



Prefix Sorting DFAs: A Recursive Algorithm

Nicola Cotumaccio  

Gran Sasso Science Institute, L'Aquila, Italy
Dalhousie University, Halifax, Canada

Abstract

In the past thirty years, numerous algorithms for building the suffix array of a string have been proposed. In 2021, the notion of suffix array was extended from strings to DFAs, and it was shown that the resulting data structure can be built in $O(m^2 + n^{5/2})$ time, where n is the number of states and m is the number of edges [SODA 2021]. Recently, algorithms running in $O(mn)$ and $O(n^2 \log n)$ time have been described [CPM 2023].

In this paper, we improve the previous bounds by proposing an $O(n^2)$ recursive algorithm inspired by Farach's algorithm for building a suffix tree [FOCS 1997]. To this end, we provide insight into the rich lexicographic and combinatorial structure of a graph, so contributing to the fascinating journey which might lead to solve the long-standing open problem of building the suffix tree of a graph.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Pattern matching

Keywords and phrases Suffix Array, Burrows-Wheeler Transform, FM-index, Recursive Algorithms, Graph Theory, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2023.22

Related Version *Full Version*: <https://arxiv.org/abs/2305.02526>

Funding This work was partially funded by Dante Labs.

Acknowledgements I thank Nicola Prezza for pointing out the paper [22].

1 Introduction

The *suffix tree* [31] of a string is a versatile data structure introduced by Weiner in 1973 which allows solving a myriad of combinatorial problems, such as determining whether a pattern occurs in the string, computing matching statistics, searching for regular expressions, computing the Lempel-Ziv decomposition of the string and finding palindromes. The book by Gusfield [18] devotes almost 150 pages to the applications of suffix trees, stressing the importance of these applications in bioinformatics. However, the massive increase of genomic data in the last decades requires space-efficient data structures able to efficiently support pattern matching queries, and the space consumption of suffix trees is too high. In 1990, Manber and Myers invented *suffix arrays* [25] as a space-efficient alternative to suffix trees. While suffix arrays do not have the full functionality of suffix trees, they still allow solving pattern matching queries. Suffix arrays started a new chapter in data compression, which culminated in the invention of data structures closely related to suffix arrays, notably, the Burrows-Wheeler Transform [4] and the FM-index [14, 16], which have heavily influenced sequence assembly [30].

The impact of suffix arrays has led to a big effort in the attempt of designing efficient algorithms to construct suffix arrays, where “efficient” refers to various metrics (worst-case running time, average running time, space, performance on real data and so on); see [28] for a comprehensive survey on the topic. Let us focus on worst-case running time. Manber and Myers build the suffix array of a string of length n in $O(n \log n)$ by means of a *prefix-doubling* algorithm [25]. In 1997, Farach proposed a recursive algorithm to build the *suffix tree* of a



© Nicola Cotumaccio;

licensed under Creative Commons License CC-BY 4.0

34th International Symposium on Algorithms and Computation (ISAAC 2023).

Editors: Satoru Iwata and Naonori Kakimura; Article No. 22; pp. 22:1–22:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

string in linear time for integer alphabets [12]. In the following years, the recursive paradigm of Farach’s algorithm was used to develop a multitude of linear-time algorithms for building the suffix array [23, 20, 21, 26]. All these algorithms carefully exploit the lexicographic structure of the suffixes of a string, recursively reducing the problem of computing the suffix array of a string to the problem of computing the suffix array of a smaller string (*induced sorting*).

The problem of solving pattern matching queries not only on strings, but also on labeled graphs, is an active topic of research. Recently, Equi et al. showed that no algorithm can solve pattern matching queries *on arbitrary graphs* in $O(m^{1-\epsilon}|P|)$ time or $O(m|P|^{1-\epsilon})$ (where m is the number of edges, P is the pattern and $\epsilon > 0$), unless the Orthogonal Vectors hypothesis fails [11, 10]. On the other hand, over the years the idea of (lexicographically) sorting the suffixes of a string has been generalized to graphs, thus leading to compact data structures that are able to support pattern matching queries on graphs. The mechanism behind the suffix array, the Burrows-Wheeler Transform and the FM-index was first generalized to trees [13, 15]; later on, it was generalized to De Bruijn graphs [3, 24] (which can be used for Eulerian sequence assembly [19]). Subsequently, it was extended to the so-called Wheeler graphs [17, 1], and finally to arbitrary graphs and automata [9, 6]. The idea of lexicographically sorting the strings reaching the states of an automaton has also deep theoretical consequences in automata theory: for example, it leads to a parametrization of the powerset construction, which implies fixed-parameter tractable algorithms for PSPACE-complete problems such as deciding the equivalence of two non-deterministic finite automata (NFAs) [8].

The case of deterministic finite automata (DFAs) is of particular interest, because in this case the notion of “suffix array” of a DFA has a simple interpretation in terms of strings. Assume that there is a fixed total order \preceq on the alphabet Σ of a DFA \mathcal{A} , and extend \preceq lexicographically to the set of all infinite strings on Σ . Assume that each state has at least one incoming edge. If u is a state, consider the set I_u of all infinite strings that can be read starting from u and following edges *in a backward fashion*, and let \min_u and \max_u be the lexicographically smallest (largest, respectively) string in I_u (see Figure 1). Consider the partial order $\preceq_{\mathcal{A}}$ on the set of all states Q such that for every $u, v \in Q$, with $u \neq v$, it holds $u \prec_{\mathcal{A}} v$ if and only if $\max_u \preceq \min_v$. Then, the partial order $\preceq_{\mathcal{A}}$ induces a (partial) permutation of the set of all states that plays the same role of the permutation of text positions induced by the suffix array of a string [9, 22], so $\preceq_{\mathcal{A}}$ is essentially the “suffix array” of the DFA. If we are able to compute the partial order $\preceq_{\mathcal{A}}$ (and a minimum-size partition of Q into sets such that the restriction of $\preceq_{\mathcal{A}}$ to each set is a *total* order), then we can efficiently and compactly solve pattern matching queries on the DFA by means of techniques that extend the Burrows-Wheeler transform and the FM-index from strings to graphs. As a consequence, we now have to solve the problem of efficiently building the “suffix array” of a DFA, that is, the problem of computing $\preceq_{\mathcal{A}}$.

The first paper on the topic [9] presents an algorithm that builds $\preceq_{\mathcal{A}}$ in $O(m^2 + n^{5/2})$ time, where n is the number of states and m is the number of edges. In a recent paper [22], Kim et al. describe two algorithms running in $O(mn)$ and $O(n^2 \log n)$ time. The key observation is that, if we build the *min/max-partition* of the set of all states, then we can determine the partial order $\preceq_{\mathcal{A}}$ in linear time by means of a reduction to the interval partitioning problem. Determining the min/max-partition of the set of all states means picking the string \min_u and \max_u for every state $u \in Q$, and sorting these $2|Q|$ strings (note that some of these strings may be equal, see Figure 1 for an example). As a consequence, we are only left with the problem of determining the min/max-partition efficiently.

