Faster Cache-Efficient Pattern Matching for Deterministic Wheeler Pangenome Graphs

Riccardo Maso*,†, Nicola Prezza† and Carlo Tosoni†

DAIS, Ca' Foscari University of Venice, Via Torino 155, Venice, Italy

Abstract

Pattern matching on strings is regarded as one of the core operations in computer science. Although many solutions have been proposed several solutions to this problem, some of the most elegant and widely used approaches are based on the renowned Burrows-Wheeler transform (BWT). The success of the BWT lies in its pattern matching algorithm known as backward search, which is not only near-optimal in the RAM model, but also runs directly on a compressed representation of the input string. More recently, this backward search has been generalized to Wheeler deterministic finite automata (DFAs), a subclass of standard DFAs, without losing its near-optimal time efficiency. Similarly to the case of strings, this pattern matching algorithm for Wheeler DFAs has found applications in bioinformatics, where researchers have shown that specific pangenome graphs of human chromosomes can be transformed into Wheeler DFAs and consequently indexed using this strategy. However, this BWT-based index on Wheeler DFAs inherited a significant drawback from the original backward search, namely the high number of I/O operations triggered during the algorithm execution, which are in the worst-case lower-bounded by the length of the pattern. In this paper, we address this limitation by proposing the first cache-friendly algorithm specifically designed for Wheeler DFAs. Our new data structure reduces the number of I/O operations by employing a strategy analogous to the suffix array: it interleaves a binary search with fast sequential scans of the automaton. We empirically validate this new indexing strategy by running our algorithm on real-world Wheeler pangenome graphs. We show that while our data structure can use up to 15 times the space required by the backward search index, it can also be 500 times faster and able to process a single character of the pattern in less than 3 ns.

Keywords

Burrows-Wheeler transform, Pattern matching on graphs, Human chromosomes, I/O model

1. Introduction

Pattern matching, the process of identifying a pattern P within a text T, is a fundamental operation in computer science. This task finds applications in several domains, with bioinformatics being a notable example, where locating the occurrences of a pattern can be used to perform sequence alignment or map short sequencing reads to a large reference genome [1, 2]. Due to these reasons, this problem has been deeply studied in the literature where various solutions have been proposed. Some of these solutions pre-process the pattern P to achieve search times proportional to the length of the text T [3], others reverse this paradigm by preprocessing the text T and their query times depend on the length of the pattern P [4, 5, 6]. This latter group includes pattern matching indexes based on the Burrows-Wheeler transform (BWT) [7], including the renowned FM-index [8]. The importance of the FM-index lies in its dual capability to provide efficient query time, while compressing the text into its empirical entropy. This *pattern matching* problem naturally generalizes to a (potentially infinite) collection of strings, where the objective is to identify whether a pattern P occurs as a substring within this collection. A common approach to represent this collection is to employ a finite-state automaton, accepting the collection of strings as its regular language. However, contrary to the case of strings, pattern matching on general automata cannot be solved in strongly subquadratic time unless the Orthogonal Vectors Hypothesis [9] is false, as proven by Equi et al. [10, 11]. In order to overcome this complexity constraint,

ICTCS 2026: Italian Conference on Theoretical Computer Science, September 7–9, 2026, Udine, Italy

*Corresponding author.

†These authors contributed equally.

riccardo.maso@unive.it (R. Maso); nicola.prezza@unive.it (N. Prezza); carlo.tosoni@unive.it (C. Tosoni)

0009-0004-8773-1523 (R. Maso); 0000-0003-3553-4953 (N. Prezza); 0009-0009-5149-2416 (C. Tosoni)

research has focused on identifying subclasses of automata that allow for efficient pattern matching. This led to the definition of *Wheeler NFAs* [12], which represents a natural generalization of the original BWT to automata. Informally, an NFA is said to be Wheeler if there exists a total order of its states *consistent* with the strings entering into them. This total ordering represents a natural generalization of the permutation returned by the original string-BWT, which sorts the string characters based on their subsequent suffixes. This line of research has found applications in bioinformatics, where genomic collections may be represented by using a pangenome graph, encoding the genetic diversity of a population [13]. Indeed, Sirén et al. [14] showed that some pangenome graphs built from a collection of human chromosomes can be transformed into Wheeler NFAs for supporting queries across the genetic variation of a population. The current state-of-the-art algorithm for indexing Wheeler NFAs relies on the *forward search*, the analogue of the well-known backward search used by the string-BWT, but processing the pattern in a left-to-right fashion. While in the RAM model this forward search runs in near-optimal $\widetilde{O}(|P|)$ time, in the I/O model of Aggarwal and Vitter [15] it inherited a notable bottleneck from the backward search algorithm, namely the high number of cache misses, which in the worst-case are lower-bounded by the length of the pattern.

In the string domain, the suffix array [6] and tree [4, 5] support cache-friendly solutions for this problem. Indeed, the occurrences of an input pattern can be located by means of a binary search on the suffix array, combined with fast sequential scans of the matching regions in the text. However, these solutions have not yet been generalized to Wheeler NFAs. This represents a significant drawback for the usage of Wheeler NFAs in real-world applications, since in the era of the so-called big data applications time efficiency has become as paramount as space complexity for indexing large-scale datasets.

