Wavelet trees

Wavelet Trees provide an elegant and efficient implementation of rank-select data structures for strings of symbols from an arbitrary multisymbol alphabet. They generalize the well-known rank-select data structures such as RRR [31] that handle binary strings.

Conceptually, the Wavelet Tree is most simply described as a full (and often balanced) binary tree (see the example in Figure 1). The root node is a bit array $\mathcal{B}$ of the same length as the input text $\mathcal{T}$ and partitions the text's alphabet into two sets, where a 0-bit indicates that the corresponding text symbol is in the left set and a 1-bit indicates that the text symbol is in the right set. (See Table 1 for the bit array $\mathcal{B}$ applied to the string L.) Such a bit array representation happens recursively at each internal node, where the text at the internal node is of the symbols dispatched from the parent node with their order in the text preserved. Each leaf node represents a distinct symbol of the alphabet of the input text, with its multiplicity preserved.

Operations of *rank* and *select* on binary strings, as defined in Definition 1, are used to navigate up and down the Wavelet Tree. For example, in the root node of the Wavelet Tree of Figure 1, consider the symbol a in the sixth position of L, which is represented by a 0, which means that its corresponding entry on the next level is in the left subtree. To find that entry, we compute $rank_0(L, 6) = 5$ using a data structure such as RRR; we thus find that the corresponding entry is entry 5 in the left subtree. To go from that entry on level 2 back to the position of the corresponding entry in root node, we compute $select_0(L, 5) = 6$.

Of course, extra data structures are needed in order to support the needed query operations and functionality in an efficient way. Those details are not given here in this survey. Instead we focus on the main ideas that make wavelet trees such an elegant and powerful method.

The key aspect of Wavelet Trees is that they cleverly decompose the text into a hierarchy of bit arrays without introducing any redundancy, so that the total size of the raw bit arrays in the Wavelet Tree is exactly the same as the size (in bits) of the input text, regardless of the shape of the Wavelet Tree. If the bit array at each node is 0th-order entropy compressed, the resulting cumulative space usage of the Wavelet Tree is equal to n times the 0th-order entropy-compressed size of the input text, again regardless of the shape of the tree [16, 11].

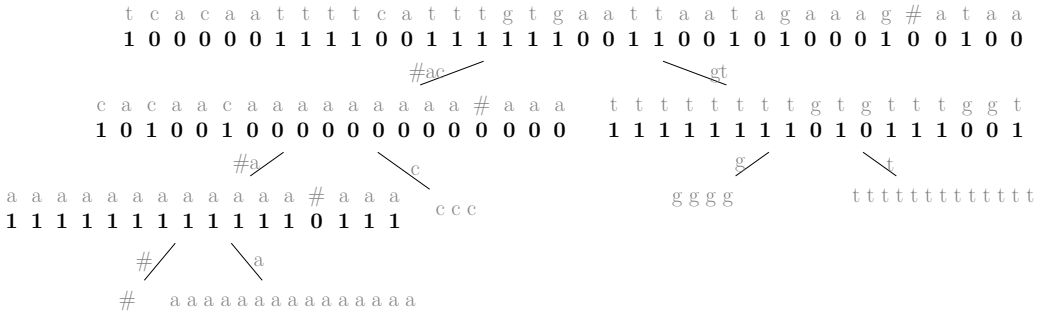


Figure 1 The Wavelet Tree for string $L =$ `tcacaattttcatttgtgaattaatagaaag#ataa` of Table 1.

4 Properties of Wavelet Trees

The Wavelet Tree pictured in Figure 1 is a balanced binary tree, in which the number of symbols at each internal node are partitioned as evenly as possible for the level below. As such it has at most $\log \sigma$ levels, and each level (in raw uncompressed form) consists of a bit array of length n. Using RRR [31] to encode each internal node allows fast rank and select queries for general alphabets on the overall tree.

For an internal node of m bits with ℓ 1-bits and $m - \ell$ 0-bits, the RRR encoding [31] uses $\log \binom{m}{\ell} + o(\ell) + \mathcal{O}(\log \log m)$ bits, where $\log \binom{m}{\ell}$ is the information theoretic minimum. This form of entropy is called 0th-order finite-set entropy; it is always less than or equal to the empirical entropy of Definition 2, since it removes dependence from the statistical model. Summing the $\log \binom{m}{\ell}$ terms over all the internal nodes of the Wavelet Tree cascades into the multinomial coefficient $\binom{n}{n_1, n_2, \ldots, n_\sigma}$ [17], which is less than or equal n times the 0th-order empirical entropy of the text $\mathcal{T}$.

There is much interesting work on efficient ways to construct Wavelet Trees. For example, Munro *et al.* [27] show that construction can be done in $\mathcal{O}\big(n(\log \sigma)/\sqrt{\log n}\big)$ time.

5 Varieties of Wavelet Trees and Their Characteristics

There are several variants of Wavelet Trees. In this section, we focus on Huffman Wavelet Trees, Pruned Wavelet Trees, and Wavelet Trees with multiway branching.

In [24, 28] the authors propose giving the Wavelet Tree the Huffman shape of the frequencies with which the symbols appear in the text string $\mathcal{T}$. It has been shown that the total number of bits stored in the Huffman Wavelet Tree is exactly the number of bits output by a Huffman compressor that takes the symbol frequencies in $\mathcal{T}$, which is upper-bounded by $n(\mathcal{H}_0(\mathcal{T}) + 1)$. Moreover, the accesses to the leaves in the Huffman Wavelet Tree can be obtained with frequency proportional to their number of occurrences in T. Huffman shapes can be combined with multiway Wavelet Trees and entropy compression of the bitmaps [1]. In this case, one can achieve space $n\mathcal{H}_0(\mathcal{T}) + o(n)$ bits, worst-case time $\mathcal{O}(1 + (\log \sigma)/\log \log n)$, and average case time $\mathcal{O}(1 + \mathcal{H}_0(\mathcal{T})/\log \log n)$ for both rank and select queries.

Ferragina, Giancarlo, and Manzini [11] addressed the impact of the Wavelet Tree data structure on data compression by providing a theoretical analysis of a wide class of compression algorithms based upon Wavelet Trees, and proposing a novel framework, called *pruned Wavelet Trees*, that aims for the best combination of Wavelet Trees of properly-designed shapes and compressors either binary (like, run-length encoders) or non-binary (like, Huffman and arithmetic encoders). There were three main contributions in that paper.

First, they proposed a thorough analysis of Wavelet Trees as *stand-alone* general-purpose compressor, by considering two specific cases (extending [13]) in which either the binary strings associated to their nodes are compressed via RLE (referred to as the Rle Wavelet Tree) or via Gap Encoding (referred to as the Ge Wavelet Tree). These analyses provided compression bounds that depended on the features of these prefix-free encoders and the 0th-order entropy of the input string. As a result, the authors were able to prove that Rle Wavelet Trees can achieve a compression bound in terms of the kth order empirical entropy, whereas Ge Wavelet Trees *cannot*. This result has been then strengthened in [25] where the authors showed that Rle can be replaced by the succinct dictionaries of [31].

Ferragina *et al.* [11] introduced the *pruned Wavelet Trees* that generalize and improve Wavelet Trees by working on (C.1) the shape (or topology) of the binary tree underlying their structure; and on (C.2) the assignment of alphabet symbols to the binary-tree leaves. The authors showed that it is possible to exhibit an infinite family of strings over an alphabet Σ for which changing the Wavelet Tree shape influences the coding cost by a factor $\Theta(\log \sigma)$, and changing the assignment of symbols to leaves influences the coding cost by a factor $\Theta(\sigma)$. Moreover, they showed that (C.3) Wavelet Trees commit to binary compressors, losing the potential advantage that might come from a mixed strategy in which only some strings are binary and the others are defined on an arbitrary alphabet (and can be compressed via general-purpose 0th-order compressors, such as arithmetic and Huffman coding). Again, that paper showed that it is possible to exhibit an infinite family of strings for which a mixed strategy yields a *constant* multiplicative factor improvement over standard Wavelet Trees.

It is exactly to address these questions that Ferragina *et al.* [11] introduced the pruned Wavelet Trees where only a subset of their nodes is binary, and then developed a combinatorial optimization strategy addressing (C1)–(C3) simultaneously. As a corollary, they specialized that strategy to design a polynomial-time algorithm for finding the optimal pruned Wavelet Tree of fixed shape and assignment of alphabet symbols to its leaves; or, for selecting the optimal tree shape when only the assignment of symbols to the leaves of the tree is fixed.

Ferragina *et al.* [12] showed that increasing the branching factor in the wavelet tree nodes can reduce the time for rank and select queries to constant time, assuming that the alphabet size satisfies $\sigma = \mathcal{O}(\text{polylog}\, n)$.

6 Practical Implementations of Wavelet Trees

In this section, we review some approaches for designing fast and compact implementations of Wavelet Trees. Grossi, Vitter, and Xu [18] provided a full generic package of Wavelet Trees for a wide range of options on the dimensions of coding schemes and tree shapes. Their experimental study reveals the practical performance of the various modifications. It provides potential users a rationale for how to implement a Wavelet Tree based upon the particular characteristics of the underlying datat.

The authors[18] considered three styles of Wavelet Trees:

- Normal: Each node is partitioned by alphabetically dividing the symbols represented in the node into two nearly *equal-sized* halves;
- Alphabetic weight-balanced: Each node is partitioned by alphabetically dividing the symbols represented in the node into two nearly *equal-weighted* halves, where the weights are designated by the frequencies of the symbols; and
- Huffman: Each node is partitioned as in Huffman compression.

They also considered several styles of encoding the conceptual bit array in each node, the main ones being

- None: The raw bit array is used at each node; support for member, rank, and select queries is provided by $o(n)$-space auxiliary structures using superblocks and lookup-table based popcount operations [15];
- RRR: The practical implementation of the RRR method [31] given in [6];
- RLE+γ: Run-length encoding with Elias γ encoding [8] for the runs; queries are supported using the $o(n)$-space auxiliary structures [15];
- RLE+δ: Same as RLE+γ, but using Elias δ encoding [8] instead of γ;
- SC: Small-integer t-subset encoding [22];
- AC: Arithmetic coding [19]; and
- LP/$*$: Bit arrays are compressed by a specified method iff there are at most three distinct symbols represented in the node, since nodes close to the leaf level are typically highly compressible.

The overall results of experiments on 33 Wavelet Tree options on texts from English, Protein, and Genome sources (several with very low entropy) suggest that RLE-encoded Wavelet Trees are the best in space efficiency when coding BWTs or sequences with very low 0th-order entropy, but are slower in query performance. RLE+δ is especially space-efficient for low-entropy sequences. AAWT and Huffman-shaped Wavelet Trees have similar performance, with the latter more space efficient for low-entropy sequences. When used with RRR encoding, both offer good all-around space and query time performance. AAWT+None is generally the fastest in all the experiments for queries and construction time.

Many existing implementations construct the Wavelet Matrix [7], which is an alternative representation for the Wavelet Tree, dropping its structure and thus achieving faster performance in practice. It is especially effective for large alphabets.

At the first level of the Wavelet Matrix, the most significant bits (MSBs) of the symbols are stored, analogous to the first level of the Wavelet Tree. Then, to construct the next level ℓ, starting with the second, the text is stably sorted using the $(\ell-1)$th MSB as key. Just as with the Wavelet Tree, the symbols are represented using their ℓth MSB on each level ℓ. However, the order of the symbols on each level is given by the stably sorted text. This removes the tree structure of the Wavelet Tree. However, the same intervals as in the Wavelet Tree occur on each level, just in a bit-reversal permutation order [1]. The number of 0s in each level is stored in an array. This information is needed to answer queries using one less binary rank and/or select query per level compared to Wavelet Trees. In the following, we use Wavelet Tree to refer to both Wavelet Tree and Wavelet Matrix, and refer the interested reader to [7] for more details.

Practical Binary Rank and Select Data Structures

As we mentioned before, rank and select data structures with constant query time can be constructed in linear time requiring $o(n)$ extra space for a binary vector of size n [5, 20]. The currently most space-efficient rank and select support for a size-n binary array that contains m ones requires only $\log\binom{n}{m} + n/\log n + \tilde{\mathcal{O}}(n^{3/4})$ bits (including the bit vector) [30], which however is not of practical interest.

Almost all practical data structures to support rank queries follow the same layout. The main idea is to split the binary array into blocks of b bits each and superblocks of B blocks each. This way, every bit belongs to a block and every block belongs to a superblock. Then

[1] See `https://oeis.org/A030109`, last accessed 2025-02-24.

rank queries are supported by storing a counter for each superblock and block. For each superblock, we store the number of ones in the binary array up to the beginning of the superblock. Similarly, for each block, we store the number of ones from the beginning of its superblock to the beginning of the block. This hierarchy of counters is used to limit the number of bits to be used for each counter. The counter of a superblock is stored in $\Theta(\log n)$ bits, while the counter of a block is stored in $\Theta(\log b + \log B)$ bits.

A rank query at a certain position is computed by summing up the counter of the superblock and the counter of the block that contains the queried position, and the rank intra-block. The latter is computed with a popcount operation, which is supported by any modern CPU (see e.g. [9]).

Different superblock and block sizes are chosen by different implementations. For example, many implementations [32, 14] use superblocks of size $B = 8$ and blocks of size $b = 64$ bits. Wider block sizes $B = 4$ and $b = 512$ are chosen by the most recent implementations [34, 23] and computations intra-block with such a large block size are done with SIMD operations. These implementations have a smaller space overhead than the ones with smaller block sizes, but they require more time to answer rank queries. However, when the binary vector is large, the query time is dominated by the cost of accessing the block content from memory rather than the CPU time to perform the operations within the block. Thus, the disadvantage of having larger block sizes becomes almost negligible, and these solutions should be preferred because of their smaller space overhead.

The support for select queries is more involved. There exist two possible approaches: *rank-based* and *position-based*. Rank-based select structures rely on the rank data structure that is augmented by a lightweight index storing sampled positions for every kth occurrence of a 1 or 0. To answer a select query, the first step is to locate the nearest block using the sampled positions. Subsequently, consecutive blocks are examined until the basic block containing the target bit (with the specified rank) is found. The exact position within the basic block can then be determined directly. Although this method typically does not guarantee constant query time, it is efficient in practice. On the other hand, position-based select structures split the binary vector into blocks and sample positions of the block based on its density [32]. For example, if the block is sparse, we can store the answer of every select query directly. The main advantage of position-based select data structures is their constant worst-case query time. However, this comes at the cost of higher space usage.

Faster Wavelet Tree Implementations

When answering queries using a Wavelet Tree, the query is translated to $\Theta(\log \sigma)$ binary rank or select queries as described above. Most of the time to answer a query on the Wavelet Tree is spent answering these binary rank and select queries. Additionally, on each level of the Wavelet Tree, the binary rank and select queries will result in at least one cache miss, which again is the dominant cost of these binary queries. The currently fastest implementation [3] reduces the number of cache misses by reducing the number of levels. This is done by making use of a 4-ary Wavelet Tree. By doubling the number of children, it (roughly) halves the number of levels. A 4-ary Wavelet Tree represents the symbols on each level using two bits stored in a quad vector, i.e., a vector over the alphabet $\{0, 1, 2, 3\}$ with access, rank, and select support. The rank and select support for the quad vector is implemented using strategies similar to the ones of rank and select data structures for binary vectors.

The speed-up of this implementation over the other existing ones is approximately a factor 2 for all the queries. The rank queries can be further improved using a small prediction model designed to anticipate and pre-fetch the cache lines required for rank queries. This could give a further improvement up to a factor of 1.6 for rank query [3].

References

1 Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013. doi:10.1016/j.tcs.2013.10.019.

2 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm, 1994.

3 Matteo Ceregini, Florian Kurpicz, and Rossano Venturini. Faster wavelet tree queries. In *Data Compression Conference*, pages 223–232. IEEE, 2024. doi:10.1109/DCC58796.2024.00030.

4 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988. doi:10.1137/0217026.

5 David Clark and J. Ian Munro. Efficicent suffix trees on secondary storage. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, pages 383–391, New York, NY, 1996. Association for Computing Machinery.

6 Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE '08)*, pages 176–187, November 2008. doi:10.1007/978-3-540-89097-3_18.

7 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015. doi:10.1016/j.is.2014.06.002.

8 Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.1975.1055349.

9 Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.

10 Paolo Ferragina, Raffaele Giancarlo, Roberto Grossi, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter. Wavelet Tree, Part II: Text indexing. submitted to Festschrift's honoree Giovanni Manzini.

11 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009. doi:10.1016/j.ic.2008.12.010.

12 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):Article 20, 2007.

13 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. This work combined earlier versions from *SIAM Symposium on Discrete Algorithms (SODA)*, January 2004, and "Fast compression with a static model in high-order entropy," *Data Compression Conference (DCC)*, March 2004. doi:10.1145/1198513.1198521.

14 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

15 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *WEA*, pages 27–38, 2005.

16 Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 841–850, New York, NY, 2003. Association for Computing Machinery.

17 Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. An earlier version appeared in *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC'00)*, May 2000. doi:10.1137/S0097539702402354.

18 Roberto Grossi, Jeffrey S. Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *2011 First International Conference on Data Compression, Communications and Processing*, pages 210–221, 2011. doi:10.1109/CCP.2011.16.

19 P. G. Howard and Jeffrey S. Vitter. Arithmetic coding for data compression. *Proceedings of the IEEE*, 82(6):857–865, June 1994. doi:10.1109/5.286189.

20 Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS'89)*, pages 549–554, Washington, DC, 1989. IEEE Computer Society. doi:10.1109/SFCS.1989.63533.

21 Juha Kärkkäinen. *Repetition-based Text Indexixng*. Ph.d., University of Helsinki, Finland, 1999.

22 Donald E. Knuth. *The Art of Computer Programming*, volume Volumg 3: Sorting and Searching. Addison-Welsey, 2nd edition edition, 1998.

23 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. doi:10.1007/978-3-031-20643-6_19.

24 Veli Mäkinen and Gonzalo Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical Report Tech. Rep. C-2004-20, University of Helsinki, April 2004.

25 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE'07)*, pages 229–241, Heidelberg, Germany, 2007. Springer-Verlag Berlin. doi:10.1007/978-3-540-75530-2_21.

26 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. An earlier version appeared in *Proceedings of the 10th Symposium on Discrete Algorithms (SODA '99)*, January 1999, 669–677. doi:10.1145/382780.382782.

27 J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.

28 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.

29 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

30 Mihai Patrascu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

31 Rajeev Raman, Venkatesh Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007. doi:10.1145/1290672.1290680.

32 Sebastiano Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.

33 J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 193–204, Philadelphia, PA, June 1999. Awarded the 2009 SIGMOD Test of Time Award for the most impactful paper from the SIGMOD conference 10 years earlier.

34 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA*, volume 7933 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2013. doi:10.1007/978-3-642-38527-8_15.

Faster Range LCP Queries in Linear Space

Yakov Nekirch
Michigan Technological University, Houghton, MI, USA

Sharma V. Thankachan
North Carolina State University, Raleigh, NC, USA

Abstract

A range LCP query $\mathsf{rlcp}(\alpha, \beta)$ on a text $T[1..n]$ asks to return the length of the longest common prefix of any two suffixes of T with starting positions in a range $[\alpha, \beta]$. In this paper we describe a data structure that uses $O(n)$ space and supports range LCP queries in time $O(\log^{\varepsilon} n)$ for any constant $\varepsilon > 0$. Our result is the fastest currently known linear-space solution for this problem.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Data Structures, String Algorithms, Longest Common Prefix

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.16

Category Research

Funding *Yakov Nekirch*: Supported by the National Science Foundation under NSF grant 2203278.
Sharma V. Thankachan: Supported by the National Science Foundation under NSF grant 2315822.

Problem Definition and Previous Work

In this note we consider a variant of the longest common prefix (LCP) problem, called the *range LCP problem*. In this problem we store a text $T[1..n]$ in a data structure so that range LCP queries can be answered efficiently. A range LCP query $[\alpha, \beta]$ asks to return the length of the longest common prefix of any two suffixes with starting positions in a range $[\alpha, \beta]$,

$$\mathsf{rlcp}(\alpha, \beta) = \max\{\mathsf{LCP}(i, j) \mid i \neq j \text{ and } i, j \in [\alpha, \beta]\},$$

where $\mathsf{LCP}(i, j)$ denotes the length of the longest common prefix of $T[i..n]$ and $T[j..n]$.

This problem and its variants were considered in several papers, see e.g., [5, 9, 2, 3, 1, 10, 7]. The currently fastest data structure by Amir et al. [2] uses $O(n \log n)$ words of space and answers range LCP queries in $O(\log \log n)$ time. Henceforth we assume that a word of space consists of $\log n$ bits. The data structure with $O(n)$ space usage by Abedin et al. [1] supports queries in $O(\log^{1+\varepsilon} n)$ time for any constant $\varepsilon > 0$. The data structure of Matsuda et al. [10] uses $O(nH_0)$ bits of space where H_0 is the 0-order entropy of the text T; however this space usage is achieved at a cost of significantly higher query time as their data structure supports queries in time $O(n^{\varepsilon})$.

In this note we describe a new trade-off between the space usage and the query time: Our data structure uses linear space and supports queries in time $O(\log^{\varepsilon} n)$ for any constant $\varepsilon > 0$. Thus we achieve the same space usage as in [1] and query time that is close to [2]. Our solution combines the techniques from some previous papers with some new ideas. The compact data structure for predecessor queries by Grossi et al. [8] is also used in our construction.

Notation

We will say that a triple (i, j, h) is a *bridge* if $1 \leq i < j \leq n$ and $\mathsf{LCP}(i, j) = h$. The total number of bridges is $O(n^2)$. However in order to answer a range LCP query it is sufficient to consider a subset of all bridges of size $O(n \log n)$ [2, 1]. Following the method of Abedin et al. [1], we consider *special bridges* that are defined below.

We consider the suffix tree of the text T and divide its nodes into heavy and light nodes [11]. Let size(u) denote the number of leaves in the subtree rooted at u. The root is a light node. Exactly one child u' of every internal node u is designated as a heavy node, specifically one with the largest size(u') (ties are broken arbitrarily). All other nodes are light. Let ℓ_i and ℓ_j denote the leaves in the suffix tree that hold suffixes $T[i..n]$ and $T[j..n]$ respectively. Let u denote the lowest common ancestor of ℓ_i and ℓ_j and let u_i and u_j denote the children of u that are ancestors of ℓ_i and ℓ_j respectively. A bridge (i, j, h) is a special bridge if one of the following conditions is satisfied:

1. u_i is a light node **and** $j = \min\{x \mid (i, x, h) \text{ is a bridge}\}$
2. u_j is a light node **and** $i = \max\{x \mid (x, j, h) \text{ is a bridge}\}$

▶ **Lemma 1** ([1]). *There are $O(n \log n)$ special bridges. For any α and β such that $1 \leq \alpha \leq \beta \leq n$, we have $\mathsf{rlcp}(\alpha, \beta) = \max\{\, h \mid (i, j, h)$ is a special bridge and $\alpha \leq i, j \leq \beta \,\}$*

In the rest of this paper a bridge will denote a special bridge.

Bridge Classification

Let $\Delta = \log n$. For a bridge (i, j, h) we will say that i is its *left leg*, j is its *right leg*, and h is its *height*. We will say that a bridge (i, j, h) is in the interval $[a, b]$ if $a \leq i \leq j \leq b$. Let $\mathcal{B}_t$ denote the set of all bridges of height t.

By pigeonhole principle, there exists some value π, $1 \leq \pi \leq \Delta$, such that the total number of bridges in $\cup_{k \geq 0} \mathcal{B}_{\pi + k\Delta}$ is bounded by $O(\frac{n \log n}{\Delta}) = O(n)$; see also [1, Section 5.1] We will say that all bridges in $\mathcal{B}_1 \cup (\cup_{k \geq 0} \mathcal{B}_{\pi + k\Delta})$ are *base bridges*. All other bridges are *implicit* bridges. Data structures for different categories of bridges are described below.

Base bridges

The total number of base bridges is $O(n)$. Furthermore we can find the maximum height base bridge in a query range by answering a variant of an orthogonal range searching query. Our data structure for base bridges is summarized in the following lemma.

▶ **Lemma 2.** *There exists an $O(n)$-word data structure that finds, for any interval $[\alpha, \beta]$, the base bridge with maximal height in $[\alpha, \beta]$. The query time is $O(\log \log n)$.*

Proof. There is at most one bridge $(i, j, 1)$ for every value of i, $1 \leq i \leq n$ and the total number of bridges in $\mathcal{B}_1$ is $O(n)$. Hence the total number of base bridges is $O(n)$. In order to answer a query, we must find the largest h such that $\alpha \leq i \leq \beta$, $\alpha \leq j \leq \beta$, and there is a base bridge (i, j, h). Since $i \leq j$, this is equivalent to finding the triple (i, j, h) such that $i \geq \alpha$, $j \leq \beta$, and h is maximized. The latter query is equivalent to a two-dimensional dominance maxima query. Using a data structure for top-k dominance queries with $k = 1$, such a query can be answered in $O(\log \log n)$ time using $O(n)$ space, see [4, Theorem 7]. ◀

Implicit Bridges

Using Lemma 2, we can find the largest h_0 such that there is a base bridge of height h_0 in the query range $[\alpha, \beta]$. Now we explain how to find the largest h, $h_0 < h < h_0 + \Delta$, such that there is (an implicit) bridge of height h in $[\alpha, \beta - \Delta]$.

The following properties of special bridges will be used.

▶ **Lemma 3** ([1], Lemma 6). *If there is a special bridge (i, j, h), then for any $d < h$ there is a special bridge $(i + d, j' + d, h - d)$ for some $j' \leq j$ and a special bridge $(i' + d, j + d, h - d)$ for some $i' \geq i$.*

▶ **Lemma 4** ([1], Lemma 5). *There exists a data structure that uses $O(n)$ space and supports the following queries in $O(\log^{\varepsilon} n)$ time:*

- *find the right leg j of a special bridge (i, j, h) if its left leg i and its height h are known*
- *find the left leg i of a special bridge (i, j, h) if its right leg j and its height h are known*

Let H_0 denote the set of heights of base bridges. For every $h_0 \in H_0$ we consider all bridges of height h_0 and construct the list of their left legs sorted in increasing order $L(h_0) = \{ s_1, s_2, \ldots, s_{n_0} \}$, where n_0 is the number of special bridges with height h_0. For each k, $1 \leq k \leq \Delta$, we also construct an array $R_{h_0,k}[1..n_0]$ so that $R_{h_0,k}[i] = \min\{ j \mid \mathsf{LCP}(s_i - k, j) \geq h_0 + k \}$. Finally let

$$\mathrm{Min}(h_0, k, a) = \min\{R_{h_0,k}[i] \mid i_a \leq i \},$$

where s_{i_a} is the successor of $(a + k)$ in $L(h_0)$ and i_a is the position of s_{i_a} in $L(h_0)$.

▶ **Lemma 5.** *If $\mathrm{Min}(h_0, k, a) \leq b$, then there is a bridge of height $h_0 + k$ in $[a, b]$.*
If $\mathrm{Min}(h_0, k, a) > b$, then there is no bridge (i, j, h') in $[a, b - \Delta]$ such that $h' \geq h_0 + k$

Proof. The first statement directly follows from the definition of Min: if $\mathrm{Min}(h_0, k, a) \leq b$, then there is an index i_m, such that $s_{i_m} \geq a + k$ and a position j, such that $s_{i_m} - k < j \leq b$ and $\mathsf{LCP}(s_{i_m} - k, j) \geq h_0 + k$. Hence there is also a special bridge $(s_{i_m} - k, j', h_0 + k)$ where $j' \leq j \leq b$ and $a \leq s_{i_m} - k \leq b$.
To prove the second part of the lemma, suppose that there is a bridge (i, j, h') where $a \leq i \leq b - \Delta$, $a \leq j \leq b - \Delta$ and $h' = h_0 + k + f$ for some f, $0 \leq f \leq \Delta - k$. Then $\mathsf{LCP}(i, j) = h_0 + k + f$ and $\mathsf{LCP}(i + k + f, j + k + f) = h_0$. Hence there is a base bridge $(i + k + f, j'_0, h_0)$ for some $j'_0 \leq j + k + f$. Let t denote the position of $(i + k + f, j'_0, h_0)$ in $L(h_0)$. Furthermore $\mathsf{LCP}(i + f, j + f) = h_0 + k$ and there is an implicit bridge $(i + f, j'_1, h_0 + k)$ for some $j'_1 \leq j + f$. Since $j + f \leq b$, $R_{h_0,k}[t] \leq b$ and $\mathrm{Min}(h_0, k, a) \leq b$. ◀

▶ **Lemma 6.** *For any $h_0 \in H_0$, there exists a data structure that uses $O(n_0 \log n)$ bits and determines whether $\mathrm{Min}(h_0, k, a) \leq b$ in $O(\log^{\varepsilon} n)$ time for any $1 \leq a \leq b \leq n$ and for any $1 \leq k \leq \Delta$.*

Proof. For every k, $1 \leq k \leq \Delta$, we store a compact data structure that supports range minimum queries on $R_{h_0,k}$ in $O(1)$ time and uses $O(n_0)$ bits of space. We can use the data structure from [6] for this purpose. All compact range minima data structures use $O(n_0 \Delta) = O(n_0 \log n)$ bits. Arrays $R_{h_0,k}$ are not stored. Additionally we store $L(h_0)$ in a data structure that uses $O(n_0 \log n)$ bits and supports successor queries in $O(\log \log n)$ time [12].

To compute $\mathrm{Min}(h_0, k, a)$, we find the smallest $s \in L(h_0)$ such that $s \geq a + k$. Then we use the range minimum data structure and find the index i_m such that $i_m \geq s$ and $R_{h_0,k}[i_m] \leq R_{h_0,k}[i]$ for any $i \geq s$. Finally we compute the right leg j_m of the bridge $(i_m, j_m, h_0 + k)$ using Lemma 4. If $j_m > b$, then $\mathrm{Min}(h_0, k, a) > b$. If $j_m \leq b$, then $\mathrm{Min}(h_0, k, a) \leq b$. ◀

We keep a data structure from Lemma 6 for every $h_0 \in H_0$. The total space usage of these data structures is $O(n \log n)$ bits. Using Lemma 6 and binary search, we find the largest k such that $\mathrm{Min}(h_0, k, a) \leq b$. By Lemma 5, there is a bridge of height $h_0 + k$ in $[a, b]$; hence $\mathsf{LCP}(a, b) \geq h$. Also, by Lemma 5 there is no bridge (i, j, h) in $[a, b - \Delta]$, such that $h \geq h_0 + k + 1$. The total time is $O(\log^\varepsilon n \log\log n)$. We thus proved the following lemma.

▶ **Lemma 7.** *There exists an $O(n)$-word data structure that finds, for any interval $[\alpha, \beta]$, the implicit bridge with height h_i in $[\alpha, \beta]$, so that there is no bridge of height $h > h_i$ in $[\alpha, \beta - \Delta]$. The query time is $O(\log^\varepsilon n \log\log n)$.*

Block Bridges

It remains to consider the case of bridges with right leg in the interval $[\beta - \Delta, \beta]$ and of height h, $h_0 < h < h_0 + \Delta$. We apply the pigeonhole principle again. Let $\Delta_1 = \log\log n$. There exists some value π_1, $1 \leq \pi_1 \leq \Delta_1$, such that the total number of bridges in $\cup_{k \geq 0} \mathcal{B}_{\pi_1 + k\Delta_1}$ is bounded by $O(\frac{n \log n}{\Delta_1}) = O(n \frac{\log n}{\log\log n})$. Let $H_1 = \{\, \pi_1 + k\Delta_1 \mid 1 \leq k \leq (n - \pi_1)/\Delta_1 \,\}$. We will say that all bridges $\mathcal{B}_t$ where $t \in H_0 \cup H_1$ are *good bridges*. All other bridges are *bad* bridges.

We will denote by Block_{t,h_0} the set of all bridges (i, r, h) such that (a) $h \in H_0 \cup H_1$ and $h_0 \leq h < h_0 + \Delta$ for some $h_0 \in H$ and (b) $t\Delta + 1 \leq r \leq (t+1)\Delta$ for some k, $0 \leq k < \frac{n}{\Delta}$.

▶ **Lemma 8.** *Let m denote the number of bridges in Block_{t,h_0}. There exists a data structure that finds, for any interval $[\alpha, \beta]$, the bridge from Block_{t,h_0} with maximal height such that its right and left legs are in $[\alpha, \beta]$. This data structure uses $O(\log n + m \log\log n)$ bits and supports queries in $O(\log^\varepsilon n)$ time.*

Proof. Let I_{t,h_0} denote the set that contains left legs of all bridges in Block_{t,h_0}. Every bridge (i, j, h) from Block_{t,h_0} is represented as follows: We replace the right leg j with $d(j) = j - t\Delta$ and the height h with $d(h) = h - h_0$. We replace the left leg i with $r(i)$, where $r(i) = |\{\, x \leq i \mid x \in I_{t,h_0} \,\}|$ is the rank of i in I_{t,h_0}. Thus a bridge $(i, j, h) \in \mathrm{Block}_{t,h_0}$ is represented by a triple $(r(i), d(j), d(h))$. Since $r(i)$, $d(j)$, and $d(h)$ are bounded by Δ we can store each triple $(r(i), d(j), d(h))$ using $O(\log\log n)$ bits. For each $(r(i), d(j), d(h))$, we can retrieve the corresponding values of j and h with one addition. If j and h of some bridge (i, j, h) are known, we can obtain the value of its left leg i in $O(\log^\varepsilon n)$ time using the data structure from Lemma 4.

In addition, we store all elements of I_{t,h_0} in a compact data structure that is described by Grossi et al. in [8, Lemma 3.3]. This data structure supports successor queries on a set of integers S; provided that we can access an arbitrary element of S in time t_{acc}, a successor query can be answered in time $O(\log m / \log\log n + t_{\mathrm{acc}})$ where m is the number of elements in S. The data structure uses $O(\log\log u)$ bits per element, where u is the size of the universe (in addition to the space required to store S). In our case, I_{t,h_0} has $O(\Delta^2)$ elements and the size of the universe is n. Hence for every left leg $i \in I_{t,h_0}$, the data structure uses $O(\log\log n)$ bits. We can obtain the value of any left leg in time $O(\log^\varepsilon n)$. Hence successor queries are answered in $O((\log \Delta^2)/\log\log n + \log^\varepsilon n) = O(\log^\varepsilon n)$ time. That is, we can find for any α the smallest $i_\alpha \in I_{t,h_0}$ such that $i_\alpha \geq \alpha$.

Finally all triples $(r(i), d(j), d(h))$ are stored in the data structure described in [4, Theorem 7] [1]. This data structure uses $O(m \log m) = O(m \log \log n)$ bits and supports the following range maxima queries in $O(\log \log n)$ time: for any r_α and d_β, find the highest $d(h)$, among all tuples $(r(i), d(j), d(h))$ satisfying $r(i) \geq r_\alpha$ and $d(j) \geq d_\beta$.

