McDag: Indexing Maximal Common Subsequences in Practice

Giovanni Buzzega
University of Pisa, Italy

Alessio Conte
University of Pisa, Italy

Roberto Grossi
University of Pisa, Italy

Giulia Punzi
University of Pisa, Italy

Abstract

Analyzing and comparing sequences of symbols is among the most fundamental problems in computer science, possibly even more so in bioinformatics. Maximal Common Subsequences (MCSs), i.e., inclusion-maximal sequences of non-contiguous symbols common to two or more strings, have only recently received attention in this area, despite being a basic notion and a natural generalization of more common tools like Longest Common Substrings/Subsequences. In this paper we simplify and engineer recent advancements on MCSs into a practical tool called McDag, the first publicly available tool that can index MCSs of real genomic data. We demonstrate that our tool can index sequences exceeding 10,000 base pairs within minutes, utilizing only 4-7% more than the minimum required nodes, while also extracting relevant insights.

2012 ACM Subject Classification Theory of computation → Pattern matching; Applied computing → Molecular sequence analysis; Applied computing → Computational genomics

Keywords and phrases Index data structure, DAG, Common subsequence, Inclusion-wise maximality, LCS

Digital Object Identifier 10.4230/LIPIcs.WABI.2024.21

Supplementary Material *Software (Source Code)*: `https://github.com/giovanni-buzzega/McDag` archived at `swh:1:dir:a37c7031ca6d202fa93adc92df453b880a1a56c3`

Funding *Giovanni Buzzega*, *Roberto Grossi*, *Giulia Punzi*: Received funding from the European Union's Horizon 2020 Research and Innovation Staff Exchange programme under the Marie Skłodowska-Curie grant agreement No. 872539.
Giovanni Buzzega, *Alessio Conte*, *Roberto Grossi*: Supported, in part, by MUR of Italy, under PRIN Project n. 2022TS4Y3N - EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data.
Giulia Punzi: Supported by the Italian Ministry of Research, under the complementary 36 actions to the NRRP "Fit4MedRob - Fit for Medical Robotics" Grant (#PNC0000007).

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments and suggestions.

1 Introduction

Strings are fundamental in computer science, and their analysis, indexing, and processing are among the oldest and best-studied problems. A central aspect of these problems is searching for relevant patterns in strings, which vary based on the application. We focus here on patterns that are *common* between two or more strings.

In some real-world domains, a common substring may be too strict of a requirement: to make an example, a sequence of bases in genetic data may represent an important gene, but different specimens may have undergone different micro-variations in their genetic code that very slightly altered the gene, and even the act of sequencing introduces noise in the data, so that an exact match is not guaranteed even when comparing samples from the same specimen. In these domains, it is relevant to consider the *common subsequence*: an ordered sequence of characters that occurs in all given strings, but not necessarily contiguously, i.e., the characters of the sequence may be interleaved with others.

As the number of common subsequences between two strings can be exponentially high, a common idea is looking at just the one of maximum length, *longest common subsequence* (LCS hereafter). LCSs are used to see how well two or more sequences align, or how similar they are [25]. Sometimes, the sequence itself is ignored, and a similarity metric is simply the length of the LCS compared to n, the length of the original strings. While LCS-based approaches can be effective, they have significant limitations: firstly, efficiency is limited as finding a single LCS among an arbitrary number of strings is NP-complete [19], and still takes quadratic time with just two strings (see the conditional lower bounds in [1, 4]). Also, Figure 1 shows an example case where a critical but relatively short sequence cannot be extended to a common subsequence as long as an LCS, thus any analysis based on LCSs would completely disregard this information.

On the other extreme, it is possible to consider *all* common subsequences with a Common Subsequence Automaton [8]. However, these will include many solutions (possibly most) that are included within other longer solutions, and thus pollute the set with redundant information. For these reasons, we focus on a generalization of LCSs called *Maximal Common Subsequences* (MCS hereafter), which provides an interesting middle point between LCS and all subsequences: an MCS S between two (or more) strings is a sequence of characters that occurs as a subsequence in each of the strings and that is *(inclusion) maximal*, that is, S cannot be extended with any character in any position and still be a common subsequence. For example in Figure 1, the critical shorter common subsequence may be included in the set of MCSs, and could improve the alignment of some critical common parts.

Once established our interest in MCSs for string analysis, the natural question is: which tools should be employed? In fact, very little was possible until recently. We now have an algorithm to enumerate all MCSs efficiently [6] but they can be exponential in number (in fact, this is true for LCSs too [12]) and it would be inconvenient to both enumerate and store all of them. To facilitate analysis using MCSs, a useful tool would be a sufficiently small index that allows us to efficiently retrieve and query MCSs.

First steps in this direction have been taken: [7] proves the existence of a DAG (Directed Acyclic Graph) of polynomial size that is able to represent all MCSs, count them, and reconstruct them as needed. Similar structures have also been shown in [13].

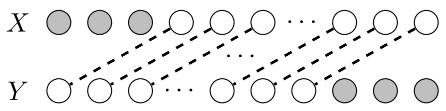

Figure 1 In the example, the LCS only shows the longer white "promoter sequence", a common occurrence in genomic sequences. The critical (but shorter) shaded common part cannot be extended to a common subsequence of the same length, is thus not shown by LCS-based analysis.

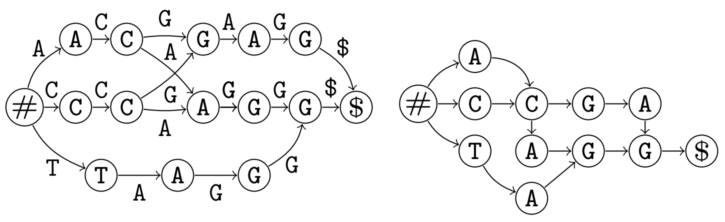

Figure 2 M-DAG (taken from [7]), CSA-MAXIMAL (derived from [8] and our filtering methods), and McDAG (this paper), for input strings X = TCACAGAGA and Y = ACCCGTAGG. Here, $MCX(X,Y)$ = ACGAG, ACAGG, CCGAG, CCAGG, TAGG.

1.1 Contribution

Based on these results, the contribution of this paper is to build a practical indexing tool, which we call McDAG, that is simultaneously simpler in concept and faster to construct, and to show that it can provide insight on real genomic data. Figure 2 shows our proposed McDAG on the right, along with two other DAGs obtained from the literature [7, 8].[1] Apart from the different number of nodes and edges, all share the same conceptual structure:

- There exist a single source s, labeled with marker #, and a single sink t, labeled with marker \$. All other nodes are labeled with characters from the alphabet Σ of X and Y.
- Each st-path is associated with a unique $Z \in MCS(X,Y)$ spelled out in its traversed nodes; vice versa, each $Z \in MCS(X,Y)$ has a unique st-path associated.
- The out-neighbors of each node are labeled with distinct characters from Σ, so the out-degree is at most $|\Sigma|$ (ignoring # and \$ labeling s and t respectively).