1.1. Our contributions

In this paper, we address this limitation by proposing the first cache-friendly algorithm specifically designed for *deterministic* Wheeler automata. We term our new indexing strategy the *Graph Suffix Array* as it represents a natural generalization of the aforementioned pattern matching algorithm based on the suffix array of a string. Indeed, our algorithm locates all the occurrences of a pattern by alternating a binary search with fast linear scans of the input Wheeler DFA, thus achieving a drop in the overall number of cache misses triggered during the algorithm execution. Specifically, our algorithm works in three distinct phases. During the first phase, we identify the longest prefix of the input pattern P occurring at least twice in the Wheeler automaton. We show that this operation can be performed efficiently in the I/O model by running a binary search on the list of *infimum* and *supremum* strings [16, 17], which are respectively the smallest and largest strings entering into each state of the Wheeler DFA. These infimum and supremum strings can be encoded using a pseudoforest, i.e., a forest where every tree node has at most one incoming edge, which ensures that every tree contains at most one cycle. As a consequence of this, it follows that we can employ a specialized version of the *heavy-light decomposition* to achieve a cache-efficient traversal of these pseudotrees. Due to these observations, we show that the following result holds.

Lemma 1.1. *Let $\mathcal{D}$ be a Wheeler DFA of n states, and let P be an input pattern of length m. Then we can compute the longest prefix β of P reaching at least two distinct states in $\mathcal{D}$ using $\mathcal{O}((\frac{m}{B}+\log n)\log n \log m)$ operations in the I/O model, where B is the block size.*

Therefore, if this prefix β is equal to P, then we return the states reached by β and we are done. Otherwise, we know that the prefix βa of P can reach at most a single state of the DFA. Thus, in the second phase of the algorithm execution we use standard algorithms and data structures on Wheeler automata [12] to check in $\mathcal{O}(\log\log\sigma)$ I/O operations if there exists a transition labeled a outgoing from these states reached by β. If this transition exists, then we conclude that the string βa reaches a state u of the DFA. Therefore, in this last phase of the algorithm we start navigating the DFA from u, by following the remaining string suffix γ, where $P = \beta a \gamma$. In order to speed up this third phase, we partition the Wheeler DFA in *unary paths*, that is, maximal sequences of connected states having in-degree equal to 1. By doing so, we bound the number of cache misses in this last phase by $\mathcal{O}(\frac{m}{B}+d)$,

where the parameter d indicates the number of unary paths traversed while following γ in the DFA. Summing up the time complexities of these three phases, we obtain the following result.

Theorem 1.2. *Let $\mathcal{D}$ be a Wheeler DFA of n states, and let P be an input pattern of length m. Then there exists a data structure taking $\mathcal{O}(|\mathcal{D}|)$ RAM words and able to locate the occurrences of P in $\mathcal{D}$ in at most $\mathcal{O}((\frac{m}{B} + \log n) \log n \log m + d)$ operations in the I/O model, with B being the block size.*

In this theorem, the value $|\mathcal{D}|$ denotes the number of states and transitions of $\mathcal{D}$. Specifically, the space complexity of $\mathcal{O}(|\mathcal{D}|)$ stems from the third phase of the algorithm, which requires a plain representation of the input automaton. As the parameter d is crucial for the time efficiency of the Graph Suffix Array, we carry out experiments on real-world Wheeler deterministic pangenomes, built over human chromosomes, to experimentally assess the typical values of d in these graphs. We empirically show that, due to the high sparseness of these pangenome graphs [18], this parameter d is about two orders of magnitude smaller than the pattern length. We also run experiments showing that our index is up to 500 times faster than the original pattern matching index for Wheeler automata based on the forward search algorithm. As a limitation of our approach, the space usage of $\mathcal{O}(|\mathcal{D}|)$ RAM words is asymptotically higher than the $O(|\mathcal{D}| \log \sigma)$ bits required by the forward search [12, Lemma 4], where σ is the alphabet size. This spatial analysis is also empirically validated, as we show that our data structure can use up to 15 times the space required by the forward search. However, in our experiments the space usage of our data structures never exceeds 1.5 GB, which implies that our solution can be easily stored in the RAM.

Finally, as a direct consequence of Lemma 1.1, we show that if we aim to locate only those patterns occurring at least twice in the automaton, then we can achieve a tighter time and space complexity.

Corollary 1.3. *Let $\mathcal{D}$ be a Wheeler DFA of n states, and let P be an input pattern of length m. Then there exists a data structure taking $\mathcal{O}(n)$ RAM words which returns the set T of states reached by P, if $|T| \geq 2$, and the empty set $\emptyset$ otherwise. The overall number of operations in the I/O model is bounded by $\mathcal{O}((\frac{m}{B} + \log n) \log n)$, where B is the block size.*

The rest of the paper is organized as follows. In Section 2, we introduce the notation and the preliminary concepts used in the paper. In Section 3, we prove some properties on Wheeler DFAs and we introduce our Graph Suffix Array. In Section 4, we prove the time and space complexity of our novel pattern matching algorithms. Finally, in Section 5 we conduct experiments on real pangenome graphs using our new indexing strategy.