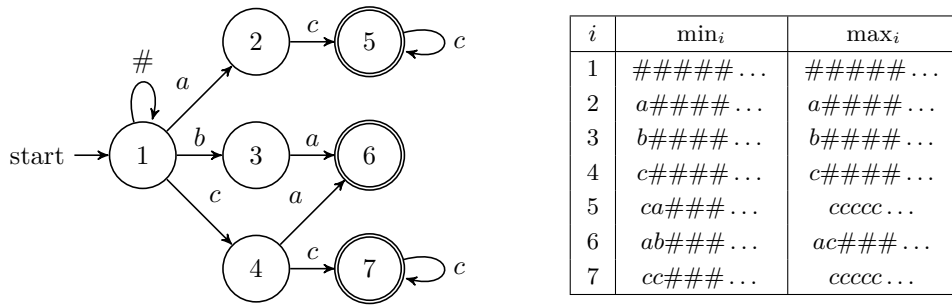


Figure 1 A DFA \mathcal{A} , with the minimum and maximum string reaching each state (we assume $\# < a < b < c$). The min/max partition is given by $\{(1, \min), (1, \max)\} < \{(2, \min), (2, \max)\} < \{(6, \min)\} < \{(6, \max)\} < \{(3, \min), (3, \max)\} < \{(4, \min), (4, \max)\} < \{(5, \min)\} < \{(7, \min)\} < \{(5, \max), (7, \max)\}$, meaning that $\min_1 = \max_1 < \min_2 = \max_2 < \min_6 < \max_6 < \min_3 = \max_3 < \min_4 = \max_4 < \min_5 < \min_7 < \max_5 = \max_7$.

The $O(n^2 \log n)$ algorithm builds the min/max-partition by generalizing Manber and Myers’s $O(n \log n)$ algorithm from strings to DFAs. However, since it is possible to build the suffix array of a string in $O(n)$ time, it is natural to wonder whether it is possible to determine the min/max-partition in $O(n^2)$ time.

In this paper, we show that, indeed, it is possible to build the min/max-partition in $O(n^2)$ time by adopting a recursive approach inspired by one of the linear-time algorithms that we have mentioned earlier, namely, Ko and Aluru’s algorithm [23]. As a consequence, our algorithm is asymptotically faster than all previous algorithms.

A long-standing open problem is whether it is possible to define a suffix tree of a graph. Some recent work [5] suggests that it is possible to define data structures that *simulate* the behavior of a suffix tree by carefully studying the lexicographic structure of the graph (the results in [5] only hold for Wheeler graphs, but we believe that they can be extended to arbitrary graphs). More specifically, it is reasonable to believe that it is possible to generalize the notion of *compressed suffix tree* of a string [29] to graphs. A compressed suffix tree is a compressed representation of a suffix tree which consists of some components, including a suffix array. We have already seen that $\preceq_{\mathcal{A}}$ generalizes the suffix array to a graph structure, and [5] suggests that the remaining components may also be generalized to graphs. The complements of the suffix tree of a string heavily exploit the lexicographic and combinatorial structure of a string. Since the algorithm that we present in this paper deeply relies on the richness of the lexicographic structure (which becomes even more challenging and surprising when switching from a string setting to a graph setting), we believe that our results also provide a solid conceptual contribution towards extending suffix tree functionality to graphs.

We remark that, a few days after we submitted this paper to arXiv, a new arXiv preprint showed how to determine the min/max partition in $O(m \log n)$ time, where n is the number of states and m is the number of edges (this new arXiv preprint was also accepted for publication [2]). If the graph underlying the DFA is sparse, then the algorithm in [2] improves our $O(n^2)$ algorithm. Since the $O(m \log n)$ algorithm uses different techniques (it is obtained by adapting Paige and Tarjan’s partition refinement algorithm [27]), we are left with the intriguing open problem of determining whether, by possibly combining the ideas behind our algorithm and the algorithm in [2], it is possible to build the min/max partition in $O(m)$ time.

Due to space constraints, all proofs can be found in the full version of this paper [7].

2 Preliminaries

2.1 Relation with Previous Work

In the setting of the previous works on the topic [1, 9, 22], the problem that we want to solve is defined as follows. Consider a deterministic finite automaton (DFA) such that (i) all edges entering the same state have the same label, (ii) each state is reachable from the initial state, (iii) each state is co-reachable, that is, it is either final or it allows reaching a final state, (iv) the initial state has no incoming edges. Then, determine the min/max partition of the set of states (see Section 2.2 for the formal definition of min/max partition, and see Figure 1 for an example).

- Assumptions (ii) and (iii) are standard assumptions in automata theory, because all states that do not satisfy these assumptions can be removed without changing the accepted language.
- In this setting, all the non-initial states have an incoming edge, but the initial state has no incoming edges. This implies for some state u it may hold $I_u = \emptyset$ (remember that I_u is the set of all infinite strings that can be read starting from u and following edges in a backward fashion, see the introduction), so Kim et al. [22] need to perform a tedious case analysis which also takes finite strings into account in order to define the min/max-partition (in particular, the minimum and maximum strings reaching the initial state are both equal to the empty string). However, we can easily avoid this complication by means of the same trick used in [5]; we can add a self-loop to the initial state, and the label of the self-loop is a special character $\#$ *smaller* than any character in the alphabet. Intuitively, $\#$ plays the same role as the termination character in the Burrows-Wheeler transform of a string, and since $\#$ is the smallest character, adding this self-loop does not affect the min/max-partition (see [22] for details).
- Notice that the initial state and the set of all final states play no role in the definition of the min/max partition; this explains why, more generally, it will be expedient to consider deterministic graphs rather than DFAs (otherwise we would need to artificially add an initial state and add a set of final states when we recursively build a graph in our algorithm). Equivalently, one may assume to work with semiautomata in which the transition function is not necessarily total. This justifies the assumptions that we will make in Section 2.2.
- Some recent papers [6, 8] have shown that assumptions (i) and (iv) can be removed. The partial order $\preceq_{\mathcal{A}}$ is defined analogously, and all the algorithms for building $\preceq_{\mathcal{A}}$ that we have mentioned still work. Indeed, if a state u is reached by edges with the distinct labels, we need to only consider all edges with the smallest label when computing \min_u and all edges with the largest label when computing \max_u ; moreover, we once again assume that the initial state has a self loop labeled $\#$. The only difference is that assumption (i) implies that $m \leq n^2$ (n being the number of states, m being the number of edges) because each state can have at most n incoming edges, but this is no longer true if we remove assumption (i). As a consequence, the running time of our algorithm is no longer $O(n^2)$ but $O(m + n^2)$ (and the running time of the $O(n^2 \log n)$ algorithm in [22] becomes $O(m + n^2 \log n)$) because we still need to process all edges in the DFA.

To sum up, all the algorithms for computing $\preceq_{\mathcal{A}}$ work on arbitrary DFAs.

2.2 Notation and First Definitions

Let Σ be a finite alphabet. We consider finite, edge-labeled graphs $G = (V, E)$, where V is the set of all nodes and $E \subseteq V \times V \times \Sigma$ is the set of all edges. Up to taking a subset of Σ , we assume that all $c \in \Sigma$ label some edge in the graph. We assume that all nodes have at least one incoming edge, and all edges entering the same node u have the same label $\lambda(u)$ (*input consistency*). This implies that an edge $(u, v, a) \in E$ can be simply denoted as (u, v) , because it must be $a = \lambda(v)$. In particular, it must be $|E| \leq |V|^2$ (and so an $O(|E|)$ algorithm is also a $O(|V|^2)$ algorithm). If we do not know the $\lambda(u)$'s, we can easily compute them by scanning all edges. In addition, we always assume that G is *deterministic*, that is, for every $u \in V$ and for every $a \in \Sigma$ there exists at most one $v \in V$ such that $(u, v) \in E$ and $\lambda(v) = a$.