As a result, each prefix of an MCS has a unique path from s. For example, ACA is found in McDAG following #, A, C, and A in this order, each time with a unique branching choice on the current node. The McDAG has less than $|X| \times |Y|$ nodes in our experimental study of Section 3, only 4-7% more than the minimum required nodes. It takes quadratic time in practice, which allows us to index sequences exceeding 10,000 base pairs within minutes. Note that, in general, no DAG storing $MCS(X,Y)$ can take sub-quadratic time in the worst case, unless SETH or OVH fail, as an LCS is an MCS of maximum length, and the problem of finding the LCS length has a quadratic conditional lower bound [1, 4].

The benefit of the above conceptual structure is that it fits several efficient algorithms on the state of the art for querying deterministic acyclic automata. For instance, listing all the strings in $MCS(X,Y)$, reporting only those of (up to) a given length, or matching a simple regular expression, and counting the number of the above strings (e.g. see [7]). Moreover, McDAG can be easily extended to store MCS of an arbitrary number k of strings.

A further example, showcasing both the capabilities of our tool and the significance of MCSs, is shown in Figure 3: we took segments of DNA data from different sources (viruses, fish) as well as a randomized sequence as control, we used McDAG to compute the MCSs of each pair of types, and plot their size distribution, without generating all the exponentially many MCSs. Looking at the distributions involving the random string we see remarkably similar bell curves, due to the sheer amount of shorter subsequences which occur in all sequences. Strikingly, when comparing virus and fish DNA, the curve aligns rather precisely with the random ones; this is a key insight to the fact that such subsequences are unlikely to

[1] Based on our understanding of [13], the DAG introduced therein could potentially fit our definition. However, we lack sufficient details to make this assertion with certainty.

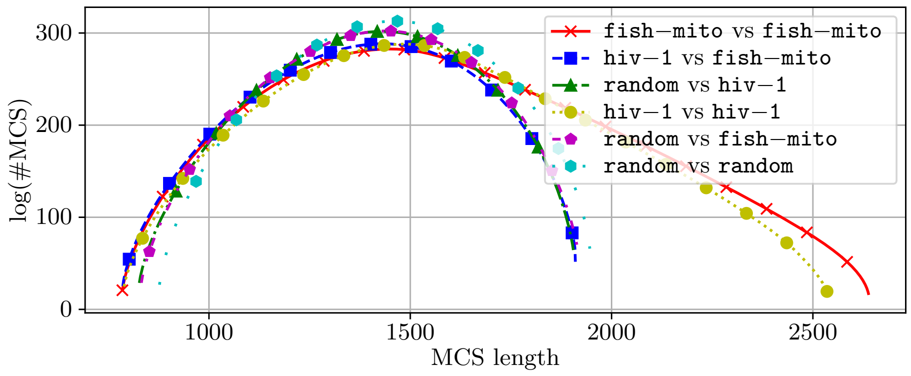

Figure 3 Length distribution of MCSs among different pairs of DNA sequences (see Section 3).

hold much significance. On the other hand, when comparing two sequences of the same kind (fish vs fish, and virus vs virus, in red and yellow), the distribution lines initially start in the same bell-curve shape, but take a completely different behaviour on the right part of the graph, showing a high amount of larger MCSs. This suggests that beyond a certain length threshold, some MCS may exhibit valuable alignment properties. Notably, this threshold might be shorter than the length of the LCS.

This suggests that MCS-based analysis can determine not only when two sequences have significant similarities, but also which common sequences are relevant and which are likely noise. We argue that McDag is a significant first step in this type of analysis. It is – to the best of our knowledge–the first publicly available tool that allows for efficiently indexing and analyzing MCSs, and can process sequences of over 10000 symbols in just a few minutes (`https://github.com/giovanni-buzzega/McDag`). While of course complex genes such as human ones are orders of magnitude longer and require further development of the tool, this already allows for a deeper analysis of simpler genomic data or selected segments.

1.2 Related work

The concept of MCSs first appeared within a general form in the data mining community [2]. In this context, the authors considered ordered sequences of sets of items rather than strings. A subsequence is obtained from a sequence by deleting any number of items from any set at any position. The focus was on finding frequent subsequences, which are subsequences that appear in more than a user-defined number of sequences in the database. One of the problems proposed was to find inclusion-maximal frequent subsequences, which are not subsequences of any other frequent subsequences. Our problem can be seen as a special case of this framework by considering only two sequences of singletons and setting the frequency threshold to two.

The MCS problem was later formalized in [11], along with several variations of the common subsequences, for which they studied the computational complexity and dynamic programming solutions in some cases. Further solutions to this problem have been proposed in various studies. Sakai provided the first (almost) linear-time algorithm to extract one MCS between two strings [23]. Bulteau et al. [5] used MCSs as a tool for a new parameterized LCS algorithm. Hirota and Sakai explored MCSs for multiple strings [14]. Conte et al. [7] and Hirota and Sakai [13] independently proposed DAGs for enumerating MCSs of two strings. Conte et al. [7] published the first polynomial-size DAG in the literature, where each node represents at least one prefix of some MCS.

We give some detail of the latter: if there is an edge from node u to node v, all prefixes of u are prefixes of some MCS and when extended with the character associated with that edge they do not lose this property. This allows for the direct construction of an MCS index, but maintaining it can be costly [6], as finding the right character to extend a prefix may require expensive computation. For instance, finding a character that extends an MCS prefix to a CS prefix is simple, but it may yield prefixes that do not lead to any MCS. The current approach for finding suitable extensions associates a distinct quadruple of integers to each node, causing the automaton size to be n^3 or more in terms of nodes.

Subsequence-related problems have been previously addressed using automata. Baeza-Yates [3] introduced the Directed Acyclic Subsequence Graph (DASG), that accepts all subsequences of a given string, and can be generalized to accept subsequences of any string in a set. A subsequent result was the common subsequence automaton (CSA) [8, 9, 26]: it accepts common subsequences of a set of strings, including non-maximal ones, and it is similar in concept to the common subsequence tree of [15]. The CSA can also be used to find an LCS between two strings [20]. Moreover, automaton-inspired tools such as binary decision diagrams like ZDD [21] and SeqBDD [17] can be used to compactly represent the set $MCS(X, Y)$, but construction is non trivial: one potentially needs to first generate all MCSs and this can take exponential time and space.

As for LCSs, some algorithms for their computation can be seen as dynamic programming on some DAG [16, 18]. Furthermore, a DAG representation of LCSs was also recently used in [24] for the problem of finding diverse LCSs.

2 The McDag Index

Before defining McDag, we provide some preliminary notions.

2.1 Preliminaries

We consider a string $X = X[0] \dots X[|X| - 1]$ as a sequence of characters from an 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. We use special characters $\{\#, \$\}$ as markers delimiting input strings.