2. Notation and preliminaries

Relations and strings. A binary relation R over a set U is a subset of the set $U \times U$. In this paper, we work with *total orders*, which are relations satisfying (i) reflexivity, (ii) antisymmetry, (iii) transitivity, and (iv) totality. We use the symbols $\leq$ and $\preceq$ to denote total orders. Given an alphabet Σ, we denote by Σ^* the set of all finite strings over Σ, where $\varepsilon \in \Sigma^*$ denotes the empty string. In addition, we define Σ^ω as the set of all strings formed by an infinite countable concatenation of characters from Σ. In particular, we work with *left-infinite* strings, meaning that $\alpha \in \Sigma^\omega$ is constructed from ε by prepending an infinite number of characters to it. For a string $\alpha \in \Sigma^*$, we write $\alpha[i, j]$ to denote the substring consisting of characters from position i through j, where $1 \leq i \leq j \leq |\alpha|$. For two strings $\alpha \in \Sigma^*$ and $\beta \in \Sigma^* \cup \Sigma^\omega$, we write $\alpha \dashv \beta$ to denote that α is a *suffix* of β. Moreover, we assume to have a fixed total order $\preceq$ over the alphabet Σ, and we extend this order $\preceq$ to $\Sigma^* \cup \Sigma^\omega$ *co-lexicographically* as follows. For every two strings $\alpha, \beta \in \Sigma^* \cup \Sigma^\omega$ we have that (i) $\varepsilon \preceq \alpha$ for every $\alpha \in \Sigma^* \cup \Sigma^\omega$, and (ii) if $\alpha = \alpha' a$ and $\beta = \beta' b$ for some $\alpha', \beta' \in \Sigma^* \cup \Sigma^\omega$ and $a, b \in \Sigma$, then $\alpha \prec \beta$ if and only if $(a \prec b) \vee (a = b \wedge \alpha' \prec \beta')$.

DFA. A deterministic finite automaton (DFA) is a 4-tuple $\mathcal{D} = (Q, \Sigma, \delta, s)$ defined as follows. Q is the set of states, Σ is a finite, nonempty set called the alphabet. δ is a function $\delta : Q \times \Sigma \to Q \cup \{\perp\}$ mapping each pair (u, a), with $u \in Q$ and $a \in \Sigma$, to either a state in Q or to a special symbol $\perp$,

indicating that no transition labeled by a leaves the state u. Finally, $s \in Q$ is the unique initial state of the automaton. Typically, a DFA is defined by a 5-tuple, including the set of final states $F \subseteq Q$, which we omit since we assume all states are final. Given a DFA $\mathcal{D} = (Q, \Sigma, \delta, s)$, a state $u \in Q$, and a character $a \in \Sigma$, we may use the shortcut $\delta_a(u)$ for $\delta(u, a)$. Moreover, we extend the transition function to words $\alpha \in \Sigma^*$ as follows: let $a \in \Sigma, \alpha \in \Sigma^*$, and $u \in Q$, then $\delta(u, \alpha a) = \delta(\delta(u, \alpha), a)$ and $\delta(u, \varepsilon) = u$. We make the following assumptions on the DFAs we treat: (i) We assume the alphabet Σ to be *effective*; meaning that every character labels at least one transition in the automaton. (ii) We assume that every state is reachable from the initial state. (iii) The initial state s has a single incoming transition $s = \delta(s, \#)$, where $\# \in \Sigma$ is a special symbol of the alphabet not labeling any other transition, and satisfying $\# \preceq a$ for all $a \in \Sigma$. We note that these assumptions are not restrictive, as any DFA can be transformed to satisfy them without changing its regular language. In the rest of the paper, we denote by n the number of states in the input DFA, formally $n = |Q|$.

Infimum and supremum strings. Let $\mathcal{D}$ be a DFA, and let $u \in Q$. We define I_u as the set of all left-infinite strings obtained by following transitions backward starting from u, and prepending the corresponding character at every step. Formally, we define the set I_u as follows.

Definition 2.1. *Let $\mathcal{D} = (Q, \Sigma, \delta, s)$ be a DFA and $u \in Q$. We define I_u as:*

$$I_u = \{\alpha \in \Sigma^\omega \mid \text{there exist an infinite sequence of states } u_1, u_2, \dots \text{ in } Q \text{ such that } (i)\ u_1 = u, (ii)\ u_i = \delta(u_{i+1}, \alpha_i), \text{ for every } i \geq 1 \text{ where } \alpha = \dots \alpha_3 \alpha_2 \alpha_1\}$$

We now define the infimum and supremum strings of a state $u \in Q$, as introduced by Conte et al. [16] and Kim et al. [17], representing the lower and upper bounds of the strings in I_u, respectively. Our definition aligns with that of Becker et al. [19].

Definition 2.2 (Infimum and supremum strings)**.** *Let $\mathcal{D} = (Q, \Sigma, \delta, s)$ be a DFA, and let u be a node in Q. Then, the infimum string of u, denoted by $\inf I_u$, is the co-lexicographically smallest string in I_u. The supremum string of u, denoted by $\sup I_u$, is the co-lexicographically largest string in I_u.*

See Figure 1 for an example. This definition is equivalent to the formulation originally introduced by Conte et al. [16] and Kim et al. [17], with the distinction that in Definition 2.2 the infimum and supremum strings are always infinite strings. The existence of $\inf I_u$ and $\sup I_u$ for each $u \in Q$ is guaranteed by Observation 8 of the article of Kim et al. [17].