Let Σ^* be the set of all finite strings on Σ , and let Σ^ω be the set of all (countably) right-infinite strings on Σ . If $\alpha \in \Sigma^* \cup \Sigma^\omega$ and $i \geq 1$, we denote by $\alpha[i] \in \Sigma$ the i^{th} character of α (that is, $\alpha = \alpha[1]\alpha[2]\alpha[3]\dots$). If $1 \leq i \leq j$, we define $\alpha[i, j] = \alpha[i]\alpha[i+1]\dots\alpha[j-1]\alpha[j]$, and if $j < i$, then $\alpha[i, j]$ is the empty string ϵ . If $\alpha \in \Sigma^*$, then $|\alpha|$ is length of α ; for every $0 \leq j \leq |\alpha|$ the string $\alpha[1, j]$ is a *prefix* of α , and if $0 \leq j < |\alpha|$ it is a *strict prefix* of α ; analogously, one defines suffixes and strict suffixes of α . An *occurrence* of $\alpha \in \Sigma^*$ starting at $u \in V$ and ending at $u' \in V$ is a sequence of nodes $u_1, u_2, \dots, u_{|\alpha|+1}$ of V such that (i) $u_1 = u$, (ii) $u_{|\alpha|+1} = u'$, (iii) $(u_{i+1}, u_i) \in E$ for every $1 \leq i \leq |\alpha|$ and (iv) $\lambda(u_i) = \alpha[i]$ for every $1 \leq i \leq |\alpha|$. An *occurrence* of $\alpha \in \Sigma^\omega$ starting at $u \in V$ is a sequence of nodes $(u_i)_{i \geq 1}$ of V such that (i) $u_1 = u$, (ii) $(u_{i+1}, u_i) \in E$ for every $i \geq 1$ and (iii) $\lambda(u_i) = \alpha[i]$ for every $i \geq 1$. Intuitively, a string $\alpha \in \Sigma^* \cup \Sigma^\omega$ has an occurrence starting at $u \in V$ if we can read α on the graph starting from u and following edges *in a backward fashion*.

In the paper, occurrences of strings in Σ^ω will play a key role, while occurrences of strings in Σ^* will be used as a technical tool. For every $u \in V$, we denote by I_u the set of all strings in Σ^ω admitting an occurrence starting at u . Since every node has at least one incoming edge, then $I_u \neq \emptyset$.

A *total order* \leq on a set V if a reflexive, antisymmetric and transitive relation on V . If $u, v \in V$, we write $u < v$ if $u \leq v$ and $u \neq v$.

Let \preceq be a fixed total order on Σ . We extend \preceq to $\Sigma^* \cup \Sigma^\omega$ *lexicographically*. It is easy to show that in every I_u there is a lexicographically smallest string \min_u and a lexicographically largest string \max_u (for example, it follows from [22, Observation 8]).

We will often use the following immediate observation. Let $u \in V$, and let $(u_i)_{i \geq 1}$ be an occurrence of \min_u . Fix $i \geq 1$. Then, $(u_j)_{j \geq i}$ is an occurrence of \min_{u_i} , and $\min_u = \min_u[1, i-1] \min_{u_i}$.

Let $V' \subseteq V$. Let \mathcal{A} be the unique partition of V' and let \leq be the unique total order on \mathcal{A} such that, for every $I, J \in \mathcal{A}$ and for every $u \in I$ and $v \in J$, (i) if $I = J$, then $\min_u = \min_v$ and (ii) if $I < J$, then $\min_u < \min_v$. Then, we say that (\mathcal{A}, \leq) , or more simply \mathcal{A} , is the *min-partition* of V' . The *max-partition* of V' is defined analogously. Now, consider the set $V' \times \{\min, \max\}$, and define $\rho((u, \min)) = \min_u$ and $\rho((u, \max)) = \max_u$ for every $u \in V'$. Let \mathcal{B} be the unique partition of $V' \times \{\min, \max\}$ and let \leq be the unique total order on \mathcal{B} such that, for every $I, J \in \mathcal{B}$ and for every $x \in I$ and $y \in J$, (i) if $I = J$, then $\rho(x) = \rho(y)$ and (ii) if $I < J$, then $\rho(x) < \rho(y)$. Then, we say that (\mathcal{B}, \leq) , or more simply \mathcal{B} , is the *min/max-partition* of V' .

The main result of this paper will be proving that the min/max partition of V can be determined in $O(|V|^2)$ time.

2.3 Our Approach

Let $G = (V, E)$ be a graph. We will first show how to build the min-partition of V in $O(n^2)$ time, where $n = |V|$ (Section 4); then, we will show how the algorithm can be adapted so that it builds the min/max-partition in $O(n^2)$ time (Section 5).

In order to build a min-partition of V , we will first classify all minima into three categories (Section 3), so that we can split V into three pairwise-disjoint sets V_1, V_2, V_3 . Then, we will show that in $O(n^2)$ time:

- we can compute V_1, V_2, V_3 (Section 4.1);
- we can define a graph $\bar{G} = (\bar{V}, \bar{E})$ having $|V_3|$ nodes (Section 4.2);
- assuming that we have already determined the min-partition of \bar{V} , we can determine the min-partition of V (Section 4.3).

Analogously, in $O(n^2)$ time we can reduce the problem of determining the min-partition of V to the problem of determining the min-partition of the set of all nodes of a graph having $|V_1|$ (not $|V_3|$) nodes (Section 4.4). As a consequence, since $\min\{|V_1|, |V_3|\} \leq |V|/2 = n/2$, we obtain a recursive algorithm whose running time is given by the recurrence:

$$T(n) = T(n/2) + O(n^2)$$

and we conclude that the running time of our algorithm is $O(n^2)$.

3 Classifying Strings

In [23], Ko and Aluru divide the suffixes of a string into two groups. Here we follow an approach purely based on stringology, without fixing a string or a graph from the start. We divide the strings of Σ^ω into three groups, which we call group 1, group 2 and group 3 (Corollary 3 provides the intuition behind this choice).

► **Definition 1.** Let $\alpha \in \Sigma^\omega$. Let $a \in \Sigma$ and $\alpha' \in \Sigma^\omega$ such that $\alpha = a\alpha'$. Then, we define $\tau(\alpha)$ as follows:

1. $\tau(\alpha) = 1$ if $\alpha' \prec \alpha$.
2. $\tau(\alpha) = 2$ if $\alpha' = \alpha$.
3. $\tau(\alpha) = 3$ if $\alpha \prec \alpha'$.

We will constantly use the following characterization.

► **Lemma 2.** Let $\alpha \in \Sigma^\omega$. Let $a \in \Sigma$ and $\alpha' \in \Sigma^\omega$ such that $\alpha = a\alpha'$. Then:

1. $\tau(\alpha) = 2$ if and only if $\alpha' = a^\omega$, if and only if $\alpha = a^\omega$.
2. $\tau(\alpha) \neq 2$ if and only if $\alpha' \neq a^\omega$, if and only if $\alpha \neq a^\omega$.