In order to find the maximum-height bridge in $[\alpha, \beta]$ from Block_{t,h_0}, we find the successor of α and its rank $r(\alpha)$, using the compact successor data structure. We also compute $d(\beta) = \beta - t\Delta$. Then we find the highest value $d(h_{\max})$ among all $(r(i), d(j), d(h))$ satisfying $r(i) \geq r_\alpha$ and $d(j) \leq d_\beta$. The maximum height of a bridge in $[\alpha, \beta]$ is $h_{\max} = d(h_{\max}) + h_0$. The total time required to answer a query is $O(\log^\varepsilon n + \log \log n) = O(\log^\varepsilon n)$. ◀

▶ **Lemma 9.** *There are $O(n)$ non-empty blocks.*

Proof. Suppose that there is at least one implicit bridge (i, j, h) in Block_{t,h_0}. Let $k = h - h_0$. Then by Lemma 3 there is a special bridge $(i', j + k, h_0)$ such that $i + k \leq i' \leq j + k$. Since $1 \leq k < \Delta$ and $t\Delta < j \leq (t+1)\Delta$, we have $t\Delta < j + k \leq t + 2\Delta$. Thus for every base bridge (i', j', h_0) where $h_0 \in H_0$ there are at most two non-empty blocks. Since there are $O(n)$ base bridges, the number of non-empty blocks is $O(n)$. ◀

▶ **Lemma 10.** *There exists a data structure that finds, for any $\alpha \leq \beta$, and any $h_0 \in H_0$, the highest good bridge (i, j, h) in $[\alpha, \beta]$ such that its right leg j is in $[\beta - \Delta, \beta]$ and its height h is in $[h_0, h_0 + \Delta]$. The data structure uses $O(n)$ words of space and supports queries in $O(\log^\varepsilon n)$ time.*

Proof. We store a block data structure from Lemma 8 for each non-empty block Block_{t,h_0}. The total number of bridges in all blocks is equal to the total number of good bridges. By Lemma 9, the total number of non-empty blocks is $O(n)$. Hence the total space usage of all block data structures is $O(n \log n + (n \frac{\log n}{\log \log n}) \log \log n) = O(n \log n)$ bits. The range $[\beta - \Delta, \beta]$ intersects with at most two blocks. Hence we can find the highest bridge satisfying the conditions of this lemma in time $O(\log^\varepsilon n)$ by answering two queries to block data structures. ◀

Putting All Parts Together

In order to answer a range LCP query $[\alpha, \beta]$ we need to identify the largest h such that there is a bridge (i, j, h) in $[\alpha, \beta]$. Our algorithm works in four stages:

1. First, we find the largest h_0 such that there is a base bridge of height h_0 in $[\alpha, \beta]$. This step takes $O(\log \log n)$ time by Lemma 2.
2. Then we find the largest h_i, where $h_0 < h_i < h_0 + \Delta$, such that there is an implicit bridge of height h_i in $[\alpha, \beta - \Delta]$. This can be done in $O(\log^\varepsilon n \log \log n)$ time by Lemma 7
3. We find the largest h_n such that there is a good bridge of height $h_n > h_0$ with right leg in $[\beta - \Delta, \beta]$. This step takes $O(\log^\varepsilon n)$ time by Lemma 10.
4. Let $h_1 = \max(h_0, h_i, h_n)$. We check if there is a bridge of height h in $[\alpha, \beta]$ for each h, $h_1 < h < h_1 + \Delta_1$. By Lemma 5 we can check each candidate value of h in $O(\log^\varepsilon n)$ time. Hence this step takes $O(\log^\varepsilon \Delta_1) = O(\log^\varepsilon n \log \log n)$ time.

The total query time is $O(\log^\varepsilon n \log \log n)$. By replacing ε with a constant $\varepsilon' < \varepsilon$ in the above construction, we obtain our final result.

▶ **Theorem 11.** *There exists a data structure that uses $O(n)$ words of space and answers range LCP queries in time $O(\log^\varepsilon n)$ time.*

[1] The same data structure was also used in Lemma 2.

References

1. Paniz Abedin, Arnab Ganguly, Wing-Kai Hon, Kotaro Matsuda, Yakov Nekrich, Kunihiko Sadakane, Rahul Shah, and Sharma V. Thankachan. A linear-space data structure for range-lcp queries in poly-logarithmic time. *Theor. Comput. Sci.*, 822:15–22, 2020. doi:10.1016/J.TCS.2020.04.009.
2. Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi:10.1016/J.JCSS.2014.02.010.
3. Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP queries revisited. In *22nd International Symposium String Processing and Information Retrieval (SPIRE)*, pages 350–361, 2015. doi:10.1007/978-3-319-23826-5_33.
4. Timothy M. Chan, Yakov Nekrich, Saladi Rahul, and Konstantinos Tsakalidis. Orthogonal point location and rectangle stabbing queries in 3-d. *J. Comput. Geom.*, 13(1):399–428, 2022. doi:10.20382/JOCG.V13I1A15.
5. Graham Cormode and S. Muthukrishnan. Substring compression problems. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 321–330, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070478.
6. Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.
7. Arnab Ganguly, Manish Patil, Rahul Shah, and Sharma V. Thankachan. A linear space data structure for range LCP queries. *Fundam. Informaticae*, 163(3):245–251, 2018. doi:10.3233/FI-2018-1741.
8. Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 517–528, 2009. doi:10.4230/LIPICS.STACS.2009.1847.
9. Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.
10. Kotaro Matsuda, Kunihiko Sadakane, Tatiana Starikovskaya, and Masakazu Tateshita. Compressed orthogonal search on suffix arrays with applications to range LCP. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 23:1–23:13, 2020. doi:10.4230/LIPICS.CPM.2020.23.
11. Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.
12. Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

On Inverting the Burrows-Wheeler Transform

Nicola Cotumaccio
University of Helsinki, Finland

Abstract

We study the relationship between four fundamental problems: sorting, suffix sorting, element distinctness and BWT inversion. Our main contribution is an $\Omega(n \log n)$ lower bound for BWT inversion in the comparison model. As a corollary, we obtain a new proof of the classical $\Omega(n \log n)$ lower bound for sorting, which we believe to be of didactic interest for those who are not familiar with the Burrows-Wheeler transform.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Models of computation

Keywords and phrases Burrows-Wheeler transform, sorting, suffix array, element distinctness

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.17

Category Research

Funding Funded by the Helsinki Institute for Information Technology (HIIT).

1 Introduction

Sorting is typically one of the introductory topics in a first course on algorithms and data structures. Knuth devotes almost four hundred pages to the problem in the *The Art of Computer Programming* [12] and Cormen et al. use insertion sort as a first example of an algorithm [3].

Consider a sorted alphabet (i.e., an alphabet endowed with a total order). The problem of sorting can be stated as follows.

▶ **Problem 1** (Sorting). *Given a string $T = a_1 a_2 \dots a_n$, compute a permutation ψ of $\{1, 2, \dots, n\}$ such that $a_{\psi(i)} \leq a_{\psi(i+1)}$ for every $1 \leq i \leq n-1$.*

For example, if $T = cdba$, then the output of Problem 1 is the permutation ψ of $\{1, 2, 3, 4\}$ such that $\psi(1) = 4$, $\psi(2) = 3$, $\psi(3) = 1$ and $\psi(4) = 2$. Note that the permutation ψ of Problem 1 is uniquely defined if and only the a_i's are pairwise distinct. More generally, ψ becomes uniquely defined if we add the additional requirement that, for every $1 \leq i \leq n-1$, if $a_{\psi(i)} = a_{\psi(i+1)}$, then $\psi(i) < \psi(i+1)$. The permutation ψ that satisfies this additional requirement is the *stable sort* of T.

Both Knuth and Cormen at al.'s consider the *comparison model*, in which no restriction on the (possibly infinite) sorted alphabet is assumed, and the only way to obtain information on the mutual order between the characters in T is by solving queries $c(i, j)$ of the following type: given $1 \leq i, j \leq n$, decide whether $a_i \leq a_j$. We assume that each query $c(i, j)$ takes $O(1)$ time.

In the comparison model, the (worst-case) complexity of Problem 1 is $\Theta(n \log n)$: there exist algorithms (e.g., merge sort, which computes the stable sort of T) solving Problem 1 in $O(n \log n)$ time, and any algorithm solving Problem 1 has complexity $\Omega(n \log n)$. The (classical) proof of the $\Omega(n \log n)$ lower bound [12, 3] shows a stronger result: to solve Problem 1, in the worst case we need $\Omega(n \log n)$ queries $c(i, j)$'s, and this is true even if we know that the a_i's are pairwise distinct. In other words, $\Omega(n \log n)$ is not only an algorithmic lower bound, but it also captures the minimum number of operations required to have enough information for solving Problem 1.

The comparison model yields a simple, informative, and mathematically appealing setting for studying the complexity of sorting, but it can hardly be considered a realistic computational model. For example, if the a_i's are known to be integers in a polynomial range, Problem 1 can be solved in $O(n)$ time via radix sort, which also computes the stable sort of T [10].

The landscape becomes more complex if one considers alternative models of computations. On the one hand, it is possible to obtain models that are more flexible than the comparison model by allowing more complex queries in $O(1)$ time. Assume that the alphabet is the set of all real numbers, and consider the function $f(x, y) = x - y$. In the comparison model, in $O(1)$ time we can test whether $f(x, y) \leq 0$ for any choice of $x, y \in \{a_1, a_2, \ldots, a_n\}$. In the *linear decision tree model*, one is allowed more general tests of the type $f(x_1, x_2, \ldots, x_n) \lessgtr 0$, where we consider a linear function $f(x_1, x_2, \ldots, x_n) = c_0 + c_1x_1 + c_2x_2 + \cdots + c_nx_n$. In the *algebraic decision tree model*, we can choose $f(x_1, x_2, \ldots, x_n)$ to be an arbitrary polynomial. Even if the linear decision tree model and the algebraic decision tree model are more general than the comparison model, the complexity of Problem 1 in all these models is still $\Theta(n \log n)$ [4, 1]. On the other hand, it is possible to consider (realistic) variants of the RAM model. In the word-RAM model with word size $w \geq \log N$, where N is the size of the input, the complexity of Problem 1 is still open, even though it is known to be $o(n \log n)$ [8, 9]. Other variants of these models are possible, but we will focus only on the comparison model and the case of integers in a polynomial range, which are arguably the most common settings in the literature.

In this paper, we show that the complexity of Problem 1 (Sorting) is the same as the complexity of other fundamental problems: suffix sorting, element distinctness and BWT inversion. The complexity of these problems in $\Theta(n \log n)$ in the comparison model and $\Theta(n)$ in the case of integers in a polynomial range. Our main contribution is an $\Omega(n \log n)$ lower bound for BWT inversion in the comparison model (Theorem 1). We also show that Theorem 1 implies a new proof of the classical $\Omega(n \log n)$ lower bound for sorting (Corollary 2), which we believe to be of didactic interest for those who are not familiar with the Burrows-Wheeler transform.

2 Sorting, BWT Inversion and Related Problems

Problem 1 is closely related to several fundamental problems. Here are some examples.

▶ **Problem 2** (Suffix sorting). *Given a string $T = a_1a_2 \ldots a_n$, compute the permutation ψ of $\{1, 2, \ldots, n\}$ such that $a_{\psi(i)}a_{\psi(i)+1} \ldots a_{n-1}a_n$ is the i-th lexicographically smallest suffix of T for every $1 \leq i \leq n$.*

For example, if $T = banana$, then the output of Problem 2 is the permutation ψ of $\{1, 2, 3, 4, 5, 6\}$ such that $\psi(1) = 6$, $\psi(2) = 4$, $\psi(3) = 2$, $\psi(4) = 1$, $\psi(5) = 5$ and $\psi(6) = 3$. Note that ψ is uniquely defined because the suffixes of a string are pairwise distinct.

▶ **Problem 3** (Element distinctness). *Given a string $T = a_1a_2 \ldots a_n$, decide whether there exist $1 \leq i < j \leq n$ such that $a_i = a_j$.*

For example, if $T = cdba$, then the output of Problem 3 is "no".

The complexity of Problems 2 and 3 is $\Theta(n)$ in the case of integers in a polynomial range, and $\Theta(n \log n)$ in the comparison model. Let us show how to obtain these bounds by reducing these problems to Problem 1 (Sorting).

Let us consider the case of integers in a polynomial range.

- Problem 2 can be solved in $O(n)$ time by using any linear-time algorithm for building the suffix array (see [15] for a survey). Historically, the linear time complexity was first proved by Farach, who solved the more general problem of building the suffix tree of a string in linear time [5]. Conceptually, the easiest algorithm is probably Kärkkäinen et al.'s algorithm [11].
- Problem 3 can be solved in $O(n)$ time by first computing a permutation ψ as in Problem 1 (Sorting) and then checking in $O(n)$ time if there exists $1 \leq i \leq n-1$ such that $a_{\psi(i)} = a_{\psi(i+1)}$.

Let us consider the comparison model.

- On the one hand, Problem 2 can be solved in $O(n \log n)$ time by (i) sorting the a_i's via any $O(n \log n)$ comparison-based algorithm, (ii) replacing each a_i with its rank in the sorted list of all a_i's (which does not affect the mutual order of the suffixes) and (iii) applying any $O(n)$ suffix sorting algorithm mentioned earlier. On the other hand, to solve Problem 2 we need $\Omega(n \log n)$ queries $c(i,j)$'s in the worst case, and the same lower bound is true even if we know that the a_i's are pairwise distinct. Indeed, given the permutation ψ of Problem 2, we have $a_{\psi(i)} \leq a_{\psi(i+1)}$ for every $1 \leq i \leq n-1$ (because the suffix $a_{\psi(i)}a_{\psi(i)+1} \dots a_{n-1}a_n$ is lexicographically smaller than the suffix $a_{\psi(i+1)}a_{\psi(i+1)+1} \dots a_{n-1}a_n$), so ψ is also a correct output for Problem 1, and the conclusion follows from the lower bound for Problem 1 mentioned earlier.
- On the one hand, Problem 3 has complexity $O(n \log n)$: we can argue as in the case of integers in a polynomial range, but we use any $O(n \log n)$ comparison-based algorithm to compute a permutation ψ as in Problem 1. On the other hand, to solve Problem 3 we need $\Omega(n \log n)$ queries $c(i,j)$'s in the worst case, as we show next. Fix an integer n, and consider an algorithm that solves Problem 3 for every string $T = a_1 a_2 \dots a_n$. Let $f(n)$ be the number of queries $c(i,j)$'s solved by the algorithm in the worst case. If the a_i's are pairwise distinct, the permutation ψ of Problem 1 is uniquely defined, and the algorithm for Problem 3 must return "no". To this end, the algorithm must solve the query $c(\psi(i+1), \psi(i))$ for every $1 \leq i \leq n-1$ because otherwise the algorithm could not infer that $a_{\psi(i)} \neq a_{\psi(i+1)}$ from the other queries that it solves and so it would not have enough information to solve Problem 3 correctly on input T. Consequently, if for every query $c(i,j)$ solved by the algorithm we also consider the query $c(j,i)$, we obtain at most $2f(n)$ queries. In particular, we consider both the query $c(\psi(i+1), \psi(i))$ and the query $c(\psi(i), \psi(i+1))$ for every $1 \leq i \leq n-1$, from which we can infer that $a_{\psi(i)} < a_{\psi(i+1)}$ for every $1 \leq i \leq n-1$. We conclude that at most $2f(n)$ queries $c(i,j)$'s are sufficient in the worst case to compute ψ and so solve Problem 1 for every $T = a_1 a_2 \dots a_n$ in which the a_i's are pairwise distinct. Hence, we obtain $f(n) = \Omega(n \log n)$ from the lower bound for Problem 1 mentioned earlier.

Let us show that Problem 1 is related to another fundamental problem in string processing and compression. To this end, we need to introduce the *Burrows-Wheeler transform (BWT)* of a string [2]. Let $S = b_1 b_2 \dots b_n$ be a string such that $b_n = \$$, where \$ is a special character such that (i) \$ does not occur anywhere else in the string and (ii) \$ is smaller than all the other characters. For example, we can consider the string $S = banana\$$ (where $n = 7$). For $1 \leq i \leq n$, let $S_i = b_i b_{i+1} \dots b_n b_1 b_2 \dots b_{i-1}$ be the i-th *circular suffix* of S. For example, if $S = banana\$$, we have $S_1 = banana\$$, $S_2 = anana\$b$, $S_3 = nana\$ba$, $S_4 = ana\$ban$, $S_5 = na\$bana$, $S_6 = a\$banan$ and $S_7 = \$banana$. Note that the circular suffixes of S are pairwise distinct because \$ occurs in a different position in each of them.

Table 1 The Burrows-Wheeler transform of $banana\$$.

i	$F[i]$						$L[i]$	ψ	A
1	\$	b	a	n	a	n	a	5	7
2	a	\$	b	a	n	a	n	1	6
3	a	n	a	\$	b	a	n	6	4
4	a	n	a	n	a	\$	b	7	2
5	b	a	n	a	n	a	\$	4	1
6	n	a	\$	b	a	n	a	2	5
7	n	a	n	a	\$	b	a	3	3

We build the square matrix M of size $n \times n$ such that, for every $1 \leq i \leq n$, the i-th row $R_i[1, n]$ is equal to i-th (lexicograhically) smallest circular suffix of S (see Table 1 for the matrix M of size 7×7 obtained from $S = banana\$$). Note that $R_1 = b_n b_1 b_2 \dots b_{n-1} = S_n$ because $b_n = \$$ is the smallest character.

For $1 \leq i \leq n$, let $C_i[1, n]$ be the i-th column of M from left to right. Notice that every C_i is a rearrangement of the characters in S. Let $F = C_1$ and $L = C_n$ be the first column and the last column of M, respectively. In Table 1, we have $F = \$aaabnn$ and $L = annb\$aa$. Note that F can be obtained by sorting the characters of S. By definition, the Burrows-Wheeler transform $\mathsf{BWT}[S]$ of S is the column F, that is, $\mathsf{BWT}[S] = F = C_n$. In Table 1, we have $\mathsf{BWT}[S] = annb\$aa$.

Crucially, the Burrows-Wheeler transform $\mathsf{BWT}[S]$ of a string S is an *encoding* of the string: given $\mathsf{BWT}[S]$, we can retrieve the string S. In Table 1, given $\mathsf{BWT}[S] = annb\$aa$, we can retrieve $S = banana\$$. To prove this, we only need to show that from $\mathsf{BWT}[S]$ we can retrieve the matrix M, because then the unique row of the matrix ending with \$ yields the original string S. We can retrieve M by computing all columns C_i's. We know that $C_n = \mathsf{BWT}(S)$, and we can retrieve C_1 by sorting all characters in C_n. Let us show how to retrieve C_2. We know C_n and C_1, so we know all pairs of consecutive characters in S. If we sort these pairs lexicographically and we pick the last element of each pair, we retrieve C_2. In Table 1, all pairs of consecutive characters in S are $a\$$, na, na, ba, $\$b$, an, an. By sorting these pairs, we obtain $\$b$, $a\$$, an, an, ba, na, na, and by picking the last element of each pair, we can infer that $C_2 = b\$nnaaa$. Let us show how to retrieve C_3. We know C_n, C_1 and C_2, so we know all triples of consecutive characters in S. If we sort these triples lexicographically and we pick the last element of each triple, we retrieve C_3. In Table 1, all triples of consecutive characters in S are $a\$b$, $na\$$, nan, ban, $\$ba$, ana, ana. By sorting these triples, we obtain $\$ba$, $a\$b$, ana, ana, ban, $na\$$, nan, and by picking the last element of each triple, we can infer that $C_3 = abaan\$n$. In the same way, we can retrieve $C_4, C_5, \dots, C_{n-1}$.

Since $\mathsf{BWT}(S)$ is an encoding of S, we can store $\mathsf{BWT}(S)$ instead of S without losing information. The reason why storing $\mathsf{BWT}(S)$ may be more beneficial than storing S is that the string $\mathsf{BWT}(S)$ tends to be repetitive if in S several substrings have multiple occurrences, so we can exploit the repetitiveness of S to *compress* $\mathsf{BWT}(S)$. This property motivated the introduction of the Burrows-Wheeler transform in the original paper [2] and can be stated in precise mathematical terms via the notion of *entropy* [13]. Surprisingly, it is also possible to solve pattern matching *on the original string* by augmenting the compressed representation of the Burrows-Wheeler transform with space-efficient data structures, thus obtaining the *FM-index* [6].

We described an (inefficient) algorithm to *invert* the Burrows-Wheeler transform. Let us state the problem formally.

▶ **Problem 4** (BWT inversion). *Given a string $T = a_1a_2 \dots a_n$ such that $T = \mathsf{BWT}(S)$ for some (\$-terminated) string S, compute S.*

For example, if $T = annb\$aa$, then $S = banana\$$ (see Table 1).

3 Our Results

In this section, we show that the complexity of Problem 4 in $\Theta(n \log n)$ in the comparison model and $\Theta(n)$ in the case of integers in a polynomial range. The only bound that cannot be inferred from the original paper on the Burrows-Wheeler transform [2] is the $\Omega(n \log n)$ lower bound in the comparison model. Notice that the four problems considered in this paper have the same complexity.

To prove the $\Omega(n \log n)$ bound for Problem 4, we will proceed differently from Problems 2 and 3. We will prove the lower bound directly, without relying on the lower bound for Problem 1 (Sorting). Then, we will use the lower bound for Problem 4 (BWT inversion) to infer the lower bound for Problem 1 (Sorting). In addition to establishing interesting relationships between fundamental problems, our approach yields a new proof of the celebrated $\Omega(n \log n)$ bound for sorting, which we believe to be of didactic interest.

Let us start with the lower bound for BWT inversion.

▶ **Theorem 1.** *In the comparison model, to solve Problem 4 we need $\Omega(n \log n)$ queries $c(i, j)$'s in the worst case. The same lower bound holds even if we know that the a_i's are pairwise distinct.*

Proof. Fix an integer n. Consider $n - 1$ distinct characters $b_1, b_2, \dots, b_{n-1}$ such that $\$ < b_1 < b_2 < \dots < b_{n-1}$. Then, the set:

$$\mathcal{S} = \{b_{\phi(1)}b_{\phi(2)} \dots b_{\phi(n-1)}\$ \mid \phi \text{ is a permutation of } \{1, 2, \dots, n-1\}\}$$

has size $(n - 1)!$. For every $S \in \mathcal{S}$, the string $\mathsf{BWT}(S)$ is an encoding of S, so the set:

$$\mathcal{T} = \{\mathsf{BWT}(S) \mid S \in \mathcal{S}\}$$

has also size $(n - 1)!$. Notice that for every string $T = a_1a_2 \dots a_n$, if $T \in \mathcal{T}$, then the a_i's are pairwise distinct.

Consider *any* algorithm solving Problem 4 for every input $T = a_1a_2 \dots a_n \in \mathcal{T}$. The algorithm can gather information on the mutual order between the a_i's only by solving queries $c(i, j)$'s. The decision on the next query $c(i, j)$ can depend on the outcome of the previous queries $c(i, j)$'s, so we can describe the behavior of the algorithm on all inputs $T \in \mathcal{T}$ through a *decision tree* (see Figure 1 for the case $n = 4$). Since the algorithm correctly solves Problem 4 for every input $T = a_1a_2 \dots a_n \in \mathcal{T}$, then the outcome of all queries $c(i, j)$'s on a path from the leaf to a roof cannot be consistent with two distinct elements of $\mathcal{T}$, otherwise the algorithm would not have enough information to compute S . For example, in Figure 1, if the output of "$a_1 \leq a_3$" is "no" and then the output "$a_1 \leq a_2$" is "yes", then the algorithm must necessarily solve an additional query $c(i, j)$: both $T_1 = b_1b_3\$b_2$ and $T_2 = b_2b_3b_1\$$ are strings in $\mathcal{T}$ for which $\neg(a_1 \leq a_3) \wedge (a_1 \leq a_2)$, and we have $T_1 = \mathsf{BWT}(S_1)$ and $T_2 = \mathsf{BWT}(S_2)$ for two distinct $S_1, S_2 \in \mathcal{S}$, where $S_1 = b_2b_3b_1\$$ and $S_2 = b_3b_1b_2\$$.

Assume that the longest path in the tree consists of k edges. Then, the number of paths from the root to a leaf is upper bounded by 2^k, so we must have $2^k \geq (n - 1)!$, which implies $k = \Omega(n \log n)$. This means that there exists $T \in \mathcal{T}$ for which the algorithm needs to solve $\Omega(n \log n)$ queries $c(i, j)$'s. ◀

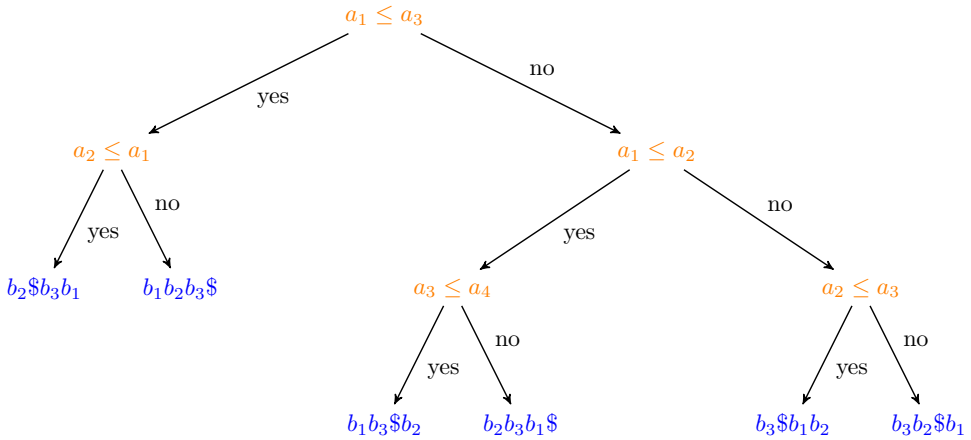


Figure 1 The decision tree of a possible algorithm solving Problem 4 (see the proof of Theorem 1) for $n = 4$. We have $|\mathcal{S}| = |\mathcal{T}| = (4-1)! = 6$. The tree describes the sequence of all queries $c(i,j)$'s for every input $T = a_1a_2a_3a_4 \in \mathcal{T}$. Recall that we assume $\$ < b_1 < b_2 < b_3$. We have $\mathsf{BWT}(b_1b_2b_3\$) = b_3\$b_1b_2$, $\mathsf{BWT}(b_1b_3b_2\$) = b_2\$b_3b_1$, $\mathsf{BWT}(b_2b_1b_3\$) = b_3b_2\$b_1$, $\mathsf{BWT}(b_2b_3b_1\$) = b_1b_3\$b_2$, $\mathsf{BWT}(b_3b_1b_2\$) = b_2b_3b_1\$$ and $\mathsf{BWT}(b_3b_2b_1\$) = b_1b_2b_3\$$. Every element of $\mathcal{T}$ corresponds to a path from the root to a leaf.

The proof of Theorem 1 is similar to the classical proof of the $\Omega(n \log n)$ lower bound for sorting [12, 3]. In the classical proof, one typically uses the fact that a binary tree with $n!$ leaves must have height at least $\log(n!)$. Here we used a slightly more direct pigeonhole argument to prove the inequality $2^k \ge (n-1)!$. The *worst-case entropy* of a set S is $\log|S|$ by a similar pigeonhole argument [14], so one may argue that in the proof of Theorem 1 we used an entropy-based argument. This appears to establish an interesting analogy because we have already mentioned that the compressibility of the Burrows-Wheeler transform can be described through the notion of entropy [13].

Now, let us describe an efficient algorithm to solve Problem 4. The original paper by Burrows and Wheeler [2] follows an approach based on a permutation called *LF-mapping*. Here we will use a different approach based on the *permutation* ψ, which is the inverse of the LF-mapping. The permutation ψ plays a crucial role in compressed suffix arrays [7], and it captures the connection between Problem 4 (BWT inversion) and Problem 1 (Sorting) more explicitly. We are given $T = a_1a_2\ldots a_n$ such that $T = \mathsf{BWT}(S)$ for some $S = b_1b_1\ldots b_n$, where $b_n = \$$, and we need to compute S. By definition, $T = L$, where L is the last column of the matrix M, so we know that $L = a_1a_2\ldots a_n$ and we must retrieve S.

Let ψ be the stable sort of L (see Problem 1 and Table 1). Since \$ is the smallest character, we have $L[\psi(1)] = \$ = b_n$. We will prove that $b_i = L[\psi^{i+1}(1)]$ for every $1 \le i \le n-1$. For example, in Table 1 we have $b_1 = L[\psi^2(1)] = L[4] = b$, $b_2 = L[\psi^3(1)] = L[7] = a$, $b_3 = L[\psi^4(1)] = L[3] = n$, $b_4 = L[\psi^5(1)] = L[6] = a$, $b_5 = L[\psi^6(1)] = L[2] = n$ and $b_6 = L[\psi^7(1)] = L[1] = a$, so $S = banana\$$. After computing ψ, we can retrieve all the b_i's in $O(n)$ time by computing all the powers $\psi^{i+1}(1)$'s. In the comparison model, we can compute ψ in $O(n \log n)$ time, and in the case of integers in a polynomial range we can compute ψ in $O(n)$ time, so the complexity of Problem 4 is $O(n \log n)$ in the comparison model and $O(n)$ in the case of integers in a polynomial range. We are left with proving that $b_i = L[\psi^{i+1}(1)]$ for every $1 \le i \le n-1$.

Let $A[1,n]$ be the array such that, for every $1 \leq i \leq n$, the row R_i of the matrix M is equal to the circular suffix $S_{A[i]}$ (see Table 1). Then, $A[1,n]$ yields a permutation of $\{1,2,\dots,n\}$. Moreover, we have $F[i] = b_{A[i]}$ and $L[i] = b_{A[i]-1}$ for every $1 \leq i \leq n$, where we assume $b_0 = b_n$. In particular, $b_{A[\psi(1)]-1} = L[\psi(1)] = \$$, so we have $A[\psi(1)] = 1$ and $2 \leq A[\psi(i)] \leq n$ for $2 \leq i \leq n$. Notice that A yields a permutation of $\{1,2,\dots,n\}$. From $R_1 = S_n$ we obtain $A[1] = n$.

Let us prove that $A[\psi(i)] = A[i] + 1$ for every $2 \leq i \leq n$ (for example, in Table 1 we have $A[\psi(2)] = A[1] = 7 = 6 + 1 = A[2] + 1$). Fix a character $c \neq \$$ that occurs in S. Let $1 \leq i_1 \leq n$ be the smallest integer such that $F[i_1] = c$, and let $1 \leq i_2 \leq n$ be the largest integer such that $F[i_2] = c$. We only have to prove that $A[\psi(i)] = A[i] + 1$ for every $i_1 \leq i \leq i_2$, because by picking all possible values $c \neq \$$ from smallest to largest we cover every i between 2 and n ($i = 1$ corresponds to \$). Let us prove that $A[\psi(i)] = A[i] + 1$ for every $i_1 \leq i \leq i_2$. From the definitions of i_1 and i_2 we obtain that in every row of M and in every column of M there are exactly $i_2 - i_1 + 1$ characters equal to c, and in particular $L[\psi(i_1)] = L[\psi(i_1 + 1)] = \cdots = L[\psi(i_2 - 1)] = L[\psi(i_2)] = c$. Since ψ is the stable sort of L, we obtain $\psi(i_1) < \psi(i_1 + 1) < \cdots < \psi(i_2 - 1) < \psi(i_2)$. This implies that $S_{A[\psi(i_1)]}$ is lexicographically smaller than $S_{A[\psi(i_1+1)]}$, $S_{A[\psi(i_1+1)]}$ is lexicographically smaller than $S_{A[\psi(i_1+2)]}$, …, $S_{A[\psi(i_2-1)]}$ is lexicographically smaller than $S_{A[\psi(i_2)]}$. We have $b_{A[\psi(i)]-1} = L[\psi(i)] = c$ for every $i_1 \leq i \leq i_2$, so we conclude that $S_{A[\psi(i_1)]-1}$ is lexicographically smaller than $S_{A[\psi(i_1+1)]-1}$, $S_{A[\psi(i_1+1)]-1}$ is lexicographically smaller than $S_{A[\psi(i_1+2)]-1}$, …, $S_{A[\psi(i_2-1)]-1}$ is lexicographically smaller than $S_{A[\psi(i_2)]-1}$, where $b_{A[\psi(i)]-1} = c$ for every $i_1 \leq i \leq i_2$. At the same time, for every $1 \leq i \leq n$ we have $b_{A[i]} = c$ if and only if $i_1 \leq i \leq i_2$, and $S_{A[i_1]}$ is lexicographically smaller than $S_{A[i_1+1]}$, $S_{A[i_1+1]}$ is lexicographically smaller than $S_{A[i_1+2]}$, …, $S_{A[i_2-1]}$ is lexicographically smaller than $S_{A[i_2]}$. We obtain $A[\psi(i)] - 1 = A[i]$ for every $i_1 \leq i \leq i_2$, so $A[\psi(i)] = A[i] + 1$ for every $i_1 \leq i \leq i_2$, as claimed.

Let us prove that $A[\psi^i(1)] = i$ for every $1 \leq i \leq n$ (for example, in Table 1 we have $A[\psi^2(1)] = A[\psi(5)] = A[4] = 2$). We proceed by induction on i. For $i = 1$, we know that $A[\psi(1)] = 1$. Now assume that $2 \leq i \leq n$. By the inductive hypothesis, we know that $A[\psi^{i-1}(1)] = i - 1$. In particular, $A[\psi^{i-1}(1)] \neq n$, so $2 \leq \psi^{i-1}(1) \leq n$ and we obtain $A[\psi^i(1)] = A[\psi(\psi^{i-1}(1))] = A[\psi^{i-1}(1)] + 1 = (i-1) + 1 = i$.

We are now ready to prove the main claim. We have $b_i = b_{(i+1)-1} = b_{A[\psi^{i+1}(1)]-1} = L[\psi^{i+1}(1)]$ for every $1 \leq i \leq n-1$.

We conclude our paper by showing that Theorem 1 implies a new proof of the lower bound for sorting.