Wheeler DFA. In the following we report the definition of Wheeler DFAs, which was first introduced by Gagie et al. [12].

Definition 2.3 (Wheeler DFA)**.** *Let $\mathcal{D} = (Q, \Sigma, \delta, s)$ be a DFA. A Wheeler order on $\mathcal{D}$ is a total order $\leq$ on Q such that for every $v = \delta(u, a)$ and $v' = \delta(u', a')$ the following two axioms hold:*

(Axiom 1) *If $a \prec a'$, then $v < v'$.*

(Axiom 2) *If $a = a'$ and $u < u'$, then $v \leq v'$.*

A DFA that admits a Wheeler order is called a Wheeler DFA *(WDFA).*

(Axiom 1) implies a property known in the literature as *input-consistency*: in a Wheeler DFA, all transitions entering a state u must share the same label $a \in \Sigma$. Indeed, if two transitions entering u had distinct labels, this would imply $u < u$, a contradiction. Generally, Gibney and Thankachan [20] showed that recognizing a Wheeler automaton is an NP-complete problem [20]. However, Alanko et al. [21, Theorem 3.3] proved that in the deterministic setting this problem can be solved in linear time, with respect to the DFA size. Furthermore, in the same paper they also showed that a Wheeler DFA admits a unique Wheeler order $\leq$ which can be computed in linear time with respect to the automaton dimension as well. Henceforth, given a Wheeler DFA $\mathcal{D}$ we denote by $u_1, u_2, \dots, u_n$ the sequence of

states of $\mathcal{D}$ sorted according to its unique Wheeler order $\leq$, i.e., $u_i \leq u_j$ holds if and only if $i \leq j$. In the following, we recall that WDFAs satisfy a fundamental structural property, known in the literature as *path coherence*. This property ensures that the states reached by any string form a contiguous interval in the Wheeler state order.

Lemma 2.4 (Path-coherence). *[12, Lemma 3] Let $\mathcal{D}$ be a Wheeler DFA and $\leq$ its Wheeler order. Then for every string $\alpha \in \Sigma^*$ there exists an interval $[i, j]$, with $1 \leq i, j \leq n$, such that $u_i, u_{i+1}, \ldots, u_j$ are those and only states of $\mathcal{D}$ reached by the string α. If no state of $\mathcal{D}$ can be reached by α, then $i > j$ holds.*

We say that a string $\alpha \in \Sigma^*$ reaches a state u, if there exists v such that $u = \delta(v, \alpha)$. As a consequence of path-coherence, it follows that a Wheeler order permutes the states on an automaton consistently with the strings reaching them. Path-coherence also ensures that Wheeler NFAs can be indexed using a strategy analogous to other BWT-based data structures [12], which is known in the literature as *forward search*. The next lemma from the article of Conte et al. [16] establishes a connection between infimum/supremum strings and Wheeler orders in DFAs.

Lemma 2.5. *[16, Lemma 3] Let $\mathcal{D} = (Q, \Sigma, \delta, s)$ be a WDFA and let $\leq$ be its Wheeler order. For any distinct states $u, v \in Q$ with $u < v$, it holds that* $\sup I_u \preceq \inf I_v$.

Finally, for an input DFA we define $T(\alpha)$ as the set of states reached by a walk whose transitions are labeled by the string $\alpha \in \Sigma^*$. This set can be formally defined using the sets I_u as follows: $T(\alpha) = \{u \in Q \mid (\exists \beta \in I_u)(\alpha \dashv \beta)\}$. We recall that by Lemma 2.4 in Wheeler DFAs this set $T(\alpha)$ forms an interval according to its Wheeler order $\leq$.

Computational model. We analyze the time complexity of our algorithms in the I/O model of Aggarwal and Vitter [15]. Specifically, our two-level memory hierarchy is formed by the RAM and the cache. The RAM is formed by contiguous blocks of size B and the performance of our algorithms is evaluated by the number of block transfers from the RAM to the cache. For the analysis of the space complexity, we assume to have RAM words of size $\Omega(\log|\mathcal{D}|)$, where $|\mathcal{D}|$ is the input size.

3. The Graph Suffix Array

In this section, we formally introduce our Graph Suffix Array (GrSA) for solving `count` and `locate` queries over a WDFA. Formally, for every string $\alpha \in \Sigma^*$ the query `count`(α) returns the cardinality of the set $T(\alpha)$, while `locate`(α) returns the set $T(\alpha)$ itself. We refer to these two queries as *subpath queries*. As the set $T(\alpha)$ plays a central role for both queries, in this section we propose an efficient algorithm to compute it. We start this section by proving some preliminary properties on this set $T(\alpha)$, which are at the core of the indexing strategy proposed in this article.