Assume that $\tau(\alpha) \neq 2$. Then, there exist unique $c \in \Sigma \setminus \{a\}$, $\alpha'' \in \Sigma^\omega$ and $i \geq 0$ such that $\alpha' = a^i c \alpha''$ (and so $\alpha = a^{i+1} c \alpha''$). Moreover:

1. $\tau(\alpha) = 1$ if and only if $c \prec a$, if and only if $\alpha' \prec a^\omega$, if and only if $\alpha \prec a^\omega$.
2. $\tau(\alpha) = 3$ if and only if $a \prec c$, if and only if $a^\omega \prec \alpha'$, if and only if $a^\omega \prec \alpha$,

The following corollary will be a key ingredient in our recursive approach.

► **Corollary 3.** Let $\alpha, \beta \in \Sigma^\omega$. Let $a, b \in \Sigma$ and $\alpha', \beta' \in \Sigma^\omega$ such that $\alpha = a\alpha'$ and $\beta = b\beta'$. Then:

1. If $a = b$ and $\tau(\alpha) = \tau(\beta) = 2$, then $\alpha = \beta$.
2. If $a = b$ and $\tau(\alpha) < \tau(\beta)$, then $\alpha \prec \beta$. Equivalently, if $a = b$ and $\alpha \preceq \beta$, then $\tau(\alpha) \leq \tau(\beta)$.

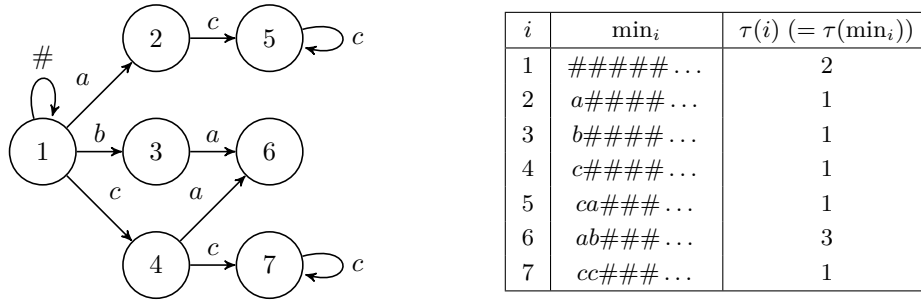


Figure 2 The graph from Figure 1, with the values \min_i 's and $\tau(i)$'s.

4 Computing the min-partition

Let $G = (V, E)$ be a graph. We will prove that we can compute the min-partition of V in $O(|V|^2)$ time. In this section, for every $u \in V$ we define $\tau(u) = \tau(\min_u)$ (see Figure 2).

Let $u \in V$, and let $(u_i)_{i \geq 1}$ be an occurrence of \min_u starting at u . It is immediate to realize that (i) if $\tau(u) = 1$, then $\lambda(u_2) \preceq \lambda(u_1)$, (ii) if $\tau(u) = 2$, then $\lambda(u_k) = \lambda(u_1)$ for every $k \geq 1$ and (iii) if $\tau(u) = 3$, then $\lambda(u_1) \preceq \lambda(u_2)$.

As a first step, let us prove that without loss of generality we can remove some edges from G without affecting the min/max-partition. This preprocessing will be helpful in Lemma 23.

► **Definition 4.** Let $G = (V, E)$ be a graph. We say that G is trimmed if it contains no edge $(u, v) \in E$ such that $\tau(v) = 1$ and $\lambda(v) \prec \lambda(u)$.

In order to simplify the readability of our proofs, we will not directly remove some edges from $G = (V, E)$, but we will first build a copy of G where every node u is a mapped to a node u^* , and then we will trim the graph. In this way, when we write \min_u and \min_{u^*} it will be always clear whether we refer to the original graph or the trimmed graph. We will use the same convention in Section 4.2 when we define the graph $\bar{G} = (\bar{V}, \bar{E})$ that we will use for the recursive step.

► **Lemma 5.** Let $G = (V, E)$ be a graph. Then, in $O(|E|)$ time we can build a trimmed graph $G^* = (V^*, E^*)$, with $V^* = \{u^* \mid u \in V\}$, such that for every $u \in V$ it holds $\min_{u^*} = \min_u$. In particular, $\tau(u^*) = \tau(u)$ for every $u \in V$.

4.1 Classifying Minima

Let us first show how to compute all $u \in V$ such that $\tau(u) = 1$.

► **Lemma 6.** Let $G = (V, E)$ be a graph, and let $u, v \in V$.

1. If $(u, v) \in E$ and $\lambda(u) \prec \lambda(v)$, then $\tau(v) = 1$.
2. If $(u, v) \in E$, $\lambda(u) = \lambda(v)$ and $\tau(u) = 1$, then $\tau(v) = 1$.

► **Corollary 7.** Let $G = (V, E)$ be a graph, and let $u \in V$. Then, $\tau(u) = 1$ if and only if there exist $k \geq 2$ and $z_1, \dots, z_k \in V$ such that (i) $(z_i, z_{i+1}) \in E$ for every $1 \leq i \leq k - 1$, (ii) $z_k = u$, (iii) $\lambda(z_1) \prec \lambda(z_2)$ and (iv) $\lambda(z_2) = \lambda(z_3) = \dots = \lambda(z_k)$.

Corollary 7 yields an algorithm to decide whether $u \in V$ is such that $\tau(u) = 1$.

► **Corollary 8.** Let $G = (V, E)$ be a graph. We can determine all $u \in V$ such that $\tau(u) = 1$ in time $O(|E|)$.

Now, let us show how to determine all $u \in V$ such that $\tau(u) = 2$. We can assume that we have already determined all $u \in V$ such that $\tau(u) = 1$.

► **Lemma 9.** *Let $G = (V, E)$ be a graph, and let $u \in V$ such that $\tau(u) \neq 1$. Then, we have $\tau(u) = 2$ if and only if there exist $k \geq 2$ and $z_1, \dots, z_k \in V$ such that (i) $(z_{i+1}, z_i) \in E$ for every $1 \leq i \leq k-1$, (ii) $z_1 = u$, (iii) $z_k = z_j$ for some $1 \leq j \leq k-1$ and (iv) $\lambda(z_1) = \lambda(z_2) = \dots = \lambda(z_k)$.*

In particular, such $z_1, \dots, z_k \in V$ must satisfy $\tau(z_i) = 2$ for every $1 \leq i \leq k$.

► **Corollary 10.** *Let $G = (V, E)$ be a graph. We can determine all $u \in V$ such that $\tau(u) = 2$ in time $O(|E|)$.*

From Corollary 8 and Corollary 10 we immediately obtain the following result.

► **Corollary 11.** *Let $G = (V, E)$ be a graph. Then, in time $O(|E|)$ we can compute $\tau(u)$ for every $u \in V$.*

4.2 Recursive Step

Let us sketch the general idea to build a smaller graph for the recursive step. We consider each $u \in V$ such that $\tau(u) = 3$, and we follow edges in a backward fashion, aiming to determine a prefix of \min_u . As a consequence, we discard edges through which no occurrence of \min_u can go, and by Corollary 3 we can restrict our attention to the nodes v such that $\tau(v)$ is minimal. We proceed like this until we encounter nodes v' such that $\tau(v') = 3$.