▶ **Corollary 2.** *In the comparison model, to solve Problem 1 we need $\Omega(n \log n)$ queries $c(i,j)$'s in the worst case. The same lower bound holds even if we know that the a_i's are pairwise distinct.*

Proof. Consider *any* algorithm solving Problem 4 for every input $T = a_1 a_2 \dots a_n$ such that the a_i's are pairwise distinct. Since the a_i's are pairwise distinct, the permutation ψ of Problem 1 is uniquely defined, and ψ is also the stable sort of T. Let $f(n)$ be the number of queries $c(i,j)$'s solved by the algorithm in the worst case to compute ψ. Assume that $T = \mathsf{BWT}(S)$, where $S = b_1 b_2 \dots b_n$. After computing ψ, we can compute S in $O(n)$ time (because $b_i = T[\psi^{i+1}(1)]$ for every $1 \leq i \leq n-1$, as we have seen before) without solving any additional query $c(i,j)$. Consequently, $f(n)$ queries are sufficient in the worst case to solve Problem 4 for every input $T = a_1 a_2 \dots a_n$ such that the a_i's are pairwise distinct. By Theorem 1, we conclude $f(n) = \Omega(n \log n)$. ◀

4 Conclusions

In this paper, we have shown that, in the comparison model, inverting the Burrows-Wheeler transform has complexity $\Theta(n \log n)$. As a corollary, we have obtained a new proof of the $\Omega(n \log n)$ sorting lower bound. Our main goal was to highlight how the ideas behind the Burrows-Wheeler transform are deeply intertwined with the most fundamental results in Computer Science.

References

1 Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.

2 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, 1994.

3 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

4 David P Dobkin and Richard J Lipton. On the complexity of computations under varying sets of primitives. *Journal of Computer and System Sciences*, 18(1):86–91, 1979. doi:10.1016/0022-0000(79)90054-0.

5 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997. doi:10.1109/SFCS.1997.646102.

6 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000. doi:10.1109/SFCS.2000.892127.

7 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the thirty-second Annual ACM Symposium on Theory of Computing*, pages 397–406, 2000.

8 Torben Hagerup. Sorting and searching on the word RAM. In *STACS 98: 15th Annual Symposium on Theoretical Aspects of Computer Science Paris, France, February 25–27, 1998 Proceedings 15*, pages 366–398. Springer, 1998. doi:10.1007/BFB0028575.

9 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004. doi:10.1016/J.JALGOR.2003.09.001.

10 John E Hopcroft, Jeffrey D Ullman, and Alfred Vaino Aho. *Data structures and algorithms*, volume 175. Addison-wesley Boston, MA, USA:, 1983.

11 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

12 Donald E Knuth. *The Art of Computer Programming: Sorting and Searching, volume 3*. Addison-Wesley Professional, 1998.

13 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM (JACM)*, 48(3):407–430, 2001. doi:10.1145/382780.382782.

14 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

15 Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4–es, 2007.

DNA Is a Puzzle Enthusiast

Roberto Marangoni
Department of Biology, University of Pisa, Italy

Abstract

This article presents a concise summary of research projects in which Roberto Grossi participated, yielding interesting results that were never previously published in research papers. At the time, these studies were deemed too limited and in need of further extensions and generalizations, which were never realized due to a lack of resources. The researches focused on methods for inferring possible three-dimensional DNA conformations based on nucleotide sequence characteristics. Specifically, two key approaches were investigated: the identification of structured motifs for detecting Transcription Factor Binding Sites (TFBS) and the study of nested permutations using PQ-trees. This article describes the obtained results in selected case studies, their potential implications, and the current state of the art in these research areas.

2012 ACM Subject Classification Applied computing → Bioinformatics; Theory of computation → Design and analysis of algorithms

Keywords and phrases DNA, 3D structure, PQ trees, structural motifs

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.18

Category Research

1 Introduction

From a biochemical perspective, DNA is a polymeric macromolecule composed of four fundamental nucleotides: Adenine, Cytosine, Guanine, and Thymine. The genetic information encoded in DNA is determined by the specific sequence of these nucleotides. A well-known characteristic of DNA is its bilinear structure, consisting of two antiparallel strands. The chemical bonds linking the nucleotides are directional, giving each strand a specific orientation. Complementary base pairing follows strict rules: Adenine pairs with Thymine, while Cytosine pairs with Guanine.

The structural organization of DNA was first described in 1953 through the "double helix" model proposed by Watson and Crick, based on Rosalind Franklin's X-ray diffraction data. This model, referred to as the "B-form double helix," represents one of several possible DNA conformations. In fact, in living cells, DNA can assume more complex three-dimensional conformations, including single-, double-, triple-, and even quadruple-stranded regions [19]. Additionally, local complex structures, such as single-hairpin loops, cruciform DNA (double-hairpin structures), and other intricate conformations, have been observed [20].

Double-helix conformations result from the planar pairing of complementary bases between the two strands. More complex structures emerge from non-trivial base pairings within the same strand or between different strands. For example, if two sequences on the same strand are mutually inverted complemented, their pairing may generate cruciform (double-hairpin) structure (see Figure 1 for a schematic representation).

The study of DNA's three-dimensional structures is crucial because, while the primary sequence encodes genetic information, its expression is mediated by proteins such as polymerases, transcription factors, and gene inhibitors and others. These proteins recognize specific DNA conformations rather than nucleotide sequences themselves. Molecular interactions, in fact, rely on "tactile" recognition, requiring precise surface contact to explicate biological functions. For example, a transcription factor may specifically recognize a hairpin structure with a defined size and folding pattern, and such a 3D structure is generated by a specific paring pattern of the nucleotides.

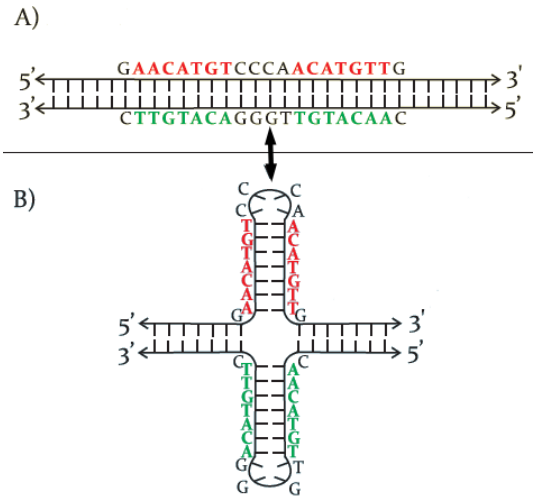


Figure 1 An example of a DNA region containing two sequences that are mutually inverted complemented. At room temperature, this region can oscillate between a linear conformation (A) and a double-hairpin (cruciform) conformation (B). Modified from [12].

In bioinformatics applications, DNA is often represented as a string in the alphabet {A, T, C, G}, corresponding to the primary sequence of one strand, from which the complementary strand can be inferred relying on pairing rules. This representation is primarily used to characterize the informational content of DNA, but it is also indirectly linked to possible local geometries, given that these geometries arise from non-trivial base pairings within the same strand or between different strands.

Bioinformatics research has frequently focused on comparing DNA sequences, identifying similarities, character substitutions, insertions, deletions, and even more significant modifications at the chromosomal level. Various algorithms have been proposed (and continue to be developed) for comparing genomic sequences to detect differences between healthy individuals and those affected by specific diseases, reconstruct the phylogenetic history of evolutionarily related species, and support numerous other applications that have become common with the rise of omics sciences.

Attempts to investigate possible local structures based on the arrangement of letters within DNA sequences have been relatively more limited. Identifying repetitions, palindromes, inversions, and complementations with translocations, for example, provides a means to infer structures with probable functional significance. Roberto Grossi and Nadia Pisanti have made a significant contribution to this type of investigation, along with their collaborators. In particular, two biologically important directions of research have emerged: the identification of structured motifs [15, 10, 9, 4, 16, 17] and the representation of DNA's primary sequence using PQ-trees, an efficient data structure that can identify sequences derived from each another through character permutations [3, 5, 7, 11].

2 Structured motifs in the DNA

A case study of structural investigation by motifs search concerned the heat-shock genes activation in *Tetrahymena termophila*. Heat shock genes are a class of genes that are usually activated when a thermal stress might damage the structure of cellular proteins. This class of genes includes functionally different molecular tools (mainly chaperonins: i.e. proteins

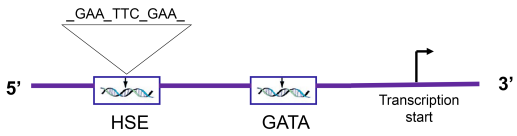
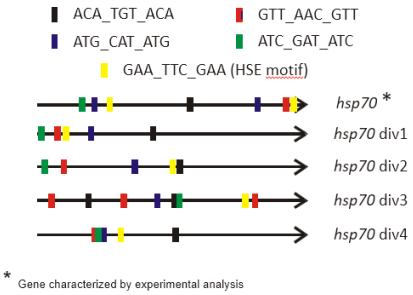


Figure 2 Schematic representation of the *T. thermophila* hsp70 promoter region including among others, the HSE and GATA regulatory motifs involved in the hsp70 gene activation as shown by experimental analysis. The structure of HSE sequence is reported above the corresponding box (underscore character = an interval ranging from 2 up to 8 of any nucleotides).

that help other proteins to assume the correct folding), usually named after the molecular weight of their protein product. For example, the *hsp70* genes refer to the heat shock gene subclass, the protein products of which weight about 70 kD. *T. thermophila* genome presents several (about 30) hsp70 genes, called isoforms, which are very similar each other, but not perfectly identical, and they may have a different regulation mechanism. The firstly studied copy, named hsp70_1 appears to be regulated through a complex mechanism involving two seemingly unrelated sequences [1]. The first regulatory element is a simple GATA tetra-nucleotide, while the second, named HSE (heat-shock enhancer), exhibiting a relatively more intricate structure. The HSE is composed of three blocks: the first block contains the GAA three-nucleotide, separated from the second block by an interval of two arbitrary nucleotides. The second block features the TTC three-nucleotide, which is the inverted complement of the first block. Following an interval of approximately eight arbitrary nucleotides, the third block contains again the same three-nucleotide of the first block (see Figure 2 for a schematic representation). This arrangement allows the second block to pair with both the first and third blocks, leading to the formation of two distinct double-hairpin structures: one in which the first and second blocks pair within the same strand, and another in which the pairing is established between the second and third block [1]. The HSE sequence can oscillate between three distinct conformations: (i) a linear conformation with no intra-strand base pairing, and (ii) two alternative conformations featuring double-hairpin structures. The proposed mechanism for gene expression regulation involves structural stabilization: the transcription factor is likely to bind to one of the double-hairpin conformations, thereby stabilizing the proximal DNA structure and facilitating RNA polymerase access to the operator site. The functional role of the HSE sequence has been experimentally validated [1]: site-directed mutagenesis studies have demonstrated a significant reduction in HSP70 gene expression upon disruption of the HSE motif. A key biological question arising from this discovery was whether other hsp70 gene copies exhibited analogous regulatory sequences. To address this, we developed a variant of the SMILE algorithm, named BioMotif [2], specifically designed for the identification and statistical assessment of structured sequences based on grammatical properties of the DNA sequence itself (an evolution of SMILE algorithm has been published thereafter with the name RISOTTO [15]). We systematically searched for sequences located within 500 nucleotides upstream of gene start sites, structured into three distinct boxes, each three nucleotides long, with the central box containing an inverted complement of the sequence found in both the first and third boxes. In essence, the objective was to identify sequences capable of forming double-hairpin structures within close genomic proximity, a key structural characteristic identified in the experimentally studied sequence. The search yielded statistically significant results, with identified sequences located near the transcription start sites of each hsp70 isoform. The Figure 3 illustrates the localization of all detected sequences. Notably, the experimentally validated sequence was also identified, despite not

Motif	Score
ACA_TGT_ACA	1.01
ATG_CAT_ATG	0.70
GTT_AAC_GTT	0.70
ATC_GAT_ATC	0.55
TGA_TCA_TGA	0.44
CTA_TAG_CTA	0.38
TAG_CTA_TAG	0.34
TTG_CAA_TTG	0.25
CAA_TTG_CAA	0.22
AGA_TCT_AGA	0.22
TCT_AGA_TCT	0.21
GAA_TTC_GAA	0.12
TTC_GAA_TTC	0.11
CTT_AAG_CTT	0.10

(a) Top significant motifs (the experimentally found one is highlighted).



(b) Localization of top-score significant motifs on the upstream region of various hsp70 isoforms.

Figure 3 Significant motifs found by BioMotif.

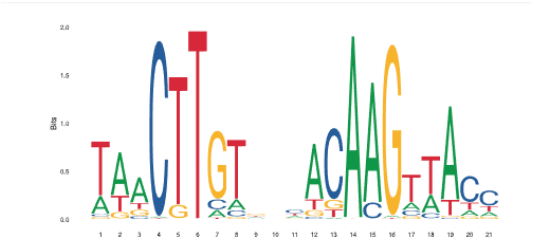


Figure 4 Picture of the consensus matrix MA2009.2 in *A. thaliana*; its whole Jaspar entry can be accessed at: `https://jaspar.elixir.no/matrix/MA2009.2/`.

being explicitly provided to the search algorithm. This finding strengthens the hypothesis that these sequences may also play functional roles, potentially acting as regulatory elements for the expression of hsp70 isoforms.

At this stage, the investigation was handed over to experimental biologists for site-directed mutagenesis studies to assess the involvement of additional candidate regulatory sequences. However, the termination of the research project and the disbanding of the experimental team indefinitely postponed this experimental validation, which is the primary reason why these findings were never published.

Nonetheless, we emphasize that this type of investigation, though relatively underutilized in biological research, can provide far more informative and effective insights than usual consensus sequence searches, which rely on conservation of bases rather than structural and functional properties. The largest (and regularly updated: the last version is v.10, 2024) database of TFBS (Transcription Factor Binding Sites), Jaspar ([8], [18]), still encode the TFBS regions as consensus matrices. Jaspar does not include *T. thermophila* in its species' collection, therefore it is not possible to use it for validating BioMotif output. We anyhow performed a search of them, to investigate their presence in other organisms. We firstly transformed the most significant BioMotif results into consensus matrices following two simple criteria: a) conserved nucleotides get 100% of the consensus column, the other bases get 0; b) in "don't care" positions, each nucleotide gets 25%. We assign a length of 3 to each "don't care" spacer between conserved sequences. These matrices have been assigned to Jaspar for searching.

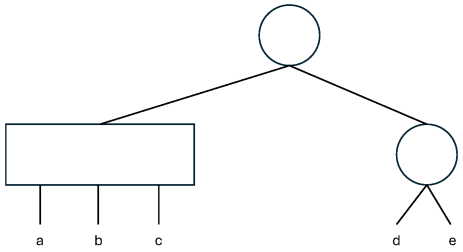


Figure 5 Example of a PQ tree, that, in an alphabet of five letters A={ a, b, c, d, e}, describes any of the following strings S={ abcde, abced, cbade, cbaed, edabc, deabc, edcba, edabc.

We have found no exact results in any taxonomic category, even when we searched for the experimentally discovered HSE motif, but this is not surprising, given that the Jaspar database does not contain *T. thermopila*, nor any evolutionary affine organism. But the search found several consensus sequences containing sub-strings of BioMotif results. For example, the search for the motif "CTT_AAG_CTT" (score = 0.10), returns a result in *Arabidopsis thaliana*, the consensus sequence of which is displayed in Figure 4. This is a TFBS of a NAC-gene family, involved in stress-responses [13], then with analogous function of hsp70 genes. The first two boxes of the query sequence, "CTT_AAG", are clearly present in it, but looking more in the detail, we find that the middle position of the first box shows the "T" nucleotide with about 80% of conservation, while the other letter is an "C", with about 20%. A symmetric situation takes place for the second position in the second box: 80% "A" and 20% "G". Jaspar does not give more details in summarized results, but this situation is compatible with the existence of 80% "CTT_AAG", and 20% "CGT_ACG": it is noticeable that both sequences are configuring the same structural motif: the second box is the complemented inverted of the first, thus probably giving rise to the same 3D conformation. In other words, it is reasonable to think that the 3D structure has been more preserved than the sequence itself: describing TFBS as structured motifs instead of consensus sequences could make this situation more explicit, with a significant improvement of the biological knowledge.

3 Searching for permutations using PQ trees

A further generalization beyond structured motifs consists of considering all possible permutations contained within, and often nested in, a DNA sequence. We regard this as a generalization because we do not predefine either a consensus sequence or a structured motif but instead investigate potential internal organizations within the sequence.

Permutations occurring at variable distances may give rise to a wide variety of 3D DNA conformations, which are probably not experimentally described but likely have functional significance. The primary data structure used in this research was the PQ tree, a notation introduced in the 1970s to represent maximal permutative structures within a sequence [6]. A PQ tree possesses two kind of nodes: type-P nodes (graphically represented by a circle), whose children can be permutated in any order, and type-Q nodes (graphically represented by a rectangle), whose chilren can occur only in the given order or in the reverse one, but not in other orders: Figure 5 presents an example of a PQ tree. A PQ tree is a natural way to represent a sequence containing substrings which are permutated each other, a very common situation for genomic DNA sequences. The application of

Table 1 Distribution of PQ tree heights and their frequency.

PQ tree height	# trees
1	18018
2	13583
3	10897
4	3412
5	3513
6	514
7	281
8	8
9	5
10	3
11	5
12	1
13	1
14	0
15	1
16	0

this approach to biological sequences was the subject of two master's theses, for which Roberto Grossi was among the advisors [7, 11]. From a computational perspective, these theses focused on the problem of efficiently generating the PQ tree representation of a given input sequence, minimizing both execution time and memory usage. The research led to the development of algorithms and procedures capable of extracting PQ trees in nearly linear time. Biological applications were implemented as simple test cases of the procedure. However, they yielded promising results that, unfortunately, were never further developed. These preliminary analyses were conducted on the gene encoding the principal glutamate receptor in *Rattus norvegicus*, called mGluR1, and accessible at ENA Nucleotide database at: `https://www.ebi.ac.uk/ena/browser/view/M61099`. The study did not aim to characterizing the generated PQ trees (they are too many, accounting for a huge number of permutations) but rather to examine the distribution of their height. The height of a PQ tree is, in practice, proportional by the degree of nesting of the permuted sequences, i.e. the number of times by which permutations of long regions contain, within them, permutations of shorter regions. One more nesting level increases the height of the relative PQ tree by 1; for example, the PQ tree in Figure 5 has height = 2. The Table 1 reports the number of PQ trees generated for each height. There are no trees with height > 16, but there are several trees with height > 7, thus showing a high-degree of nested permutations. This situation probably has functional significance, as a random shuffling of the sequence leads to a very different distribution, with a maximum height = 6 and not 15. In other words, the high level of permutation nesting is not random, but reasonably it is linked to some local arrangement of the DNA conformation. Most of the studies on PQ-trees and their potential applications in computational molecular biology were published between 2005 and 2012. In more recent years, only a few works have been published, with those exploring applications in biology being particularly scarce and almost exclusively focused on comparative genomics. Even recently, a PQ-tree structure has been proposed to address the problem of identifying and comparing gene clusters in bacterial genomes of strains/species closely related to already annotated genomes [14]. However, searches in SCOPUS bibliographic database have not retrieved any publication that investigates the height of PQ-trees to test its potential association to functional properties of genomic sequences.

4 Conclusions

We can conclude that there is an intriguing and still poorly investigated link between the various arrangements in the letters composing the DNA, when it is represented as a string, and the local conformation assumed by the DNA as a biological macromolecule. The presented approaches, structural motifs search and PQ-trees representation, are still valid and promising even though they have been proposed around 15-20 years ago. It is clear that DNA loves to play word games: it is surely a puzzle enthusiast and any approach that is able to establish a link between a peculiar arrangement of its letters and a biological function can produce very useful biological insights.

References

1 Sabrina Barchetta, Antonietta La Terza, Patrizia Ballarini, Sandra Pucciarelli, and Cristina Miceli. Combination of two regulatory elements in the tetrahymena thermophila HSP70-1 gene controls heat shock activation. *Eukaryotic cell*, 7(2):379–386, 2008.

2 Alessandro Bartolomei. BioMotif: un metodo per la ricerca di motivi altamente strutturati in sequenze genomiche. Master's thesis, University of Pisa, IT, June 2007. Available at `https://etd.adm.unipi.it/theses/available/etd-09252007-092605/unrestricted/Tesi.pdf`.

3 Giovanni Battaglia. *Discovery of unconventional patterns for sequence analysis: theory and algorithms*. Phd thesis, University of Pisa, Italy, June 2011. Available at `https://tesidottorato.depositolegale.it/handle/20.500.14242/128506`. URL: `https://etd.adm.unipi.it/theses/available/etd-12052011-215104/`.

4 Giovanni Battaglia, Davide Cangelosi, Roberto Grossi, and Nadia Pisanti. Masking patterns in sequences: A new class of motif discovery with don't cares. *Theoretical Computer Science*, 410(43):4327–4340, 2009. `doi:10.1016/J.TCS.2009.07.014`.

5 Giovanni Battaglia, Roberto Grossi, and Noemi Scutella. Consecutive ones property and PQ-trees for multisets: Hardness of counting their orderings. *Information and Computation*, 219:58–70, 2012. `doi:10.1016/J.IC.2012.08.005`.

6 Kellogg S Booth and George S Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of computer and system sciences*, 13(3):335–379, 1976. `doi:10.1016/S0022-0000(76)80045-1`.

7 Giuseppe Camposeo. Scoperta di pattern ripetuti mediante l'uso di alberi pq. Master's thesis, University of Pisa, IT, June 2007. Available at `https://etd.adm.unipi.it/theses/available/etd-03132008-103339/unrestricted/Tesi.pdf`.

8 Jaime A Castro-Mondragon, Rafael Riudavets-Puig, Ieva Rauluseviciute, Roza Berhanu Lemma, Laura Turchi, Romain Blanc-Mathieu, Jeremy Lucas, Paul Boddie, Aziz Khan, Nicolás Manosalva Pérez, et al. JASPAR 2022: the 9th release of the open-access database of transcription factor binding profiles. *Nucleic acids research*, 50(D1):D165–D173, 2022.

9 Roberto Grossi, Giulia Menconi, Nadia Pisanti, Roberto Trani, and Søren Vind. Motif trie: An efficient text index for pattern discovery with don't cares. *Theoretical Computer Science*, 710:74–87, 2018. `doi:10.1016/J.TCS.2017.04.012`.

10 Roberto Grossi, Andrea Pietracaprina, Nadia Pisanti, Geppino Pucci, Eli Upfal, and Fabio Vandin. MADMX: A strategy for maximal dense motif extraction. *Journal of Computational Biology*, 18(4):535–545, 2011. `doi:10.1089/CMB.2010.0177`.

11 Rosario Lombardo. Algoritmi efficienti per la scoperta di pattern ripetuti a intervalli. Master's thesis, University of Pisa, IT, June 2008. Available at `https://etd.adm.unipi.it/theses/available/etd-06182008-085952/unrestricted/Lombardo_2008__Laurea_Magistrale_in_Informatica.pdf`.

12 SG Lushnikov, AV Dmitriev, Alexander Ivanovich Fedoseev, Gennady Aleksandrovich Zakharov, AV Zhuravlev, Anna Vladimirovna Medvedeva, BF Schegolev, and EV Savvateeva-Popova. Low-frequency dynamics of DNA in Brillouin light scattering spectra. *JETP letters*, 98:735–741, 2014.

13 Hisako Ooka, Kouji Satoh, Koji Doi, Toshifumi Nagata, Yasuhiro Otomo, Kazuo Murakami, Kenichi Matsubara, Naoki Osato, Jun Kawai, Piero Carninci, et al. Comprehensive analysis of NAC family genes in oryza sativa and arabidopsis thaliana. *DNA research*, 10(6):239–247, 2003.

14 Eden Ozeri, Meirav Zehavi, and Michal Ziv-Ukelson. New algorithms for structure informed genome rearrangement. *Algorithms for Molecular Biology*, 18(1):17, 2023. doi:10.1186/S13015-023-00239-X.

15 Nadia Pisanti, Alexandra M Carvalho, Laurent Marsan, and Marie-France Sagot. RISOTTO: fast extraction of motifs with mismatches. In *LATIN 2006: Theoretical Informatics: 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006. Proceedings 7*, pages 757–768. Springer, 2006. doi:10.1007/11682462_69.

16 Nadia Pisanti, Maxime Crochemore, Roberto Grossi, and M F Sagot. A basis of tiling motifs for generating repeated patterns and its complexity for higher quorum. In *Mathematical Foundations of Computer Science 2003: 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003. Proceedings 28*, pages 622–631. Springer, 2003. doi:10.1007/978-3-540-45138-9_56.

17 Nadia Pisanti, Maxime Crochemore, Roberto Grossi, and Marie-France Sagot. Bases of motifs for generating repeated patterns with wild cards. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(1):40–50, 2005. doi:10.1109/TCBB.2005.5.

18 Ieva Rauluseviciute, Rafael Riudavets-Puig, Romain Blanc-Mathieu, Jaime A Castro-Mondragon, Katalin Ferenc, Vipin Kumar, Roza Berhanu Lemma, Jérémy Lucas, Jeanne Chèneby, Damir Baranasic, et al. JASPAR 2024: 20th anniversary of the open-access database of transcription factor binding profiles. *Nucleic acids research*, 52(D1):D174–D182, 2024.

19 Alexander Rich. DNA comes in many forms. *Gene*, 135(1-2):99–109, 1993.

20 Andrew Travers and Georgi Muskhelishvili. DNA structure and function. *The FEBS journal*, 282(12):2279–2295, 2015.

Designing Output Sensitive Algorithms for Subgraph Enumeration

Alessio Conte
University of Pisa, Italy

Kazuhiro Kurita
Nagoya University, Japan

Andrea Marino
University of Florence, Italy

Giulia Punzi
University of Pisa, Italy

Takeaki Uno
National Institute of Informatics, Tokyo, Japan

Kunihiro Wasa
Hosei University, Tokyo, Japan

Abstract

The enumeration of all subgraphs respecting some structural property is a fundamental task in theoretical computer science, with practical applications in many branches of data mining and network analysis. It is often of interest to only consider solutions (subgraphs) that are maximal under inclusion, and to achieve output-sensitive complexity, i.e., bounding the running time with respect to the number of subgraphs produced. In this paper, we provide a survey of techniques for designing output-sensitive algorithms for subgraph enumeration, including partition-based approaches such as flashlight search, solution-graph traversal methods such as reverse search, and cost amortization strategies such as push-out amortization. We also briefly discuss classes of efficiency, hardness of enumeration, and variants such as approximate enumeration. The paper is meant as an accessible handbook for learning the basics of the field and as a practical reference for selecting state-of-the-art subgraph enumeration strategies fitting to one's needs.

2012 ACM Subject Classification Mathematics of computing → Graph enumeration; Mathematics of computing → Graph algorithms

Keywords and phrases Graph algorithms, Graph enumeration, Output sensitive enumeration

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.19

Category Research

Funding *Alessio Conte*: Partially supported by MUR PRIN 2022 project EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data (#2022TS4Y3N).
Kazuhiro Kurita: Partially supported by Kakenhi JP23K24806, JP25K00136, JP25K21273, and JP25K03080.
Andrea Marino: Partially supported by Italian PNRR CN4 Centro Nazionale per la Mobilità Sostenibile, NextGeneration EU – CUP, B13C22001000001. MUR of Italy, under PRIN Project n. 2022ME9Z78 – NextGRAAL: Next-generation algorithms for constrained GRAph visuALization, PRIN PNRR Project n. P2022NZPJA – DLT-FRUIT: A user centered framework for facilitating DLTs FRUITion.
Giulia Punzi: Supported by the Italian Ministry of Research, under the complementary actions to the NRRP "Fit4MedRob – Fit for Medical Robotics" Grant (#PNC0000007).

1 Introduction

The goal of enumeration is to systematically list all feasible solutions to a given problem. Unlike optimization problems, which seek to find a single best solution according to an objective function – i.e., an extreme case – enumeration problems aim to identify all solutions that satisfy given constraints, representing local extreme cases.

A well-constructed enumeration model strikes a balance between the size and the number of solutions. When solutions are large, it is preferable to have fewer solutions. To achieve this, models often incorporate parameters such as solution size, frequency, and weight, or they unify similar solutions to manage complexity. Also, in order to limit the number of solutions, we are often interested in listing only minimal or maximal solutions, i.e., solutions that cannot be reduced or enlarged. It is important to note that the number of solutions grows with the size of the input. When dealing with small input sizes, brute-force algorithms and simple implementations can effectively solve the problem. However, for large-scale data, more sophisticated algorithmic techniques are required to ensure computation time increases in a controlled manner as the input size grows.

The development of algorithms for enumerating all possible solutions to a specific combinatorial problem has a long history, dating back at least to the 1960s. During this period, researchers began tackling the problem of enumerating specific graph-theoretic structures, such as shortest paths and cycles. As noted by David Eppstein [27], enumeration problems have numerous applications, including:

1. Identifying structures that satisfy additional constraints that are difficult to optimize,
2. Evaluating the quality of a model for a given problem by assessing the number of incorrect structures,
3. Analyzing the sensitivity of structures to variations in problem parameters,
4. Exploring a broader class of structures beyond just the optimal ones to gain deeper insight into the problem.

Over the past fifty years, a wide range of enumeration problems have been studied in the literature, spanning various domains such as geometry, graph and hypergraph theory, order and permutation problems, logic, set theory, and string problems. A recent compendium compiled by part of the authors of this paper includes more than 500 combinatorial problems along with more than 300 references, highlighting the depth and breadth of research in this field.

1.1 Our Contribution

Despite significant progress, the field of enumeration algorithms remains highly active, with many intriguing open problems still under investigation. This paper seeks to contribute to this ongoing research by first providing an overview of the key computational challenges in designing and analyzing enumeration algorithms, and then presenting some of the most effective techniques for developing efficient enumeration methods.

In particular, this paper serves as an introductory survey for newcomers interested in the design of efficient output-sensitive enumeration algorithms. At the same time, it may also serve as a valuable reference for experienced researchers, offering insights into state-of-the-art techniques and pointers to relevant literature on well-established concepts in the field.

1.2 Structure of the paper and roadmap

In Section 2, we discuss the general algorithmic challenges of enumeration and briefly examine brute-force approaches. While these methods do not require sophisticated algorithmic design, they provide the basis for understanding enumeration tasks, independently of efficiency considerations. On this latter note, Section 3 presents the various ways efficiency is defined in the context of listing algorithms.

Section 4 focuses on partition-based approaches for enumeration. We fist introduce a backtracking technique based on recursively partitioning the set of solutions. Depending on how duplication is managed, this approach can either be straightforward, or leverage forbidden sets. To improve efficiency, we present enhancements using the so-called *flashlight* method and the *extension problem*. As a byproduct of these techniques, we present their implications for *assessment*. Section 5 introduces strategies that intuitively navigate the space of solutions, often modeled as a *solution graph*. These include reverse search, the input-restricted problem approach, and proximity search.

In Section 6, we introduce amortization techniques, which are frequently used to refine partition-based methods and achieve (sub)linear average delay. We cover several such techniques, including amortization by children and grandchildren, push-out amortization, and geometric amortization.

Finally, in Section 7, we discuss approaches for determining the inherent difficulty of an enumeration problem. These arguments rely on the complexity of enumerating hypergraph transversals or employ suitable reductions from NP-hard problems. We also highlight here parameterized and approximate enumeration as promising directions for future research.

1.3 Preliminaries

This paper focuses on enumeration problems on graphs, so let us first give some preliminaries on the topic. A graph is a pair $G = (V(G), E(G))$, where $V(G)$ is its set of nodes or vertices, and $E(G) \subseteq V(G) \times V(G)$ is the set of edges. When the graph G is clear from the context, we just write $V = V(G)$ and $E = E(V)$. For an edge $(u, v) \in E(G)$, we refer to nodes u and v as its *endpoints*. A *self-loop* is an edge with equal endpoints. When multiple edges are allowed with the same endpoints, we call the graph a *multigraph*, and we call *multiplicity* of an edge the number of times it occurs.

We say that H is a *subgraph* of G, and we write $H \subseteq G$, if both $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. A subgraph H of G is said to be *induced* by the set of nodes $V(H)$, and denoted by $G[V(H)]$ if $E(H)$ is formed by all edges of G that connect nodes of $V(H)$: $E(H) = E(G) \cap (V(H) \times V(H))$. Given a node $v \in V(G)$, we denote with $G \setminus v$ the subgraph $G[V \setminus \{v\}]$.

There are two main types of graphs: undirected or directed. A graph is *undirected* if its edges are represented as *non-ordered pairs*. In other words, two edges are equal if they have the same endpoints, i.e., $(u, v) = (v, u)$. In this case, the *degree* of a node u is the number of edges incident to u, $d(u) = |\{(u, v) \in E\}|$, and its *neighbors* are the set of nodes connected to it with an edge: $N(u) = \{v \in V(G) | (u, v) \in E(G)\}$. A graph is instead said to be *directed* if the edges of E are ordered pairs, that is, edge (u, v) is different from edge (v, u). In this case, we say that edge (u, v) is directed *from* node u *to* node v. Given a node $u \in V(G)$, we define its *out-neighbors* as $N^+_G(u) = \{v \in V(G) \mid (u, v) \in E(G)\}$, and symmetrically its *in-neighbors* as $N^-_G(u) = \{v \in V(G) \mid (v, u) \in E(G)\}$. We have now two forms of degree for a node u: the *outdegree* $d^+_G(u)$ is the number of out-neighbors, while the *indegree* $d^-_G(u)$ is the number of in-neighbors. We note that, in the case of multigraphs (both directed and undirected), the degrees also account for the multiplicity of the incident edges.

A *clique* K of an undirected graph G is a subgraph such that every pair of distinct nodes of K is connected by an edge. A *matching* of a graph is a set of (non-loop) edges that share no endpoints.

Paths and Cycles

A sequence of distinct adjacent nodes of G is called a *path*: $P = u_1 u_2 \cdots u_k$ is a path if $u_i \in V$ for all $i = 1, ..., k$, where $u_i \neq u_j$ for any $i \neq j$, and $(u_i, u_{i+1}) \in E$ for all $i = 1, ..., k-1$. Note that this definition is independent of whether G is directed or not. In this case we say that P *traverses* nodes $u_1, ..., u_k$, has endpoints u_1 and u_k, and has *length* $|P| = k$. A path with endpoints u and v is sometimes called an *uv-path*. Given two nodes $s, t \in V$, the set of all *st*-paths of G is denoted by $\mathcal{P}_{s,t}(G)$, where $\mathcal{P}_{t,t}(G)$ only contains the trivial path from t to itself. A *cycle* is a path P whose endpoints coincide: in the previous notation, we have $u_1 = u_k$. A graph with no cycles is called *acyclic*; if it also is directed, it is referred to as a directed acyclic graph, or *DAG*.

Connectivity

We give here some connectivity notions which slightly differ according to whether the graph is directed or not. An undirected graph G is called *connected* if there is a path connecting every pair of nodes. A graph is *biconnected* if for any $u \in V$ the graph $G \setminus u$ is still connected. On the contrary, an *articulation point* is a node u such that $G \setminus u$ is *not* connected. The maximal biconnected subgraphs of G are called its *biconnected components* (BCCs). Analogously, a graph is *2-edge-connected* if $G \setminus e$ is connected for each $e \in E$, and edges whose removal disconnect the graph are called *bridges*. For directed graphs, we have two main notions of connectivity. A directed graph G is *weakly connected* if the *underlying undirected graph*, that is, the graph obtained by replacing all directed edges with undirected ones, is connected. Graph G is *strongly connected* if for each pair $u, v \in V$ there are both a *uv*-path and a *vu*-path.