We say that string Z is a subsequence of X if there exist indices $0 \le i_0 < \dots < i_{|Z|-1} < |X|$ such that $X[i_k] = Z[k]$ for $0 \le k < |Z|$. Moreover, Z is a common subsequence of strings X and Y if Z is a subsequence of both X and Y: letting $0 \le j_0 < \dots < j_{|Z|-1} < |Y|$ be the indices such that $Y[j_k] = Z[k]$ for $0 \le k < |Z|$, we say that each pair (i_k, j_k) is a *match* (as $X[i_k] = Y[j_k]$) and the pairs $(i_0, j_0), \dots, (i_{|Z|-1}, j_{|Z|-1})$ form a *matching* in X and Y, whose corresponding string is Z. In general, we call a pair (i, j) a match when $X[i] = Y[j]$, observing that the pairs induce a partial order, defined as $(i, j) < (i', j')$ iff $i < i'$ and $j < j'$, which is total if pairs belong to the same matching; $(i, j) \le (i', j')$ is analogously defined.

In our example of Figure 2, $Z = \texttt{CGA}$ is a common subsequence of $X = \texttt{T\underline{C}ACA\underline{G}AG\underline{A}}$ and $Y = \texttt{AC\underline{C}C\underline{G}T\underline{A}GG}$, and one of its matchings is underlined in X and Y as $(1, 2), (5, 4), (8, 6)$.

We say that Z is a *longest common subsequence* (LCS), or belongs to $LCS(X, Y)$, if there is no common subsequence that is strictly longer than Z. Finally, Z is a *maximal common subsequence* (MCS) of X and Y if there is no string W that satisfies both conditions: *(i)* W is a common subsequence of X and Y, and *(ii)* Z is a proper subsequence of W. The set of all strings that are maximal common subsequences is denoted by $MCS(X, Y)$. Note that $LCS(X, Y) \subseteq MCS(X, Y)$, as an LCS is an MCS of maximum length.

We next introduce some graph notions. A *directed graph* $G = (V, E)$, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges, is a graph so that each edge (u, v) has direction from u to v. Specifically, two edges (u, v) and (w, z) are 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* $G = (V, E)$ 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 $d^+(u) = 0$. We consider a *labeled* DAG $G = (V, E, \ell)$, where each node u is associated with a character $\ell(u) \in \Sigma \cup \{\#, \$\}$.

2.2 Definition of index for MCS

Given two strings X and Y, a labeled DAG $G = (V, E, \ell)$ 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 $\ell(u) = X[i] = Y[j]$, where $0 \leq i < |X|$ and $0 \leq j < |Y|$.
2. There exist a single source s and a single sink t, with special values $m(s) = (-1, -1)$, $\ell(s) = \#$, and $m(t) = (|X|, |Y|)$, $\ell(t) = \$$.
3. Each st-path $P = s, x_0, ..., x_{h-1}, t$ is associated with unique string $Z = \ell(x_0), ..., \ell(x_{h-1}) \in MCS(X, Y)$, and the associated matching for P must satisfy $m(x_0) < \cdots < m(x_{h-1})$.
4. Each $Z \in MCS(X, Y)$ has a corresponding st-path $P = s, x_0, ..., x_{h-1}, t$ such that $Z = \ell(x_0), ..., \ell(x_{h-1})$.

We say that the DAG is *deterministic* if each node has out-neighbors labeled with distinct characters (and so its out-degree is at most $|\Sigma|$ and each prefix of a MCS has a unique path from s), and *co-deterministic* if the condition applies to the in-neighbors of each node (which has in-degree at most $|\Sigma|$). In both cases, there cannot be two distinct st-paths corresponding to the same string. Moreover, we get an *approximate* index for $MCS(X, Y)$ when condition 3 is relaxed, so that Z is not necessarily maximal, and so there could be st-paths in the DAG that store non-maximal common subsequences.

The DAGs in Figure 2 are all deterministic indices for the same set $MCS(X, Y)$, and they all satisfy the above conditions. The leftmost is M-DAG and has been introduced in [7]. The central one is CSA-MAXIMAL and has been derived from the Common Subsequence Automaton [8] by filtering out the non-maximal common subsequences. The rightmost is McDag, our proposed index that further reduces the number of nodes.

When any of the above deterministic indices for the set $MCS(X, Y)$ is available, a number of classical operations can be supported. For instance:

- List all the strings in $MCS(X, Y)$.
- Report only the strings in $MCS(X, Y)$ of (up to) given length.
- List the strings in $MCS(X, Y)$ containing a given string S, or similar regular expressions.
- Count the number of the above strings (all kinds).

We refer the reader to [7] for these operations, which can be implemented using standard algorithms from the literature on strings and automata, following the above definition of a deterministic index for MCS. As a final remark, the associated matches $m(u)$ for the nodes u in the DAGs are not strictly necessary for these operations, but help to quickly reconstruct a possible matching of a given MCS Z to X and Y.

2.3 Operational definition of McDag

The best way to define McDag for input X and Y is to employ a two-phase scheme. In the first phase, an approximate co-deterministic index A for $MCS(X,Y)$ is built. We recall that A stores $MCS(X,Y)$ along with some non-maximal common subsequences. We allow for this relaxed version as we can run the second phase, which eliminates from A the non-maximal ones, and makes the outcome a deterministic index, thus yielding our McDag.

First phase. We build $A = (V_A, E_A, \ell_A)$ with source s_A, so that A satisfies the rightmost property: for each $(u,v) \in E_A$, let $c = \ell_A(u)$ be the character labeling u, and $m(u) = (i_u, j_u)$ and $m(v) = (i_v, j_v)$ be the matches (recalling that $m(u) < m(v)$ by definition of MCS index); then, c does not appear in $X[i_u+1]\dots X[i_v-1]$ and $Y[j_u+1]\dots Y[j_v-1]$.

We can obtain A as a vanilla version of the Common Subsequence Automaton [8] as follows. Since we want it to be co-deterministic, we read X and Y from right to left, and start building A from its sink t_A backwards. Consider the generic step for a node u, initially $u = t_A$ with $m(t_A) = (|X|, |Y|)$. We need to link u to its in-neighbors, possibly creating some of the latter ones, which are at most $|\Sigma|+1$, one per character c and one for the source. Hence, for c we find the largest pair $(i_c, j_c) < m(u)$ such that $c = X[i_c] = Y[j_c]$: if a node v with $m(v) = (i_c, j_c)$ does not exist, we create v with $m(v) = (i_c, j_c)$ and $\ell(v) = c$, and add edge (s_A, v); in any case, we add edge (v, u).

Apart from its source and sink, A has as many nodes as the matches involved in its construction. As there cannot be two distinct nodes u and v of A with the same match $m(u) = m(v)$, we derive that $|V_A| \leq n^2+2$ and $|E_A| \leq (|\Sigma|+1)(n^2+1)$ as the max in-degree is $|\Sigma|+1$. Its construction time is $O(n^2|\Sigma|\log n)$, and its space is $O(n^2|\Sigma|)$. Here we assume wlog that $n = |X| = |Y|$.

An optimized version of the first phase has been implemented in McDag, as discussed in the experimental Section 3.3.

Second phase. Given $A = (V_A, E_A, \ell_A)$ with source s_A from the first phase, we apply Algorithm 1 to get a graph $G = (V, E, \ell)$ that becomes our McDag with source s and sink t. During the construction we associate each node $u \in V$ with a set $F(u)$ of nodes from V_A, all having the same label as u's (initially $F(s) = \{s_A\}$ with label $\#$). At each step we expand a node $u \neq t$ with its out-neighbors, filtering out the nodes of A whose matches are to the right of some match $(i_c, j_c) > m(u)$, as they cannot lead to MCS: (i_c, j_c) is a witness to defy their maximality. After that, we create new nodes in G for the filtered set of nodes with the same label coming from A, and their edges in G. We end up having a single sink t, corresponding to \$, only occurring at the end of both strings.