Let us formalize our intuition. We will first present some properties that the occurrences of a string \min_u must satisfy.

► **Lemma 12.** *Let $G = (V, E)$ be a graph. Let $u, v \in V$ be such that $\min_u = \min_v$. Let $(u_i)_{i \geq 1}$ be an occurrence of \min_u and let $(v_i)_{i \geq 1}$ be an occurrence of \min_v . Then:*

1. $\lambda(u_i) = \lambda(v_i)$ for every $i \geq 1$.
2. $\min_{u_i} = \min_{v_i}$ for every $i \geq 1$.
3. $\tau(u_i) = \tau(v_i)$ for every $i \geq 1$.

In particular, the previous results hold if $u = v$ and $(u_i)_{i \geq 1}$ and $(v_i)_{i \geq 1}$ are two distinct occurrences of \min_u .

► **Lemma 13.** *Let $G = (V, E)$ be a graph. Let $u \in V$ and let $(u_i)_{i \geq 1}$ an occurrence of \min_u starting at u . Let $k \geq 1$ be such that $\tau(u_1) = \tau(u_2) = \dots = \tau(u_{k-1}) = \tau(u_k) \neq 2$. Then, u_1, \dots, u_k are pairwise distinct. In particular, $k \leq |V|$.*

The previous results allow us to give the following definition.

► **Definition 14.** *Let $G = (V, E)$ be a graph. Let $u \in V$ such that $\tau(u) = 3$. Let ℓ_u to be the smallest integer $k \geq 2$ such that $\tau(u_k) \geq 2$, where $(u_i)_{i \geq 1}$ is an occurrence of \min_u starting at u .*

Note that ℓ_u is well-defined, because (i) it cannot hold $\tau(u_k) = 1$ for every $k \geq 2$ by Lemma 13 (indeed, if $\tau(u_2) = 1$, then $(u_i)_{i \geq 2}$ is an occurrence of \min_{u_2} starting at u_2 , and by Lemma 13 there exists $2 \leq k \leq |V| + 2$ such that $\tau(u_k) \neq 1$) and (ii) ℓ_u does not depend on the choice of $(u_i)_{i \geq 1}$ by Lemma 12. In particular, it must be $\ell_u \leq |V| + 1$ because $u_1, u_2, \dots, u_{\ell_u-1}$ are pairwise distinct (u_1 is distinct from u_2, \dots, u_{ℓ_u-1} because $\tau(u_1) = 3$ and $\tau(u_2) = \tau(u_3) = \dots = \tau(u_{\ell_u-1}) = 1$ by the minimality of ℓ_u).

► **Lemma 15.** *Let $G = (V, E)$ be a graph. Let $u \in V$ such that $\tau(u) = 3$. Then, $\min_u[i+1] \preceq \min_u[i]$ for every $2 \leq i \leq \ell_u - 1$. In particular, if $2 \leq i \leq j \leq \ell_u$, then $\min_u[j] \preceq \min_u[i]$.*

If $R \subseteq Q$ is a nonempty set of nodes such that for every $u, v \in R$ it holds $\lambda(u) = \lambda(v)$, we define $\lambda(R) = \lambda(u) = \lambda(v)$. If $R \subseteq Q$ is a nonempty set of nodes such that for every $u, v \in R$ it holds $\tau(u) = \tau(v)$, we define $\tau(R) = \tau(u) = \tau(v)$.

Let $R \subseteq Q$ be a nonempty set of states. Let $F(R) = \arg \min_{u \in R'} \tau(u)$, where $R' = \arg \min_{v \in R} \lambda(v)$. Notice that $F(R)$ is nonempty, and both $\lambda(F(R))$ and $\tau(F(R))$ are well-defined. In other words, $F(R)$ is obtained by first considering the subset $R' \subseteq F(R)$ of all nodes v such that $\lambda(v)$ is as small as possible, and then considering the subset of R' of all nodes v such that $\tau(v)$ is as small as possible. This is consistent with our intuition on how we should be looking for a prefix of \min_u .

Define:

$$G_i(u) = \begin{cases} \{u\} & \text{if } i = 1; \\ F(\{v' \in Q \mid (\exists v \in G_{i-1}(u))((v', v) \in E)\}) \setminus \bigcup_{j=2}^{i-1} G_j(u) & \text{if } 1 < i \leq \ell_u. \end{cases}$$

Notice that we also require that a node in $G_i(u)$ has not been encountered before. Intuitively, this does not affect our search for a prefix of \min_u because, if we met the same node twice, then we would have a cycle where all edges are equally labeled (because by Lemma 15 labels can only decrease), and since $\tau(G_i(u)) = 1$ for every $2 \leq i \leq \ell_u - 1$, then no occurrence of the minimum can go through the cycle because if we remove the cycle from the occurrence we obtain a smaller string by Lemma 2.

The following technical lemma is crucial to prove that our intuition is correct.

► **Lemma 16.** *Let $G = (V, E)$ be a graph. Let $u \in V$ such that $\tau(u) = 3$.*

1. $G_i(u)$ is well-defined and nonempty for every $1 \leq i \leq \ell_u$.
2. Let $(u_i)_{i \geq 1}$ be an occurrence of \min_u starting at u . Then, $u_i \in G_i(u)$ for every $1 \leq i \leq \ell_u$. In particular, $\tau(u_i) = \tau(G_i(u))$ and $\min_u[i] = \lambda(u_i) = \lambda(G_i(u))$ for every $1 \leq i \leq \ell_u$.
3. For every $1 \leq i \leq \ell_u$ and for every $v \in G_i(u)$ there exists an occurrence of $\min_u[1, i - 1]$ starting at u and ending at v .

Let $u \in V$ such that $\tau(u) = 3$. We define:

- $\gamma_u = \min_u[1, \ell_u]$;
- $\mathbf{t}_u = \tau(G_{\ell_u}(u)) \in \{2, 3\}$

Now, in order to define the smaller graph for the recursive step, we also need a new alphabet (Σ', \preceq') , which must be defined consistently with the mutual ordering of the minima. The next lemma yields all the information that we need.

► **Lemma 17.** *Let $G = (V, E)$ be a graph. Let $u, v \in V$ such that $\tau(u) = \tau(v) = 3$. Assume that one of the following statements is true:*

1. γ_u is not a prefix of γ_v and $\gamma_u \prec \gamma_v$.
2. $\gamma_u = \gamma_v$, $\mathbf{t}_u = 2$ and $\mathbf{t}_v = 3$.
3. γ_v is a strict prefix of γ_u .

Then, $\min_u \prec \min_v$.

Equivalently, if $\min_u \preceq \min_v$, then one the following is true: (i) γ_u is not a prefix of γ_v and $\gamma_u \prec \gamma_v$; (ii) $\gamma_u = \gamma_v$ and $\mathbf{t}_u \leq \mathbf{t}_v$; (iii) γ_v is a strict prefix of γ_u .

Now, let $\Sigma' = \{(\gamma_u, \mathbf{t}_u) \mid u \in V, \tau(u) = 3\}$, and let \preceq' be the total order on Σ' such that for every distinct $(\alpha, x), (\beta, y) \in \Sigma'$, it holds $(\alpha, x) \prec' (\beta, y)$ if and only if one of the following is true:

1. α is not a prefix of β and $\alpha \prec \beta$.
2. $\alpha = \beta$, $x = 2$ and $y = 3$.
3. β is a strict prefix of α .