Trees

A *tree* is a connected acyclic undirected graph. A *spanning tree* of a graph G is a subgraph T of G such that (i) T is a tree and (*ii*) $V(T) = V(G)$. Except for a single vertex, called *root* and denoted as r, every node u of the tree has a unique *parent*, defined as the only neighbor of u on the *ru*-path. The neighbors of a node that are not its parent are called its *children*. A node with no children is called a *leaf*. For a node u, we define its *rooted subtree* as the subgraph formed by node u, and all nodes (and edges) that it can reach without traversing its parent.

2 Algorithmic Challenges and Brute Force Approaches

The design of enumeration algorithms involves several crucial aspects that must be considered to ensure both correctness and efficiency. Specifically, an enumeration algorithm must guarantee that each solution is output exactly once, thereby avoiding duplication. A straightforward approach to achieve this is to store all previously found solutions in memory and, upon encountering a new solution, check whether it has already been output. However, this approach can be highly memory-inefficient when solutions are relatively large wrt the available memory. Addressing this issue would require dynamic memory allocation mechanisms and efficient search structures, such as hash functions. Consequently, a more practical strategy employed by many enumeration algorithms is to determine whether a solution has already been output without explicitly storing all generated solutions.

Algorithm 1 BruteForce(i, X). The notion of *valid* and *feasible* depends on the problem at hand. For instance, if all the valid sequences to be enumerated are the ones without repetition having prefix X and formed by even numbers in a set N, feasible may refer to the even numbers in N not in X.

Input: An integer $i \geq 1$, a sequence of values $X = \langle x_0, \ldots, x_{i-1} \rangle$, eventually empty
Output: All the valid sequences of length n whose prefix is X

```
1 if no solution includes X then return;
2 if i ≥ n then
3 |  if X is a solution then output X;
4 else
5 |  foreach feasible value e of x_i do
6 |  |  BruteForce(i + 1, ⟨X, e⟩)
```

Algorithm 2 BruteForce(X, D).

Input: A set X, a reference to a global database D
Output: All the distinct solutions containing X not contained in D

```
1 D ← D ∪ {X}
2 if no solution includes X then return;
3 if X is a solution then output X;
4 foreach X' obtained by adding an element to X do
5 |  if ∄Z ∈ D such that Z = X' then
6 |  |  BruteForce(X', D)
```

In addition to direct duplication, implicit forms of redundancy should also be avoided – for instance, ensuring that isomorphic solutions are not redundantly output. To achieve this, it is often beneficial to define a canonical representation for solutions, enabling efficient comparisons. An ideal canonical form establishes a one-to-one mapping between objects and their representations without significantly increasing their size. This transformation allows the enumeration of certain objects to be reframed as the enumeration of their canonical forms. However, for structures such as graphs, sequence data, and matrices, determining isomorphism remains computationally challenging, even when using canonical forms. Nonetheless, in such cases, isomorphism can still be checked using exponential-time algorithms, which, in practice, often perform efficiently when the number of solutions is relatively small.

Simple structures, such as cliques and paths, are generally easy to enumerate. Cliques can be constructed by iteratively adding vertices, while the set of paths can be easily partitioned. However, more complex structures – such as maximal structures (where no additional element can be added without violating a required property), minimal structures (where no element can be removed without violating a required property), or constrained structures – are significantly more challenging to enumerate. In these cases, even if finding a single solution is feasible in polynomial time, the main challenge lies in devising a method to generate additional solutions from a given one, i.e., *defining a solution neighborhood*, to enable traversal through all solutions by iteratively exploring these neighborhoods.

Using an exponential-time approach to identify each neighboring solution or having an exponential number of neighbors can lead to inefficiency. When a solution requires exponentially many modifications to generate potential neighbors, the enumeration process may take exponential time per solution, as there is no guarantee that any given modification

will yield a valid solution. This issue frequently arises in maximal solution enumeration: iteratively removing and adding elements to obtain maximality can allow traversal between solutions, but when the number of such modifications is exponential, the computational cost per solution also becomes exponential. In such cases, reducing the number of neighbors or employing pruning strategies to eliminate redundant computations can significantly improve efficiency.

Even more complex scenarios involve problems where finding a single solution is NP-complete, such as SAT or the Hamiltonian cycle problem. Despite this, heuristics often prove effective, particularly when the problem is *usually easy* (as in SAT), the *solutions are not excessively large* (as in maximal and minimal structure enumeration), and the *size of the solution space is bounded*.

For small instance sizes, brute-force algorithms can be a viable approach. These methods include using a divide-and-conquer strategy to generate candidate solutions and selecting feasible ones, or iteratively constructing solutions while removing isomorphic duplicates. Two fundamental brute-force algorithmic schemas are outlined in Algorithms 1 and 2. In Algorithm 1, each solution is represented as an ordered sequence of values. By invoking BruteForce$(1,\emptyset)$, feasible values are recursively determined by extending the current solution, requiring only a check to determine whether X constitutes a valid solution. Conversely, Algorithm 2 also attempts to extend the current solution but incorporates a check at each step to determine whether the solution has been previously encountered, storing results in a database D.

For both algorithms, it is crucial to establish a method for transforming a candidate solution X into another candidate X'. Additionally, in both cases, performing an accurate preliminary check to determine whether X belongs to any feasible solution set can prevent unnecessary computations, thereby enhancing efficiency.

3 Classes of Efficiency

For many enumeration problems, the number of solutions is often exponential in the size of the input instance, which inherently leads to enumeration algorithms requiring at least exponential time. In other words, for enumeration algorithms, the *worst case overall time*, which bounds the total time to end with a function depending on the input size, is often exponential.[1]

However, when the number of solutions is polynomially bounded, it is natural to seek a polynomial-time algorithm. Hence, the most natural approach to enhance the analysis of enumeration algorithms is to assess how the total computation time for generating all solutions correlates with both the input size and the output size. Algorithms analyzed in this manner are commonly referred to as output-sensitive, in contrast to input-sensitive algorithms [32]. In this context, the complexity classes of enumeration problems defined in this section are based on the number of solutions, ensuring that if the solution set is small, an efficient algorithm terminates in polynomial time, while larger solution spaces permit longer runtimes [37]. For more formal definitions, we refer the reader to the recent survey in [60]. We adopt the same notation as [60], noting that the same notions have been referred to by the literature with different names.

[1] These types of bounds can be found by applying *measure & conquer* approaches [31], but a discussion of these is outside the scope of this paper.

We are interested only in problems whose number of solutions is finite and which are polynomially balanced, i.e., the size of each solution is polynomial in the input size [60].

▶ **Definition 1.** EnumP *is the class of problems whose solutions can be checked in polynomial time.*

The problems in EnumP can be seen as the task of listing the solutions (or witnesses) of NP problems [60]. This is referred to as $\mathcal{LP}$ in [56]. In this paper, we will only consider problems in EnumP.

▶ **Definition 2** (Output-polynomial time)**.** *An enumeration algorithm operates in* output-polynomial time *if the time required to output all solutions is bounded by a polynomial in the size of the input and the number of solutions.*

This notion has been referred to as *polynomial total time* by Johnson, Yannakakis, and Papadimitriou [37], and implicitly used before, for instance by Tarjan [63], and by Paull and Unger [53]. As pointed out by Goldberg [34], we can ask for output-polynomial time algorithms only if the decision problem is not NP-complete. More precisely, the listing version of an NP-complete problem cannot admit an output-polynomial time unless P = NP. Indeed, suppose that we have an output-polynomial time algorithm for the listing version running within some polynomial time-bound. We can use this listing algorithm to answer whether there is at least a solution, by waiting the time bound and checking whether there has been output at least a solution or not.

When generating all solutions is impractical due to excessive runtime, it becomes important to generate at least a subset of them efficiently. Therefore, we should evaluate and limit the total time required to produce a specified number of solutions.

▶ **Definition 3** (Incremental polynomial time)**.** *An algorithm is incremental polynomial time if it generates the i distinct solutions in time polynomial in i and the input size.*

▶ **Definition 4** (Average polynomial time)**.** *A listing algorithm takes average polynomial time if the time required to output all the solutions is bounded by $O(n^k\alpha)$, where n is the input size, k is a constant, α is the number of solutions.*

In other words, if the average cost per solution is polynomial. This notion has been previously referred by Valiant [71] using the name P-ENUMERABLE. It is worth remarking that the average time per solution has been the subject of a lot of research, which focused on constant amortized time. An algorithm has a constant amortized time (CAT) if its total time divided by the number of solutions is constant (see the book [57]).

The following measure concerns the delay between consecutive solutions, i.e., bounding not the average time between two consecutive solutions but the worst-case time bound between them.

▶ **Definition 5** (Polynomial Delay)**.** *An enumeration algorithm operates with* polynomial delay *if it generates solutions sequentially, ensuring that the time before outputting the first solution, the time between any two consecutive solutions, and the time between the last solution and the end of the algorithm is bounded by a polynomial in the input size.*

Also, it is worth remarking that a lot of research on enumeration algorithms has focused on constant delay. Concerning this we mention the *gray coding* technique exposed in "The Art of Computer Programming", Volume 4 Fascicle 2A, by Knuth [43]. In this context, in order to go from one solution to the next one in constant time, two consecutive solutions must only differ by a fixed number of elements.

Algorithm 3 BACKTRACK(S). We denote as $\pi(x)$ the index associated to an element $x \in U$.

Input: $S \subseteq U$ a set (eventually empty)
Output: All the solutions containing S

```
1 output S
2 foreach e > max_{x∈S} π(x) do
3     if S ∪ {e} is a solution then
4         BACKTRACK(S ∪ {e})
```

It is worth noting that having a polynomial delay implies having an average polynomial time. Having an average polynomial time implies having incremental polynomial time. Having incremental polynomial time implies being output-polynomial. The converse of each of these implications is not true.

4 Partition Techniques for Enumeration

In this section, we detail the first main enumeration technique. The *partition technique* is a powerful approach for enumeration problems that systematically divides the solution space into two or more disjoint subsets, ensuring complete coverage of all the solutions while avoiding redundancy. In general, it is based on recursive decomposition, where the solution space is divided into smaller, non-overlapping subsets, allowing each subset to be processed independently. Moreover, each partition is *distinct*, ensuring that no solution is counted multiple times.

This can be realized by using a *backtracking* strategy, where partial solutions are incrementally constructed and extended until a complete solution is found or determined to be infeasible. A typical backtracking enumeration algorithm follows these steps:

1. Start with an empty or partial solution S.
2. Extend the solution S by making a choice from available candidates $c_1, \ldots, c_k$ for some k.
3. If the extended solution satisfies some constraints, recursively explore further.
4. If a valid complete solution is found, output it.
5. If constraints are violated or no further extensions are possible, backtrack by undoing the last choice and exploring other possibilities.

From a partition perspective, in Item 2 the set of all the solutions extending S is partitioned into the subsets of solutions extending $S \cup \{c_1\}, \ldots, S \cup \{c_k\}$. The constraints in Item 3 help guarantee that such subsets do not overlap, i.e., the corresponding subproblems do not contain the same solutions.

For the sake of explanation, we start by describing *plain backtracking* where the indexing of the elements is used to get non-overlapping subproblems. In this case, we enlarge solutions only guided by the candidates' indexes. We then introduce backtracking *with forbidden sets*, where while recurring, still trying to avoid duplication, we more carefully reduce the sets of candidates, explicitly specifying to the recursive calls which candidates should not be considered. We then introduce the *flashlight method*, which introduces suitable checks before recurring, solving the so-called *extension problem*, in order to improve efficiency.

4.1 Plain Backtracking

A set $F \subseteq 2^U$ (of subsets of U) satisfies *downward closure* if for any $X \in F$ and for any $X' \subseteq X$, we have $X' \in F$; in other words, for any X belonging to F we have that any subset of X also belongs to F. Given a set U and an oracle to decide whether $X \subset U$ belongs to F, an unknown set of 2^U satisfying the downward closure, we consider the problem of enumerating all (eventually maximal) elements of F. The backtracking technique is often applied to problems of this type.

In this approach we start from an empty set, and the elements are recursively added to a solution. The elements are usually indexed, so that in each iteration, in order to avoid duplication, only an element whose index is greater than the one of the current maximum element is added. After performing all examinations concerning one element, by backtracking, all the other possibilities are explored. The basic schema of backtracking algorithms is shown in Algorithm 3. Note that, whenever it is possible to apply this schema, we obtain a polynomial delay algorithm whose space complexity is also polynomial. The technique proposed relies on a depth-first search approach. However, it is worth observing that in some cases of enumeration of families of subsets exhibiting the downward closure property, arising in the mining of frequent patterns (e.g., mining of frequent itemsets), a breadth-first approach can also be successfully used instead of depth-first backtracking. For instance, this is the case of the Apriori algorithm for discovering frequent itemsets [62].

4.1.1 Enumeration of Subsets of Bounded Sum: Backtracking

Consider the problem of enumerating all the subsets of a collection $U = \{a_1, \ldots, a_n\}$ whose sum is less than a certain threshold b. By using the backtracking schema, it is possible to solve the problem as shown in Algorithm 4. Each iteration outputs a solution, and takes $O(n)$ time, so that we have $O(n)$ time per solution. It is worth observing that if we sort the elements of U, then each recursive call can generate a solution in $O(1)$ time, yielding $O(1)$ time per solution.

4.2 Backtracking with Forbidden Set

In the previous section, namely Section 4.1, the elements' indexing was helpful to get non-overlapping subproblems, by enlarging solutions only guided by the candidates' indexes. In the case of backtracking with forbidden sets, while recurring we try to avoid duplication by more carefully reducing the sets of candidates. We do so by explicitly specifying to the recursive calls which candidates should not be considered, through a forbidden set. We showcase this approach through an example, the Bron-Kerbosch algorithm for maximal clique enumeration.

Algorithm 4 SubsetSum(S).

Input: S a set (eventually empty) of integers belonging to the collection $U = \{a_1, \ldots, a_n\}$, a threshold b
Output: All the subsets of U containing S whose sum is less than b.

```
output S
Let π(x) be the index associated to an element x
foreach i > max_{x∈S} π(x) do
    if a_i + Σ_{x∈S} x < b then
        SubsetSum(S ∪ {a_i})
```

4.2.1 Bron-Kerbosch Algorithm for Maximal Clique Enumeration

The Bron-Kerbosch algorithm is a recursive backtracking algorithm used to enumerate all maximal cliques in an undirected graph. Given a graph $G = (V, E)$, the algorithm maintains three sets:

- R: The current clique being constructed.
- P: The set of potential candidates that can be added to R, meaning that all the vertices in P are connected to all the vertices in R.
- X: The set of already processed vertices to avoid redundant solutions.

A recursive call aims to produce all the maximal cliques containing R, some vertices in P, and no vertices in X. Hence, we partition the set of maximal cliques satisfying this property by analysing all the possible candidates in P. When considering a candidate $v \in P$, v is added to the partial solution R, while P and X are reduced to include only neighbors of v.

We output solutions only if P and X are empty. If indeed P contains some element, it means we can still do further recursion to enlarge R, as R is not maximal. If X is non-empty, it also means that R is not maximal, as to be maximal, it should include elements of X. Whenever $P = \emptyset$ and $X \neq \emptyset$, no further addition is possible to R, but the clique is still not maximal, and the algorithm thus backtracks without performing any output. This is typically called a *dead-end*, and the presence of such events is the main reason why this algorithm is not output-sensitive.

Algorithm 5 Bron-Kerbosch Algorithm.

Input: Graph $G = (V, E)$, sets $R, P, X \subseteq V$, where all vertices in P are connected to all the vertices in R.

Output: All maximal cliques containing R, some vertices in P, no vertices in X

```
1  if P = ∅ and X = ∅ then
2  |  Output R as a maximal clique
3  end
4  else
5  |  foreach v ∈ P do
6  |  |  BRON-KERBOSCH(R ∪ {v}, P ∩ N(v), X ∩ N(v))
7  |  |  P ← P \ {v}
8  |  |  X ← X ∪ {v}
9  |  end
10 end
```

The algorithm recursively explores the search space, ensuring that each maximal clique is output exactly once. The basic Bron-Kerbosch algorithm (Algorithm 5) recursively builds cliques and removes processed vertices to avoid duplicates. A more efficient version of the algorithm introduces a pivoting strategy [66]. Choosing a pivot vertex u from $P \cup X$ helps reduce the number of recursive calls by limiting the number of candidates that need to be processed. To obtain this algorithm it is sufficient to replace line 5 with the following two lines: "Choose a pivot vertex u from $P \cup X$" and "**for each** $v \in P \setminus N(u)$ **do**". The idea of this pivoting strategy is to avoid iterating at each expansion on the P set. The results will have to contain either the pivot or one of its non-neighbors, since if none of the non-neighbors of the pivot is included, then we can add the pivot itself to the result. Hence we can avoid iterating on the neighbors of the pivot at this step.

An alternative approach to enhancing the basic Bron–Kerbosch algorithm involves omitting pivoting at the outermost recursion level and instead carefully selecting the order of recursive calls to minimize the size of the candidate vertex set P within each call [28]. The *degeneracy* of a graph G is defined as the smallest integer d such that every subgraph of G contains at least one vertex with a degree of at most d. Every graph admits a *degeneracy ordering*, where each vertex has at most d later neighbors in the sequence. This ordering can be computed in linear time by repeatedly removing the vertex with the smallest degree among the remaining ones. By processing the vertices in the Bron–Kerbosch algorithm according to a degeneracy ordering, the candidate set P (i.e., the neighbors of the current vertex that appear later in the ordering) is guaranteed to have at most d elements. Conversely, the exclusion set X, which consists of previously processed neighbors, may be significantly larger. Although pivoting is omitted at the outermost recursion level, it can still be employed in deeper recursive calls to further optimize performance.

Time Bounds

The Bron–Kerbosch algorithm is not output-sensitive – unlike some other algorithms for the same task (as shown later), it does not run in polynomial time per maximal clique generated. However, it remains efficient in the worst-case sense. By a result by Moon and Moser (1965), any n-vertex graph has at most $3^{n/3}$ maximal cliques [52]. When using a pivot strategy that minimizes recursive calls, the Bron–Kerbosch algorithm runs in $O(3^{n/3})$, matching this bound [66]. For sparse graphs, tighter bounds apply. Specifically, the degeneracy-ordering variant of the algorithm runs in $O(dn3^{d/3})$. Some d-degenerate graphs contain up to $(n-d)3^{d/3}$ maximal cliques, making this bound nearly tight [28].

4.3 Flashlight Search

In the following, for the sake of simplicity, we use the *binary partition* approach, which is a backtracking strategy where each recursive node has at most two children as explained next, in order to easily introduce a more refined backtracking strategy, called *flashlight*. The idea can be easily extended to the case where recursive nodes have more than two children.

Let X be a subset of F, the set of solutions, such that all elements of X satisfy a property P. The binary partition method outputs X only if the set is a singleton, otherwise, it partitions X into two sets X_1 and X_2, whose solutions are characterized by the disjoint properties P_1 and P_2 respectively. This procedure is repeated until the current set of solutions is a singleton. The bipartition schema can be successfully applied to the problem of enumeration of paths of a graph connecting two vertices s and t (as we will see in this section), of the perfect matchings of a bipartite graph [68], of the spanning trees of a graph [59]. The flashlight method relies on checking whether there is at least a solution before recurring, i.e., checking whether there is at least a solution in X_1 (resp. X_2) satisfying P_1 (resp. P_2). This is usually called an *extension problem*, as we are checking while enlarging a partial solution whether this can be extended/completed to a final solution. This check is crucial for efficiency as if every partition is non-empty, i.e., all the internal nodes of the recursion tree are binary, then the number of internal nodes is bounded by the number of leaves. In addition, if we have that solving the extension problem takes polynomial time, since every leaf outputs a solution, we have that the resulting algorithm is output-polynomial. On the other hand, even if there are empty partitions, i.e., internal unary nodes in the recursion tree, if the height of tree is bounded by a polynomial in the size of the input and the partition oracle takes polynomial time, then the resulting algorithm is polynomial delay.

We will discuss more about extension problems in Section 4.5.

4.3.1 Enumerating all *st*-paths in a graph

We showcase the flashlight strategy through a binary partition algorithm for the problem of *st*-path enumeration in an undirected graph $G = (V, E)$. The partition schema chooses an arc $e = (s, r)$ incident to s, and partitions the set of all the *st*-paths into the ones including e and the ones not including e. The *st*-paths including e are obtained by removing all the arcs incident to s, and enumerating the *rt*-paths in this new graph, denoted by $G \setminus s$. The *st*-paths not including e are obtained by removing e and enumerating the *st*-paths in the new graph, denoted by $G \setminus e$. The corresponding pseudocode is shown by Algorithm 6. It is worth observing that if the arc e is badly chosen, a subproblem could not generate any solution; in particular, the set of the *rt*-paths in the graph $G \setminus s$ is empty if t is not reachable from r, while the set of the *st*-paths in $G \setminus e$ is empty if t is not reachable from s. Thus, before performing the recursive call to the subproblems it could be useful to test the validity of e, by testing the reachability of t in these modified graphs. This will be our "flashlight" indicating which partitions lead to at least one solution. Note that the height of the corresponding recursion tree is bounded by $O(|E|)$, since at every level the size of the graph is reduced by at least one arc. The cost per iteration amounts to the reachability test, so $O(|E|)$ time. Therefore, the algorithm has $O(|E|^2)$ delay.

This problem has been studied in [55, 63, 65], and in [38], guaranteeing a linear delay. In [5], the latter algorithm has been modified in order to enumerate *bubbles*. In the particular case of undirected graphs, in Section 4.3.2 we will show an algorithm based on this bipartition approach having an output sensitive amortized complexity, as shown in [6]. In the particular case of shortest paths, the enumeration problem has been studied in [27]. It is worth observing that the problem of enumerating all the *st*-paths in a graph is equivalent to the problem of enumerating all the cycles passing through a vertex.

Algorithm 6 PATHS(G, s, t, S).

Input: A graph G, the vertices s and t, a sequence of vertices S (potentially empty)
Output: All the paths from s to t in G

```
if s = t then
    output S
    return
choose an arc e = (s, r)
if there is a (r, t)-path in G \ s then
    PATHS(G \ s, r, t, S · r)
if there is a (s, t)-path in G \ e then
    PATHS(G \ e, s, t, S)
```

4.3.2 Improving *st*-paths Enumeration: Dynamic Certificate

We can improve the *st*-path enumeration algorithm proposed in the previous section to achieve optimal complexity, by using a dynamic certificate based on the biconnected components of the graph [6]. Let $P_{st}(G)$ denote the set of *st*-paths in G and, for an *st*-path $\pi \in P_{st}(G)$, let $|\pi|$ be the number of it edges. We note that the problem of *st*-path enumeration necessarily requires $\Omega(\sum_{\pi \in P_{st}(G)} |\pi|)$ time to just list the output. The algorithm we present here is then optimal, as it lists all the *st*-paths of G in $O(m + \sum_{\pi \in P_{st}(G)} |\pi|)$ time, where m is the number of edges of the input graph.

BCCs and Bead Strings

Recall that the *biconnected components* (BCCs) of G are the inclusion-maximal biconnected subgraphs of G. A BCC is called *non-trivial* if it is made up of at least three nodes. It is known that the BCCs of a connected graph form a tree:

▶ **Definition 6.** *Let G be a connected graph, and consider the following graph: add a vertex for each BCC of G, and a vertex for each articulation point of G. Then, add an edge between an articulation point a and a BCC B if and only if $a \in B$. The resulting graph $\mathcal{B}_G$ is a tree, called the* block-cut tree*, or* BC-tree*, of G.*

The vertices of the BC-tree that correspond to BCCs are called *graph-vertices*, while the ones that correspond to articulation points are called *node-vertices*. By construction, the tree alternates levels of node-vertices to levels of graph-vertices. We observe that, given any two nodes x, y of G, there is a unique corresponding shortest path in the BC-tree, which we call the *bead string from x to y*, and we denote with $B_{x,y}$. If the path is trivial, then x and y belong to the same BCC. Otherwise, the two endpoints of $B_{x,y}$ are distinct BCCs: one of them contains x, and the other y. Given $B_{x,y}$, we define its *head* H_x as the first biconnected component along the bead string (in other words, the component of $B_{x,y}$ that contains x).

In the following, we give a characterization of how st-paths behave with respect to the BCCs of G. Consider the bead string $B_{s,t}$ in the BC-tree of G: its extremities $B_s = H_s, B_t$ (possibly equal) must contain s and t, respectively. Keeping this notation in mind, we can restate the following result by Birmelé et al.:

▶ **Lemma 7** (Lemma 3.2 from [6])**.** *All the st-paths in $\mathcal{P}_{s,t}(G)$ are contained in the subgraph of G corresponding to $B_{s,t}$. Moreover, all the articulation points in $B_{s,t}$ are traversed by each of these paths.*

Thus, when looking for st-paths, we are actually only concerned with the subgraph of G formed by the BCCs along the bead string $B_{s,t}$ from B_s to B_t in the BC-tree.

Validity Check with Bead Strings

Our aim is to improve the partition schema introduced in the previous section, by employing the structure and properties of the bead strings to perform an efficient validity test for the edge e that is chosen at partition time (line 4 of Algorithm 6). More specifically, let G, s, t be the instance at the current iteration; we wish to avoid choosing $e = (s, r)$ such that t is not reachable from r in $G \setminus s$ (that is, $\mathcal{P}_{r,t}(G \setminus s) = \emptyset$), and not reachable from s in $G \setminus e$ ($\mathcal{P}_{s,t}(G \setminus e) = \emptyset$). Indeed, for such a bad choice both partition subproblems yield no solution. In the previous section, we checked both of these conditions by performing a linear time reachability test at each iteration. We can avoid this computation by retaining a dynamic certificate based on the bead strings.

Indeed, by Lemma 7 we can immediately see that edges $e = (s, r)$ with $r \notin H_s$ are bad choices, leading to no solutions in their subproblems. On the other hand, any choice with $r \in H_s$ will lead to at least a solution, since by definition of the bead string, each $e = (s, r) \in H_s$ is such that $\mathcal{P}_{r,t}(G \setminus s) \neq \emptyset$. We call this branch of the partition the *left child*. Furthermore, if the head H_s is non-trivial (that is, s has at least two neighbors), then there is also always a solution in $G \setminus e$. Indeed, since H_s is biconnected, there are always at least two st-paths in the bead string. Thus, there is always an st-path that traverses e (which is the left child from before), and one that does not (which we call the *right child*): thus $\mathcal{P}_{s,t}(G \setminus e) \neq \emptyset$.

The downside of the bead string approach is that maintaining this information can be costly if done naively: computing the bead string at each step would require linear time in the worst case, giving us no advantage with respect to the previous algorithm. It is for this reason that the notion of certificate is introduced. By employing such certificates, coupled with an amortized analysis based on the fact that only the BCCs of the head H_s need to be updated at each step, the authors of [6] were able to provide the optimal amortized complexity.

Certificate

The certificate C is a compacted and augmented DFS tree of $B_{s,t}$ rooted at s, which changes over time along with the corresponding bead string. Edges of the bead string are classified according to the tree as either *tree edges* (the ones belonging to the DFS tree), or *back edges*[2], and this information is used by the certificate for making certain choices. The certificate is able to perform the following operations:

1. CHOOSE(C, s): returns an edge $e = (s, r)$ with $r \in H_s$ (leading to the left child). As mentioned before, there always exists one such edge. This function chooses r as the last such neighbor of s in DFS postorder. This will ensure that, the (only) tree edge leaving s is returned last.
2. LEFT_UPDATE(C, e): given $e = (s, r)$, it computes $B_{r,t}$ in $G \setminus s$ from $B_{s,t}$ in G. This also updates H_s and C, and returns bookkeeping information I so that everything can be reverted to the status before this operation when exiting the recursive call.
3. RIGHT_UPDATE(C, e): given $e = (s, r)$, it computes $B_{s,t}$ in graph $G \setminus e$ from $B_{s,t}$ in G. As before, it updates H_s and C, and returns bookkeeping information I like LEFT UPDATE(C, e).
4. RESTORE(C, I): reverts all information (bead string, head, certificate) to their status before the corresponding operation I (left or right update).

With these notions, we can rewrite the pseudocode for st-path enumeration as in PATHS_CERTIFICATE (Algorithm 7). Note that, since the tree edge incident to s is the last one to be returned by CHOOSE(C, s), then all remaining st-paths must traverse this edge, and the recursive call at line 7 would yield no solutions. Therefore, the check at line 5 is the only necessary one to ensure that all recursive calls produce solutions.

With a careful implementation and amortized analysis, it can be shown that a certificate to retain bead strings using DFS tree information achieves optimal $O(m + \sum_{\pi \in P_{st}(G)} |\pi|)$ time for the enumeration of st-paths of an input graph with m edges.

4.4 Assessment of *st*-paths

The same BCC structure highlighed in the previous section can also be used to perform the *assessment* task for st-paths.

In an assessment problem, we are asked to determine whether the number of solutions exceeds a given input threshold z. Assessment algorithms are positioned between counting and enumeration: their performance is akin to counting (the output is only related to the number of solutions, with no listing required), while their structure is often similar to a truncated enumeration procedure. Not only can assessment be helpful in applications where we only need to guarantee a certain amount of solutions, and thus full counting is redundant

[2] Note that there are no *cross edges*, as the graph is undirected.

Algorithm 7 PATHS_CERTIFICATE(G, s, t, S).

Input: A graph G, the vertices s and t, a sequence of vertices S (potentially empty)
Output: All the paths from s to t in G

```
if s = t then
    output S
    return
e = CHOOSE(C, s); let e = (s, r)
if e is back edge then
    I = RIGHT_UPDATE(C, e)
    PATHS_CERTIFICATE(G \ e, s, t, S)
    RESTORE(C, I)
I = LEFT_UPDATE(C, e)
PATHS_CERTIFICATE(G \ s, r, t, S · r)
RESTORE(C, I)
```

(e.g., z-reverse safe data structures [4,17]), but it can be even more useful for devising efficient algorithms for #P-complete counting problems. Such problems have no polynomial-time counting algorithm unless P= NP [71], but they can still have assessment algorithms running in time polynomial in the input size and the value of the threshold z. An example of this is the problem of counting the number of st-paths in an undirected graph [72], which will be the topic of this section. For this problem, an assessment algorithm running in $O(|E|z)$ time and $O(|E||V|)$ space was proposed in [54]. This algorithm decomposes and arranges the st-paths in a tree-like structure (based on the BCCs), which is in turn used to maintain and update a combinatorial lower bound.

We start again from Lemma 7, further noting that the paths inside each of the components of $B_{s,t}$ can be combined independently, thus obtaining:

▶ **Corollary 8** (Corollary 2.1 of [54]). *Let $B_{s,t} = B_s a_0 B_1 a_1 B_2 \cdots B_k a_k B_t$ be the path in $\mathcal{B}_G$ between B_s and B_t, where $a_0, ..., a_k$ are node-vertices, and $B_1, ..., B_k$ are graph-vertices. Then:*

$$\mathcal{P}_{s,t}(G) = \mathcal{P}_{s,a_0}(B_s) \times \left(\prod_{i=1}^{k} \mathcal{P}_{a_{i-1},a_i}(B_i) \right) \times \mathcal{P}_{a_k,t}(B_t). \tag{1}$$

4.4.1 Main Idea: Expanding a Structural Lower Bound

Assume that we are given a structural lower bounding function $L_{s,t}(\cdot)$: that is, for any biconnected graph G with $s, t \in V(G)$, we have $L_{s,t}(G) \leq |\mathcal{P}_{s,t}(G)|$. Then, if we take

$$lbP = L_{s,a_0}(B_s) \times \left(\prod_{i=1}^{k} L_{a_{i-1},a_i}(B_i) \right) \times L_{a_k,t}(B_t), \tag{2}$$

we obtain that, by Corollary 8, lbP is a lower bound on the total number of st-paths. At this point, if $lbP \geq z$, we can output YES. Otherwise, we need some way to expand (1). We can do so by exploiting the following disjoint union:

$$\mathcal{P}_{s,t}(G) = \bigcup_{u \in N_{B_s}(s)} s \cdot \mathcal{P}_{u,t}(G \setminus s), \tag{3}$$

where $N_{B_s}(s) = \{u \in B_s \mid (s,u) \in E\}$ is the set of neighbors of node s inside B_s. Note that this can be seen as exploring the binary partition tree of the previous section in a different way: given a current source s, we only consider what we called its "left children", and we explore all of them at the same step (by considering all the neighbors of s that lead to t). Note that this leads us to conceptually create "new sources" whose disjoint union covers all paths from the original source, as we now focus on ut-paths for each u neighbor of s.

The latter formula, when the lower bound is applied, is translated to

$$L_{s,t}(G) = \sum_{u \in N_{B_s}(s)} L_{u,t}(G \setminus s).$$

We can plug this lower bound expansion in (2), improving the total bound lbP:

$$lbP = L_{s,a_0}(B_s) \times \left(\prod_{i=1}^{k} L_{a_{i-1},a_i}(B_i) \right) \times L_{a_k,t}(B_t) =$$

$$= \sum_{u \in N_{B_s}(s)} L_{u,a_0}(B_s \setminus s) \times \left(\prod_{i=1}^{k} L_{a_{i-1},a_i}(B_i) \right) \times L_{a_k,t}(B_t).$$

This refinement can be recursively applied again and again, by taking the reduced graph $B_s \setminus s$, recomputing its bead string, and applying Corollary 8 to the new bead string. In this way, we can proceed until either lbP reaches z, or when we have effectively counted all st-paths, and their total number does not reach z: here we can safely output NO.

To be employed in such a procedure, we need a *structural lower bounding function*, meaning that it must satisfy the following:

1. Given a biconnected graph G' and two nodes $s', t' \in V(G')$, we have $1 \leq L_{s',t'}(G') \leq N_{s',t'}(G')$.
2. If G' is trivial (size 2), then $L_{s',t'}(G') = 1$.

In [54], the authors prove that the function $\mathcal{C}(G) := |E| - |V| + 2$ satisfies such properties, and they employ it in the subsequent implementation.

Multi Source Tree Data Structure

To efficiently retain the information concerning the nested lower bounds, the authors use a tree structure called the *Multi Source Tree Data Structure (MSTDS)*. Such a tree has one node for each of the BCCs currently contributing to the global lower bound lbP (that is, the ones appearing in the formula expanded so far).

The starting tree will consist of a path, given by the bead string $B_{s,t}$ of the original graph G. The root of the tree is the BCC B_t containing t, and there is one leaf for each BCC containing a current source (that is, a node u that was expanded as per (3)). Each root-to-leaf path is a bead string for one of the current sources, and their disjoint union covers all st-paths of the original graph. We note that, since we start from $B_{s,t}$, the structure obtained by repeatedly expanding the BCC corresponding to a source will always be a tree.[3]

Given the tree at a current step, the global bound lbP can be computed as the sum, over all root-to-leaf paths, of the product of the components along each path.