The proof of correctness of procedure McConstruct in Algorithm 1 is non-trivial, as we have to show that we retain all and only the strings in $MCS(X,Y)$ along all the st-paths in G. We postpone the correctness to Section 4, after providing the experimental analysis of Algorithm 1 in Section 3, which yields a quadratic cost in practice.

3 Experimental Analysis

3.1 Experimental setup

Our algorithms were implemented in C++ using `g++ 11.4.0` and compiled with the `-O3` and `-march=native` flags. All tests were conducted on a DELL PowerEdge R750 machine in a non-exclusive mode. This platform features 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.

Algorithm 1 Construction algorithm for McDag.

```
    Input $A = (V_A, E_A, \ell_A)$: rightmost approximate co-deterministic MCS index with source $s_A$
 1: procedure McConstruct($A$)
 2:     Initialize $G = (V, E, \ell)$, where $V = \{s\}$, $E = \emptyset$, $m(s) = m(s_A) = (-1, -1)$, $\ell(s) = \#$
 3:     $F(s) = \{s_A\}$        ▷ $F(u)$ is the set of nodes in $A$ corresponding to $u$ in $G$
 4:     while there exists $u \in V$ with $d^+(u) = 0$ and $\ell(u) \neq \$$ do
 5:         Initialize $N_c = \emptyset$ for all $c \in \Sigma \cup \{\$\}$
 6:         for all $(x, y) \in E_A$ such that $x \in F(u)$ do
 7:             Add $y$ to $N_c$, where $c = \ell_A(y)$
 8:         Initialize $P = \emptyset$
 9:         for all $N_c \neq \emptyset$ do
10:             $i_c = \min\{i \mid (i, j) = m(z), z \in N_c\}$
11:             $j_c = \min\{j \mid (i, j) = m(z), z \in N_c\}$
12:             Add $(i_c, j_c)$ to $P$
13:         for all $N_c \neq \emptyset$ and $p \in P$ do
14:             Remove all $z$ from $N_c$ such that $p < m(z)$
15:         for all $N_c \neq \emptyset$ do
16:             if no node $v \in V$ has $F(v) = N_c$ then
17:                 Add new node $w$ to $V$
18:                 Set $F(w) = N_c$, $m(w) = (i_c, j_c)$, $\ell(w) = c$
19:             Add edge $(u, w)$ to $E$
20:     return $G = (V, E)$
```

Datasets. To evaluate the effectiveness of our methods, we selected three datasets with diverse compositions and varying sequence lengths, as shown in Table 1:

- `random`: Random sequences of 4 symbols generated using the uniform distribution from the standard C++ library.
- `fish−mito`: 25 mitochondrial genomes of fish species from the suborder Labroidei, sourced from [10].
- `hiv−1`: 43 complete HIV-1 genomes, referenced in the literature [27].

Table 1 Datasets.

Datasets	*Composition*	*# Sequences*	*Avg # Base Pairs*	*Source*
`random`	Random sequences	2	3000	stdlib
`fish−mito`	Fish assemblies	25	16624	[10]
`hiv−1`	HIV genomes	43	9267	[27]

Index data structures. We implemented the following indexing data structures in C++ and evaluated them based on the number of nodes, edges, and construction time:

- M-DAG: The DAG storing MCSs presented in [7].
- CSA-all: The Common Subsequence Automaton [8] storing all common sequences, both non-maximal and maximal, and implemented as a labeled DAG (as presented in the first phase of Section 2.3).

- CSA-maximal: The automaton storing only MCS, derived from CSA-all using the method described in Section 2.3.
- CSA-filtered: An optimized version of CSA-all, which filters out many non-maximal common subsequences during construction, as detailed below in Section 3.3.
- McDag: Our proposed data structure from Section 2, implemented as the DAG storing only MCS derived from CSA-filtered, using the method described in Section 2.3.
- MCS-minimized: The minimized version of M-DAG, CSA-maximal, and McDag.Note that, as these three DAGs encode the same language, when minimized they converge to the unique optimal index MCS-minimized.

Qualitative data. The baseline for our experimental study involves finding $MCS(X,Y)$ where X and Y are sequences selected from `random`, `fish−mito`, and `hiv−1`. This results in six unordered pairs: `random` vs `random`, `fish−mito` vs `fish−mito`, `hiv−1` vs `hiv−1`, `random` vs `fish−mito`, `random` vs `hiv−1`, and `fish−mito` vs `hiv−1`. The distribution by length of $MCS(X,Y)$, displaying the number of MCS for each length, was shown in Figure 3 of the introduction. For uniformity, we truncated all sequences to $n = 3000$ base pairs, though longer sequences have been analyzed and resemble the current results.

Table 2 provides statistics for the comparisons among these sequences. For each pair X, Y, the columns report the number of sequences in $MCS(X,Y)$, the number of sequences in $LCS(X,Y) \subseteq MCS(X,Y)$, the length of an LCS, and the number of nodes and edges of the minimal automaton MCS-minimized for storing $MCS(X,Y)$. Note that for random data, the number of MCSs quickly explodes and reaches the `max-val` of the machine word. Intuitively, a larger #MCS indicates that X and Y are farther apart. The table also emphasizes the need for a compact index, since it appears evident that generating all MCSs or all LCSs is unfeasible.

Table 2 Statistics on compared pairs, all sequences truncated to 3000 base pairs.

Comparison Pair X vs Y	*# MCS*	*# LCS*	*Len(LCS)*	*# Nodes*	*# Edges*
`fish−mito` vs `fish−mito`	2.33×10^{284}	1.42×10^{17}	2638	2235772	3656773
`hiv−1` vs `hiv−1`	4.45×10^{289}	1.92×10^{11}	2541	2255709	3623730
`random` vs `random`	`max-val`	$8{,}79\times10^{65}$	1946	2152163	3704365
`hiv−1` vs `fish−mito`	1.06×10^{295}	4.97×10^{56}	1910	2239742	3684099
`random` vs `hiv−1`	2.78×10^{303}	2.57×10^{69}	1905	2181229	3710081
`random` vs `fish−mito`	7.62×10^{303}	7.44×10^{61}	1919	2139712	3627623

3.2 Index size

We analyzed the computational cost of the index data structures M-DAG, CSA-maximal, McDag, and MCS-minimized in terms of the number of nodes, edges, and construction time. Figure 4 shows two plots: the left plot displays the number of nodes and the right plot shows the number of edges for M-DAG, CSA-maximal, McDag, and MCS-minimized as the sequence length n increases. The x-axis reports the sequence length n, while the y-axis reports the corresponding number of nodes/edges. Data from `hiv−1` are presented.