22:10 Prefix Sorting DFAs: A Recursive Algorithm

It is immediate to verify that \preceq' is a total order: indeed, \preceq' is obtained (i) by first comparing the γ_u 's using the variant of the (total) lexicographic order on Σ^* in which a string is smaller than every strict prefix of it and (ii) if the γ_u 's are equal by comparing the \mathfrak{t}_u 's, which are elements in $\{2, 3\}$.

Starting from $G = (V, E)$, we define a new graph $\bar{G} = (\bar{V}, \bar{E})$ as follows:

- $\bar{V} = \{\bar{u} \mid u \in V, \tau(u) = 3\}$.
- The new totally-ordered alphabet is (Σ', \preceq') .
- For every $\bar{u} \in \bar{V}$, we define $\lambda(\bar{u}) = (\gamma_u, \mathfrak{t}_u)$.
- $\bar{E} = \{(\bar{v}, \bar{v}') \mid \mathfrak{t}_v = 2\} \cup \{(\bar{u}, \bar{v}) \mid \mathfrak{t}_v = 3, u \in G_{\ell_v}(v)\}$.

Note that for every $\bar{v} \in \bar{V}$ such that $\mathfrak{t}_v = 3$ and for every $u \in G_{\ell_v}(v)$ it holds $\tau(u) = \tau(G_{\ell_v}(v)) = \mathfrak{t}_v = 3$, so $\bar{u} \in \bar{V}$ and $(\bar{u}, \bar{v}) \in \bar{E}$. Moreover, $\bar{G} = (\bar{V}, \bar{E})$ satisfies all the assumptions about graphs that we use in this paper: (i) all edges entering the same node have the same label (by definition), (ii) every node has at least one incoming edge (because if $\bar{v} \in \bar{V}$, then $G_{\ell_v}(v) \neq \emptyset$ by Lemma 16) and (iii) \bar{G} is deterministic (because if $(\bar{u}, \bar{v}), (\bar{u}, \bar{v}') \in \bar{E}$ and $\lambda(\bar{v}) = \lambda(\bar{v}')$, then $\gamma_v = \gamma_{v'}$ and $\mathfrak{t}_v = \mathfrak{t}_{v'}$, so by the definition of \bar{E} if $\mathfrak{t}_v = \mathfrak{t}_{v'} = 2$ we immediately obtain $\bar{v} = \bar{u} = \bar{v}'$, and if $\mathfrak{t}_v = \mathfrak{t}_{v'} = 3$ we obtain $u \in G_{\ell_v}(v) \cap G_{\ell_{v'}}(v')$; since by Lemma 16 there exist two occurrences of $\min_{v'}[1, \ell_{v'} - 1] = \gamma_{v'}[1, \ell_{v'} - 1] = \gamma_v[1, \ell_v - 1] = \min_v[1, \ell_v - 1]$ starting at v and v' and both ending at u , the determinism of G implies $v = v'$ and so $\bar{v} = \bar{v}'$).

Notice that if $\bar{v} \in \bar{V}$ is such that $\mathfrak{t}_v = 2$, then $I_{\bar{v}}$ contains exactly one string, namely, $\lambda(\bar{v})^\omega$; in particular, $\min_{\bar{v}} = \max_{\bar{v}} = \lambda(\bar{v})^\omega$.

When we implement $G = (V, E)$ and $\bar{G} = (\bar{V}, \bar{E})$, we use integer alphabets $\Sigma = \{0, 1, \dots, |\Sigma| - 1\}$ and $\Sigma' = \{0, 1, \dots, |\Sigma'| - 1\}$; in particular, we will not store Σ' by means of pairs $(\gamma_u, \mathfrak{t}_u)$'s, but we will remap Σ' to an integer alphabet consistently with the total order \preceq' on Σ' , so that the mutual order of the $\min_{\bar{u}}$'s is not affected.

Let us prove that we can use $\bar{G} = (\bar{V}, \bar{E})$ for the recursive step. We will start with some preliminary results.

► **Lemma 18.** *Let $G = (V, E)$ be a graph. Let $u, v \in V$ be such that $\tau(u) = \tau(v) = 3$, $\gamma_u = \gamma_v$ and $\mathfrak{t}_u = \mathfrak{t}_v = 2$. Then, $\min_u = \min_v$.*

► **Lemma 19.** *Let $G = (V, E)$ be a graph. Let $u \in V$, and let $(u_i)_{i \geq 1}$ be an occurrence of \min_u starting at u . Then, exactly one of the following holds true:*

1. *There exists $i_0 \geq 1$ such that $\tau(u_i) \neq 2$ for every $1 \leq i < i_0$ and $\tau(u_i) = 2$ for every $i \geq i_0$.*
2. *$\tau(u_i) \neq 2$ for every $i \geq 1$, and both $\tau(u_i) = 1$ and $\tau(u_i) = 3$ are true for infinitely many i 's.*

Crucially, the next lemma establishes a correspondence between minima of nodes in $G = (V, E)$ and minima of nodes in $\bar{G} = (\bar{V}, \bar{E})$.

► **Lemma 20.** *Let $G = (V, E)$ be a graph. Let $u \in V$ such that $\tau(u) = 3$. Let $(u_i)_{i \geq 1}$ be an occurrence of \min_u starting at u . Let $(u'_i)_{i \geq 1}$ be the infinite sequence of nodes in V obtained as follows. Consider $L = \{k \geq 1 \mid \tau(u_k) = 3\}$, and for every $i \geq 1$, let $j_i \geq 1$ be the i^{th} smallest element of L , if it exists. For every $i \geq 1$ such that j_i is defined, let $u'_i = u_{j_i}$, and if $i \geq 1$ is such that j_i is not defined (so L is a finite set), let $u'_i = u'_{|L|}$. Then, $(\bar{u}'_i)_{i \geq 1}$ is an occurrence of $\min_{\bar{u}}$ starting at \bar{u} in $\bar{G} = (\bar{V}, \bar{E})$.*

The following theorem shows that our reduction to $\bar{G} = (\bar{V}, \bar{E})$ is correct.

► **Theorem 21.** *Let $G = (V, E)$ be a graph. Let $u, v \in V$ be such that $\tau(u) = \tau(v) = 3$.*

1. *If $\min_u = \min_v$, then $\min_{\bar{u}} = \min_{\bar{v}}$.*
2. *If $\min_u \prec \min_v$, then $\min_{\bar{u}} \prec' \min_{\bar{v}}$.*

Since \preceq is a total order (so exactly one among $\min_u \prec \min_v$, $\min_u = \min_v$ and $\min_v \prec \min_u$ holds true), from Theorem 21 we immediately obtain the following result.

► **Corollary 22.** *Let $G = (V, E)$ be a graph. Let $u, v \in V$ be such that $\tau(u) = \tau(v) = 3$.*

1. *It holds $\min_u = \min_v$ if and only if $\min_{\bar{u}} = \min_{\bar{v}}$.*
2. *It holds $\min_u \prec \min_v$ if and only if $\min_{\bar{u}} \prec' \min_{\bar{v}}$.*

In particular, if we have the min-partition of \bar{V} (with respect to \bar{G}), then we also have the min-partition of $\{u \in V \mid \tau(u) = 3\}$ (with respect to G).