[3] Some care must be taken to avoid repetition of BCCs, as the same node can become source multiple times, but it can be easily handled by keeping multiplicities for each current source.

4.4.2 The Assessment Algorithm

The assessment algorithm can now follow quite immediately: start from a single bead string, and the MSTDS $\mathcal{T}$ given by the corresponding path. At each step, choose a leaf B_u of $\mathcal{T}$ with source u, and expand it as per (3). Recompute the BCCs of $B_u \setminus u$, and add them in $\mathcal{T}$ as a subtree in place of B_u. Then, update the total lower bound lbP accordingly. By keeping some information in the MSTDS nodes concerning the lower bounds of the paths, such bound update can be performed in $O(1)$ time.

The lower bound lbP is improved by at least one every time a source with at least two neighbors is considered. We can find a source with this property in at most linear time (we traverse its unique neighbor, and thus trivial BCC, for at most $|V|$ steps). Once we find such a source, and we expand the lower bound following (3), we need $O(|E|)$ time to update the BCCs after the source removal. Thus, we spend linear time per non-trivial step where we update lbP. Since we stop when the lower bound reaches z, we do no more than z such steps, leading to a time complexity of $O(|E|z)$.

4.5 More about the Extension Problem

In Section 4.3, we introduced the so-called extension problem, which consists of determining whether a partial solution can be extended into a complete one. We have seen that this check is fundamental for the flashlight method, which ensures polynomial delay by verifying the existence of at least one solution before making a recursive call. This mechanism effectively avoids dead-ends, such as those encountered in the Bron-Kerbosch algorithm when listing maximal cliques (see Section 4.2.1).

Since these dead-ends prevent the Bron-Kerbosch algorithm from being output-sensitive, one might consider defining an extension problem to guide its recursion. Specifically, in Algorithm 5, before making the recursive call at line 6, it would be desirable to check whether there exists a maximal clique containing $R \cup \{v\}$, using some vertices from $P \cap N(v)$ and none from $X \cap N(v)$. However, this problem has been proven to be NP-complete [22], implying that such a check would make each recursive step intractable unless P = NP. Consequently, the Bron-Kerbosch algorithm has exponential delay, even when different pivoting strategies are considered [22].

It is important to note that the hardness of the extension problem for maximal cliques is not an isolated case [9]. In fact, Brosse et al. have shown that the extension problem is NP-complete for every "interesting" hereditary property. Notable examples include maximal k-degenerate induced subgraphs of a chordal graph [21] and maximal Steiner subgraphs [16], the latter of which remains hard even for just three terminals.

Furthermore, it is worth emphasizing that the hardness of the extension problem does not preclude the existence of output-sensitive algorithms. In some cases, two possible approaches can be considered. First, we may focus on restricted versions of the extension problem; for instance, in the case of maximal Steiner subgraphs with three terminals, if the partial solution is already connected, the extension problem becomes polynomial. Second, alternative techniques can be employed, such as reverse search for maximal clique enumeration, as discussed later.

5 Solution Graph Techniques for Enumeration

In Section 4, we introduced partition-type enumeration algorithms. This type of algorithm simply explores a solution space by dividing it into smaller subspaces. However, when enumerating only maximal or minimal solutions, this approach usually faces hard subproblems, like extension problems. Hence, we need to study the "structure" of the solution space to obtain an efficient enumeration algorithm.

In this section, we present another approach, called the *solution graph* technique. Intuitively, algorithms based on this technique traverse a graph, called the *solution graph*, defined on the set of solutions. The vertex set of the solution graph is the set of solutions. The main difficulty of this technique concerns how to define a good edge set for the solution graph, that is, how to define the neighbourhood of each solution. Fortunately, we have several standard strategies to do so. In the following, we will introduce some popular techniques such as the reverse search method, the input restricted problem method, and proximity search.

5.1 Reverse Search

The *reverse search* method [3] is one of the most common ways to define a solution graph $\mathcal{G}$. When we enumerate all the solutions from $\mathcal{G}$, we perform a depth-first search on $\mathcal{G}$. However, if $\mathcal{G}$ is disconnected, then we need to enumerate all connected components in $\mathcal{G}$. Moreover, if $\mathcal{G}$ is cyclic, then we may need an exponential-size stack to avoid outputting duplicate solutions. Hence, to achieve space-efficient enumeration, it is desirable for $\mathcal{G}$ to be connected and acyclic, that is, $\mathcal{G}$ forms a tree.

A solution graph defined by the reverse search method is called the *family tree*. As its name suggests, the family tree is connected and acyclic. To define the tree, it is important to define the "root" and a good "parent rule" in the tree. When we traverse the tree, performing enumeration, we will start from the root and go downwards, traversing the tree edges in reverse direction from parents to children (hence, the name reverse search). Therefore, for obtaining an efficient algorithm we also need to provide an efficient procedure for finding all the "children" of each solution. In the following subsections, we will give two examples of constructing a family tree for enumeration problems. We will start by re-considering the problem of enumerating all subsets of bounded sum (as seen in Section 4.1.1) with this new approach. Then, we will move to a more involved example concerning maximal clique enumeration, finally achieving output-sensitive time complexity for this problem, contrarily to the previous algorithm of Section 4.2.1.

5.1.1 Enumeration of Subsets of Bounded Sum: Reverse Search

In Section 4.1.1, we gave an enumeration algorithm based on a partition approach (Algorithm 4). Interestingly, this algorithm can be interpreted as a reverse search-based algorithm. Let $\textsc{Sol}(U)$ be the set of solutions on U. The root of the family tree is the empty set. Let S be a non-root solution $S = \{s_{f(1)}, \ldots, s_{f(k)}\} \subseteq U$ with cardinality k. Here, $f : \{1, \ldots, k\} \to \{1, \ldots, n\}$ is an injection such that $f(i) < f(j)$ for each $i < j$. Then, we can define the parent $Par(S)$ of S as the set obtained by removing the element with largest index in S: that is, $Par(S) = \{s_{f(1)}, \ldots, s_{f(k-1)}\}$. Note that $Par(S)$ is clearly in $\textsc{Sol}(U)$. Now, we define the family tree $\mathcal{F}(U)$ as follows:

$$\mathcal{F}(U) = (\textsc{Sol}(U), \mathcal{E}(U)), \tag{4}$$

where $\mathcal{E}(U) = \{(Par(S), S) \mid S \neq \emptyset\}$.

We first show that the family tree is connected and acyclic. A typical approach to show this is to use a function $g : \textsc{Sol}(U) \to \mathbb{R}$ such that

1. For the root R, $g(R)$ has the minimum value among the solutions, and
2. For any non-root solutions $S, S' = Par(S)$, we have $g(S) > g(S')$.

Here, a candidate such function is the cardinality of a solution. By using such a g, we can then easily obtain the following lemma:

▶ **Lemma 9.** *For any solution $S \in \textsc{Sol}(U)$, there is a path from S to $\emptyset$ in $\mathcal{F}(U)$. Moreover, $\mathcal{F}(U)$ is acyclic.*

Proof. For any non-root solution S, $|S| > |Par(S)|$. Hence, by recursively obtaining the parent of S, we can find the empty set, that is, the root. Thus, there is a path from S to the root. Moreover, if there is a cycle in $\mathcal{F}(U)$, then there must be an edge (S, S') in $\mathcal{F}(U)$ such that $|S| > |S'|$. However, this contradicts that S is the parent of S'. Thus, the lemma holds. ◀

Next, we show the definition of the children. Suppose that for an element $x \in U$, $\textsc{idx}(x)$ is the index of x in U. Then, we define the children of S is as follows:

$$Ch(S) = \left\{ S \cup \{u\} \;\middle|\; \textsc{idx}(u) > \max_{x \in S} \textsc{idx}(x) \text{ and } u + \sum_{x \in S} x < b \right\} \tag{5}$$

This definition is reasonable by the following lemma:

▶ **Lemma 10.** *For any solution S, $Ch(S) = \{S' \mid Par(S') = S\}$.*

Proof. Let S and S' be any pair of solutions. Then, first we show that if $Par(S') = S$ then $S' \in Ch(S)$. Since S' is a child of S, for some $u \in U$, $S' = S \cup \{u\}$. Moreover, u has the maximum index in S'. Thus, from the definition of the parent relationship, the lemma holds.

Next, we show the other direction. Assume that $S' \in Ch(S)$. Then, from the definition of $Ch(\cdot)$, S' contains an element u that has a larger index than any element in S. Thus, from the definition of the parent, $Par(S') = S$. ◀

From the above discussion, we can give another proof for the existence of an enumeration algorithm for this problem with $O(1)$ time per solution, given by the traversal of this family tree.

5.1.2 Maximal Clique Enumeration via Reverse Search

Next, we give a (slightly) more complicated example. Recall that a *clique* of a graph $G = (V, E)$ is a vertex subset C of the graph such that for any pair of vertices in C, there is an edge in G. Moreover, a clique C is *maximal* if there is no clique C' such that $C \subsetneq C'$. Several enumeration algorithms for maximal cliques can be found in the literature [2, 40, 50]. In this section, we explain the algorithm based on reverse search by Makino and Uno [50]. As in the previous section, we need to define the root and the parent-child relation of the family tree.

First, we define the root of the family tree. Suppose that $V = \{v_1, \ldots, v_n\}$ is a totally ordered set such that for any $i < j$, $v_i < v_j$. For two maximal cliques C_1, C_2, we say that C_1 is *lexicographically smaller* than C_2 if the vertex with minimum index in $(C_1 \setminus C_2) \cup (C_2 \setminus C_1)$ belongs to C_1, and write $C_1 \prec C_2$. Then, let K_0 be the maximal clique such that it is the lexicographically smallest among all the maximal cliques in G. We say a maximal clique is the *root* if the clique is equal to K_0.

Next, we define the parent of a clique $K \neq K_0$. For an integer i, let $K_{\leq i} = K \cap \{v_j \mid j \leq i\}$, and let $C(K_{\leq i})$ be the lexicographically smallest maximal clique containing $K_{\leq i}$. Note that $C(K_{\leq i})$ may be equal to K.

Thus, we define the *parent index* i of a maximal clique K as the maximum index such that $C(K_{\leq i-1}) \neq K$. Consequently we define the parent $Par(K)$ of K as $C(K_{\leq i-1})$, where i is the parent index of K.

Clearly, $Par(K) \prec K$ holds, and there can be no cycles. It also follows that every maximal clique will have a parent index, and thus a parent, other than $K_0 = C(\emptyset)$. Thus, the parent-child relationship derives a tree structure rooted at K_0.

Computing the parent of a maximal clique is easy. However, obtaining all the children of a clique K is not obvious. The crucial observation for child enumeration is the following: Let K' be a child of K; then, for any j larger than or equal to the parent index, $C(K'_{\leq j}) = K'$. Hence, children can be found by removing from K a vertex v and vertices having larger indices than v, then adding a vertex u (possibly $u = v$) to the remaining of K, and greedily adding vertices to obtain a maximal clique. By this construction, u is the vertex with the parent index in the resultant clique. In the following, we give more details. Given a maximal clique K and an index i, let

$$K[i] = C((K_{\leq i} \cap N(v_i)) \cup \{v_i\}).$$

$K[i]$ can be interpreted as follows: we first pick the vertices adjacent to v_i in K, and then greedily add vertices to K. Therefore, the following lemma holds.

▶ **Lemma 11** ([50, Lemma 2]). *Let K and K' be maximal cliques in G. Then K' is a child of K if and only if $K' = K[i]$ hold for some i such that*

1. $v_i \notin K$
2. *i is larger than the parent index of K*
3. $K[i]_{\leq i-1} = K_{\leq i} \cap N(v_i)$
4. $K_{\leq i} = C(K_{\leq i} \cap N(v_i))_{\leq i}$

Moreover, if an index i satisfies the above four conditions, then i is the parent index of $K[i]$.

From the above lemma, we can obtain the children of K in polynomial time, yielding a polynomial delay enumeration algorithm for maximal cliques.

5.2 Input-restricted Problem

While reverse search is a powerful tool, one of its drawbacks is the complexity involved in defining a working "parent-child" relationship. However, there actually exists a simple and general way to obtain a solution graph for a large class of problems, which in some cases immediately yields efficient enumeration algorithms for maximal solutions. This method is typically referred to as the *Input-Restricted Problem*, or hereafter IRP.

5.2.1 Designing the algorithm

The key intuition can be already found in Lawler et al. [49], generalizing algorithms by Paull and Unger [53] and by Tsukiyama et al. [67]. The idea given is the following: given a maximal solution S of a given enumeration problem and some element $x \notin S$, the hardness of listing solutions *maximal within* $S \cup \{x\}$ is linked to the hardness of listing maximal solutions of the general problem.

This has been then formalized as the Input-Restricted problem by Cohen et al. [15], who also restrict their attention to the enumeration of maximal solutions in graphs (although the technique can potentially be applied to any enumeration problem where solutions are maximal sets of elements). Formally:

▶ **Definition 12** (Input-Restricted Problem). *Let $G = (V, E)$ be a graph and P a property (such as being a clique, or a path). Let $S \subseteq V$ be a maximal solution, i.e., an inclusion-maximal set of nodes respecting P, and v any node in $V \setminus S$.*

The Input-Restricted Problem $IRP(S, v)$ asks to enumerate all maximal solutions of P in the induced subgraph $G[S \cup \{v\}]$.[4]

Intuitively, we can use a maximal solution X of $G[S \cup \{v\}]$ to generate a new maximal solution X' of G. Whenever this happens, we say there is an edge from S to X' in the solution graph (optionally labelled with v). Another significant upside of the technique is that X' can be any arbitrary solution that includes X. Observe that S itself is always a maximal solution of $IRP(S, v)$, but we typically avoid returning it as it would only add a futile edge from S to S.

The catch is the following: this technique assumes that the property P at hand is *hereditary*, i.e., if S is a solution, then any subset of S must itself be a solution. Even if many properties respect this condition (e.g., cliques and most of their relaxations, independent sets, forests), not all properties do, and the IRP technique may not be the most suitable for the latter (e.g., cycles and spanning trees).

If the property P is hereditary, then we can easily define a simple *maximalization* procedure COMPLETE(X), which iteratively adds an arbitrary element of $V \setminus X$ to X until it is no longer possible to do so without breaking P. With this, we have all the elements for building an enumeration algorithm, which is a DFS-like traversal of the solution graph (see Algorithm 8).

Algorithm 8 Enumeration via the Input-Restricted Problem [15].

Input: A graph $G = (V, E)$, a property P
Output: All the maximal subgraphs of G respecting P.

```
1  Let S ← COMPLETE(∅)                                   // An arbitrary solution
2  REC(S)
3  Function REC(S):
4      output S
5      foreach v ∈ (V \ S) do
6          foreach X ∈ IRP(S, v) do
7              X' ← COMPLETE(X)
8              if X' was not already found then REC(X')
9  Function IRP(S, v):                                   // Input-Restricted Problem
10     foreach Maximal solution X of P in G[S ∪ {v}] do
11         yield X
```

[4] The definition is identical for problems where solutions are sets of edges, simply requiring S to be a set of edges and v to be an edge.

The main advantage of the technique is now clear: if we simply plug in a solver for the input-restricted problem IRP(S, v), the algorithm will enumerate all maximal solutions to the general problem.

The completeness of this algorithm for hereditary properties relies on the idea that the solution graph generated with the IRP is strongly connected, thus a traversal from any starting point will visit every solution. Proving this is actually quite simple. It relies on showing that if you start at S and want to reach T, there is always a $v \in T$ and a solution X of $IRP(S, v)$, such that X has a larger intersection with T than S had: applying this repeatedly must inevitably yield T in a finite amount of steps. More detailed proofs can be found in [15].

Let us then discuss the drawbacks of this approach, and how some of them can be solved.

Firstly, checking that "X' was not already found" is necessary to avoid duplication, but it is no trivial task. A way to do this is to store every solution found so far in a dictionary, and use it to check whether X' is new. This is efficient time-wise (a trie or a block tree can answer the query in linear time), but as there can be exponentially many solutions, it makes the space usage exponential. However, Cohen et al. [15] show that, for hereditary properties, we can avoid a dictionary and induce a tree-like structure based on solving the problem incrementally. Essentially, they use the solutions of $G[\{v_1, ..., v_{i-1}\}]$ to generate those of $G[\{v_1, ..., v_i\}]$, much like Tsukiyama et al. [67], in a DFS-like fashion. This alternative algorithm has the advantage of reducing space usage, but does not operate anymore on the solution graph, as the actual maximal solutions of G correspond only to the leaves of this tree-like structure.

It is thus worth noting that Algorithm 8 has a striking similarity to reverse search, and indeed it is also possible to achieve polynomial space by inducing a parent-child relationship, identifying a parent and a parent index for X' in the same way as Section 5.1, and turning the condition in Line 8 to "if the parent of X' is S, and the parent index is v". In this light, the Input-Restricted Problem becomes a powerful and general tool for building reverse search algorithms.

5.2.2 Supported classes of properties

As discussed above, the Input-Restricted Problem is easy to deal with if the property P at hand is *hereditary* (i.e., subsets of solutions are solutions). However, Cohen et al. [15] generalizes the proof of correctness of Algorithm 8 to the more general class of *connected-hereditary* graph properties, i.e., properties for which any *connected subgraph* of a solution is itself a solution.

To give an example, *induced forests* are a hereditary property: if a set of nodes S induces a forest, any subset of S induces a forest. A connected-hereditary property are instead *induced trees*: if S induces a tree, a subset of S induces a forest, but a subset of S that induces a connected subgraph will induce a tree. Inducing a parent-child relationship in this class is more challenging, and Cohen et al. [15] rely on a solution dictionary for avoiding duplication. However, Conte et al. [20] proposes a general technique that allows the definition of a parent-child relationship within the structure of Algorithm 8 even for connected-hereditary properties,[5] thus allowing enumeration in polynomial space.

[5] Technically the class is the more general set of "strongly accessible and commutable" properties, whose definition can be found in [20], although most naturally defined properties in the class will typically fall inside connected-hereditary.

Finally, note that properties like *induced cycles* are neither hereditary nor connected-hereditary, as a subset of a cycle will never be a cycle; properties of this kind cannot be solved directly by Algorithm 8.

5.2.3 Complexity of the Algorithm and the IRP

The final piece of the puzzle is the Input-Restricted Problem itself.

It is relatively easy to see that, in Algorithm 8, the complexity of the $IRP(S, v)$ essentially corresponds to the amount of work done per solution: every time a new solution is found, we perform $IRP(S, v)$ up to $|V|$ times (for each different v), and for each solutions returned we just need to apply COMPLETE(). This means that the cost-per-solution of the algorithm is just a polynomial multiplied by the cost of the IRP[6]. While intuitively a simpler task, the IRP it does not always allow efficient resolution: the number of maximal solutions within the IRP can greatly vary from problem to problem, leading to different complexities.

One of the simpler cases are maximal cliques. Given a maximal clique S and a node v, $G[S \cup \{v\}]$ has just two maximal cliques:

- S itself (v cannot be added to it, as S was already maximal in G)
- $\{v\} \cup (S \cap N(v))$, i.e., v and all its neighbors in S (any other element of S not adjacent to v cannot extend the clique)

As it has only two solutions (and only one we care about, since S was already known), $IRP(S, v)$ for maximal cliques can be solved in polynomial time. This means the total amount of work done by Algorithm 8 per solution is polynomial, i.e., we achieve polynomial amortized time.[7]

This is not always the case: the Input-Restricted Problem may have an exponentially high number of solutions, meaning it cannot be solved in polynomial time. Even in this case, not all hope is lost: some properties, like *Maximal Connected Induced Bipartite Subgraphs* [73] or *Maximal Temporal Cliques* in a restricted class of temporal graphs [11] allow the IRP to be solved in polynomial delay.

More in general, we can say that whenever the IRP can be solved in *Incremental Polynomial Time* (which includes polynomial delay), then Algorithm 8 also runs in Incremental Polynomial Time. Intuitively, this can be proved by the fact that, if i solutions were found so far, the time used by the IRP to find solutions that we already knew is polynomial in i and the input size (e.g., $O(n^c \cdot i^c)$), and since we can apply the IRP only to the i solutions found so far, the time required until a new solution is found is bounded by $O(i \cdot n^c \cdot i^c)$, which is also polynomial in i and n.

Finally, there are cases where even the IRP simply cannot be solved efficiently: Lawler [49], for example, shows how to reduce an instance of SAT to a simple hereditary property on a set of n elements, where $O(n)$ maximal solutions can be easily identified in polynomial time, and another solution exists if-and-only-if the input formula is satisfiable. It follows that no output-sensitive algorithm may exist for this problem, unless $\mathsf{P} = \mathsf{NP}$. This tells us that even an efficient solution to the IRP would imply $\mathsf{P} = \mathsf{NP}$, as we could use it in Algorithm 8 to obtain an efficient algorithm for the problem.

---

[6] The polynomial of course varies according to whether we can limit the number of v considered and the cost of COMPLETE, but it is always polynomial for hereditary and connected-hereditary properties, as long as we can identify in polynomial time wherher S is a solution or not, as we can then simply iteratively test every node for addition until none is possible.

[7] In fact, we can turn this into polynomial delay with *alternative output* [69], that simply makes us output S at the beginning of REC(S) if the recursion depth is even, and at the end of REC(S) if the recursion depth is odd. This avoids long chains of recursive calls being closed (or opened) without new outputs being produced.

5.3 Proximity Search

We have seen above how the Input-Restricted Problem is a powerful tool for designing enumeration algorithms, but sometimes it cannot provide an efficient solution. It is natural to ask: can we go beyond it, and obtain polynomial delay even when the IRP has exponentially many solutions?

The answer is "sometimes yes", and one of the techniques to do this is called *Proximity Search* [23]. The basic idea is simple: if the number of solutions to the input-restricted problem $IRP(S, v)$ is exponential, this corresponds to S having an exponential out-degree in the solution graph; we then want to ignore some of the edges, and still prove that the solution graph is strongly connected.

The resulting algorithm is identical in structure to Algorithm 8, with the only difference being some specific function NEIGHBORS(S) in place of IRP(S, v) to produce new solutions (i.e., out-neighbors of S). The complexity relies on finding an efficient NEIGHBORS(S) function and proving that the resulting solution graph is strongly connected.

5.3.1 Completeness

To prove strong connectivity we want to say that, starting from any solution S, and traversing edges of the solution graph, we will eventually find a path to any other solution S^*. The Input-Restricted Problem strategy, on hereditary properties, relies on intersection: for any pair S, S^*, there exists a v for which IRP(S, v) finds an S' such that $|S' \cap S^*| > |S \cap S^*|$, and this tells us S' is the next step in the path towards S^*.

A key step of proximity search is replacing the monotone increase in intersection with a weaker, more general, and possibly problem-specific condition, that is called *proximity*, and denoted as $S \tilde{\cap} S^*$. In essence, a proximity search algorithm requires:

- An arbitrary starting solution S
- A function $S \tilde{\cap} T$ to determine proximity that, for any fixed T, is maximized when and only when $S = T$.[8]
- A function NEIGHBORS(S) such that, for any other solution S^*, there is $S' \in$ NEIGHBORS(S) for which $|S' \tilde{\cap} S^*| > |S \tilde{\cap} S^*|$.

The completeness then follows from the same logic above, as NEIGHBORS(S) always has an S' that gets "one step closer" to T. By the same logic as Algorithm 8, if NEIGHBORS(S) takes only polynomial time, then the enumeration has polynomial delay.

Just like Algorithm 8, we also need to check whether any solution was already found, to ignore duplicates, and that means again storing all solutions in a dictionary and use exponential space. We'll see later how it is sometimes possible to overcome this.

However, rather than solving the problem, we now simply formalized the requirement. Next we introduce a useful technique to actually design proximity search algorithms.

5.3.2 Canonical reconstruction

Canonical reconstruction, formalized in [19], is an effective method to implement proximity search.

[8] Note that we don't care about the efficiency of this function, as it is used in the proofs and never in the algorithm!

The key ingredient is defining, for each solution S, a *solution order* $s_1, s_2, \ldots, s_{|S|}$. This order will be problem-specific, and ideally allows us to exploit the problems' structure to our advantage. Then, canonical reconstruction defines proximity as follows:

▶ **Definition 13** (Proximity in canonical reconstruction). *Given two solutions S and T, let $t_1, \ldots, t_{|T|}$ be the solution order of T. Then $S\tilde{\cap}T$ is defined as the elements in the longest prefix $t_1, \ldots, t_i$ of this order that is fully contained in S.*

A couple observations are in order:

- $S\tilde{\cap}S = S$, as S contains all elements in its own solution order.
- $S\tilde{\cap}T \neq S \cap T$, and in particular if $t_1 \notin S$, then $S\tilde{\cap}T = \emptyset$. Indeed S may contain several elements of T, but we only care about those who align into a prefix of the solution order of T.
- $S\tilde{\cap}T \neq T\tilde{\cap}S$, since by definition $T\tilde{\cap}S$ will instead consider the longest prefix of the solution order of S that is fully contained in T.

Finally, given S and T with $S\tilde{\cap}T = \{t_1, \ldots, t_{i-1}\}$, we call t_i the *canonical extender* for the pair S, T, that is, the earliest element in the solution order of T that is missing from S.

The directive, in canonical reconstruction, is finding a maximal solution S' in $G[S \cup \{t_i\}]$ that contains $\{t_1, \ldots, t_{i-1}, t_i\}$. S' may very well have a smaller intersection with T than S, but its proximity will have increased by at least one.

This is by no means a conclusive guide as, again, canonical reconstruction relies on finding a problem-specific solution order and exploiting its structure. However, many graph properties are associated with specific decompositions that induce an order, and this often turns out to be a powerful way to exploit these decompositions.

For completeness, to showcase the technique, we give here the example of *Maximal Induced Chordal Subgraphs*. More examples can be found in [19].

5.3.3 Maximal Induced Chordal Subgraphs

A graph is *chordal* if it has no induced cycle longer than 3, i.e., any longer cycle has a shortcut edge (chord). Here we want to use canonical reconstruction to enumerate all maximal $S \subseteq V$ such that $G[S]$ is chordal.

Observe that this is a hereditary property, since any induced subgraph of a chordal subgraph graph S cannot introduce a new induced cycle: any shortcut edge in S would be preserved in the induced subgraph. Moreover, as shown in [19], the Input-Restricted Problem for this property can have exponentially many solutions, so it cannot be used to achieve polynomial delay.

We will use two key properties of chordal graphs:

- A chordal graph $G = (V, E)$ has $O(|V|)$ maximal cliques.
- A graph is chordal if and only if it allows a *perfect elimination order*.

A *perfect elimination order* is an elimination order of the vertices obtained by iteratively removing *simplicial* vertices, where a vertex is simplicial if its neighbors form a clique.

Given a graph S, we define its solution order as a *reversed perfect elimination order*, that is, an order $s_1, \ldots, s_{|S|}$ where, for each s_i, the neighbors of s_i preceding it in the order $(N(s_i) \cap \{s_1, \ldots, s_{i-1}\})$ form a clique.

Now, take any pair of solutions S, T, where $t_1, \ldots, t_{|T|}$ is the solution order of T, and let $S\tilde{\cap}T = \{t_1, \ldots, t_{i-1}\}$. As described above, we call t_i the *canonical extender* for S, T, and our goal will be, given S, to produce a solution S' containing $\{t_1, \ldots, t_{i-1}, t_i\}$. It is crucial to observe that *we do not know* T. The NEIGHBORS(S) function must be able to satisfy such a

condition for any of the (exponentially many) other solutions T, even though NEIGHBORS(S) must take polynomial time, so it is only allowed to compute polynomially many solutions. The good news is that we can simply try to use every vertex in V as canonical extender, so eventually we will always consider the correct t_i.

This is where the structure of the problem unravels: we know that $\{t_1, \ldots, t_{i-1}\} \subseteq S$, and crucially, we designed the solution order so that $\{t_1, \ldots, t_{i-1}\} \cap N(t_i)$ is a clique, meaning that $\{t_1, \ldots, t_{i-1}\} \cap N(t_i)$ is contained in a maximal clique of S; let this clique be Q. With knowledge of t_i and Q, we can produce a suitable S' as follows:

$$S' \leftarrow \text{COMPLETE}((S \setminus N(t_i)) \cup Q \cup \{t_i\})$$

Observe that S' must contain all of $S \tilde{\cap} T = \{t_1, \ldots, t_{i-1}\}$ because the vertices that were not neighbors of t_i were never removed from S, and the vertices that were neighbors of t_i were added by including Q. Finally, applying the COMPLETE function gives us a maximal solution, but does not remove any vertex.

Moreover, observe that S' is chordal: $(S \setminus N(t_i)) \cup Q$ is a subset of S, so it is chordal because this is a hereditary property, and the further addition of t_i does not break chordality, because t_i is simplicial, so $(S \setminus N(t_i)) \cup Q \cup \{t_i\}$ allows a perfect elimination ordering starting from t_i. By definition, applying the COMPLETE function does not break chordality.

The final piece of the puzzle is the fact that, while we do not know Q, we know that S has only $O(|S|)$ maximal cliques, so we can indeed try all possible combinations of t_i and Q, obtaining just $O(|V| \cdot |S|)$ solutions.

Specifically, NEIGHBORS(S) will produce these solutions:

- For each $v \in V \setminus S$
- For each maximal clique Q of S
- Return COMPLETE($(S \setminus N(t_i)) \cup Q \cup \{t_i\}$)

From what observed above, for any possible choice of T, there will be one correct choice of v and Q such that the S' obtained has $|S' \tilde{\cap} T| > |S \tilde{\cap} T|$. Again, observe how there are exponentially many possible T, and yet a polynomial number of S' is sufficient to satisfy the condition.

The result is a polynomial-delay algorithm for enumerating Maximal Induced Chordal Subgraphs.

5.3.4 Extensions and alternatives

Canonical reconstruction has one more advantage: [19] shows that it is sometimes possible to induce a parent-child relationship by adapting the parent-child relationship designed in [20] on connected-hereditary (commutable) properties. We remind the reader to [19] for the details, but it is remarkably easy to determine when this is applicable: as a rule-of-thumb, it is possible to induce a parent-child relationship and perform proximity search in polynomial space when the solution order of canonical reconstruction is defined via a breadth-first search.

This is *not* the case for the example shown above of chordal subgraphs, which used a reversed perfect elimination order. However, [19] shows how to apply it to several other problems such as bipartite subgraphs, induced trees, and connected acyclic subgraphs.

Moreover, [10] actually showed how to compute a suitable solution order for chordal subgraphs in a BFS-like fashion, and thus produced a proximity search algorithm for maximal chordal subgraphs with polynomial delay and polynomial space.

Finally, we remark how techniques for navigating a solution graph can get arbitrarily interesting and more complex. A great example are *retaliation-free paths* by Cao [12]: in essence, while canonical reconstruction finds a path from S to T by adding the canonical

extender t_i without losing $\{t_1, \ldots, t_{i-1}\}$, retaliation-free paths work on a similar logic but with a key difference: they consider a solution order for T, but allow the addition of t_i to remove elements from $\{t_1, \ldots, t_{i-1}\}$, as long as we can prove that, along the path, the elements sacrificed will be added again without "retaliation", i.e., without removing t_i again. Proving the existence of retaliation-free paths is not straightforward, but [12] successfully uses them to achieve polynomial delay on multiple enumeration problems for which the input-restricted problem is not solvable in polynomial time.

6 Amortization analysis

In Section 4, we presented techniques for designing efficient enumeration algorithms, based on the *partition technique*. Algorithms designed using this approach typically achieve polynomial delay and polynomial space. However, when aiming for more efficient algorithms, such as those with (sub-)linear delay or average (sub-)linear delay, a more refined algorithm design is required. In algorithms based on the partition technique, the bottleneck is often the procedure of dividing the problem into subproblems. A straightforward way to improve efficiency is to reduce the cost of this partition procedure. However, achieving (sub-)linear time improvements is challenging.

One example is enumeration problems related to connectivity, such as st-path enumeration. In this problem, partition often involves vertex or edge deletions. After deleting some vertices or edges, determining whether s and t remain in the same connected component is known as the dynamic connectivity problem. Solving this problem in $o(n+m)$ time is non-trivial, where n and m are the number of vertices and edges, respectively.

To reduce the cost of partitioning, amortized analysis can be effectively applied in algorithm design [18,46,47]. Typically, the partition method generates at most two subproblems. However, for certain problems, it is possible to generate more subproblems with nearly the same cost. In such cases, if it can be shown that the cost is proportional to the number of generated subproblems, the average delay can be improved. In enumeration algorithms, rather than focusing solely on reducing the cost for each partition, designing algorithms that consider the relationship between the number of solutions and the cost often leads to simple and efficient algorithms. In this section, we give several such examples.

When discussing algorithms with $o(n+m)$ delay or average $o(n+m)$ delay, as in this section, it is common to consider preprocessing time separately. This is because, for many problems, achieving $o(n+m)$ delay is impossible for inputs with no solutions, leading to special treatment of preprocessing time. In the following algorithms, it is easy to show that the preprocessing time is bounded by $\mathrm{poly}(n)$ time. Thus, we omit the analysis of the preprocessing time.

The correctness of the algorithms presented in this section immediately follows from the partition technique. Thus, we also omit the proof of correctness of the algorithms.

6.1 Amortization by Children and Grandchildren

A typical technique in amortized analysis is to analyze computation time based on the number of children or grandchildren. If the computation time at each recursive call X is $O(|ch(X)| \cdot f(X))$ time, where $ch(X)$ is the set of children of X and f is some function, then the total computation time can be bounded by $O(f(X))$ time. Therefore, designing an algorithm such that its computation time depends on the number of children and grandchildren can often lead to improvements in overall efficiency.

6.1.1 Enumerating all paths starting from s

We consider the problem of enumerating paths with a specified starting vertex but no specified end vertex. For this problem, we can obtain an average constant-time algorithm by using a simple partition approach. We give our algorithm in Algorithm 9. Since Algorithm 9 is a recursive algorithm, it generates a tree structure. Hereafter, we denote it as $\mathcal{T}$ and we call a vertex in $\mathcal{T}$ a *recursive call*.

In this algorithm, each recursive call runs in $O(\Delta)$ time and outputs one path starting from s, where Δ is the maximum degree of the graph. Therefore, an immediate analysis shows that the average time complexity is $O(\Delta)$ time. However, using amortized analysis, we can show that this algorithm actually achieves average constant delay.

For each recursive call X, the running time of X is $O(\Delta)$. More precisely, X runs in $O(|N(p_k)|)$ time, where p_k is the other endpoint of a current path P. On the other hand, the number of children of X is also $|N(p_k)|$. Using this observation, we analyze the total running time T of the algorithm as follows: $O(\sum_{X \in V(\mathcal{T})} |ch(X)|) = O(|V(\mathcal{T})|)$, where $ch(X)$ is the set of children of a recursive call X. Since the number of recursive calls and the number of paths starting from s are equivalent, the average time complexity is thus average constant delay.