M-DAG consistently has more nodes/edges than CSA-maximal, which in turn has more than McDag. The former two were interrupted after a timeout of 8000 seconds (giving a truncated plot, respectively, for $n = 2300$ and $n = 5700$), while McDag completed in less than 600 seconds. Indeed, the construction of McDag benefits from the optimization that we will discuss in Section 3.3.

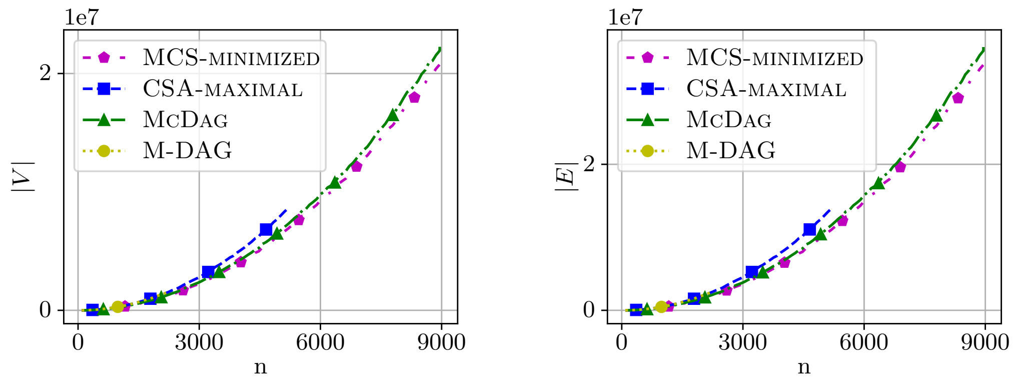

Figure 4 Number of nodes and edges in the DAGs on the dataset `hiv−1`.

All plotted curves are below n^2, empirically established for all of our datasets. We plot the data for McDag and MCS-minimized, along with the curve for n^2 to ease the comparison in Figure 5. McDag is closest to MCS-minimized, with nodes/edges only 4-7% more than MCS-minimized (compared to M-DAG's 26-31% and CSA-maximal's 19-27%).

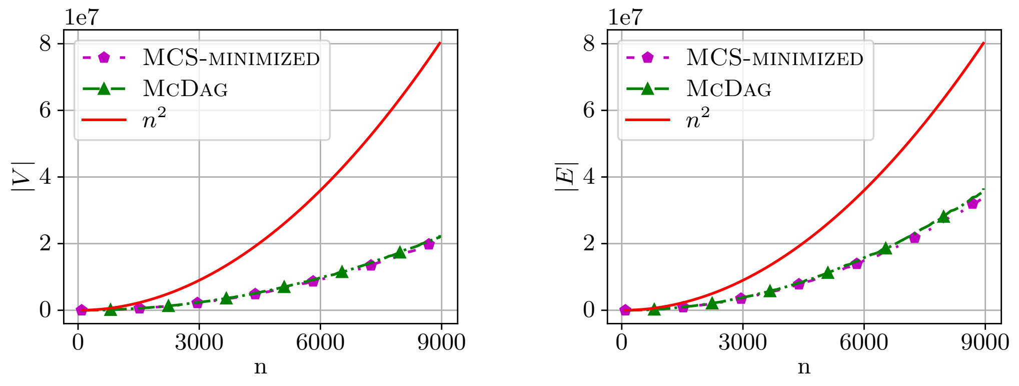

Figure 5 Number of nodes and edges for McDag and MCS-minimized, in comparison with n^2.

It remains an open problem to prove that the number of nodes and edges in McDag is always $< n^2/c$ for some constant $c > 0$, regardless of the sequence alphabet. Synthetic sequences X and Y of length n can be defined to yield $\Omega(n^2)$ nodes and edges in McDag, but we found no real-world or synthetic sequences exceeding n^2 nodes or edges. For example, in Figure 6, we fix $n = 3000$ on `random` data, and report the number of nodes and edges for McDag and MCS-minimized for varying $|\Sigma|$ compared to $n^2 = 9 \cdot 10^6$.

3.3 Filtering and construction time

We described McDag's construction in Section 2.3 using a vanilla version A of the Common Subsequence Automaton [8]. We denote this by CSA-all in our experiments, and the resulting DAG from Algorithm 1 is CSA-maximal.

However, we can further optimize this before executing Algorithm 1. We build a filtered version of the CSA by reading input strings X and Y from left to right, and building the edges forward, so that it is deterministic. To reduce the number of inserted edges, when analyzing a node x we do not add an out-neighbor y if there exists a match m such that

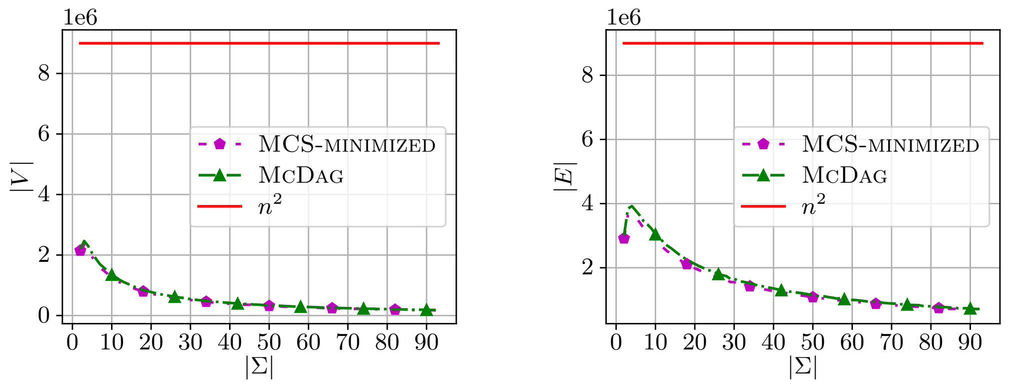

Figure 6 Number of nodes and edges in `random` for increasing alphabet size $|\Sigma|$, vs $n^2 = 9 \cdot 10^6$.

$m(x) < m < m(y)$. Note that the presence of such match implies that the all paths that pass from x to y are not maximal by definition. We then simulate an intersection with CSA-ALL to filter out some common subsequences, ultimately obtaining CSA-FILTERED, which becomes the smaller A to give as input to Algorithm 1. As we show, this process is more effective in reducing the number of nodes and edges, and its resulting graph is the choice for McDag. This choice is conceptually motivated by the computational paths shown on the left of Figure 7, where we report the number of nodes over time on dataset `hiv−1`:

- The blue path is the one in Section 2.3: we start from CSA-ALL, obtain CSA-MAXIMAL, and then, using Revuz's algorithm [22], minimize to obtain MCS-MINIMIZED.
- The green path is its optimized version: we start from the filtered version (which is symmetrical to CSA-ALL and takes the same time), obtain CSA-FILTERED, get McDag, and then MCS-MINIMIZED using Revuz's algorithm, as in the blue path.

As noted, both paths lead to the same MCS-MINIMIZED, but the green path takes less time and generates fewer intermediate nodes. Therefore, we decided to implement McDag using the green path in our experiments.