Lastly, we show that our reduction to $\bar{G} = (\bar{V}, \bar{E})$ can be computed within $O(n^2)$ time.

► **Lemma 23.** *Let $G = (V, E)$ be a trimmed graph. Then, we can build $\bar{G} = (\bar{V}, \bar{E})$ in $O(|V|^2)$ time.*

4.3 Merging

We want to determine the min-partition \mathcal{A} of V , assuming that we already have the min-partition \mathcal{B} of $\{u \in V \mid \tau(u) = 3\}$.

First, note that we can easily build the min-partition \mathcal{B}' of $\{u \in V \mid \tau(u) = 2\}$. Indeed, if $\tau(u) = 2$, then $\min_u = \lambda(u)^\omega$ by Lemma 2. As a consequence, if $\tau(u) = \tau(v) = 2$, then (i) $\min_u = \min_v$ if and only if $\lambda(u) = \lambda(v)$ and (ii) $\min_u \prec \min_v$ if and only if $\lambda(u) \prec \lambda(v)$, so we can build \mathcal{B}' in $O(|V|)$ time by using counting sort.

For every $c \in \Sigma$ and $t \in \{1, 2, 3\}$, let $V_{c,t} = \{v \in V \mid \lambda(v) = c, \tau(v) = t\}$. Consider $u, v \in V$: (i) if $\lambda(u) \prec \lambda(v)$, then $\min_u \prec \min_v$ and (ii) if $\lambda(u) = \lambda(v)$ and $\tau(u) < \tau(v)$, then $\min_u \prec \min_v$ by Corollary 3. As a consequence, in order to build \mathcal{A} , we only have to build the min-partition $\mathcal{A}_{c,t}$ of $V_{c,t}$, for every $c \in \Sigma$ and every $t \in \{1, 2, 3\}$.

A possible way to implement each $\mathcal{A}_{c,t}$ is by means of an array $A_{c,t}$ storing the elements of $V_{c,t}$, where we also use a special character to delimit the border between consecutive elements of $A_{c,t}$.

It is immediate to build incrementally $\mathcal{A}_{c,3}$ for every $c \in \Sigma$, from its smallest element to its largest element. At the beginning, $A_{c,3}$ is empty for every $c \in \Sigma$. Then, scan the elements I in \mathcal{B} from smallest to largest, and add I to $\mathcal{A}_{c,3}$, where $c = \lambda(u)$ for any $u \in I$ (the definition of c does not depend on the choice of u). We scan \mathcal{B} only once, so this step takes $O(|V|)$ time. Analogously, we can build $\mathcal{A}_{c,2}$ for every $c \in \Sigma$ by using \mathcal{B}' .

We are only left with showing how to build $\mathcal{A}_{c,1}$ for every $c \in \Sigma$. At the beginning, each $A_{c,1}$ is empty, and we will build each $\mathcal{A}_{c,1}$ from its smallest element to its largest element. During this step of the algorithm, we will gradually mark the nodes $u \in V$ such that $\tau(u) = 1$. At the beginning of the step, no such node is marked, and at the end of the step all these nodes will be marked. Let $\Sigma = \{c_1, c_2, \dots, c_\sigma\}$, with $c_1 \prec c_2 \prec \dots \prec c_\sigma$. Notice that it must be $V_{c_1,1} = \emptyset$, because if there existed $u \in V_{c_1,1}$, then it would be $\min_u \prec c_1^\omega$ by Lemma 2 and so c_1 would not be the smallest character in Σ . Now, consider $V_{c_1,2}$; we have already fully computed $\mathcal{A}_{c_1,2}$. Process each I in $\mathcal{A}_{c_1,2}$ from smallest to largest, and for every $c_k \in \Sigma$ compute the set J_k of all non-marked nodes $v \in V$ such that $\tau(v) = 1$, $\lambda(v) = c_k$, and $(u, v) \in E$ for some $u \in I$. Then, if $J_k \neq \emptyset$ add J_k to $\mathcal{A}_{c_k,1}$ and mark the nodes in J_k . After processing the elements in $\mathcal{A}_{c_1,2}$, we process the element in $\mathcal{A}_{c_1,3}$, $\mathcal{A}_{c_2,1}$, $\mathcal{A}_{c_2,2}$, $\mathcal{A}_{c_2,3}$, $\mathcal{A}_{c_3,1}$ and so on, in this order. Each $\mathcal{A}_{c_i,t}$ is processed from its (current) smallest element to its (current) largest element. We never remove or modify elements in any $\mathcal{A}_{c,t}$, but we only

22:12 Prefix Sorting DFAs: A Recursive Algorithm

add elements to the $A_{c,1}$'s. More precisely, when we process I in $A_{c,t}$, for every $c_k \in \Sigma$ we compute the set J_k of all non-marked nodes $v \in V$ such that $\tau(v) = 1$, $\lambda(v) = c_k$, and $(u, v) \in E$ for some $u \in I$ and, if $J_k \neq \emptyset$, then we add J_k to $A_{c_k,1}$ and we mark the nodes in J_k .

The following lemma shows that our approach is correct. Let us give some intuition. A *prefix* of a min-partition \mathcal{C} is a subset \mathcal{C}' of \mathcal{C} such that, if $I, J \in \mathcal{C}$, $I < J$ and $J \in \mathcal{C}'$, then $I \in \mathcal{C}'$. Notice that every prefix of \mathcal{A} is obtained by taking the union of $\mathcal{A}_{c_1,2}, \mathcal{A}_{c_1,3}, \mathcal{A}_{c_2,1}, \mathcal{A}_{c_2,2}, \mathcal{A}_{c_2,3}, \mathcal{A}_{c_3,1}, \dots$ in this order up to some element $\mathcal{A}_{c,t}$, where possibly we only pick a prefix of the last element $\mathcal{A}_{c,t}$. Then, we will show that, when we process I in $A_{c,t}$, we have already built the prefix of \mathcal{A} whose largest element is I . This means that, for every $v \in J_k$ and for any *any* occurrence $(v_i)_{i \geq 1}$ of \min_v starting at v , it must hold that v_2 is in I .

► **Lemma 24.** *Let $G = (V, E)$ be a graph. If we know the min-partition of $\{u \in V \mid \tau(u) = 3\}$, then we can build the min-partition of V in $O(|E|)$ time.*

4.4 The Complementary Case

We have shown that in $O(n^2)$ time we can reduce the problem of determining the min-partition of V to the problem of determining the min-partition of the set of all nodes of a graph having $|\{u \in V \mid \tau(u) = 3\}|$ nodes. Now, we must show that (similarly) in $O(n^2)$ time we can reduce the problem of determining the min-partition of V to the problem of determining the min-partition of the set of all nodes of a graph having $|\{u \in V \mid \tau(u) = 1\}|$ nodes. The merging step will be more complex, because the order in which we will process the $\mathcal{A}_{c,t}$ will be from largest to smallest ($\mathcal{A}_{c_\sigma,2}, \mathcal{A}_{c_\sigma,1}, \mathcal{A}_{c_{\sigma-1},3}, \mathcal{A}_{c_{\sigma-1},2}, \mathcal{A}_{c_{\sigma-1},1}, \mathcal{A}_{c_{\sigma-2},3}$ and so on) so we will need to update some elements of some $A_{c,t}$'s to include the information about minima that we may infer at a later stage of the algorithm. We provide the details in the full version of the paper [7].