Algorithm 9 s-PATH(G, P). By running s-PATH$(G, (s))$, all paths starting from s can be enumerated.

Input: A graph $G = (V, E)$ and a path $P = (p_1 = s, \ldots, p_k)$
Output: All paths starting from p_k
1 Output P.
2 **foreach** $v \in N(p_k)$ **do** s-PATH$(G[V \setminus \{p_k\}], (p_1, \ldots, p_k, v))$;

6.1.2 Enumerating all trees containing s

As a next example, we propose the enumeration of all trees containing a specified vertex s, as shown in Algorithm 10. In this algorithm, the time complexity of each recursive call is $O(\Delta^2)$ time. More precisely, the running time is $O(\sum_{w \in N[u]} |N(w)|)$ time. Although the notation G/e is used in the algorithm, its definition is as follows. For an edge $e = (u, v)$, we denote with G/e the graph obtained by contracting edge e, that is, by unifying vertices u and v into one. For a set of edges $F = \{e_1, ..., e_k\}$, we write G/F to denote in short $G/e_1 \cdots /e_k$. Since the number of recursive calls and the number of solutions are equivalent, the average time complexity boils down to $O(\Delta^2)$ time in a straightforward analysis.

We again improve the average time complexity using an amortized analysis. In Algorithm 10, the number of children of a recursive call X is the degree of a given vertex s. Let $Y_1, \ldots, Y_{|N(s)|}$ be the set of children of X. When we generate Y_i, contracting an edge $\{s, u\}$, we need $O(|N(u)|)$ time. On the other hand, the number of children of Y_i is at least $|N(u)|$. In other words, the computation time required to generate Y_i is bounded by the number of children of Y_i. The time complexity of each recursive call X is bounded by $O(|ch(X)| + |gch(X)|)$, where $gch(X)$ is the set of grandchildren of X. Hence, $\sum_{X \in V(\mathcal{T})} |ch(X)| + |gch(X)| = O(|V(\mathcal{T})|)$ holds and the average time complexity is average constant delay.

Algorithm 10 Tree(G, s, T).

Input: A graph $G = (V, E)$, a vertex s, a tree T
Output: All trees containing s and T
Output T.
foreach $u \in N(s)$ **do**
  Let w be a new vertex in $G/\{s, u\}$ with incident edges $\delta(s) \cup \delta(u)$, where $\delta(v)$ is the set of edges incident to v.
  Tree$(G/\{s, u\}, w, T \cup \{s, u\})$
  $G \leftarrow G[E \setminus \{s, u\}]$

6.2 Push out amortization

In the previous subsections, we improved the average delay of some algorithms by considering the relationship between the number of children and grandchildren, and the computation time of each recursive call. In principle, this method can be generalized to account for all descendants. However, analyzing the total number of descendants and their computation time makes the analysis extremely complex. To address this, we introduce a technique called *push out amortization* [70]. In push out amortization, if all recursive calls satisfy a condition known as the PO condition, we can distribute the computational cost proportionately across all descendants. As a result, a better upper bound can be given. The definition of the PO condition is as follows: $\alpha T(X) \leq \sum_{Y \in ch(X)} T(Y) + \beta(|ch(X)| + 1)T^*$, where $T(X)$ is the computational time of a recursive call X, T^* is an upper bound of the computation time of a leaf recursive call, and $\alpha > 1$ and $\beta \geq 0$ are constants that do not depend on the input. If all recursive calls satisfy the PO condition, the average delay of the algorithm is $O(T^*)$.

Intuitively, if the PO condition holds at every internal node, it indicates that the computation time for the entire recursion tree is dominated by the computation time of its leaves. Typically, the number of leaves and the number of solutions is equivalent, and thus the average delay is bounded by $O(T^*)$ time.

Informally speaking, one of the advantages of push out amortization is that, when aiming for constant average-delay, a linear time computation in each recursive call is often not a concern. In contrast, when using amortized analysis based on the number of children or grandchildren, proving constant average-delay becomes challenging if each recursive call takes linear time.

In the following, we provide analysis using push out amortization while examining several concrete examples.

6.2.1 Matching Enumeration

Algorithm 11 shows a simple partition-based matching enumeration algorithm. In this algorithm, each recursive call requires $O(m)$ time. We note that, if the maximum degree is at most 2, each recursive call can be done in constant time. Thus, in what follows, we consider recursive calls with the maximum degree more than 2.

From the definition of the big O notation, for each recursive call X, the computation time $T(X)$ is at most cm for some constant c. Hereafter, we assume that $T(X) = cm$. This modification does not increase the time complexity of Algorithm 11. The purpose of this assumption is to simplify the analysis of a lower bound computation time. In push out amortization, a lower bound of computation time is crucial in the context of the PO condition.

We show that the tree structure $\mathcal{T}$ generated by Algorithm 11 satisfies the PO condition. Let X be a recursive call and $ch(X)$ be the set of children of X. Since each recursive call demands $\Omega(m)$ time, the sum of the computation time of children is $c \cdot (m - |N(u)|)$ for some constant c. We denote the child that receives $G[V \setminus \{u\}]$ as the input graph as Y_0. Since the number of children is $|N(u)| + 1$, we obtain $T(Y_0) + \beta(|ch(X)| + 1)T^* \geq c \cdot m$ for some constant c by setting $\beta = c$ since $T(Y_0) \geq c \cdot (m - |N(u)|)$ and $T^* \geq 1$. We consider the remaining part of the PO condition, $\sum_{Y \in ch(X) \setminus \{Y_0\}} T(Y)$. We show that $\sum_{Y \in ch(X) \setminus \{Y_0\}} T(Y) \geq c(m - |N(u)|)$. Let e be an edge in $E \setminus \delta(u)$. Since $|\delta(u)| \geq 3$, $\delta(u)$ has an edge f such that $\{e, f\}$ is a matching. This implies that $ch(X) \setminus \{Y_0\}$ has a child Y' that receives a graph that has e. Therefore, $\sum_{Y \in ch(X) \setminus \{Y_0\}} T(X) \geq c(m - |N(u)|)$ holds and the PO condition holds for all recursive calls. Since T^* is constant, push out amortization proves that Algorithm 11 runs in constant average delay.

Algorithm 11 MATCHINGS(G, M).

Input: A graph $G = (V, E)$ and a matching M
Output: All matchings in G containing M
1 Output M
2 Let u be a vertex in V with maximum degree
3 Let $\delta(u)$ be the set of edges incident to u
4 **foreach** $e = \{u, v\} \in \delta(u)$ **do** MATCHINGS$(G[V \setminus \{u, v\}], M \cup \{e\})$;
5 MATCHINGS$(G[V \setminus \{u\}], M)$

6.2.2 Enumerating connectivity elimination orders

In the previous example, we were able to enumerate matchings in average constant delay, even when each recursive call required linear time. By using push out amortization, we can show that an algorithm can still run in constant average delay even when the computation time of each recursive call exceeds linear time, as long as the instance size received by the children is sufficiently large.

An order of vertices $(v_1, \ldots, v_n)$ is a *connected elimination ordering* if for any $1 \leq i < n$, $G[\{v_{i+1}, \ldots, v_n\}]$ is connected. Let us consider the connected elimination order algorithm presented in Algorithm 12. We note that, for any connected graph, there are at least two vertices u and w such that $G[V \setminus \{u\}]$ and $G[V \setminus \{w\}]$ are connected (as chosen in Line 2). This can be shown using the BC-tree (as defined in Section 4.3.2): indeed, for any tree, removing a leaf does not break the connectivity.

Each recursive call of Algorithm 12 can be performed in linear time, using a linear time biconnected component decomposition algorithm. Still, such a simple algorithm demands $O(nm) = O(n^3)$ time for each recursive call. We show that even if each recursion takes $O(n^3)$ time, Algorithm 12 runs in constant average delay using push out amortization.

Let X be a recursive call and Y_1 and Y_2 be the children of X. From the definition of Y_1 and Y_2, these recursive calls receive a graph with $n - 1$ vertices. Thus, $T(Y_1) + T(Y_2) = 2c(n-1)^3 = 2cn^3 - 6cn^2 + 6cn + 2c$ for some constant c. Since $T(X) = cn^3$, $T(Y_1) + T(Y_2) - \alpha T(X) = c(2 - \alpha)n^3 - 6cn(n-1) + 2c$. By setting $\alpha = 3/2$ and assuming $n \geq 12$, $T(Y_1) + T(Y_2) \geq 3T(X)/2$ and the PO condition hold. Therefore, this algorithm runs in constant average delay as well.

Algorithm 12 $\textsc{Con}(G, \pi)$.

Input: A graph $G = (V, E)$ and a vertex ordering $\pi = (v_1, \ldots, v_k)$
Output: All connected elimination orderings.
1 **if** $V = \emptyset$ **then** Output π. ;
2 Let U be the set of vertices $\{u \in V \mid G[V \setminus \{u\}]$ is connected$\}$.
3 **foreach** $u \in U$ **do** $\textsc{Con}(G[V \setminus \{u\}], (v_1, \ldots, v_k, u))$;

6.2.3 Enumerating all perfect elimination orderings

As another example of an enumeration problem related to ordering, we consider the Perfect Elimination Ordering (PEO) enumeration problem (for fundamental facts about chordal graphs, we refer the reader to [7]). As mentioned in Section 5.3.3, if a graph $G = (V, E)$ is chordal, that is, G has no induced cycle with length at least four, then $V = \{v_1, \ldots, v_n\}$ has an ordering $(v_1, \ldots, v_n)$ such that for any $1 \leq i \leq n$, v_i is simplicial in $G[\{v_i, \ldots, v_n\}]$, called a perfect elimination ordering. It is known that a chordal graph has at least two simplicial vertices [7]. Therefore, the procedure shown in Algorithm 13 correctly works.

Enumeration of simplicial vertices can be done in $O(n^3)$ time. Since the number of simplicial vertices (and thus of children of a recursive call) is at least 2, we can again achieve average constant delay by using push out amortization. Indeed, in each recursive call the computational cost equals $c \cdot n^3$, given by simplicial vertices enumeration, where c is some constant; since $2c(n-1)^3 - cn^3 \geq 0$ holds, the PO condition holds as well.

Algorithm 13 $\mathrm{PEO}(G, \pi)$.

Input: A chordal graph $G = (V, E)$ and a vertex ordering $\pi = (v_1, \ldots, v_k)$
Output: All perfect elimination orderings
1 **if** $V = \emptyset$ **then** Output π ;
2 Let S be the set of simplicial vertices in G
3 **foreach** $u \in S$ **do** $\mathrm{PEO}(G[V \setminus \{u\}], (v_1, \ldots, v_k, u))$;

6.2.4 Spanning tree enumeration

Finally, we introduce an algorithm that takes advantage of the ability to use linear time in each recursive call. Specifically, we present an amortized constant-time algorithm for enumerating spanning trees.

In spanning tree enumeration, we partition the spanning trees into the ones containing an edge e, and the ones that do not contain e. Note that if e is a bridge of G (i.e., its removal disconnects the graph), then all spanning trees contain e. In such a case, the number of subproblems is just one, and it is thus not easy to satisfy the PO condition.

Since we want to avoid this situation, we can enumerate all bridges, which takes linear time using Tarjan's bridge enumeration algorithm [64], and contract them. Moreover, to further simplify the algorithm, hereafter, we assume that G can have parallel edges due to edge contraction (that is, it is a multigraph), and we keep as an invariant that it has no bridges, that is, it is two-edge connected. Additionally, when G has edges parallel to e, we can generate subproblems easily. We denote that the set of edges between u and v as $E_{u,v}$.

In our algorithm, to ensure a sufficient number of children, if B is the set of all bridges of $G[E \setminus \{e\}]$, we generate $|B|$ children. For each edge $f \in B \cup \{e\}$, we consider the following subproblem: enumerating all spanning trees in $G[E \setminus \{f\}]/((B \cup \{e\}) \setminus \{f\})$. That is, we

consider the graph where edge f was removed, and here contract all the edges in $B \cup \{e\}$ except for f. Such subproblems can be generated in linear time; see Algorithm 14 for the details. Thus, the time complexity of each recursive call is $O(n+m)$ time. Finally, we show that this algorithm satisfies the PO condition. When we contract an edge e, the number of edges in G is reduced by $|E_{u,v}|$. However, since the number of children is at least $|E_{u,v}|$, we can amortize this cost using the factor $\beta(ch(X)+1)$ in the PO condition. Similarly, each recursive call has at least $|B|$ children. Finally, since each edge in G is not contained in $E_{u,v}$ and B is contained in at least two subproblems, this algorithm satisfies the PO condition and runs in constant average delay.

Algorithm 14 SPANNING(G, F).

Input: A two-edge connected multigraph $G = (V, E)$ with no self-loops and a forest F
Output: All spanning trees containing F.

```
if F is a spanning tree of G then Output F. ;
Let u and v be vertices such that an edge {u, v} is non-bridge
Let E_{u,v} be the set of non-bridge edges in G between u and v and B be the set of
  bridges in G[E \ {e}]
if B ≠ ∅ then
    Spanning(G/{e}, F ∪ {e})
    foreach f ∈ B ∪ {e} do
      Spanning(G[E \ {f}]/((B ∪ {e}) \ {f}), F ∪ ((B ∪ {e}) \ {f})) ;
else
    foreach f ∈ E_{u,v} do Spanning(G[E \ (E_{u,v} \ {f})]/{f}, F ∪ {f}) ;
    Spanning(G[E \ E_{u,v}], F)
```

6.3 Geometric amortization: The gap between delay and average delay

In the analysis of average delay, we often encounter unintuitive results due to the exponential number of solutions compared to the input size. This arises because the majority of the recursive calls are performed over inputs of constant size. When improving the average delay, focusing on the relationship between the number of solutions and the computation time often leads to tighter upper bounds on the overall time complexity. However, this approach does not easily extend to delay. Indeed, the class of polynomial delay solvable problems and the class of average polynomial delay solvable problems are distinct under $\mathsf{P} \neq \mathsf{NP}$, although the separation is slightly artificial. For more about this, see Proposition 3.10 from [58].

To bridge the gap between these two classes, the class of problems solvable with linear incremental delay is proposed as an intermediate complexity class. An algorithm runs in linear incremental delay if the time required to output the i-th solution is bounded by $O(i \cdot \mathrm{poly}(n))$. We introduce here *geometric amortization* [14], which allows us to prove an important result: The class of problems that can be solved with incremental polynomial delay and polynomial space is equivalent to the class of problems solvable with polynomial delay and polynomial space. The key idea behind geometric amortization is to run multiple instances of the incremental polynomial delay algorithm in parallel on multiple machines. By carefully scheduling these machines, we can ensure that solutions are output in polynomial delay and polynomial space. Capelli and Strozecki published a demo illustrating this idea,

which may help in understanding the core concept [13, 14]. Since the space usage only increases proportionally to the number of parallel machines, we can achieve polynomial delay while maintaining polynomial space complexity.

One of the applications of geometric amortization is converting enumeration algorithms with cardinality constraints to k-best or ranked enumeration. *Ranked enumeration* is the problem of outputting solutions in order of their objective function values from best to worst [35]. A typical technique to design ranked enumeration algorithms is to combine partition techniques with optimization algorithms [48]. This approach typically demands exponential space. However, by using geometric amortization, we can obtain polynomial-delay and polynomial-space ranked enumeration algorithms, if we have an incremental polynomial-delay enumeration algorithm with cardinality constraints [45].

7 Hardness of Enumeration Problems

In the previous sections, we have focused on methods for constructing efficient enumeration algorithms. However, just as with ordinary search problems, there is no guarantee that an efficient algorithm always exists. In this section, we give a brief introduction to how to deal with the "hardness" of enumeration algorithms. Moreover, we also discuss fixed-parameter tractability and approximation. This topic is currently an active area of research [30].

One of the most important open questions in the enumeration field is whether there exists an output polynomial time algorithm for all minimal dominating set enumeration (Dom-Enum for short). A *dominating set* of a graph is a set of vertices D such that each node of the graph is either in D, or is a neighbor of a node of D. Although such algorithms have been proven to exist for many special cases, including bipartite graphs, the problem remains open for general graphs, comparability graphs, and unit-disk graphs. As in the case of showing the NP-completeness of a decision problem, if we can solve an enumeration problem that is "equivalent" to Dom-Enum, then we can also solve Dom-Enum. In that sense, introducing the notion of reduction is crucial. Currently, the following notion is adopted:

▶ **Definition 14** (E.g. [61]). *Let A and B be two enumeration problems. We say that there is a* polynomial-time delay reduction *from A to B if there exist polynomial-time computable functions f and g such that:*

1. *f is a mapping from an instance of A to a set of instances of B.*
2. *g is a mapping from a solution of B to a set of solutions of A.*
3. $\bigcup_{s \in B(f(x))} g(s) = A(x)$.
4. $\left| \left\{ s \in B(f(x)) \;\middle|\; g(s) = \emptyset \right\} \right| = p(|x|)$.
5. *For any $s, s' \in B(f(x))$, $g(s) \cap g(s') = \emptyset$.*

By using such reductions, it has been shown that Dom-Enum is equivalent to the enumeration of minimal transversals[9] (Hyper-Trans, for short) in the sense that if one can be solved in polynomial delay, then the other also can be also solved in polynomial delay [39]. Furthermore, it is also known that the dualization of monotone boolean functions (Dualization, for short) is equivalent to Hyper-Trans. In other words, problems that arise in different contexts – such as graphs, set families, and logical formulas – turn out to be equivalent to each other.

[9] A hypergraph transversal is a set of vertices that intersects every hyperedge

7.1 NP-hard Enumeration Problems

Although it is unknown whether Dualization can be solved in output polynomial time, there is a quasi-polynomial time algorithm for Dualization [33]. Do harder enumeration problems exist? One obvious example is enumeration of the solutions of an NP-hard problem, e.g., the enumeration of minimum solutions of Vertex Cover.

Another example is the problem of enumerating all minimal dicuts (Dicuts, for short). Let D be a strongly connected directed graph. A set of edges F is called a *directed cut* or *dicut* if removing F from D destroys its strong connectivity. A dicut F is *minimal* if there is no proper subset F' of F such that F' is a dicut. It is obvious that finding one minimal dicut is easy. However, enumerating more minimal dicuts is not trivial. Indeed, Dicuts is known to be NP-hard [41]. More precisely, given a family $\mathcal{F}$ of minimal dicuts for D, deciding whether there is a minimal dicut not contained in $\mathcal{F}$ is NP-complete. Similarly to Dicuts, enumeration of inclusion-wise minimal separators, a minimal set of edges that satisfies a certain connectivity requirement, in a directed graph is also NP-hard [8].

Techniques for proving NP-hardness of an enumeration problem A often involve constructing an instance I of A that corresponds to an instance I' of some NP-complete problem. Typically, obtaining several solutions of I is easy. Thus, we design I such that it contains only a polynomial number of "trivial" solutions, and if an enumeration algorithm finds a non-trivial solution, then this implies that I' has a feasible solution. With this setup, NP-hardness can often be established.

7.2 Introduction to parameterized enumeration

In the previous section, we talked about problems for which finding a single solution is easy, but enumerating all solutions is difficult. In this section and the next, we introduce two approaches to deal with problems for which it is difficult even to obtain one solution: *parameterized enumeration* and *approximate enumeration*.

Parameterized enumeration is an approach where, in addition to the usual enumeration problem, we also take into account a parameter related to the problem. An example is the following: output all minimal vertex covers (i.e., an inclusion-minimal set of vertices such that all edges of the graph have at least one endpoint in the cover) of size at most k, treating k as a parameter. As shown in later, we can "efficiently" solve this problem. Now, we first need to define the efficiency of enumeration for a problem with a parameter. In the same way that we define the FPT class for parameterized search problems, we define DelayFPT for parameterized enumeration problems, as follows:

▶ **Definition 15** (DelayFPT, [26]). *Let Π be an enumeration problem with parameter k. Then, an enumeration algorithm $\mathcal{A}$ for Π is a* DelayFPT*-algorithm if there exists a computable function $t : \mathbb{N} \to \mathbb{N}$ and a polynomial p such that for every instance x, $\mathcal{A}$ outputs all the solutions of x with delay at most $t(k)p(|x|)$.*

In the same work, we can also find the definitions of IncFPT and OutputFPT [26].

Meier's Habilitation thesis [51] studies the topic of parameterized enumeration in detail. Subsequently, Golovach et al. [36] introduced the notion of a *fully-polynomial enumeration kernel*. Intuitively, an enumeration kernel is a framework in which, given an instance of a parameterized enumeration problem, we transform it into a smaller instance with respect to the parameter, enumerate the solutions of this smaller instance, and then reconstruct the solutions for the original instance from those solutions. The algorithm that performs the transformation is called the *kernelization algorithm* $\mathcal{K}$, and the algorithm that reconstructs solutions for the original instance is called the *solution-lift algorithm* $\mathcal{L}$.

▶ **Definition 16.** *For an enumeration problem* Π *with parameter* k*, if* $\mathcal{K}$ *and* $\mathcal{L}$ *satisfy the following conditions (1) and (2), the pair is called a* fully-polynomial enumeration kernel.

1. *For every instance* I*,* $\mathcal{K}$ *outputs in time polynomial in* $|I| + k$ *an instance* J *of* Π *with parameter* k' *such that* $|J| + k' \leq f(k)$ *for some computable function* f*.*
2. *For every solution* s *of* J*,* $\mathcal{L}$ *outputs with delay polynomial in* $|I| + |J| + k + k'$ *a set of solutions* $S_s \subseteq \textsc{Sol}(I)$ *such that* $\{S_s \mid s \in \textsc{Sol}(J)\}$ *is a partition of* $\textsc{Sol}(I)$*.*

Interestingly, the concept of a fully-polynomial enumeration kernel characterizes the existence of a polynomial-delay algorithm. More specifically, Golovach et al. [36] proved that there is a polynomial-delay algorithm for an enumeration problem if and only if there is a fully-polynomial enumeration kernel for the problem whose the size of the output instance is constant.

7.3 Introduction to approximate enumeration

Among NP-hard enumeration problems, the most obviously NP-hard ones are the problems where even finding one solution is already NP-hard, for instance the enumeration of subgraphs with cardinality constraints, such as k-best or ranked enumeration problems. For this problems, enumeration is trivially NP-hard.

In the field of optimization algorithms, in such cases, we often develop algorithms that allow for some error in the output, or permit an exponential computation time with respect to a certain parameter. As mentioned in the previous section, several approaches, such as FPT delay algorithms, have been proposed [24, 25]. In this section, we introduce *approximate enumeration* algorithms, which constitute an alternative approach for addressing NP-hard enumeration problems. To the best of authors' knowledge, the paper [29] is the first to apply an approximation algorithm approach to enumeration problems, including those involving NP-hard optimization problems. This approach is an approximate version of ranked enumeration. Let f be an objective function, and let us consider a minimization problem: that is, we want to enumerate all solutions in order $(S_1, \ldots, S_N)$ such that $f(S_i) \leq f(S_j)$ for any $1 \leq i < j \leq N$. An order of solutions $(S'_1, \ldots S'_N)$ is called *θ-approximate order* if for any $1 \leq i < j \leq N$, $f(S'_i) \leq \theta \cdot f(S'_j)$. Additionally, Kobayashi et al. have proposed a relaxation based on an objective value rather than on an ordering [44]. These algorithms are called *θ-approximation algorithms*. If we allow such a relaxation, we can find one solution in polynomial time.

Research on algorithms in this area is also largely based on two approaches: partition techniques and solution graph-based algorithms [1, 42]. In solution graph-based algorithms, enumeration problems that add objective function constraints to input-restricted problems play a crucial role, similar to maximal solution enumeration algorithms [44].

8 Conclusions

In this work, we provided a survey of the relatively recent but ever growing topic of output-sensitive enumeration algorithm design for graph problems. We have presented the most common techniques of the area, mostly divided between partition-type algorithms and solution graph-based techniques. We have also provided notions of amortization analysis, showing how a careful study of the structure of the algorithm can lead to low average delay. Finally, we have introduced the concept of hardness for enumeration problems, with some preliminary notions of parameterized and approximate enumeration.

References

1 Zahi Ajami and Sara Cohen. Enumerating minimal weight set covers. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 518–529, 2019. doi:10.1109/ICDE.2019.00053.

2 Eralp A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM J. Comput.*, 2(1):1–6, 1973. doi:10.1137/0202001.

3 David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, March 1996. doi:10.1016/0166-218x(95)00026-n.

4 Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe data structures for text indexing. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 199–213. SIAM, 2020. doi:10.1137/1.9781611976007.16.

5 Etienne Birmelé, Pierluigi Crescenzi, Rui A. Ferreira, Roberto Grossi, Vincent Lacroix, Andrea Marino, Nadia Pisanti, Gustavo Akio Tominaga Sacomoto, and Marie-France Sagot. Efficient bubble enumeration in directed graphs. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012*, pages 118–129, 2012. doi:10.1007/978-3-642-34109-0_13.

6 Etienne Birmelé, Rui A. Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 1884–1896, 2013. doi:10.1137/1.9781611973105.134.

7 Jean R. S. Blair and Barry Peyton. An introduction to chordal graphs and clique trees. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 1–29, New York, NY, 1993. Springer New York.

8 Caroline Brosse, Oscar Defrain, Kazuhiro Kurita, Vincent Limouzy, Takeaki Uno, and Kunihiro Wasa. On the hardness of inclusion-wise minimal separators enumeration. *Inf. Process. Lett.*, 185:106469, March 2024. doi:10.1016/j.ipl.2023.106469.

9 Caroline Brosse, Aurélie Lagoutte, Vincent Limouzy, Arnaud Mary, and Lucas Pastor. Efficient enumeration of maximal split subgraphs and induced sub-cographs and related classes. *Discrete Applied Mathematics*, 345:34–51, 2024. doi:10.1016/J.DAM.2023.10.025.

10 Caroline Brosse, Vincent Limouzy, and Arnaud Mary. Polynomial Delay Algorithm for Minimal Chordal Completions. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2022.33.

11 Filippo Brunelli, Alessio Conte, Roberto Grossi, and Andrea Marino. Output-sensitive enumeration of maximal cliques in temporal graphs. *Discrete Applied Mathematics*, 369:66–77, 2025. doi:10.1016/j.dam.2025.02.025.

12 Yixin Cao. Enumerating maximal induced subgraphs, 2020. arXiv:2004.09885.

13 Florent Capelli and Yann Strozecki. https://florent.capelli.me/coussinet/. Accessed: 2025-02-20.

14 Florent Capelli and Yann Strozecki. Geometric Amortization of Enumeration Algorithms. In Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté, editors, *40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*, volume 254 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:22, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2023.18.

15 Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *J. Comput. Syst. Sci.*, 74(7):1147–1159, 2008. doi:10.1016/J.JCSS.2008.04.003.

16 Alessio Conte, Roberto Grossi, Mamadou Moustapha Kanté, Andrea Marino, Takeaki Uno, and Kunihiro Wasa. Listing induced steiner subgraphs as a compact way to discover steiner trees in graphs. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany*, volume 138 of *LIPIcs*, pages 73:1–73:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.MFCS.2019.73.

17 Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giulia Punzi. Beyond the best theorem: Fast assessment of eulerian trails. In *Fundamentals of Computation Theory: 23rd International Symposium, FCT 2021, Athens, Greece, September 12–15, 2021, Proceedings 23*, pages 162–175. Springer, 2021. doi:10.1007/978-3-030-86593-1_11.

18 Alessio Conte, Roberto Grossi, Andrea Marino, and Romeo Rizzi. Enumerating cyclic orientations of a graph. In Zsuzsanna Lipták and William F. Smyth, editors, *Combinatorial Algorithms - 26th International Workshop, IWOCA 2015, Verona, Italy, October 5-7, 2015, Revised Selected Papers*, volume 9538 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2015. doi:10.1007/978-3-319-29516-9_8.

19 Alessio Conte, Roberto Grossi, Andrea Marino, Takeaki Uno, and Luca Versari. Proximity search for maximal subgraph enumeration. *SIAM Journal on Computing*, 51(5):1580–1625, 2022. doi:10.1137/20M1375048.

20 Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Listing maximal subgraphs satisfying strongly accessible properties. *SIAM J. Discrete Math.*, 33(2):587–613, 2019. doi:10.1137/17M1152206.

21 Alessio Conte, Mamadou Moustapha Kanté, Yota Otachi, Takeaki Uno, and Kunihiro Wasa. Efficient enumeration of maximal k-degenerate induced subgraphs of a chordal graph. *Theoretical Computer Science*, 818:2–11, 2020. doi:10.1016/J.TCS.2018.08.009.

22 Alessio Conte and Etsuji Tomita. On the overall and delay complexity of the CLIQUES and bron-kerbosch algorithms. *Theor. Comput. Sci.*, 899:1–24, 2022. doi:10.1016/J.TCS.2021.11.005.

23 Alessio Conte and Takeaki Uno. New polynomial delay bounds for maximal subgraph enumeration by proximity search. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 1179–1190, New York, NY, USA, 2019. ACM. doi:10.1145/3313276.3316402.

24 Nadia Creignou, Raïda Ktari, Arne Meier, Julian-Steffen Müller, Frédéric Olive, and Heribert Vollmer. Parameterized enumeration for modification problems. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 524–536, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-15579-1_41.

25 Nadia Creignou, Raïda Ktari, Arne Meier, Julian-Steffen Müller, Frédéric Olive, and Heribert Vollmer. Parameterised enumeration for modification problems. *Algorithms*, 12(9):189, 2019. doi:10.3390/A12090189.

26 Nadia Creignou, Arne Meier, Julian-Steffen Müller, Johannes Schmidt, and Heribert Vollmer. Paradigms for Parameterized Enumeration. *Theory Comput. Syst.*, 60(4):737–758, May 2017. doi:10.1007/s00224-016-9702-4.

27 David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998. doi:10.1137/S0097539795290477.

28 David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *Algorithms and Computation: 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I 21*, pages 403–414. Springer, 2010. doi:10.1007/978-3-642-17517-6_36.

29 Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/375551.375567.

30 Henning Fernau, Petr Golovach, Marie-France Sagot, et al. Algorithmic enumeration: Output-sensitive, input-sensitive, parameterized, approximative (dagstuhl seminar 18421). *Dagstuhl Reports*, 8(10):63–86, 2019. doi:10.4230/DAGREP.8.10.63.

31 Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM (JACM)*, 56(5):25, 2009.

32 Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2010. doi:10.1007/978-3-642-16533-7.

33 Michael L. Fredman and Leonid Khachiyan. On the Complexity of Dualization of Monotone Disjunctive Normal Forms. *J. Algorithm. Comput. Technol.*, 21(3):618–628, November 1996. doi:10.1006/jagm.1996.0062.

34 Leslie Ann Goldberg. Efficient algorithms for listing combinatorial structures, 1992. PhD dissertation thesis.

35 Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Optimizing and parallelizing ranked enumeration. *Proc. VLDB Endow.*, 4(11):1028–1039, August 2011. doi:10.14778/3402707.3402739.

36 Petr A Golovach, Christian Komusiewicz, Dieter Kratsch, and Van Bang Le. Refined notions of parameterized enumeration kernels with applications to matching cut enumeration. *J. Comput. System Sci.*, 123:76–102, February 2022. doi:10.1016/j.jcss.2021.07.005.

37 David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988. doi:10.1016/0020-0190(88)90065-8.

38 Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975. doi:10.1137/0204007.

39 Mamadou Moustapha Kanté, Vincent Limouzy, Arnaud Mary, and Lhouari Nourine. On the Enumeration of Minimal Dominating Sets and Related Notions. *SIAM J. Discrete Math.*, 28(4):1916–1929, January 2014. doi:10.1137/120862612.

40 Toshinobu Kashiwabara, Sumio Masuda, Kazuo Nakajima, and Toshio Fujisawa. Generation of maximum independent sets of a bipartite graph and maximum cliques of a circular-arc graph. *J. Algorithms*, 13(1):161–174, 1992. doi:10.1016/0196-6774(92)90012-2.

41 Leonid Khachiyan, Endre Boros, Khaled Elbassioni, and Vladimir Gurvich. On Enumerating Minimal Dicuts and Strongly Connected Subgraphs. *Algorithmica*, 50(1):159, October 2007. doi:10.1007/s00453-007-9074-x.

42 Benny Kimelfeld and Yehoshua Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 173–182, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1142351.1142377.

43 Donald E. Knuth. The art of computer programming, volume 4, pre-fascicle 2a: A draft of section 7.2. 1.1: Generating all n-tuples, 2001.

44 Yasuaki Kobayashi, Kazuhiro Kurita, and Kunihiro Wasa. Efficient constant-factor approximate enumeration of minimal subsets for monotone properties with weight constraints. *Discret. Appl. Math.*, 361:258–275, 2025. doi:10.1016/J.DAM.2024.10.014.

45 Yasuaki Kobayashi, Kazuhiro Kurita, and Kunihiro Wasa. Polynomial-delay enumeration of large maximal common independent sets in two matroids and beyond. *Information and Computation*, 304:105282, 2025. doi:10.1016/j.ic.2025.105282.

46 Kazuhiro Kurita, Kunihiro Wasa, Hiroki Arimura, and Takeaki Uno. Efficient enumeration of dominating sets for sparse graphs. *Discret. Appl. Math.*, 303:283–295, 2021. doi:10.1016/J.DAM.2021.06.004.

47 Kazuhiro Kurita, Kunihiro Wasa, Takeaki Uno, and Hiroki Arimura. A constant amortized time enumeration algorithm for independent sets in graphs with bounded clique number. *Theor. Comput. Sci.*, 874:32–41, 2021. doi:10.1016/J.TCS.2021.05.008.

48 Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972. URL: http://www.jstor.org/stable/2629357.

49 Eugene L. Lawler, Jan Karel Lenstra, and Alexander H.G. Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980. doi:10.1137/0209042.

50 Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Algorithm Theory (SWAT 2004)*, volume 3111 of *Lecture Notes in Computer Science*, pages 260–272, Humlebæk, Denmark, July 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-27810-8_23.

51 Arne Meier. *Parametrised enumeration*. Habilitation thesis, Leibniz Universität Hannover, 2020. doi:10.15488/9427.

52 John W. Moon and Leo Moser. On cliques in graphs. *Israel journal of Mathematics*, 3:23–28, 1965.

53 M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-8(3):356–367, 1959. doi:10.1109/TEC.1959.5222697.

54 Giulia Punzi, Alessio Conte, Roberto Grossi, and Andrea Marino. An efficient algorithm for assessing the number of st-paths in large graphs. In *Proceedings of the 2023 SIAM International Conference on Data Mining (SDM)*, pages 289–297. SIAM, 2023. doi:10.1137/1.9781611977653.CH33.

55 Ronald C. Read and Robert E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975. doi:10.1002/NET.1975.5.3.237.

56 Marco Rospocher. On the computational complexity of enumerating certificates of np problems, 2006. PhD dissertation thesis.