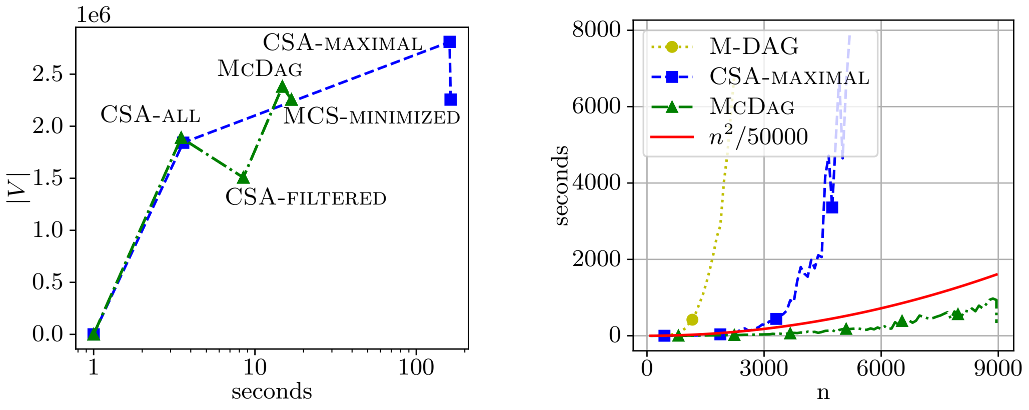

Figure 7 Filtering out and construction time on the `hiv−1` dataset.

We provide more details on how to obtain CSA-FILTERED. The idea is to build the co-deterministic CSA-ALL with an additional filter provided by the filtered version of the deterministic CSA we described earlier; we force each node v of CSA-FILTERED to correspond to the set $F(v)$ of nodes of CSA that share at least one suffix with v. With that information, we allow for the addition of an edge (u, v) only if $F(u)$ is non-empty. This way, we avoid adding strings that are not present in the filtered deterministic CSA, as they cannot be maximal. Note that, unlike in Algorithm 1, we do not permit duplicate matches, so the size of CSA-FILTERED is always smaller than the size of CSA-ALL. We apply the same filtering approach as illustrated above, not adding an edge if there is a match that inserts between the matches of the two nodes. In the end, only paths whose matchings allow for match insertions of characters are filtered out, as they are not maximal by definition. Moreover, CSA-FILTERED is a rightmost approximate co-deterministic MCS index, so it can be used as input to Algorithm 1.

Finally, the plot on the right of Figure 7 shows the running times for constructing the DAGs. All methods were implemented as fairly as possible. McDag scales well compared to the plot of $n^2/50000$, which is significant given that McDag also computes $LCS(X, Y)$, which has a quadratic conditional lower bound for computation [1, 4].

4 Correctness of McConstruct (Algorithm 1)

Here we prove the correctness of algorithm McConstruct. The algorithm takes as input the output A of the first phase, which is an approximate co-deterministic index with the rightmost property, and outputs a McDag index G as described in Section 2.3. First, we show a number of necessary properties that are satisfied by McDag. Then, to show that st-paths correspond to MCS, we conclude the proof through a characterization of the shape of non-maximal CS in similar data structures, showing that they do not occur in our McDag.

For a set of matches $X = \{(i_1, j_1), ..., (i_h, j_h)\}$ corresponding to the same character $c \in \Sigma$ we define their minimum as the match given by the minimum over both components: $\min(X) = (\min_{1\leq k\leq h} i_k, \min_{1\leq k\leq h} j_k)$. For a set of nodes $Y = \{v_1, ..., v_k\}$, we define their corresponding set of matches as $m(Y) = \{m(v_1), ..., m(v_k)\}$.

▶ **Theorem 1.** *During the algorithm, we retain the following properties for graph $G = (V, E, \ell)$ (and thus for McDag at the end):*

(i) *G is deterministic;*

(ii) *each node $v \in V$ is labeled with a symbol $\ell(v)$ from $\Sigma \cup \{\#, \$\}$, and is associated to a set of nodes $F(v)$ of A, all labeled with $\ell(v)$. Furthermore, it is associated with match $m(v) = \min(m(F(v)))$ for character $\ell(v)$.*

(iii) *G is a labeled DAG with a single source s, having $F(s) = m(s) = (-1, -1)$ and $\ell(s) = \#$;*

(iv) *If $(u, v) \in E$, then $m(u) < m(v)$.*

(v) *each path $P = s, v_1, ..., v_h$ in G is associated with unique string $str(P) = \ell(v_1), ..., \ell(v_h)$, which is a common subsequence of X and Y occurring at the pairs of (increasing) positions $m(v_1), ..., m(v_h)$.*

Proof. Conditions (i) and (ii) are immediate by construction. As for (iii), at the beginning node s is added to G with $m(s) = (-1, -1)$ and $\ell(s) = \#$. We only add out-neighbors to existing nodes, and thus we never add new sources. Furthermore, the absence of cycles follows immediately from the same property in A.

Let us now recall what happens when node u is selected to be processed. We define the sets of possible neighbors with respect to each character for all the corresponding nodes $F(u)$ in A: $N_c = \{y \in N_A^+(x) \mid x \in F(u), \ell_A(y) = c\}$. We then filter these nodes, removing the ones with corresponding matches that come after some $\min(m(N_d))$, yielding the final set N_c. For each non-empty N_c, we consider node w_c of G (or add it if it does not exist) such that $F(w_c) = N_c$, $m(w_c) = \min(m(N_c))$, $\ell(w_c) = c$, and add edge (u, w_c) to G. Let us consider a newly added edge, $(u, w_c) \in E$, for some w_c as defined above, and let us show that $m(u) < m(w_c)$ (property (iv)). It is clear that the edge corresponds to at least one $(x, y) \in E_A$ with $x \in F(u)$ and $y \in F(v)$, by definition of N_c. Let $(i, j) = m(w_c) = \min(m(F(w_c)))$. By definition of minimum, there exist $y, y' \in F(w_c)$ such that $m(y)$ has first coordinate i, and $m(y')$ has second coordinate j. By construction, there exist $x, x' \in F(u)$ such that (x, y) and (x', y') are in E_A. This implies, by definition of A, that $m(x) < m(y)$ and $m(x') < m(y')$. Since $m(x)$ and $m(x')$ contribute to the minimum for $m(u)$, we must have that the first coordinate of $m(u)$ is smaller than, or equal to, the one for $m(x)$, which is strictly smaller than i, the one for $m(y)$. With a similar reasoning over x', we show that the second coordinate of $m(u)$ is also strictly smaller than j. Therefore, $m(u) < m(w_c)$, as required.

Lastly, property (v) follows from the corresponding property 3 of approximate MCS index A: since we are never adding edges which have no corresponding ones in A, any path from s in G surely spells a subsequence of an st-path spelled by A, which are still common subsequences. ◀

Note that by construction, Algorithm 1 adds out-neighbors to all nodes u that have label $\ell(u) \neq \$$, hence the sink of McDag has label \$. Moreover, since A has an unique sink, McDag must also have a unique sink, as for all nodes $|N_\$| \leq 1$.