Remark 3.1. *Let $\mathcal{D}$ be a DFA, $\alpha \in \Sigma^*$ and $a \in \Sigma$, then $|T(\alpha a)| \leq |T(\alpha)|$.*

Proof. By definition, every state in $|T(\alpha a)|$ is a successor of some state in $|T(\alpha)|$ via a transition labeled a. Because the transition function δ is deterministic, each state has at most one successor for every symbol $a \in \Sigma$. Hence, the number of such successors cannot exceed the number of starting states, that is: $|T(\alpha a)| \leq |T(\alpha)|$. □

In the paper introducing Wheeler automata, Gagie et al. [12] presented a strategy for indexing the automaton based on the famous forward search, which characterizes BWT-based data structures. We propose a new approach based on binary search, which takes inspiration from the Compressed Suffix [22]. Our key idea is to derive $T(\alpha)$ through an intermediate set $\widetilde{T}(\alpha)$, whose purpose is to approximate it. Formally, for a given DFA this approximated set is defined as $\widetilde{T}(\alpha) = \{u \in Q \mid \alpha \dashv \inf I_u \vee \alpha \dashv \sup I_u\}$. We now prove that in Wheeler DFAs, the set $\widetilde{T}(\alpha)$ exhibits a strong relationship with $T(\alpha)$.

Lemma 3.2. *Let $\mathcal{D}$ be a Wheeler DFA and $\leq$ its Wheeler order. Then for every string $\alpha \in \Sigma^*$ the following properties hold:*

(1) If $\widetilde{T}(\alpha) \neq \emptyset$ *then* $\widetilde{T}(\alpha) = T(\alpha)$,

(2) If $\widetilde{T}(\alpha) = \emptyset$ *then* $|T(\alpha)| \leq 1$.

Proof. *(1)* By definition we know that for every $u \in \widetilde{T}(\alpha)$ it holds that $\alpha \dashv \inf I_u \vee \alpha \dashv \sup I_u$. Since by Definition 2.2 we know $\inf I_u, \sup I_u \in I_u$, it follows that $u \in T(\alpha)$. This proves $\widetilde{T}(\alpha) \subseteq T(\alpha)$. Now let us assume that $\widetilde{T}(\alpha) \neq \emptyset$. To complete the proof we have to show that $T(\alpha) \subseteq \widetilde{T}(\alpha)$. Let v and w be respectively the smallest and largest states in $\widetilde{T}(\alpha)$ according to $\leq$, and consider the infinite strings $\beta_v \in \{\inf I_v, \sup I_v\}$ and $\beta_w \in \{\inf I_w, \sup I_w\}$ satisfying $\alpha \dashv \beta_v, \beta_w$. Suppose by contradiction that there exists a state $u \in T(\alpha) \setminus \widetilde{T}(\alpha)$. If $v < u < w$, by Lemma 2.5 and by Definition 2.2 we know that $\beta_v \preceq \inf I_u \preceq \sup I_u \preceq \beta_w$. However, $\alpha \dashv \beta_v \wedge \alpha \dashv \beta_w$ implies $\alpha \dashv \inf I_u \wedge \alpha \dashv \sup I_u$, a contradiction. Therefore, let us now consider the case $u < v$, the remaining case $w < u$ is analogous. By Lemma 2.5, we know that $\sup I_u \preceq \beta_v$. Moreover, since $u \in T(\alpha)$, there exists $\beta \in I_u$ such that $\alpha \dashv \beta$, and by definition of supremum string we have $\beta \preceq \sup I_u \preceq \beta_v$. However, $\alpha \dashv \beta$ and $\alpha \dashv \beta_v$ imply $\alpha \dashv \sup I_u$, a contradiction.

(2) We show the contrapositive, i.e., that $|T(\alpha)| > 1$ implies $\widetilde{T}(\alpha) \neq \emptyset$. Let $u, v \in T(\alpha)$ be two states such that $u < v$, it follows that there exist $\beta_u \in I_u$ and $\beta_v \in I_v$ satisfying $\alpha \dashv \beta_u, \beta_v$. Moreover, Definition 2.2 and Lemma 2.5 imply that $\beta_u \preceq \sup I_u \preceq \inf I_v \preceq \beta_v$. This in turn implies that $\alpha \dashv \sup I_u$ and $\alpha \dashv \inf I_v$, since α is a proper suffix of both strings β_u and β_v. Therefore, we conclude that $\widetilde{T}(\alpha) \neq \emptyset$. □

We observe that this result implies also that the set $\widetilde{T}(\alpha)$, like $T(\alpha)$, forms a state interval with respect to the Wheeler order, since $\widetilde{T}(\alpha) = T(\alpha)$ or $\widetilde{T}(\alpha) = \emptyset$ hold. In the following corollary we show that the cardinality of the set $\widetilde{T}(\alpha)$ is non-increasing.

Corollary 3.3. *Let* $\mathcal{D}$ *be a Wheeler DFA,* $\alpha \in \Sigma^*$ *and* $a \in \Sigma$*, then* $|\widetilde{T}(\alpha a)| \leq |\widetilde{T}(\alpha)|$.