57 Frank Ruskey. Combinatorial generation. *Working version (1j-CSC 425/520)*, 2003.

58 Johannes Schmidt. Enumeration: Algorithms and complexity. *Preprint,* available at https://www.thi.uni-hannover.de/fileadmin/forschung/arbeiten/schmidt-da.pdf, 2009.

59 Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.*, 26(3):678–692, 1997. doi:10.1137/S0097539794270881.

60 Yann Strozecki. Enumeration complexity. *Bull. EATCS*, 129, 2019. URL: http://bulletin.eatcs.org/index.php/beatcs/article/view/596/605.

61 Yann Strozecki. Enumeration complexity: Incremental time, delay and space. *CoRR*, abs/2309.17042, 2023. doi:10.48550/arXiv.2309.17042.

62 Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

63 Robert E. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.*, 2(3):211–216, 1973. doi:10.1137/0202017.

64 Robert E. Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161, 1974. doi:10.1016/0020-0190(74)90003-9.

65 James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications ACM*, 13:722–726, 1970. doi:10.1145/362814.362819.

66 Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical computer science*, 363(1):28–42, 2006. doi:10.1016/J.TCS.2006.06.015.

67 Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977. doi:10.1137/0206036.

68 Takeaki Uno. A fast algorithm for enumerating bipartite perfect matchings. In *Algorithms and Computation, 12th International Symposium, ISAAC*, pages 367–379, 2001. doi:10.1007/3-540-45678-3_32.

69 Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms. *National Institute of Informatics (in Japan) Technical Report E*, 4:2003, 2003.

70 Takeaki Uno. Constant time enumeration by amortization. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures - 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*, volume 9214 of *Lecture Notes in Computer Science*, pages 593–605. Springer, 2015. doi:10.1007/978-3-319-21840-3_49.

71 Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979. doi:10.1016/0304-3975(79)90044-6.

72 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SICOMP*, 8(3):410–421, 1979. doi:10.1137/0208032.

73 Luca Versari. A new algorithmic framework for enumerating commutable set properties, 2017. Master Thesis, University of Pisa.

Subsequence-Based Indices for Genome Sequence Analysis

Giovanni Buzzega[1]
Department of Computer Science, University of Pisa, Italy

Alessio Conte
Department of Computer Science, University of Pisa, Italy

Veronica Guerrini
Department of Computer Science, University of Pisa, Italy

Giulia Punzi
Department of Computer Science, University of Pisa, Italy

Giovanna Rosone
Department of Computer Science, University of Pisa, Italy

Lorenzo Tattini
EURECOM, Biot, France
CNRS UMR 7284, INSERM U 1081, Université Côte d'Azur, Nice, France

Abstract

Compact indices are a fundamental tool in string analysis, even more so in bioinformatics, where genomic sequences can reach billions in length. This paper presents some recent results in which Roberto Grossi has been involved, showing how some of these indices do more than just efficiently represent data, but rather are able to bring out salient information within it, which can be exploited for their downstream analysis. Specifically, we first review a recently-introduced method [Guerrini et al., 2023] that employs the *Burrows-Wheeler Transform* to build reasonably accurate phylogenetic trees in an assembly-free scenario. We then describe a recent practical tool [Buzzega et al., 2025] for indexing *Maximal Common Subsequences* between strings, which can enable analysis of genomic sequence similarity. Experimentally, we show that the results produced by the one index are consistent with the expectations about the results of the other index.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases String Indices, Burrows-Wheeler Transform, Maximal Common Subsequences, Sequence Analysis, Phylogeny

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.20

Category Research

Related Version The algorithmic techniques shown in this paper are summarized from [13, 26].
Full Version: `https://doi.org/10.1186/s13015-025-00271-z` [13]
Full Version: `https://doi.org/10.1186/s13015-023-00232-4` [26]

Supplementary Material
Software (Source Code of PHYBWT): `https://github.com/veronicaguerrini/phyBWT2` [26]
Software (Source Code of MCDAG): `https://github.com/giovanni-buzzega/McDag` [13]

Funding *Giovanni Buzzega*: Partially supported by MUR PRIN 2022 project EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data (#2022TS4Y3N), and received funding from the European Union's Horizon 2020 Research and Innovation Staff Exchange programme under the Marie Skłodowska-Curie grant agreement No. 872539.

[1] corresponding author

Alessio Conte: Partially supported by MUR PRIN 2022 project EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data (#2022TS4Y3N).
Veronica Guerrini: Supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 Tuscany Health Ecosystem Spoke 6 CUP B63C2200068007 and I53C22000780001.
Giulia Punzi: Supported by the Italian Ministry of Research, under the complementary actions to the NRRP "Fit4MedRob - Fit for Medical Robotics" Grant (#PNC0000007), and received funding from the European Union's Horizon 2020 Research and Innovation Staff Exchange programme under the Marie Skłodowska-Curie grant agreement No. 872539.
Giovanna Rosone: Partially supported by the Next Generation EU PNRR MUR M4 C2 Inv 1.5 project ECS00000017 Tuscany Health Ecosystem Spoke 6 CUP B63C2200068007 and I53C22000780001, and by project "Hub multidisciplinare e interregionale di ricerca e sperimentazione clinica per il contrasto alle pandemie e all'antibioticoresistenza (PAN-HUB)" funded by the Italian Ministry of Health (POS 2014–2020, project ID: T4-AN-07, CUP: I53C22001300001).

1 Introduction

Sequence analysis is one of the core branches of bioinformatics, and it is arguably one of the most fundamental tasks due to the abundance of genomic sequences enabled by advancements in sequencing technologies. Sequence analysis methods have a huge advantage compared to "in vitro" methods: once a dataset is available, it can be instantly and easily shared with anyone, it does not deteriorate or deplete, and can be analysed repeatedly with just regular computers, without the need of expensive ad-hoc machines. This is one of the reasons why much effort is dedicated to quickly produce new and more powerful sequencers, resulting in larger datasets available to the scientific community, ranging from bacteria and viruses to humans.

The size of genomic data can be daunting, as the length of genomic sequences ranges from thousands (for some viruses, or proteins) to millions (e.g., *E. Coli* genome) or billions (e.g., human genome). An important trade-off is immediately evident: more complex and refined approaches can extract better-quality information from the data, but require more computational resources to be executed, and may be not be applicable to complex organisms. The research community has been advancing in two main directions: developing more sophisticated algorithms, and extending their applicability to increasingly complex data. A key task in the latter direction is optimizing the way data is represented and handled, since just storing sequences in an uncompressed format may already require tens of GB of space. In this scenario, compact indices are extremely valuable tools, that typically employ algorithmic tricks to provide a good trade-off between the size of the index, and ease of access to the data (as well as support for specific queries).

In this paper, our aim is to give an overview of two recent research results concerning subsequence-based indices in bioinformatics, which involve Roberto Grossi both in their past development and in their current investigation of future directions. We experimentally highlight how these indices do more than just represent genomic data: their clever processing of the input enables the extraction of salient information that can be used for sequence analysis application tasks.

Specifically, the first research result we review is a method, called PHYBWT [25, 26], that addresses the problem of phylogenetic inference employing the Burrows-Wheeler Transform (BWT) [11]. The BWT is a text transformation with the remarkable feature of clustering together repeated sequences, a property originally intended to enhance compression, but now widely used in genome indexing algorithms [32]. Phylogenetic inference refers to the

process of reconstructing the evolutionary relationships among species, or more generally, among taxa. The PHYBWT methodology for phylogeny reconstruction uses the BWT of a string collection [7, 35], precisely of a group of sequences representing different taxa, to group related taxa together and to suggest evolutionary relationships. The main features of the PHYBWT tool are its ability to work directly on raw sequencing data in an assembly-free scenario, and the fact that it does not rely on pairwise sequence comparisons, and thus on a distance matrix, but rather compares all the sequences simultaneously and efficiently.

The subsequent research result reviewed in this paper concerns the construction of a deterministic finite automaton (DFA) to efficiently index all *maximal common subsequences* (MCS) of two (or more) input strings. A common subsequence is a sequence of characters that occurs in the same order in all input strings, albeit not necessarily consecutively. An MCS is a common subsequence that is not a subsequence of any other common subsequence. The DFA, implemented as a labelled directed acyclic graph (DAG), is called MCDAG, and was presented in [12, 13]. Even if some previous works provide indices with better worst-case bounds [18, 28], the MCDAG index has been experimentally shown to be more efficient in practice for real-world genomic data, as it is significantly faster to build and typically smaller in size. Preliminary experiments (see Figure 3 from [12]) showed that the distribution of MCS lengths appears to behave differently when comparing very similar or dissimilar genomes.

Aiming to illustrate the potential of the two methods for genomic data analysis, we experimentally observe the MCS lengths distributions on genomic sequences within the same taxonomic group, to get a picture of how these distributions behave on closely related taxa versus more distant ones, using the phylogeny produced by PHYBWT as a guide.

Outline of the paper

The paper is organized as follows. In the rest of the introduction, we will provide a review of the state of the art on both PHYBWT and MCSs. Then, in Section 2 we will provide some technical preliminaries needed for the rest of the paper. Sections 3 and 4 will briefly review the main ideas behind the PHYBWT [26] and MCDAG [13] works, respectively. Finally, Section 5 will present experiments on MCS lengths distributions for sequences from the same phylogenetic trees produced by PHYBWT. The experimental comparison of PHYBWT and MCDAG with their relative state of the art is out of the scope of the present work, and the interested reader can find it in their respective papers.

Related Work: Phylogenetic inference and PhyBWT

Sequence-based phylogeny aims to reconstruct evolutionary relationships between species, or more generally taxa, by comparing their DNA (or protein) sequences. The relationships among taxa are traditionally displayed in a tree-shaped diagram called *phylogenetic tree*, which can be rooted or unrooted. The leaves of the tree represent the contemporary organisms, while internal nodes represent common ancestors from which descendant lineages diverged.

The development of next-generation sequencing technologies in the early 2000s revolutionized this field, enabling researchers to sequence entire genomes quickly and cost-effectively. Such an amount of whole-genome sequencing data has lead to the need of advanced computational algorithms and tools for efficient phylogenetic inference.

Numerous sequence-based methods have emerged and evolved over time in this research field [43]. Most of them rely on a distance matrix by computing the pairwise evolutionary distances between every pair of input sequences representing the taxa. Distance measures are typically based on sequence alignment, and once the distances are obtained, the sequences are no longer utilized in the analysis.

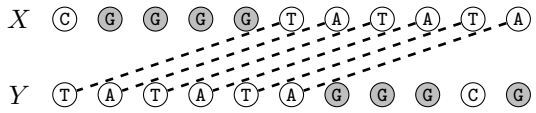


Figure 1 The two strings X, Y shown in the figure have as only LCS the string `TATATA` of length 6, shown with matching dashed edges. The MCS set is instead composed of three strings: `TATATA`, `GGGG`, and `CG`. In LCS-based analysis, the longer LCS prevents us from considering the second-longest MCS, `GGGG` (shaded), as a possible meaningful common pattern.

In 1992, Bandelt and Dress [4] introduced a technique called *split decomposition* that was shown to enhance phylogenetic analysis [5]. Based on a solid mathematical ground [4, 6], the split decomposition involves constructing a set of *splits* (binary partitions of the set of taxa) from a given dissimilarity matrix, each split being weighted by an isolation index that intuitively quantifies the strength of the split on the basis of the dissimilarity values. Given a distance matrix for ℓ taxa, the list of splits is computable in polynomial time (of order ℓ^6). Phylogenies in a tree-shaped form can be constructed by greedily selecting the splits with the highest isolation indices, as long as they are compatible[2]. Compatible splits correspond to a tree structure and, conversely, any tree can be represented by a set of compatible splits. Thus, ideal data gives rise to a phylogenetic tree, whereas phylogenetic networks, which generalize phylogenetic trees, are reconstructed when the splits are weakly compatible (see the tool SplitsTree [29]).

The increasing cost of the alignment task has led to the development of alignment-free approaches to efficiently quantify the dissimilarity between pairs of sequences [46]. Starting from the split decomposition idea, the authors of [44] introduced an alignment-free method called SANS that builds a list of splits and, from that, it infers the phylogeny by using SplitsTree. However, differently from the split decomposition theory, SANS builds the list of splits without relying on a distance matrix, and assigns weights to splits by counting fixed-length substrings shared among the sequences. According to [44], for ℓ taxa represented by sequences of length $\mathcal{O}(n)$ each, SANS runs in $\mathcal{O}(n\ell \log(n\ell))$ time.

The method phyBWT proposed in [26] and reviewed in this paper also belongs to the class of alignment-free approaches that infer phylogenetic relationships without relying on pairwise sequence comparisons. It differs from SANS as it does not build a list of splits, but rather defines a new strategy to draw a phylogenetic tree. Moreover, phyBWT evaluates sequence similarity/dissimilarity considering shared strings of varying length, without fixing a-priori the length of the common substrings. The interested reader can find a direct comparison between SANS and phyBWT in [26].

Related Work: Maximal Common Subsequences

Maximal common subsequences are a generalization of the well-known *Longest Common Subsequences* (LCSs), that is, common subsequences of maximum length: indeed, each LCS is, by definition, an MCS as well. LCSs are well-established in the context of genome sequence alignment [41], and the value of the length of an LCS can be used as a string similarity measure [8]. Still, by only considering the longest such sequences, (slightly) shorter but still relevant alignments might be discarded (see Figure 1). At the same time, going instead to the opposite extreme and considering *all* common subsequences would create too much

[2] A set of splits is compatible if, for every pair of splits, at least one of the four possible intersections between their parts is empty.

redundancy. A reasonable middle ground is thereby provided by MCSs, which can still be exponential in number (as LCSs can be too [24]), but are significantly fewer than all common subsequences.

The (shortest) MCS problem on strings[3] was proposed in [22], where the authors provided a dynamic programming algorithm for finding the shortest MCS, and other related problems. Sakai later provided the first (almost) linear-time algorithm to extract one MCS between two strings [39, 40]. This highlights a key difference from LCS computation, for which there exists a SETH-based quadratic conditional lower bound [1, 9]. When increasing the number of strings, this difference becomes even more pronounced: finding an LCS among an arbitrary number of strings is NP-hard [33], while there is a polynomial-time algorithm for extracting one MCS in the same setting [28]. MCSs were also recently employed as a tool for a parameterized LCS algorithm [10].

In the past years, several works have appeared on the topic of MCSs, more specifically in the direction of MCS enumeration and indexing, with Roberto Grossi contributing to many of them. Indeed, he took part in the first results concerning efficient MCS enumeration between two strings [16, 17], as well as in one of the two independent works that produced the first polynomial-sized indices for MCSs [18, 27]. Finally, as previously mentioned, he was involved in the development of the practical tool described and employed in the present paper [12]. While providing no formal theoretical bounds on the space complexity, experiments on genomic data have shown this algorithm to be more efficient in practice with respect to [18] (see the Experimental Analysis in [12, 13]). Worst-case complexity bounds for constructing McDag, and, more in general, for the minimum size of a DFA representing all MCSs, remain an open problem [13, Conclusions].

MCS problems have been studied for an arbitrary number m of input strings as well. Hirota and Sakai proposed a $O(nm \log n)$-time algorithm for computing one such MCS, where n is the total length of the input strings [28]. The practical indexing tool of Buzzega et al. [12] was also extended to deal with $m > 2$ strings in [13]. Moreover, the problem of efficiently indexing MCSs of an arbitrary number of strings, as well as their enumeration, has recently been shown to be unfeasible in time polynomial in the output size, unless P=NP [14].

2 Preliminaries

We consider a string $X = X[1] \dots X[|X|]$ as a sequence of characters from a finite and ordered alphabet Σ, where $X[i] \in \Sigma$ denotes the character at position i in X and $|X|$ denotes the total number of characters in X. Let $X[i, j]$ denote the *substring* $X[i] \dots X[j]$, for $1 \le i \le j \le |X|$: a substring $X[i, |X|]$ (resp. $X[1, i]$) is called *suffix* (resp. *prefix*) of X, for any $1 \le i \le |X|$. We use special characters $\{\#, \$\}$ as markers delimiting input strings.

We say that string Z is a *subsequence* of X if there exist indices $1 \le i_1 < \dots < i_{|Z|} \le |X|$ such that $X[i_k] = Z[k]$ for $0 < k \le |Z|$. If a subsequence can be mapped on X contiguously, i.e., for all $0 < k \le |Z|$, $i_k = i_1 + k - 1$, then Z is a substring of X. Moreover, Z is a *common subsequence* of strings X and Y if Z is a subsequence of both X and Y (see Figure 1). More specifically, let us call a pair (i, j) a *match* when $X[i] = Y[j]$: letting $1 \le j_1 < \dots < j_{|Z|} \le |Y|$ be the indices such that $Y[j_k] = Z[k]$ for $0 < k \le |Z|$, then each pair (i_k, j_k) is a match (as $X[i_k] = Y[j_k]$) and we say that the pairs $(i_1, j_1), \dots, (i_{|Z|}, j_{|Z|})$ form a *matching* in X and Y, whose corresponding string is Z. We observe that matches induce a partial order, defined

[3] The concept of MCS in a more general form actually first appeared in data mining applications [3], where they were defined over ordered sequences of *itemsets*, instead of over strings. The string setting we consider for MCS can be seen as a special case of this framework, where each itemset is a singleton.

as $(i,j) < (i',j')$ iff $i < i'$ and $j < j'$, which is total if the pairs belong to the same matching; $(i,j) \leq (i',j')$ is analogously defined. A string Z is a *maximal common subsequence* (MCS) of X and Y if there is no string $W \neq Z$ that satisfies both conditions: *(i)* W is a common subsequence of X and Y, and *(ii)* Z is a subsequence of W. The set of all strings that are maximal common subsequences is denoted by $MCS(X,Y)$.

We next introduce some graph notions. A *directed graph* $G = (V,E)$ consists of a set of nodes V and a set of edges $E \subseteq V \times V$, where each edge (u,v) is an ordered pair of nodes that specifies a direction from u to v. Two edges (u,v) and (w,z) are said to be adjacent if $v = w$. A path in G is a sequence of distinct edges, each adjacent to the next. If the path starts at node s and ends at node t, it is called an st-path; it is a cycle when $s = t$. A *DAG* is a directed acyclic graph. Given a node u, the set $N^+(u)$ indicates the out-neighbor nodes v such that $(u,v) \in E$, and the set $N^-(u)$ indicates the in-neighbor nodes v such that $(v,u) \in E$. The out-degree of u is $d^+(u) = |N^+(u)|$, and its in-degree is $d^-(u) = |N^-(u)|$; u is a *source* if $d^-(u) = 0$, and a *sink* if $d^+(u) = 0$. In a *labeled* DAG $G = (V,E,l)$ each node u is associated with a character $l(u) \in \Sigma \cup \{\#,\$\}$. In Section 4 we also consider labeled DAGs in which each node u is associated with a match $m(u) = (i_u, j_u)$.

3 BWT-based Phylogenetic Inference

In this section we review phyBWT, a methodology first introduced in [25] and subsequently refined in [26], for reconstructing a phylogenetic tree bypassing the standard computationally expensive steps of sequence alignment and *de novo* assembly. It can take as input any type of sequence data representing taxa, such as whole-genome sequences and raw sequencing reads.

The approach exploits the inherent combinatorial properties of the extended Burrows-Wheeler Transform (eBWT) [7, 35] to index and detect relevant common substrings of varying length. The common substrings then play a crucial role in building partition trees without performing pairwise comparisons between sequences. This is the primary feature of phyBWT: the tree structure is inferred by comparing all the sequences simultaneously and efficiently, without resorting to a distance matrix.

The second remarkable feature of phyBWT is that, to the best of our knowledge, it is the first approach to apply the properties of the eBWT to the idea of decomposition for phylogenetic inference. By indexing the sequences in the eBWT, we can identify maximal common substrings of varying lengths that are used to group sequences together and to partition groups of taxa based on their shared substrings.

Finally, the worst-case running time of phyBWT is $O(N\ell)$, where ℓ is the number of taxa and N is the total length of all the taxa sequences, using $O(N + \ell^2)$ space.

In the following, we first provide an overview of the preliminary notions. These include the extended Burrows-Wheeler Transform employed for detecting common substrings and the *positional clustering* framework, which overcomes the limitation of a priori fixing the length of the common substrings. Subsequently, we delineate the tree reconstruction methodology of phyBWT according to [26].

3.1 Burrows-Wheeler transform and Common Substrings

The Burrows-Wheeler Transform (BWT) [11] is a well-known reversible transformation that permutes the symbols of a string in such a way that, as a result, the runs of equal symbols tend to increase. In addition to enhancing the performance of memoryless compressors, the BWT plays a crucial role in the development of efficient self-indexing compressed data structures.

The BWT was first extended to a string collection in [35] (eBWT) by sorting cyclic rotations of all the strings in the collection according to a special order, called ω-order. In order to use the lexicographic order rather then the ω-order, in [7], a variant of the eBWT was defined by appending a distinct end-marker symbol to each string and lexicographically sorting the suffixes of all the strings in the collection. In the following, we call ebwt the output string[4] defined in [7], and introduce some auxiliary data structures that allow to detect common substrings in a string collection.

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ be a collection of m strings. We assume that each string $s_i \in \mathcal{S}$ is a sequence of $n_i - 1$ characters from Σ followed by a special end-marker symbol $\$_i$, i.e. $s_i[n_i] = \$_i$, which is lexicographically smaller than any other symbol in Σ and $\$_i < \$_j$, if $i < j$. The total number of characters in $\mathcal{S}$ is $N = \sum_{i=1}^{m} n_i$. The $\mathsf{ebwt}(\mathcal{S})$ string is defined by concatenating the symbols preceding each suffix of the lexicographically sorted list of suffixes of all the strings $s_1, \ldots, s_m$, where each s_i is circular. The *longest common prefix* (LCP) array [34] of $\mathcal{S}$ (denoted by $\mathsf{lcp}(\mathcal{S})$) is the array of length N storing the length of the longest common prefix between any two consecutive suffixes in lexicographically sorted list of suffixes of $s_1, \ldots, s_m$, using the convention that $\mathsf{lcp}(\mathcal{S})[1] = 0$.

Finally, if $\mathcal{S}$ comes in ℓ parts, namely $\mathcal{S} = S_1 \cup S_2 \cup \ldots \cup S_\ell$, where each S_i is a non-empty subset of $\{s_1, \ldots, s_m\}$, and all the subsets are pairwise disjoint, then the *color document array* of $\mathcal{S}$ (denoted by $\mathsf{cda}(\mathcal{S})$) is the array of length N storing the indices of the subsets to which the $\mathsf{ebwt}(\mathcal{S})$ symbols belong. The set $\mathcal{S}$ is omitted if it is clear from the context.

▶ Remark 1. Let $\mathcal{R} \subset \mathcal{S}$. The data structures $\mathsf{ebwt}(\mathcal{R})$, $\mathsf{lcp}(\mathcal{R})$, and $\mathsf{cda}(\mathcal{R})$ can be deduced through a linear scan of the larger $\mathsf{ebwt}(\mathcal{S})$, $\mathsf{lcp}(\mathcal{S})$, and $\mathsf{cda}(\mathcal{S})$, as the relative order of suffixes holds (see [7], cf. also [15]).

One technique to find common substrings of fixed length k in $\mathcal{S}$ is based on the use of LCP-intervals. An *LCP-interval* of lcp-value k is a maximal interval $[i, j]$ such that $\mathsf{lcp}[r] \geq k$ for $i < r \leq j$ (defined slightly differently from [2]); in other words, the interval $[i, j]$ corresponds to suffixes in the lexicographically sorted list that share at least the first k characters. Nevertheless, the length of the common prefix in any LCP-interval could be longer than k, possibly revealing common substrings of greater length.

To overcome the limitation of a priori fixing the length of common substrings in $\mathcal{S}$, the authors of [37] introduced a framework called *positional clustering*. According to this framework, the boundaries of the intervals in the LCP array are data-driven, and not established a-priori by a fixed k. Specifically, the intervals of interest are those enclosed between two "local minima" in the LCP array. In fact, intuitively, a local minimum in the LCP array indicates a shortening of the common prefix. Moreover, to exclude intervals associated with short random prefixes, a minimum prefix length k_m can be established. Formally, an *eBWT positional cluster* is a maximal substring $\mathsf{ebwt}[i, j]$ such that $\mathsf{lcp}[r] \geq k_m$, for all $i < r \leq j$, and none of the indices $i < r \leq j$ is a *local minimum* of the LCP array[5]. By definition, we have that any two different eBWT positional clusters are disjoint.

▶ Remark 2. Each eBWT positional cluster $\mathsf{ebwt}[i, j]$ corresponds to suffixes in the lexicographically sorted list that have a common prefix (i.e., a common substring) of length given by the minimum between $\mathsf{lcp}[i + 1]$ and $\mathsf{lcp}[j]$ (see [37, Theorem 3.3]). Thus, each eBWT positional cluster $\mathsf{ebwt}[i, j]$ corresponds to a substring in $\mathcal{S}$, and the values in $\mathsf{cda}[i, j]$ provides the information about the strings that contain it.

[4] In the literature, the extended transform of [7] is also called multi-string BWT [21] or mdolEBWT [15].

[5] According to [26], an index r is said a local minimum if $\mathsf{lcp}[r - 1] > \mathsf{lcp}[r]$ and $\mathsf{lcp}[r] < \mathsf{lcp}[r + s]$, where $s > 1$ is the number of adjacent occurrences of $\mathsf{lcp}[r]$ from position r. For instance, the local minima of $\mathsf{lcp} = [2, 1, 3, 5, 4, 4, 2, 2, 7]$ are indices 2 and 7, corresponding to LCP values of 1 and 2, respectively.

Remarks 1 and 2 are key to the PHYBWT method, which detects common variable-length substrings of a subset of taxa and uses this information to reconstruct a phylogenetic tree.

3.2 Tree Reconstruction Method

The methodology proposed in [26] reconstructs a tree T through a series of refinement steps performed on groups of taxa.

Formally, we denote the set of leaves as $\mathcal{S} = \{S_1, S_2, \ldots, S_\ell\}$ where each S_i corresponds to a taxon, which could be represented by a single sequence (e.g., genome sequence) or a string collection (e.g., sequencing reads).

The tree T is defined as a *partition tree* of the set $\mathcal{S}$:

- each node of T corresponds to a non-empty set of taxa $S' \subseteq \mathcal{S}$;
- the root of T corresponds to $\mathcal{S}$;
- each leaf of T corresponds to a distinct taxon $S_i \in \mathcal{S}$, and vice-versa;
- for each node corresponding to S', its children form a partition of S'.

We define the operation of *adding a node* to T by a set: a set $S' \subseteq \mathcal{S}$ can be added to T only if it is *compatible*, i.e., if every other node of T corresponds to a set S'' that satisfies one of these conditions: $S'' \subset S'$, $S'' \supset S'$, or $S'' \cap S' = \emptyset$ (i.e. no partial overlap between S'' and S'). If this is the case, there is only one way to add S' to T, namely, S' becomes a child of the smallest set $P \supset S'$ of T (by cardinality), and all the other children of P that are contained in S' become the children of S'. The resulting T is still a partition tree.

We describe the method by first explaining the tree reconstruction procedure, which applies a REFINEMENT procedure iteratively, and then by briefly sketching the inner REFINEMENT algorithm. The rationale of the REFINEMENT algorithm is to group together nodes of T whose associated sequences share variable-length substrings not found in other sequences, and to interpret this fact as a common feature of the group that differentiates it from the others.

Tree reconstruction via the refinement procedure. The key idea is to refine an intermediate partition tree by taking one of its internal nodes and applying the REFINEMENT procedure to the groups of taxa corresponding to its children. Here, we consider REFINEMENT as a blackbox that uses the eBWT and its related data structures to produce a list of compatible subsets, which are new nodes that can be added to the partition tree. This process allows for fine-grained node clustering, by restricting the input data to the sequences of the relevant subtree. This is repeated until all internal nodes in the partition tree have only two children, or no more refinements are possible. We report the pseudocode in Algorithm 1.

The tree produced is an unrooted tree, but for the sake of simplicity, we describe it as rooted. At the beginning the unrefined partition tree T (Line 1 in Algorithm 1) is a rooted star that has $\ell + 1$ nodes: root $\mathcal{S}$ (non-final) and children $S_1, \ldots, S_\ell$ marked as final. The mark *final* for a node indicates that no more refinement is possible at that node.

The algorithm iteratively processes any non-final node X of T (Line 3): given the list of nodes $C_1, \ldots, C_h$ that are children of X, REFINEMENT (Line 6) returns a list $L = L_1, \ldots, L_s$, with $s < h$, of compatible subsets of $\bigcup_k C_k$. The DRAW_AND_MARK function adds the nodes (possibly non-final) listed in L to T (Line 7).

To mark the node X as final, the DRAW_AND_MARK function checks if L is empty (Line 10). If L is not empty, a new internal node of T is created for each $L_i \in L$. Any inserted node, as well as X, is marked final, if it has only two children; otherwise, it needs to be further refined and is added to the queue (Lines 14-16). One possible iteration of Algorithm 1 can be found in [26, Figure 2].

Algorithm 1 Iterative refinement of the partition tree (Algorithm 1 from [26]).

input : $\ell, \mathsf{ebwt}(\mathcal{S}), \mathsf{lcp}(\mathcal{S}), \mathsf{cda}(\mathcal{S})$
output : A tree whose leaves are colored with $1 \dots \ell$, each color being a taxon of $\mathcal{S}$

Let $T \leftarrow$ Rooted star with a non-final root $\mathcal{S}$, and final leaves colored $1 \dots \ell$
Queue.push($\mathcal{S}$)
while Queue *is not empty* **do**
    $X \leftarrow$ Queue.pop()
    $C_1, \dots, C_h \leftarrow X$.children()
    $L \leftarrow$ REFINEMENT($\mathsf{ebwt}(\mathcal{S}), \mathsf{lcp}(\mathcal{S}), \mathsf{cda}(\mathcal{S}), \{C_1, \dots, C_h\}$)
    DRAW_AND_MARK(T, L, X, Queue)
Function *DRAW_AND_MARK* (*T, L, X,* Queue)
    if *L is empty* **then**
        Mark X in T as final `// cannot further refine X`
    else
        foreach *set* L_i *of* L **do**
            Add L_i as a node in T if not already present
        Mark as final every new node with two children in T
        Add to Queue all new nodes not marked final
        Mark X as final if it has two children, otherwise add it to Queue
Return T

The refinement procedure. The inner REFINEMENT function returns a list L of compatible subsets starting from a set of sibling nodes $C_1, \dots, C_h$ of T, which correspond to some (not necessarily all) taxa. Specifically, if C_i is a leaf, then C_i corresponds to one taxon (represented by a single sequence or a string collection), otherwise C_i is an internal node and corresponds to the subset $\mathcal{R}_i \subset \mathcal{S}$ comprising all the taxa associated with the leaves of the subtree rooted at C_i. Let $\mathcal{R} = \bigcup_{i=1}^{h} \mathcal{R}_i$ be the set of all the taxa corresponding to nodes $C_1, \dots, C_h$.

To quantify the similarity of a subset of taxa in $\mathcal{R}$ in terms of their common substrings, the eBWT positional clustering framework is employed and scores are assigned to some of the eBWT positional clusters detected. More precisely, given $\mathcal{R} \subset \mathcal{S}$, by Remark 1, we linearly scan the data structures $\mathsf{ebwt}(\mathcal{S})$, $\mathsf{lcp}(\mathcal{S})$ and $\mathsf{cda}(\mathcal{S})$ to detect and analyse eBWT positional clusters in $\mathsf{ebwt}(\mathcal{R})$. Among all the eBWT positional clusters detected, by Remark 2, we consider *relevant* the ones associated with common substrings that are shared by a sufficiently large number of taxa in $\mathcal{R}$ (but not by all of them) and cannot be extended on the left – the reader can find details in [26, Definition 3.5]. Since any relevant positional cluster is associated with a unique subset $R \subset \mathcal{R}$ of taxa sharing a common substring of variable length, we assign the length of that common substring as the cluster's score for subset R (see also Remark 2). After analysing all the relevant positional clusters, we have a weighted list $\mathcal{L}$ of subsets of $\mathcal{R}$. Each subset R in $\mathcal{L}$ corresponds to *at least one* relevant positional cluster, and its weight is the sum of all the scores for R over all the relevant positional clusters.

Finally, to build up the output list L, we sort the subsets in $\mathcal{L}$ by their weight and greedily select those with the highest weight that are compatible with each other. For computational efficiency, we stop the greedy procedure after a certain number of consecutive unsuccessful attempts to add elements to L (more details in [26]).

4 The McDag Compact Index for Maximal Common Subsequences

In this section we describe the main ideas of the practically efficient compact index McDag introduced in [12]. In the present paper, for the sake of simplicity, we describe the results for two input strings, even if they have also been extended to handle an arbitrary number of strings in [13].

Let us first formalize the definition of an MCS index as follows:

▶ **Definition 3** ([12], Section 2.2). *Given two strings X and Y of length $O(n)$, a labeled DAG $G = (V, E, l)$ is an index for $MCS(X, Y)$ if the following conditions hold:*

1. *Each node u (other than source or sink) is associated with a match denoted as $m(u) = (i, j)$, and has label $l(u) = X[i] = Y[j]$, where $1 \leq i \leq |X|$ and $1 \leq j \leq |Y|$.*
2. *There exist a single source s and a single sink t, with special values $m(s) = (0, 0)$, $l(s) = \#$, and $m(t) = (|X| + 1, |Y| + 1)$, $l(t) = \$$.*
3. *Each st-path $P = s, x_1, ..., x_h, t$ is associated with unique string $Z = l(x_1), ..., l(x_h) \in MCS(X, Y)$, and the associated matching for P must satisfy $m(x_1) < \cdots < m(x_h)$.*
4. *For each $Z \in MCS(X, Y)$ there is a corresponding st-path $P = s, x_1, ..., x_h, t$ such that $Z = l(x_1), ..., l(x_h)$.*

Let us note that a naive construction of such an MCS index (e.g., through a trie) could potentially require exponential time and space, as the number of nodes may be proportional to the number of MCS, which can in turn be exponential in the input size. Constructing such a compact MCS index in an efficient way is therefore not trivial.

We start by giving a high-level idea of how this problem is solved by McDag in Section 4.1, and then, in Section 4.2, we focus on explaining how to efficiently compute the frequency distribution of MCS lengths from the McDag index.

4.1 Overview of McDag Construction

The best way to define McDag is to employ a two-phase scheme. In the first phase, an *approximate rightmost co-deterministic index* $A = (V_A, E_A, l_A)$ for the set of MCSs is built. Approximate, rightmost, and co-deterministic respectively mean that (i) A indexes both the whole set of MCSs as well as some non-maximal common subsequences; (ii) for each edge (v, u) no character $l_A(v)$ appears between the positions defined by matches $m(v)$ and $m(u)$; and (iii) each node of A has at most one in-neighbor labeled with any character $c \in \Sigma$. Then, the second phase builds a *deterministic* version of A (i.e., where each node has no more than one out-neighbor per character) that does not contain any non-maximal common subsequence, yielding the final McDag. This latter procedure is called McConstruct, and its pseudocode is reported in Algorithm 2.