Theorem 1 together with the above observation implies that the graph output by McConstruct satisfies conditions 1-3 of an approximate MCS index. Thus, the only things missing for G to be an index for $MCS(X, Y)$ is that MCSs correspond to st-paths and that st-paths correspond to MCSs. We first show that all MCSs are retained as st-paths of G (condition 4 of an MCS index):

▶ **Lemma 2.** *Each $Z \in MCS(X, Y)$ has a corresponding st-path in the resulting graph $G = (V, E, \ell)$ at the end of the McConstruct procedure.*

Proof. First, recall that each MCS occurred once as an st-path of A. During the construction of G, we have a correspondence between edges $(u, v) \in E$ of G, and edges $(x, y) \in E_A$ with $x \in F(u)$ and $y \in F(v)$, which share the same respective labels. Since indeed they spell the same string, we say that an st-path $s_A = x_1, ..., x_k = t_A$ of A corresponds to an st-path $s = v_1, ..., v_k = t$ of G if $x_i \in F(v_i)$ for all i.

When building the neighbors of some node $u \in V$ during McConstruct, we may discard $(x, y) \in E_A$ with $x \in F(u)$, in the sense that $y \notin F(v)$ for any $v \in N^+_G(u)$. This happens, by construction, if and only if y is removed when filtering set N_c, that is, if and only if there exists a match $m = \min(m(N_d))$ for some $d \in \Sigma$ such that $m < m(y)$. By property (iv), we further have $m(u) < m$. Therefore, we have a pair of matching characters given by m which occur strictly between the matches given by $m(u)$, and the ones given by $m(y)$.

Let us now assume by contradiction that this happens for an edge (x, y) which is traversed by an MCS: let $P = s_A = x_1, ..., x_k = t_A$ be such that $str(P) \in MCS(X, Y)$, and let us assume that h is the minimum index such that there is a path $S = v_1, \ldots, v_h$ corresponding to prefix $x_1, \ldots, x_h$ of P and $x_{h+1} \notin F(w)$ for all $w \in N^+_G(v_h)$. By the reasoning above, there exists a match m such that $m(v_h) < m < m(x_{h+1})$. Let us now consider the following matching: $m(v_1), ..., m(v_h), m, m(x_{h+1}), ..., m(x_k)$. These matches are all strictly increasing (by (v) of Theorem 1 for the prefix, and by property 3 of approximate MCS index A for the suffix), and the consequent spelled subsequence has P as a proper subsequence, a contradiction. ◀

We now have that McDag is a deterministic approximate MCS index. To conclude the proof of correctness, the only thing left to show is that no *st*-path corresponds to a common subsequence that is not maximal. To this end, we give a characterization of the non-maximal common subsequences given by *st*-paths in data structures satisfying some of the conditions of McDag. Then, we show that during the construction of McDag, we eliminate these types of structures.

▶ **Definition 3** (Subsequence Bubble). *Consider a DAG D where each node is labeled with a symbol of $\Sigma \cup \{\#, \$\}$, and let b, e_1, e_2, e be four distinct nodes of D.*

- *A* closed subsequence bubble *is a pair of disjoint be-paths S, L such that $str(S)$ is a proper subsequence of $str(L)$.*
- *An* open subsequence bubble *is a pair of disjoint paths, where S is a be_1-path, and L is a be_2-path, such that $str(S)$ is a proper subsequence of $str(L)$.*

In both cases, S is called the short side *of the bubble, and L the* long side.

Subsequence bubbles are useful for giving a characterization of which *st*-paths correspond to common subsequences that are not maximal, under certain hypotheses:

▶ **Lemma 4.** *Let D be an approximate MCS index, and let P be an st-path of D.*

$str(P)$ is a non-maximal common subsequence if and only if there exists a closed subsequence bubble B such that P traverses the short side S of B.

Proof. If an *st*-path P traverses S, then $str(P)$ cannot be maximal: let the endpoints of B be b, e; the path given by the prefix of P up to b, then L, and then the suffix of P from e to t defines a common subsequence which has $str(P)$ as proper subsequence.

Vice versa, let $str(P)$ be a common subsequence that is not maximal. Then, there exists an *st*-path Q such that $str(P)$ is a proper subsequence of $str(Q)$, since every MCS is represented in the index. Let $b \in V$ be the first node after which $str(P)$ and $str(Q)$ differ; it is well-defined since the first node of both paths is s. Symmetrically, it is well-defined the first node after b which belongs to both paths, which we call e, since both paths end at the same node t. Then, the subpaths P' of P and Q' of Q between nodes b and e form a subsequence bubble, with P traversing the short side: $P' \neq Q'$ since they differ at the node after b, and $str(P')$ is a proper subsequence of $str(Q')$ since they start and end at the same nodes, and thus positions in the strings, with at least one more symbol appearing in Q'. ◀

As mentioned before, McDag is an approximate MCS index, and thus satisfies the hypotheses of Lemma 4. To conclude the proof of correctness of the construction procedure McConstruct, it is sufficient to show that whenever we have an open bubble, we never add a node and edges which "close it".

The first step towards this is the *subsequence mapping* λ between the short side and the long side of a subsequence bubble, to couple nodes that correspond to the same characters in the subsequences. Given a subsequence bubble (open or closed), let $S = b \to v_1 \to v_2 \to \ldots \to v_h$ be its short side and $L = b \to w_1 \to \ldots \to w_k$ be its long side. We will use the order $v_i < v_{i+1}$ and $w_j < w_{j+1}$, which is well-defined by the corresponding relationship between the associated matches, by (iv) of Theorem 1. We thereby define the injective mapping $\lambda : S \to L$ such that $\lambda(v_i) = \min\{w \in L \mid \ell(w) = \ell(v_i) \text{ and } w > \lambda(v_{i-1})\}$, where $\lambda(v_0)$ is improperly considered equal to b. In other words, it is a correspondence between the characters of the short string and the ones of the longer string, indicating the subsequence relationship.

We say that two matches m_1, m_2 are *crossing* if neither $m_1 \leq m_2$ nor $m_2 \leq m_1$. The subsequence mapping has the following properties:

▶ **Lemma 5.** *Let v be a node along the short side S of a subsequence bubble of a* McDag *(V, E). Then, if $v \neq \lambda(v)$, $\forall m_s \in m(F(v))$ and $\forall m_l \in m(F(\lambda(v)))$, it holds that $m_l \neq m_s$, and that either $m_s \leq m_l$ or m_s and m_l are crossing. In particular, note that the first condition is equivalent to $F(v) \cap F(\lambda(v)) = \emptyset$.*

Proof. Let the short side be $S = b, v_1, ..., v_h$, and the long side $L = b, w_1, ..., w_k$. We split the proof according to whether v is the first node of the bubble, or not.

First, consider $v = v_1$. By definition, v and $\lambda(v)$ are labelled with the same symbol. Therefore, by determinism of McDag, $\lambda(v) \neq w_1$. Therefore, for each $z \in F(\lambda(v))$ there are nodes $q \in F(b), y \in F(w_1)$ such that $m(q) < m(y) < m(z) =: m_l$. By construction z is filtered, that is $z \notin F(u)$, for all nodes u such that $(b, u) \in E$, and in particular $z \notin F(v)$. Thus, m_l is different from any match in $F(v)$. Let now $x \in F(v)$, and $m_s = m(x)$. If we had $m_l \leq m_s$, then we would have $m(q) < m(y) < m_s$, which again is impossible by construction. Therefore, either m_s crosses m_l, or $m_s \leq m_l$.