5 Computing the min/max-partition

Let $G = (V, E)$ be a graph. We can build the *max-partition* of V by simply considering the transpose total order \preceq^* of \preceq (the one for which $a \preceq^* b$ if and only if $b \preceq a$) and building the min-partition. As a consequence, the algorithm to build the max-partition is entirely symmetrical to the algorithm to build the min-partition.

Let $G = (V, E)$ be a graph. Let us show how we can build the min/max-partition of V in $O(|V|^2)$ time. Assume that we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ on the same alphabet (Σ, \preceq) , with $V_1 \cap V_2 = \emptyset$ (we allow G_1 and G_2 to possibly be the *null graph*, that is, the graph without vertices). Let $V'_1 \subseteq V_1$, $V'_2 \subseteq V_2$, $W = V'_1 \cup V'_2$, and for every $u \in W$ define $\rho(u) = \min_u$ if $u \in V'_1$, and $\rho(u) = \max_u$ if $u \in V'_2$. Let \mathcal{A} be the unique partition of W and let \leq be the unique total order on \mathcal{A} such that, for every $I, J \in \mathcal{A}$ and for every $u \in I$ and $u \in J$, (i) if $I = J$, then $\rho(u) = \rho(u)$ and (ii) if $I < J$, then $\rho(u) \prec \rho(u)$. Then, we say that (\mathcal{A}, \leq) , or more simply \mathcal{A} , is the *min/max-partition* of (V'_1, V'_2) . We will show that we can compute the min/max partition of (V_1, V_2) in $O((|V_1| + |V_2|)^2)$ time. In particular, if $G_1 = (V_1, E_1)$ and $G_2 = (V_1, E_2)$ are two (distinct) copies of the same graph $G = (V, E)$, then we can compute the min/max-partition of V in $O(|V|^2)$ time.

We compute $\tau(\min_u)$ for every $u \in V_1$ and we compute $\tau(\max_u)$ for every $u \in V_2$. If the number of values equal to 3 is smaller than the number of values equal to 1, then (in time $O(|V_1|^2 + |V_2|^2) = O((|V_1| + |V_2|)^2)$) we build the graphs $\bar{G}_1 = (\bar{V}_1, \bar{E}_1)$ and $\bar{G}_2 = (\bar{V}_2, \bar{E}_2)$ as defined before, where $\bar{V}_1 = \{\bar{u} \mid u \in V_1, \tau(\min_u) = 3\}$ and $\bar{V}_2 = \{\bar{u} \mid u \in V_2, \tau(\max_u) = 3\}$, otherwise we consider the complementary case (which is symmetrical). When building $\bar{G}_1 = (\bar{V}_1, \bar{E}_1)$ and $\bar{G}_2 = (\bar{V}_2, \bar{E}_2)$, we define a *unique* alphabet (Σ', \preceq') obtained by jointly sorting the $(\gamma_{\min_u}, \mathbf{t}_{\min_u})$'s and the $(\gamma_{\max_u}, \mathbf{t}_{\max_u})$'s, which is possible because Lemma 17 also applies to maxima. Note that $|\bar{V}_1| + |\bar{V}_2| \leq (|V_1| + |V_2|)/2$.

Assume that we have recursively obtained the min/max-partition of (\bar{V}_1, \bar{V}_2) with respect to \bar{G}_1 and \bar{G}_2 . This yields the min/max-partition of $(\{u \in V_1 \mid \tau(\min_u) = 3\}, \{u \in V_2 \mid \tau(\max_u) = 3\})$. Then, we can build the min/max-partition of (V_1, V_2) by jointly applying the merging step, which is possible because both the merging step for minima and the merging step for maxima require to build the $\mathcal{A}_{c,1}$'s by processing $A_{c_1,2}, A_{c_1,3}, A_{c_2,1}, A_{c_2,2}, A_{c_2,3}, A_{c_3,1}$ and so on in this order.

Since we obtain the same recursion as before, we conclude that we can compute the min/max partition of (V_1, V_2) in $O((|V_1| + |V_2|)^2)$ time.

References

- 1 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In Shuchi Chawla, editor, *Proc. of the 31st Symposium on Discrete Algorithms, (SODA'20)*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.
- 2 Ruben Becker, Manuel Cáceres, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Francisco Olivares, and Nicola Prezza. Sorting Finite Automata via Partition Refinement. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.15.
- 3 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, pages 225–235, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 4 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, 1994.
- 5 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on Wheeler DFAs. In *2023 Data Compression Conference (DCC)*, pages 150–159, 2023. doi:10.1109/DCC55655.2023.00023.
- 6 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In *2022 Data Compression Conference (DCC)*, pages 272–281, 2022. doi:10.1109/DCC52660.2022.00035.
- 7 Nicola Cotumaccio. Prefix sorting dfas: a recursive algorithm, 2023. arXiv:2305.02526.
- 8 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part i. *J. ACM*, 70(4), August 2023. doi:10.1145/3607471.
- 9 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In Daniel Marx, editor, *Proc. of the 32nd Symposium on Discrete Algorithms, (SODA'21)*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.
- 10 Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In Tomáš Bureš, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdziński, Claus Pahl, Florian Sikora, and Prudence W.H. Wong, editors, *SOFSEM 2021: Theory and Practice of Computer Science*, pages 608–622, Cham, 2021. Springer International Publishing.

- 11 Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Trans. Algorithms*, 19(3), April 2023. doi:10.1145/3588334.
- 12 M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.
- 13 P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 184–193, 2005. doi:10.1109/SFCS.2005.69.
- 14 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.
- 15 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), November 2009. doi:10.1145/1613676.1613680.
- 16 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. doi:10.1145/1082036.1082039.
- 17 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo). doi:10.1016/j.tcs.2017.06.016.
- 18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CB09780511574931.
- 19 Ramana M. Idury and Michael S. Waterman. A new algorithm for DNA sequence assembly. *Journal of computational biology: A journal of computational molecular cell biology*, 2 2:291–306, 1995.
- 20 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, November 2006. doi:10.1145/1217856.1217858.
- 21 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2):126–142, 2005. Combinatorial Pattern Matching (CPM) Special Issue. doi:10.1016/j.jda.2004.08.019.
- 22 Sung-Hwan Kim, Francisco Olivares, and Nicola Prezza. Faster prefix-sorting algorithms for deterministic finite automata. 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 16:1–16:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.CPM.2023.16.
- 23 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005. Combinatorial Pattern Matching (CPM) Special Issue. doi:10.1016/j.jda.2004.08.002.
- 24 Veli Mäkinen, Niko Välimäki, and Jouni Sirén. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11:375–388, 2014.
- 25 U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 26 Joong Chae Na. Linear-time construction of compressed suffix arrays using $o(n \log n)$ -bit working space for large alphabets. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Combinatorial Pattern Matching*, pages 57–67, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 27 Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. doi:10.1137/0216062.
- 28 Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4–es, July 2007. doi:10.1145/1242471.1242472.

- 29 Kunihiro Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.
- 30 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, June 2010. doi:10.1093/bioinformatics/btq217.
- 31 P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.