Proof. By combining Lemma 3.2 and Remark 3.1, it follows that the only case that needs to be proved is that if $|\widetilde{T}(\alpha a)| = 1$, then $|\widetilde{T}(\alpha)| \geq 1$. Therefore, consider $\widetilde{T}(\alpha a) = \{u\}$, and suppose $\alpha a \dashv \sup I_u$, the other case $\alpha a \dashv \inf I_u$ is analogous. By the maximality of $\sup I_u$, it follows that there exists a state v, such that $\alpha \dashv \sup I_v$, and consequently it holds that $\widetilde{T}(\alpha) \neq \emptyset$. □

3.1. The pattern matching algorithm

We now present our novel indexing technique, which supports both `count` and `locate` queries for any string $\alpha \in \Sigma^*$. Recall that, as previously discussed, computing the set $T(\alpha)$ suffices to answer both queries. We now begin by introducing a preliminary notion that will be used extensively in this algorithm. We recall that in the following $u_1, u_2, \ldots, u_n$ represents the ordered list of state with respect to the Wheeler order $\leq$.

Definition 3.4. *Let* $\mathcal{D} = (Q, \Sigma, \delta, s)$ *be a WDFA and let* $\leq$ *be its Wheeler order. We define the list* $\mathcal{K}_\mathcal{D}$ *as:*

$$\mathcal{K}_\mathcal{D} = \big(\inf I_{u_1}, \sup I_{u_1}, \inf I_{u_2}, \sup I_{u_2}, \ldots, \inf I_{u_n}, \sup I_{u_n}\big)$$

We emphasize that, by Lemma 2.5, the strings in $\mathcal{K}_\mathcal{D}$ are sorted co-lexicographically. Consequently, the elements of $\mathcal{K}_\mathcal{D}$ that are suffixed by the same string α constitute an interval in $\mathcal{K}_\mathcal{D}$, which implies that also $\widetilde{T}(\alpha)$ forms an interval in $\mathcal{K}_\mathcal{D}$. The algorithm assumes access to an oracle that we term *suffix match* and denote by SM. Given a string $\alpha \in \Sigma^*$, the oracle $\texttt{SM}(\alpha)$ returns the maximal interval $]i, j[$, with $0 \leq i < j \leq 2n + 1$, such that: (i) either $i = 0$ or $\mathcal{K}_\mathcal{D}[i] \prec \alpha$, (ii) either $j = 2n + 1$ or $\alpha \prec \mathcal{K}_\mathcal{D}[j]$, and (iii) $\alpha \dashv \mathcal{K}_\mathcal{D}[k]$ for every k such that $i < k < j$.

Note that, by definition of $\widetilde{T}(\alpha)$, once we obtain the interval $]i, j[$ output by $\texttt{SM}(\alpha)$ we can easily reconstruct the sequence $u_{i'}, u_{i'+1}, \ldots, u_{j'}$ corresponding to $\widetilde{T}(\alpha)$ as $i' = \lceil (i+1)/2 \rceil$ and $j' = \lceil (j-1)/2 \rceil$. Our algorithm consists of two main parts. In the first part, we start by finding the longest prefix β of α such that $\widetilde{T}(\beta) \neq \emptyset$. This operation is done by performing a binary search over the length of the input string α, where at every iteration we call our oracle SM to compute the corresponding set

$\widetilde{T}(\alpha)$. Note that, we can find such string through a binary search, since by Corollary 3.3 the cardinality of $\widetilde{T}$ is non-increasing when we consider longer prefixes of α. From Property (1) of Lemma 3.2, we know that since $\widetilde{T}(\beta)$ is not empty we have that $\widetilde{T}(\beta) = T(\beta)$. Consequently, if $\alpha = \beta$, then we can directly use $\widetilde{T}(\beta)$ to compute $\texttt{count}(\alpha)$ and $\texttt{locate}(\alpha)$ and we are done. Otherwise, let $a \in \Sigma$ and $\gamma \in \Sigma^*$ be respectively the character and the string satisfying $\alpha = \beta a \gamma$. By definition of β, we have that $\widetilde{T}(\beta a) = \emptyset$, and it follows from Property (2) of Lemma 3.2 that $|T(\beta a)| \leq 1$. Consequently, in this case we identify the unique state u_j that may belong to $T(\beta a)$. We will show that such a state u_j can be identified through a call to our oracle $\textsc{sm}(\beta a)$. Next, we check if there exists a transition labeled a, leaving a state $v \in \widetilde{T}(\beta)$. In particular, we will observe that $T(\beta a) = \{u_j\}$ holds if and only if such a transition exists, which must necessarily enter into u_j. In this case, to complete the algorithm execution it is sufficient to understand if there exists an integer k, with $1 \leq k \leq n$, such that $u_k = \delta(u_j, \gamma)$. If these conditions are not satisfied we will conclude that $\texttt{count}(\alpha) = 0$ and $\texttt{locate}(\alpha) = \emptyset$.