Empirical results show that the size of the initial approximate index A plays an important role in determining the size of the output MCS index. For this reason, we here review a method to construct A that tries to include few non-maximal common subsequences to begin with. Nevertheless, McConstruct correctly produces an MCS index for any input approximate rightmost co-deterministic index. For example, one could use a variant of the Common Subsequence Automaton [19, 20, 42], which models all common subsequences of a set of input strings.

First phase. We start by building a deterministic approximate MCS index $D = (V_D, E_D, l_D)$: we first add a source s_D with associated match $m(s_D) = (0, 0)$, corresponding to character $l(s_D) = \#$; then, we start to visit all nodes u in V_D. Throughout construction we ensure

that all nodes have distinct matches: if $m(u) = m(v)$ then $u = v$. When visiting node u, we consider for each character $c \in \Sigma$ the closest match $(i_c, j_c) > m(u)$. If there exists a match m' such that $m(u) < m' < (i_c, j_c)$, then we discard (i_c, j_c). Otherwise, we identify node v with match $m(v) = (i_c, j_c)$, or create v if it is not present. Then, we connect node u to node v. If we are not able to connect u to any node, we connect it to the sink t_D, which has match $m(t_D) = (|X| + 1, |Y| + 1)$ and label $l(t_D) = \$$.

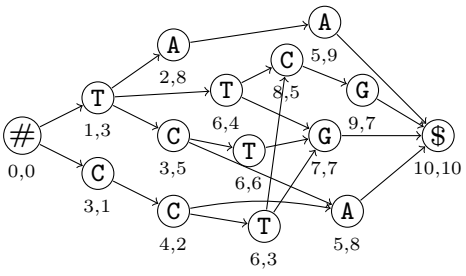
To build A, we repeat the same process in the opposite direction, reading the input strings right-to-left while using D as a guide: for each node u, we define a corresponding set of nodes $F(u) \subseteq V_D$ as the nodes that share a suffix with u. We start by adding t_A, with $F(t_A) = \{t_D\}$ and match $m(t_A) = m(t_D)$. To ensure that $F(u)$ is completely defined, we visit u only after we have visited all its potential out-neighbors, i.e., all nodes w that have match $m(u) < m(w)$. When processing a node u, we add its in-neighbors and enrich their $F(\cdot)$ sets as follows. We consider each node $x \in V_D$ such that $(x, y) \in E_D$ for some $y \in F(u)$, and if there is a match m' such that $m(x) < m' < m(u)$, we discard x. For each character c associated with the non-discarded nodes x, we select node v with match $m(v) = (i_c, j_c)$ such that c does not appear in $X[i_c + 1] \dots X[i_u - 1]$ and $Y[j_c + 1] \dots Y[j_u - 1]$, or create it if not present, and add it as an in-neighbor of u. We then add all non-discarded nodes x to $F(v)$.

Algorithm 2 McConstruct (Algorithm 1 from [12]).

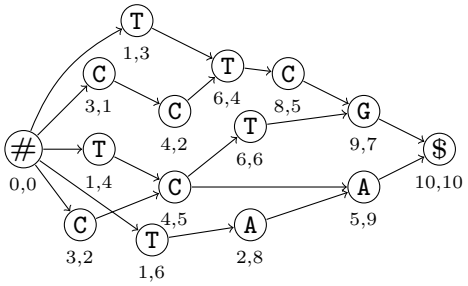
Data: Input $A = (V_A, E_A, l_A)$: rightmost approximate co-deterministic MCS index with source s_A
Result: A deterministic MCS index $G = (V, E, l)$

Initialize $G = (V, E, l)$, where $V = \{s\}$, $E = \emptyset$, $m(s) = m(s_A)$, $l(s) = \#$
`// F(u) is the set of nodes in A corresponding to u in G`
$F(s) \leftarrow \{s_A\}$
while *there exists* $u \in V$ *with no out-neighbors and* $l(u) \neq \$$ **do**
  Initialize $N_c = \emptyset$ for all $c \in \Sigma \cup \{\$\}$
  forall $(x, y) \in E_A$ *such that* $x \in F(u)$ **do**
    Add y to N_c, where $c = l(y)$
  Initialize $P = \emptyset$
  forall $N_c \neq \emptyset$ **do**
    $i_c \leftarrow \min\{i_z \mid (i_z, j_z) = m(z) \wedge z \in N_c\}$
    $j_c \leftarrow \min\{j_z \mid (i_z, j_z) = m(z) \wedge z \in N_c\}$
    Add match (i_c, j_c) to P
  forall $N_c \neq \emptyset$ *and* $p \in P$ **do**
    Remove all y from N_c such that $p < m(y)$
  forall $N_c \neq \emptyset$ **do**
    if *no node* $w \in V$ *has* $F(w) = N_c$ **then**
      Add new node w to V
      Set $F(w) = N_c$, $m(w) = (i_c, j_c)$, $l(w) = c$
    else Let $w \in V$ be the node such that $F(w) = N_c$
    Add edge (u, w) to E
return $G = (V, E, l)$

Second phase. Given $A = (V_A, E_A, l_A)$ with source s_A from the first phase, we apply Algorithm 2 (McConstruct) to obtain a graph $G = (V, E, l)$ that becomes our McDag with source s. Again, we associate each node $u \in V$ with a set $F(u)$ of nodes from V_A, all having the same label as u (initially, $F(s) = \{s_A\}$ with label $\#$). This time, a node



(a) The first deterministic approximate index D, with $|V_D| = 15$ and $|E_D| = 21$.



(b) The co-deterministic approximate index A, with $|V_A| = 15$, and $|E_A| = 19$.

Figure 2 First phase of McDag construction for input strings $X =$ `TACCATGCG` and $Y =$ `CCTTCTGAA`.

$x \in F(u)$ must share at least one prefix with u. At each step we take a node $u \neq t$ and add its out-neighbors. To do so, we take the out-neighbors of x in A and filter-out the ones whose matches are to the right of some match $(i_c, j_c) > m(u)$, as they cannot lead to an MCS: (i_c, j_c) is a witness to defy their maximality. Then, we identify a node v with that same associated set of filtered out-neighbors $F(v)$, or we add it if not present, and add edge (u, v) to G. The key difference is that in the first phase each node u is uniquely identified by match $m(u)$, while in the second phase it is uniquely identified by the set $F(u)$. We end up having a single sink t, corresponding to \$, only occurring at the end of both strings.

In the rest of this section we illustrate the described method with an example, and we highlight the key features that make McConstruct work. Consider the two input strings $X =$ `TACCATGCG` and $Y =$ `CCTTCTGAA`. Figure 2 depicts the indices constructed during the first phase. More specifically, Figure 2a depicts the deterministic approximate index D, built by reading both strings left-to-right, while Figure 2b shows the co-deterministic approximate index A which is constructed using D. Upon careful inspection, one can observe that A does not contain the non-maximal sequence `TTG` as a source-to-sink path, whereas D does. However, A still contains `CCTG`, which is also not maximal because of `CCTCG`.

Non-maximal common subsequences such as `TTG` or `CCTG` can be characterized in both of these data structures by using the concept of *subsequence bubbles*. A subsequence bubble is formed by a pair of paths that start and end at common nodes but are otherwise node-distinct, with the added condition that the shorter path spells a subsequence of the longer one. If a non-maximal common subsequence is contained in the index, the corresponding st-path must pass through the shorter path of at least one subsequence bubble. For instance, in Figure 2a a subsequence bubble is given by the pair of paths $(1,3),(6,4),(7,7)$ and $(1,3),(3,5),(6,6),(7,7)$, in which the short path spells `TTG` and the long path spells `TCTG`, thus witnessing the non-maximality of `TTG`. The main goal of McConstruct is to ensure that the resulting DAG contains no such subsequence bubbles. We refer the interested reader to [12] for the proof of correctness.

Figure 3 finally shows the McDag index, resulting from using the index A as input for McConstruct. Empirically, McDag has been shown to usually produce smaller indices with respect to the provably polinomially-bounded index of [18]. Despite this, finding a polynomial theoretical bound on the size of McDag remains an open problem. Note that here we are comparing the size of the indices as output by the respective construction methods, without applying any node removal. Otherwise, a simple automaton minimization

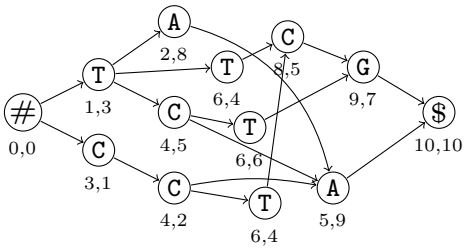


Figure 3 The McDag index for input strings $X =$ `TACCATGCG` and $Y =$ `CCTTCTGAA` has $|V| = 13$, $|E| = 17$ ($F(\cdot)$ sets omitted for compactness). Note that there are two nodes for match $(6, 4)$: this necessarily means that the two nodes have different $F(\cdot)$ sets.

algorithm (such as Revuz's algorithm [38] for acyclic deterministic finite automata) could reduce the number of nodes to a minimum, independently of the starting MCS index. Finding a tight bound on the size of the resulting minimal index also remains an open problem.

4.2 Generation of MCS Lengths Distribution

Building a deterministic index for MCSs allows us to perform a number of interesting operations such as enumeration, counting, and random access. One particular operation we might be interested in is related to counting: in Section 5 we will show experiments comparing the distribution of the MCS lengths for different pairs of genomic sequences. Here, we explain how one can generate such a distribution, by counting the number of paths for each path-length inside a DAG, using dynamic programming. The code for generating this distribution has been made available with the original paper [12], but the underlying algorithm was not detailed therein. We give a brief description here.

Consider a DAG $G = (V, E)$ and a generic node $u \in V$. Let $d(u)_i$ be the number of paths of length i that start from node u and end in a sink. These values can be computed as follows. For any sink t we define $d(t)_0 = 1$. Then, for the remaining nodes we can compute $d(u)_{i+1} = \sum_{(u,v)\in E} d(v)_i$, for all $i > 0$. Indeed, if all out-neighbors v of u have already computed their $d(v)_i$ values, node u can gather the sum of paths of length i and set the result as the number of paths of length $i + 1$ starting from u. At the end of the procedure, the distribution of the path lengths will be stored at the sources of the DAG.

The main problem of the procedure we just described is that every counter $d(u)_i$ can take non-negligible space and may not fit into a machine word. As previously mentioned, the number of MCSs can be exponential in the length of the input strings. This in turn means that the space required to store the number of paths of a given length is $O(n)$ bits. Since for our purposes we are interested in the *qualitative* distribution of the MCS lengths, we can use a trick (commonly known as log-sum-exp) to ensure that each $d(u)_i$ value can fit into a machine word. Namely, instead of storing the number of paths in $d(u)_i$, we store the logarithm of that number, as $d^*(u)_i = \log(d(u)_i) = O(n)$ for all u and i. To directly compute the value $d^*(u)_{i+1}$ we do the following: first, we find the maximum number of paths of length i among all out-neighbors as $\alpha_i = \max_{(u,v)\in E} d^*(v)_i$; then we compute $d^*(u)_{i+1} = \alpha_i + \log\left(\sum_{(u,v)\in E} 2^{(d^*(v)_i - \alpha_i)}\right)$.

5 Experiments

In this section, we experimentally review that the BWT-based tool phyBWT can achieve benchmark-level accuracy in phylogenetic reconstruction by exploiting the common substrings among taxa. Furthermore, we provide experimental evidence suggesting that the tool for MCSs indexing could offer valuable insights for inferring evolutionary relationships.

Specifically, the plots showing the distribution of the MCS lengths reveal a notable correlation between sequences associated with taxa that are close in the phylogenetic tree.

Datasets. For this study, we selected two datasets from two well-known viruses: the Human immunodeficiency virus (HIV) and the Ebola virus. Since viruses can evolve rapidly, viral phylogenies are challenging and often look very different. However, clade classification plays a crucial role in virology, since each clade (or subtype) represents a group with shared genetic similarities.

The HIV dataset comprises 43 HIV-1 complete genomes that have been used in the literature [45]. Thirty-five sequences belong to the major group (Group M) which is divided into subtypes A, B, C, D, F, G, H, J, K; seven sequences are from the minor Groups N and O, and one CPZ sequence (CIV strain AF447763) is an outgroup. The average length of the sequences is 9267 base pairs. The reference sequences have been carefully selected in [31] according to several criteria, and can be downloaded from the Los Alamos National Laboratory HIV Sequence Database[6].

The Ebola dataset comprises 20 published sequences from [23] selected in [30]. The Ebolavirus genus includes five viral species: Ebola virus (*Zaire ebolavirus*, EBOV), *Sudan virus* (SUDV), *Tai Forest virus* (TAFV), *Bundibugyo virus* (BDBV), and *Reston virus* (RESTV). The average length of the sequences is 18900 base pairs.

Phylogeny reconstruction. For the HIV dataset, Figure 4 depicts the phylogeny produced by phyBWT in [26]. Resembling the benchmark phylogeny depicted in [45, Fig. 2], subtypes are distinctly grouped together in different branches: subtypes B and D (resp. C and H) are closer to each other than to the others, and subtype F (resp. A) contains two distinguishable sub-subtypes F1 and F2 (resp. A1 and A2) that are closely related to subtypes K and J (resp. G), while subtypes N and O are external.

For the Ebola dataset, Figure 5 depicts the phylogeny produced by phyBWT in [26]. According to the benchmark phylogeny depicted in [30, Fig. 4], phyBWT exactly separated the five species. The EBOV sequences are clustered into a monophyletic clade, and BDBV and TAFV viruses are positioned close and then clustered with the EBOV branch. The SUDV clade is placed as sister to the EBOV, TAFV and BDBV clade, like in [30, Fig. 4E].

Given the required data structures, phyBWT reconstructs the proposed phylogeny for each dataset in less than one second by performing only two iterations of Algorithm 1 for the Ebola dataset and three iterations for the HIV dataset, with a RAM usage of approximately 8.5 MB.

MCS length distribution. We report in Figures 6 and 7 the logarithmic distribution of the MCS lengths of different viruses taken from the HIV and Ebola datasets. On the x-axis we find the various lengths of the MCSs, while on the y-axis the logarithm of their quantity. Specifically, Figure 6 considers the four taxa *AF286238 A2, AF286237 A2, U51190 A1,*

[6] `http://www.hiv.lanl.gov/`

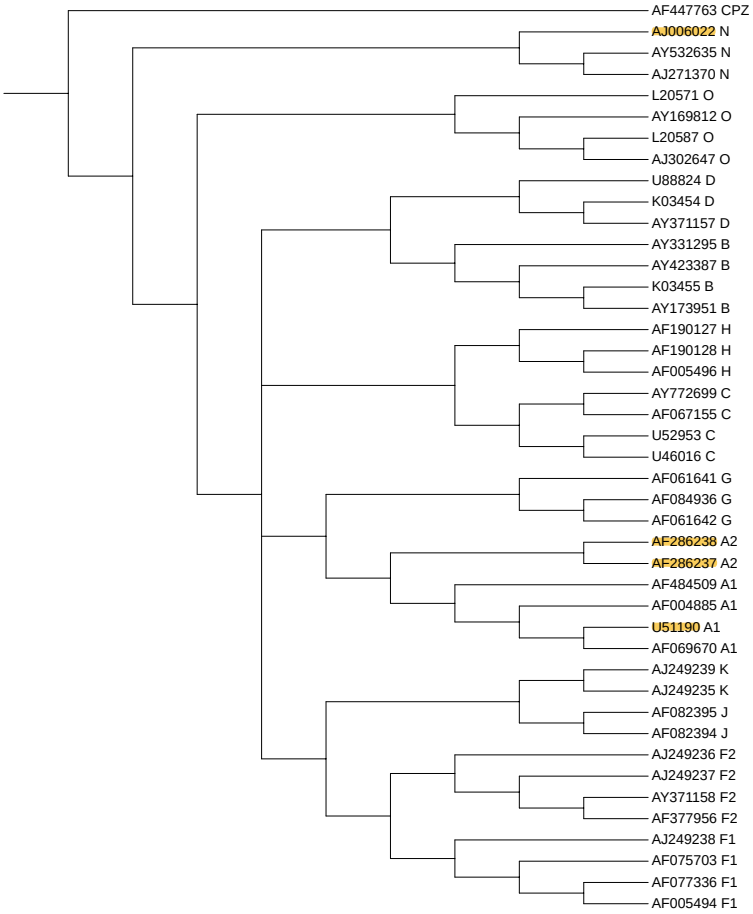


Figure 4 The phylogenetic tree on the 43 HIV sequences by PHYBWT [26, Figure 11]. Re-root the tree in CIV strain AF447763, as it is set outgroup in the reference tree in [45]. We highlight the strains that are used in the following experiments.

AJ006022 N of the HIV virus dataset, and Figure 7 considers the four taxa *BDBV 2012 KC545393, BDBV 2007 FJ217161, TAFV 1994 FJ217162, RESTV 1996 AB050936* of the Ebola virus dataset. For both datasets, we selected two taxa that are very similar (*AF286238 A2* and *AF286237 A2* for HIV, and *BDBV 2012 KC545393* and *BDBV 2007 FJ217161* for Ebola), one that is not too far from the first two (*U51190 A1* for HIV, and *TAFV 1994 FJ217162* for Ebola), and one last taxon that is far from every other considered taxon in the phylogeny (*AJ006022 N* for HIV, and *RESTV 1996 AB050936* for Ebola).

The algorithms for building McDAG and computing the distributions were implemented in C++, compiled with `g++ 11.4.0` using the `-O3` and `-march=native` flags. The source code is available at `https://github.com/giovanni-buzzega/McDag` [12]. We carried out the experiments on a DELL PowerEdge R750 machine in a non-exclusive mode, featuring 24 cores with 2 Intel(R) Xeon(R) Gold 5318Y CPUs at 2.10 GHz, and 989 GB of RAM. The operating system is Ubuntu 22.04.2 LTS.

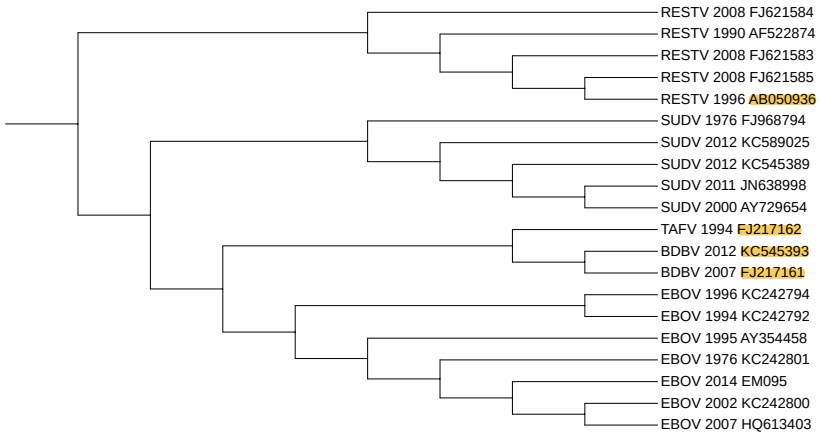


Figure 5 The phylogenetic tree on Ebolavirus dataset by PHYBWT [26, Figure 13]. We highlight the strains that are used in the following experiments.

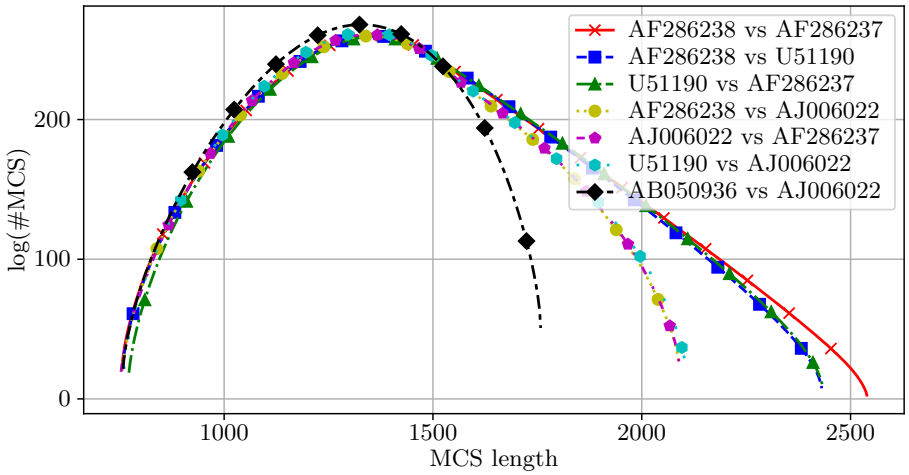


Figure 6 Length distribution of MCSs among the selected pairs of DNA sequences from the HIV dataset. The black line is the distribution of MCSs between HIV and Ebola virus taxa. The y-axis is logarithmic (base 10).

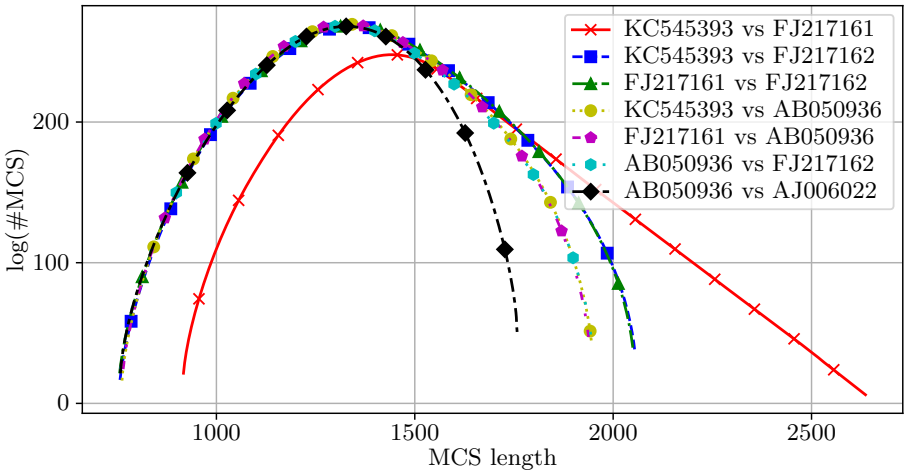


Figure 7 Length distribution of MCSs among different pairs of DNA sequences from the Ebola dataset. The black line is the distribution of MCSs between HIV and Ebola virus taxa. The y-axis is logarithmic (base 10).

For all strings we considered the substring between position 2500 and 5200 (chosen arbitrarily), and we built McDag. On average, index construction took 13.675 ± 0.673 seconds, followed by 15.763 ± 0.95 seconds to compute the MCS length distribution. As shown on the y-axis on both Figure 6 and Figure 7, the number of paths (and hence MCSs) in McDag is quite large, reaching values on the order of 10^{270}. Since such large numbers still remain within the representable range of a double, we did not use the the log-sum-exp trick in Section 4.2. However, when dealing with larger numbers of MCSs, we may have to resort to this technique to avoid overflow errors. In this case, the execution time may grow by a constant factor, due to the additional computational cost of the log and exp functions: in some preliminary testing on our data, we saw that the execution time increased to an average of 108.186 ± 10.973 seconds.

We now briefly discuss the outcome of the experiments. Since the LCS length is known to correlate well with string similarity [36], we see in both Figure 6 and 7, as expected, that the two strings considered most similar have the far right tail of their distribution ending at higher values on the x-axis. The most evident behaviour of the plots is that all lines, from left to right, start with a bell shape and, after the peak, decrease following a straight line before curving down again. This feature is more evident in the line that plots the distribution of MCSs between the taxa considered most similar.

Interestingly, we see that the two lines (in blue and green) that correspond to the MCS distribution between the taxon of medium distance and the first two taxa, are slightly detached from the first red line. For instance, in the case of HIV, Figure 6 shows an almost-perfect overlap up to lengths of 2100 on the x-axis; after that the lower similarity translates to a smaller number of long MCSs, with the straight part of the bell shape decaying earlier than in the red line. In the case of Ebola (Figure 7), there is again a gradual difference in where the lines drop on the right side (more similar pairs drop further right), but also a stark difference of the red line, representing the two closest taxa, in the left side of the graph: the start of the line is shifted right compared to the others, meaning that every MCS is longer than about 800. This behaviour suggests a particularly strong similarity, and further investigation into how it arises is an interesting direction of work.

The next three lines, in yellow, magenta, and cyan, represent the relations with the more dissimilar taxon. We see in both figures that the three lines again overlap, and the right side detaches from the other lines on lower values on the x-axis.

Finally, in both figures we added a black line that depicts the distribution of two completely unrelated string: in both plots, we used taxon *RESTV 1996 AB050936* of Ebola and taxon *AJ006022 N* of HIV. In both cases we have that the black line closely follows a bell shape, with no part of it showing a straight line behavior; moreover, on a large portion of the left side, it overlaps with the yellow, magenta and cyan lines.

This suggests that there is some baseline set of MCSs of any unrelated strings that acts as a *background noise*; after a given threshold length, the number of "non-noisy" MCSs seems to be a good indicator of string similarity, and, as an extension, of taxon similarity.

6 Conclusions

We have reviewed two recent results that use compact string indices to naturally highlight relevant information in a genomic context. The BWT-based approach PHYBWT infers evolutionary links by clustering similar substrings, and the DAG-based index MCDAG can be used to show the distribution of Maximal Common Subsequences, which exposes similarities among strings. We have also shown experimentally that MCS length distributions vary among closely related and more distantly related taxa, using the phylogeny generated by PHYBWT as a reference. Further work in visualizing and analysing the information emerging from these indices, as well as extending the analysis to new indices, is an interesting direction to explore and may yield positive results in phylogeny and, more generally, in the analysis of genomic sequences.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE, IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14.

2 M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

3 Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*, pages 3–14. IEEE, 1995. doi:10.1109/ICDE.1995.380415.

4 H.-J. Bandelt and A. W. M. Dress. A canonical decomposition theory for metrics on a finite set. *Advances in mathematics*, 92(1):47–105, 1992.

5 H.-J. Bandelt and A. W. M. Dress. Split decomposition: A new and useful approach to phylogenetic analysis of distance data. *Molecular Phylogenetics and Evolution*, 1(3):242–252, 1992. doi:10.1016/1055-7903(92)90021-8.

6 H.-J. Bandelt, K. T. Huber, J. H. Koolen, V. Moulton, and A. Spillner. *Basic Phylogenetic Combinatorics*. Cambridge University Press, 2012. URL: http://www.cambridge.org/de/knowledge/isbn/item6439332/.

7 M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013. doi:10.1016/j.tcs.2012.02.002.

8 Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE, 2000. doi:10.1109/SPIRE.2000.878178.

9 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 79–97. IEEE, 2015. doi:10.1109/FOCS.2015.15.

10 Laurent Bulteau, Mark Jones, Rolf Niedermeier, and Till Tantau. An FPT-algorithm for longest common subsequence parameterized by the maximum number of deletions. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 6:1–6:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CPM.2022.6.

11 M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DIGITAL System Research Center, 1994.

12 Giovanni Buzzega, Alessio Conte, Roberto Grossi, and Giulia Punzi. Mcdag: Indexing maximal common subsequences in practice. In *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, pages 21–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

13 Giovanni Buzzega, Alessio Conte, Roberto Grossi, and Giulia Punzi. McDag: indexing maximal common subsequences for k strings. *Algorithms for Molecular Biology*, 20(1):6, 2025. doi:10.1186/s13015-025-00271-z.

14 Giovanni Buzzega, Alessio Conte, Yasuaki Kobayashi, Kazuhiro Kurita, and Giulia Punzi. The complexity of maximal common subsequence enumeration. *Proc. ACM Manag. Data*, to appear, 2025.

15 Davide Cenzato, Zsuzsanna Lipták, Nadia Pisanti, Giovanna Rosone, and Marinella Sciortino. BWT for string collections. Accepted to Festschrift's honoree Giovanni Manzini, 2025. arXiv:2506.01092.

16 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. Polynomial-delay enumeration of maximal common subsequences. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 189–202, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32686-9_14.

17 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. Enumeration of maximal common subsequences between two strings. *Algorithmica*, 84(3):757–783, 2022. doi:10.1007/s00453-021-00898-5.

18 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. A compact DAG for storing and searching maximal common subsequences. In Satoru Iwata and Naonori Kakimura, editors, *34th International Symposium on Algorithms and Computation, ISAAC 2023, December 3-6, 2023, Kyoto, Japan*, volume 283 of *LIPIcs*, pages 21:1–21:15. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ISAAC.2023.21.

19 Maxime Crochemore, Borivoj Melichar, and Zdenek Tronícek. Directed acyclic subsequence graph - overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003. doi:10.1016/S1570-8667(03)00029-7.

20 Maxime Crochemore and Zdeněk Troníček. Directed acyclic subsequence graph for multiple texts. *Rapport IGM*, pages 99–13, 1999.

21 L. Egidi, F. A. Louza, G. Manzini, and G. P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15, 2019. doi:10.1186/s13015-019-0140-0.

22 Campbell Fraser, Robert W. Irving, and Martin Middendorf. Maximal common subsequences and minimal common supersequences. *Inf. Comput.*, 124(2):145–153, 1996. doi:10.1006/inco.1996.0011.

23 Stephen K. Gire, Augustine Goba, Kristian G. Andersen, Rachel S. G. Sealfon, Daniel J. Park, Lansana Kanneh, Simbirie Jalloh, Mambu Momoh, Mohamed Fullah, Gytis Dudas, Shirlee Wohl, Lina M. Moses, Nathan L. Yozwiak, Sarah Winnicki, Christian B. Matranga, Christine M. Malboeuf, James Qu, Adrianne D. Gladden, Stephen F. Schaffner, Xiao Yang, Pan-Pan Jiang, Mahan Nekoui, Andres Colubri, Moinya Ruth Coomber, Mbalu Fonnie, Alex Moigboi, Michael Gbakie, Fatima K. Kamara, Veronica Tucker, Edwin Konuwa, Sidiki Saffa, Josephine Sellu, Abdul Azziz Jalloh, Alice Kovoma, James Koninga, Ibrahim Mustapha, Kandeh Kargbo, Momoh Foday, Mohamed Yillah, Franklyn Kanneh, Willie Robert, James L. B. Massally, Sinéad B. Chapman, James Bochicchio, Cheryl Murphy, Chad Nusbaum, Sarah Young, Bruce W. Birren, Donald S. Grant, John S. Scheiffelin, Eric S. Lander, Christian Happi, Sahr M. Gevao, Andreas Gnirke, Andrew Rambaut, Robert F. Garry, S. Humarr Khan, and Pardis C. Sabeti. Genomic surveillance elucidates ebola virus origin and transmission during the 2014 outbreak. *Science*, 345(6202):1369–1372, 2014. doi:10.1126/science.1259657.

24 Ronald I. Greenberg. Bounds on the number of longest common subsequences. *CoRR*, cs.DM/0301030, 2003. arXiv:cs/0301030.

25 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT: Alignment-Free Phylogeny via eBWT Positional Clustering. In Christina Boucher and Sven Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*, volume 242 of *LIPIcs*, pages 23:1–23:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2022.23.

26 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT2: phylogeny reconstruction via eBWT positional clustering. *Algorithms Mol. Biol.*, 18(1):11, 2023. doi:10.1186/S13015-023-00232-4.

27 Miyuji Hirota and Yoshifumi Sakai. Efficient algorithms for enumerating maximal common subsequences of two strings. *CoRR*, abs/2307.10552, 2023. doi:10.48550/arXiv.2307.10552.

28 Miyuji Hirota and Yoshifumi Sakai. A fast algorithm for finding a maximal common subsequence of multiple strings. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 106(9):1191–1194, 2023. doi:10.1587/transfun.2022dml0002.

29 D. H. Huson and D. Bryant. Application of Phylogenetic Networks in Evolutionary Studies. *Molecular Biology and Evolution*, 23(2):254–267, October 2005.

30 Michelle Kendall and Caroline Colijn. Mapping Phylogenetic Trees to Reveal Distinct Patterns of Evolution. *Molecular Biology and Evolution*, 33(10):2735–2743, 2016. doi:10.1093/molbev/msw124.

31 Thomas Leitner, Bette Korber, Marcus Daniels, Charles Calef, and Brian Foley. HIV-1 Subtype and Circulating Recombinant Form (CRF) Reference Sequences, 2005. URL: https://www.hiv.lanl.gov/content/sequence/HIV/REVIEWS/LEITNER2005/leitner.html.

32 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.

33 David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336, 1978. doi:10.1145/322063.322075.

34 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *ACM-SIAM SODA*, pages 319–327, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

35 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theoret. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

36 Mike Paterson and Vlado Dančík. Longest common subsequences. In *International symposium on mathematical foundations of computer science*, pages 127–142. Springer, 1994.

37 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. Variable-order reference-free variant discovery with the Burrows-Wheeler transform. *BMC Bioinformatics*, 21, 2020. doi:10.1186/s12859-020-03586-3.

38 Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992. doi:10.1016/0304-3975(92)90142-3.

39 Yoshifumi Sakai. Maximal Common Subsequence Algorithms. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2018.1.

40 Yoshifumi Sakai. Maximal common subsequence algorithms. *Theor. Comput. Sci.*, 793:132–139, 2019. doi:10.1016/j.tcs.2019.06.020.

41 Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

42 Zdenek Tronícek. Common subsequence automaton. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata, 7th International Conference, CIAA 2002, Tours, France, July 3-5, 2002, Revised Papers*, volume 2608 of *Lecture Notes in Computer Science*, pages 270–275. Springer, Springer, 2002. doi:10.1007/3-540-44977-9_28.

43 Tandy Warnow. *Computational Phylogenetics: An Introduction to Designing Methods for Phylogeny Estimation*. Cambridge University Press, 2017.

44 R. Wittler. Alignment- and Reference-Free Phylogenomics with Colored de Bruijn Graphs. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143, pages 2:1–2:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2019.2.

45 Xiaomeng Wu, Zhipeng Cai, Xiu-Feng Wan, Tin Hoang, Randy Goebel, and Guohui Lin. Nucleotide composition string selection in HIV-1 subtyping using whole genomes. *Bioinformatics*, 23(14):1744–1752, May 2007. doi:10.1093/bioinformatics/btm248.

46 Andrzej Zielezinski, Hani Z Girgis, Guillaume Bernard, Chris-Andre Leimeister, Kujin Tang, Thomas Dencker, Anna Katharina Lau, Sophie Röhling, Jae Jin Choi, Michael S Waterman, Matteo Comin, Sung-Hou Kim, Susana Vinga, Jonas S Almeida, Cheong Xin Chan, Benjamin T James, Fengzhu Sun, Burkhard Morgenstern, and Wojciech M Karlowski. Benchmarking of alignment-free sequence comparison methods. *Genome Biology*, 20:144, 2019. doi:10.1186/s13059-019-1755-7.