Let us now prove the inductive case. Assume that the thesis holds for all $1 \leq j < i$, and let $v = v_i$. Suppose by contradiction that there exist $x \in F(v_i)$ and $z \in F(\lambda(v_i))$ such that $m_l := m(z) \leq m(x) =: m_s$. Since there are paths from v_{i-1} to v_i, and from $\lambda(v_{i-1})$ to $\lambda(v_i)$, we have that there exist $x' \in F(v_{i-1}), z' \in F(\lambda(v_{i-1}))$ such that $m(x') < m(x) = m_s$ and $m(z') < m(z) = m_l$. By contradiction hypothesis, $m(z') < m_l \leq m_s$, and thus we have $m(x'), m(z') < m_s$. This contradicts the rightmost property of A: indeed, consider the match given by the maximum of the coordinates of $m(x'), m(z')$. This match is different from x' by induction hypothesis, as it does not hold that $m(z') \leq m(x')$. Thus, the match follows $m(x')$ and strictly precedes m_s, and still this match does not correspond to the in-neighbor for character $\ell_A(x')$ of x, contradiction. ◀

▶ **Theorem 6.** *The graph $G = (V, E)$ obtained at the end of procedure* McConstruct *does not have any closed bubbles.*

Proof. By contradiction, let $S = b, v_1, ..., v_h, e$ and $L = b, w_1, ..., w_k, e$ be respectively the short and long sides of a subsequence bubble in G. We show that the edge (v_h, e) could not have been added to G during construction. By contradiciton, assume that it happens. Let $\lambda(v_h) = w_i$ and let $y \in F(e)$ and $m_e := m(y)$ the match for y. Since we have edge (v_h, e) in G, by construction we have $x \in F(v_h)$ such that $(x, y) \in E_A$. We have no occurrences of $\ell_A(x)$ between $m(x)$ and m_e. Furthermore, we have $m(v_h) \leq m(x)$ by definition. Now, consider node $\lambda(v_h) = w_i$, which has the same associated label $\ell_A(x)$. Since there is a path from w_i to e, we can choose node $x' \in F(w_i)$ such that we have a path from x' to y in A. In particular, this implies that $m(x') < m(y) = m_e$. By Lemma 5, we have $x' \neq x$, and we further have either $m(x') \geq m(x)$, or $m(x')$ crossing $m(x)$. Both lead to a contradiction: there would be an occurrence of character $\ell_A(x) = \ell_A(x')$ after one of the two endpoints of $m(x)$, but before the endpoints of m_e. ◀

5 Conclusions

In this paper, we presented a novel method for building a compact index of all maximal common subsequences (MCS) of two strings, which is simple to understand and implement. We empirically evaluated our method on synthetic and DNA sequences from public datasets, demonstrating its effectiveness and efficiency.

Our method introduces a significant advancement in the indexing of MCS, providing a practical solution for applications in bioinformatics, text processing, and other fields requiring efficient sequence analysis. By focusing on a co-deterministic approach and utilizing filtering techniques, we have shown that it is possible to construct a compact and precise index that can handle substantial datasets, utilizing only 4-7% more than the minimum required nodes. It remains an open problem to prove whether McDag has always less than n^2 nodes and edges, independently of the alphabet size $|\Sigma|$, and give tight bounds on its construction cost.

In future research, we will investigate the relevance of the proposed approximate MCS indices and how they compete with their exact counterparts. We aim to explore their performance in various real-world applications, identifying scenarios where approximate indices may offer substantial advantages in terms of speed and memory usage. To elicit interest in subsequence-based methods, we plan to investigate the performance of MCSs and how they compare with the k-mers methods commonly used in tasks such as sequence alignment and sequence similarity estimation.

Moreover, we plan to extend the scalability of our approach. This includes pushing further the length n of the analyzed sequences by employing advanced space-saving techniques, leveraging parallelism, and utilizing SIMD (Single Instruction, Multiple Data) instructions. Ultimately, our goal is to provide a robust and versatile tool for MCS indexing that balances accuracy, efficiency, and practicality, contributing to the advancement of sequence analysis methodologies and their applications.

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 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.

3 Ricardo A Baeza-Yates. Searching subsequences. *Theoretical Computer Science*, 78(2):363–376, 1991. doi:10.1016/0304-3975(91)90358-9.

4 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.

5 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.

6 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.

7 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.

8 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.

9 Maxime Crochemore and Zdeněk Troníček. Directed acyclic subsequence graph for multiple texts. *Rapport IGM*, pages 99–13, 1999.

10 Christoph Fischer, Stephan Koblmüller, Christian Gülly, Christian Schlötterer, Christian Sturmbauer, and Gerhard G. Thallinger. Complete mitochondrial dna sequences of the threadfin cichlid (petrochromis trewavasae) and the blunthead cichlid (tropheus moorii) and patterns of mitochondrial genome evolution in cichlid fishes. *PLOS ONE*, 8(6):1–14, June 2013. doi:10.1371/journal.pone.0067048.

11 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.

12 Ronald I. Greenberg. Bounds on the number of longest common subsequences. *CoRR*, cs.DM/0301030, 2003. URL: http://arxiv.org/abs/cs/0301030.

13 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.

14 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.

15 W. J. Hsu and M. W. Du. Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, 24(1):45–59, 1984. doi:10.1007/BF01934514.

16 Robert W Irving and Campbell B Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In *Combinatorial Pattern Matching: Third Annual Symposium Tucson, Arizona, USA, April 29–May 1, 1992 Proceedings 3*, pages 214–229. Springer, 1992. doi:10.1007/3-540-56024-6_18.

17 Elsa Loekito, James Bailey, and Jian Pei. A binary decision diagram based approach for mining frequent subsequences. *Knowl. Inf. Syst.*, 24(2):235–268, 2010. doi:10.1007/s10115-009-0252-9.

18 Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994. doi:10.1109/71.298210.

19 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.

20 Borivoj Melichar and Tomás Polcar. The longest common subsequence problem A finite automata approach. In Oscar H. Ibarra and Zhe Dang, editors, *Implementation and Application of Automata, 8th International Conference, CIAA 2003, Santa Barbara, California, USA, July 16-18, 2003, Proceedings*, volume 2759 of *Lecture Notes in Computer Science*, pages 294–296. Springer, Springer, 2003. doi:10.1007/3-540-45089-0_27.

21 Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*, DAC '93, pages 272–277, New York, NY, USA, 1993. ACM Press. doi:10.1145/157485.164890.

22 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.

23 Yoshifumi Sakai. Maximal common subsequence algorithms. *Theor. Comput. Sci.*, 793:132–139, 2019. doi:10.1016/j.tcs.2019.06.020.

24 Yuto Shida, Giulia Punzi, Yasuaki Kobayashi, Takeaki Uno, and Hiroki Arimura. Finding diverse strings and longest common subsequences in a graph. *CoRR*, abs/2405.00131, 2024. doi:10.48550/arXiv.2405.00131.

25 Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

26 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`.

27 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`.