Algorithm execution. We now present a formal description of the algorithm, a pseudocode formulation is presented in Algorithm 1. The inputs of our algorithm are the input string $\alpha \in \Sigma^*$, $\mathcal{D} = (Q, \Sigma, \delta, s)$ the Wheeler DFA and the oracle $\textsc{sm}$. The algorithm starts by performing a binary search over the length of α to find the longest prefix β of α such that $\widetilde{T}(\beta)$ is not empty. To do so, we initialize the endpoints of the binary search $[hi, lo]$ to $[1, |\alpha|]$, and set the prefix β to the empty string ε. At each iteration of the binary search, we define β' as the prefix of α of length $mid = lo + \lfloor (hi - lo)/2 \rfloor$, and we compute the open interval $]i, j[$ by calling the oracle $\textsc{sm}(\beta')$. If $j > i + 1$, the interval contains at least one value, implying that $\widetilde{T}(\beta') \neq \emptyset$ due to the oracle definition. Therefore, in this case, we update the endpoint lo to $mid + 1$ to search for a longer prefix satisfying the desired property, and we update β to β' since $|\beta| < |\beta'|$. Otherwise, if the interval is empty (i.e., $j = i + 1$), then $\widetilde{T}(\beta')$ is also empty. It follows from Corollary 3.3 that any prefix β'' longer than β' must necessarily satisfy the condition $\widetilde{T}(\beta'') = \emptyset$. Therefore, in this second case we update the endpoint hi to $mid - 1$, to search if a shorter prefix satisfies the desired property. As a consequence of that, upon termination, β is guaranteed to be the longest prefix of α such that $\widetilde{T}(\beta) \neq \emptyset$. Once β is found, we compute the open interval $]i, j[$ returned by the oracle $\textsc{sm}(\beta)$. We then determine the indices relative to the Wheeler order $\leq$ by updating the values to $i = \lceil (i+1)/2 \rceil$ and $j = \lceil (j-1)/2 \rceil$, such that $\widetilde{T}(\beta) = \{u_i, u_{i+1}, \ldots, u_j\}$. If $\alpha = \beta$, we return this set, since $T(\alpha) = \widetilde{T}(\beta)$. Otherwise, let a be the character succeeding β in α. We query the oracle $\textsc{sm}(\beta a)$ to obtain its corresponding interval $]i', j'[$. Since $\widetilde{T}(\beta a) = \emptyset$, it follows that $i' + 1 = j'$, since $i' + 1 < j'$ would imply the existence of a state u such that either $\beta a \dashv \inf I_u$ or $\beta a \dashv \sup I_u$. We then update the indices to $i' = \lceil i'/2 \rceil$ and $j' = \lceil j'/2 \rceil$. If $i' + 1 = j'$, it follows that βa does not reach any state in the Wheeler DFA, since in this case $\sup I_{u_{i'}} \prec \beta a \prec \inf I_{u_{j'}}$ and therefore βa cannot suffix any string in I_u, for every $u \in Q$. Therefore, in this case we conclude that $T(\beta a)$ is empty and thus we return $\emptyset$. Otherwise, if $i' = j'$, then we know that for the state $u_{i'}$ it holds that $\inf I_{u_{i'}} \prec \beta a \prec \sup I_{u_{i'}}$. Therefore, in this case $u_{i'}$ is the only candidate state that can belong to $T(\beta a)$, thus we want to check if there exists a string $\alpha' \in I_{u_{i'}}$ such that $\beta a \dashv \alpha'$. Consider, the sequence $u_i, u_{i+1}, \ldots, u_j$ forming $T(\beta)$. To do so, we check if there exists $k \in [i, j]$ such that $\delta(u_k, a) \neq \perp$. This verifies whether there exists a transition labeled a leaving any state in $T(\beta)$. If such an integer k does not exist, then since $u_i, u_{i+1}, \ldots, u_j$ are those and only states reached by β, we conclude that $T(\beta a)$ is empty and so we return $\emptyset$. On the other hand, if such an integer k exists, then the state reached by this transition $\delta(u_k, a)$ belongs to $T(\beta a)$ by construction, and therefore since $u_{i'}$ is the only candidate that can be part of $T(\beta a)$, in this case we conclude that $\delta(u_k, a) = u_{i'}$. Therefore, to terminate the algorithm we initialize γ, such that $\alpha = \beta a \gamma$, and we compute $w = \delta(u_{i'}, \gamma)$. Since $\alpha = \beta a \gamma$, it follows that in this latter case that if $w = \perp$, then $T(\alpha) = \emptyset$ and so we return the empty set. Otherwise, if $w \in Q$, due to an analogous reasoning, we return the set $T(\alpha) = \{w\}$ and we are done.

Computing $\widetilde{\mathbf{T}}$. In this section, we have seen that the binary search shown in Line 1 of Algorithm 1 returns the longest prefix β of α for which $\widetilde{T}(\beta) \neq \emptyset$ holds. Therefore, by combining this result with Lemma 3.2 and Corollary 3.3, it follows that we can also compute $T(\alpha)$, if $|T(\alpha)| \geq 2$, by just calling

the oracle SM once over the string α without having to navigate the input DFA. To achieve this, once we have called $\texttt{SM}(\alpha)$, we return the set $\widetilde{T}(\alpha)$ if $|\widetilde{T}(\alpha)| \geq 2$, and $\emptyset$ otherwise. See Algorithm 2 for a pseudocode.

Algorithm 1: $\texttt{ComputeT}(\alpha)$

Input : A WDFA $\mathcal{D} = (Q, \Sigma, \delta, s)$, a string $\alpha \in \Sigma^*$, the oracle SM
Output: The set $T(\alpha)$

Function $\texttt{LocateStates}(\alpha, \mathit{SM})$:
  $lo \leftarrow 1,\ hi \leftarrow |\alpha|$ // Extremes of the binary search
  $\beta \leftarrow \varepsilon$ // Initialise β as the empty string
  while $lo \leq hi$ **do** // Compute longest prefix β with $\widetilde{T}(\beta) \neq \emptyset$
    $mid \leftarrow lo + \lfloor (hi - lo)/2 \rfloor$
    $\beta' \leftarrow \alpha[1, mid]$ // Candidate to update β
    $i, j \leftarrow \texttt{SM}(\beta')$
    if $j > i + 1$ **then** // If $\widetilde{T}(\beta') \neq \emptyset$
      $lo \leftarrow mid + 1,\ \beta \leftarrow \beta'$ // update lowest extreme and β
    else $hi \leftarrow mid - 1$ // Otherwise update the highest extreme
  $i, j \leftarrow \texttt{SM}(\beta)$
  $i \leftarrow \lceil (i+1)/2 \rceil,\ j \leftarrow \lceil (j-1)/2 \rceil$ // $T(\beta) = \{u_i, u_{i+1}, \ldots, u_j\}$
  return i, j, β
$i, j, \beta \leftarrow \texttt{LocateStates}(\alpha, \mathit{SM})$
if $\beta = \alpha$ **then return** $\{u_i, u_{i+1}, \ldots, u_j\}$ // If $T(\alpha) = T(\beta)$
$a \leftarrow \alpha[|\beta| + 1]$
$i', j' \leftarrow \texttt{SM}(\beta a)$
$i' \leftarrow \lceil i'/2 \rceil,\ j' \leftarrow \lceil j'/2 \rceil$ // Either $T(\beta a) = \emptyset$ or $T(\beta a) = \{u_{i'}\}$
if $i' < j'$ **then return** $\emptyset$ // Case 1) $T(\beta a) = \emptyset$
if $\exists k \in [i, j]$ *s.t.* $\delta(u_k, a) \neq \perp$ **then** // Case 2) $T(\beta a) = \{u_{i'}\}$
  $\gamma \leftarrow \alpha[|\beta| + 2, |\alpha|]$ // $\alpha = \beta a \gamma$
  $w \leftarrow \delta(u_{i'}, \gamma)$ // compute state w reached by α
  if $w \neq \perp$ **then return** $\{w\}$
return $\emptyset$

Algorithm 2: $\texttt{Compute}\widetilde{T}(\alpha)$

Input : A WDFA $\mathcal{D} = (Q, \Sigma, \delta, s)$, a string $\alpha \in \Sigma^*$, and the oracle SM
Output: The set $T(\alpha)$, if $|T(\alpha)| \geq 2$ and $\emptyset$ otherwise

$i, j \leftarrow \texttt{SM}(\alpha)$
$i \leftarrow \lceil (i+1)/2 \rceil, j \leftarrow \lceil (j-1)/2 \rceil$
if $i < j$ **then** // If $|T(\alpha)| \geq 2$
  return $\{u_i, u_{i+1}, \ldots, u_j\}$ // Return $T(\alpha) = \{u_i, u_{i+1}, \ldots, u_j\}$
return $\emptyset$

4. Time and space complexity

In Section 3, we presented our new indexing strategy and we showed its correctness. Thus, to complete the proofs of Lemma 1.1, Theorem 1.2, and Corollary 1.3 we now analyze the space and time complexities of our novel pattern matching algorithms. Specifically, we partition this analysis into three different parts, each corresponding to one of the three substrings into which the pattern is split (i.e., $\alpha = \beta a \gamma$).

Part 1, the Oracle. We present an implementation of the oracle SM which, given a string $\beta' \in \Sigma^*$, returns the endpoints of the maximal interval $]i, j[$ defined in Section 3. Our approach requires storing the *infimum and supremum pseudoforests* of the automaton $\mathcal{D}$ as defined by Kim et al. [17]. In each of these pseudoforests, every node has exactly one incoming edge; following the unique infinite walk entering a node yields its corresponding infimum (or supremum) string. Figure 1 provides a graphical representation of these pseudoforests. At every call of the oracle $\text{SM}(\beta')$, for a string $\beta' \in \Sigma^*$, we perform a binary search over the interval $[1, 2n]$ to identify the greatest value i such that $\mathcal{K}_\mathcal{D}[i] \prec \beta'$, if this value does not exist we set $i = 0$. At each step of the binary search, we need to co-lexicographically compare β' with an entry in $\mathcal{K}_\mathcal{D}$, that is either an infimum or a supremum string depending on the index. Each comparison requires reconstructing the corresponding infimum (supremum) string by traversing the infimum (supremum) pseudoforest in a backward fashion, i.e., towards the tree root, starting from the corresponding node. Once we have retrieved this integer i, we perform a second binary search in the interval $[i + 1, 2n]$ to identify the smallest value j such that $\beta' \prec \mathcal{K}_\mathcal{D}[j] \wedge \neg(\beta' \dashv \mathcal{K}_\mathcal{D}[j])$ by using the same strategy.

To traverse these pseudoforests in a cache-friendly manner, we extend the standard Heavy-Light Decomposition (HLD) [23] from rooted trees to pseudotrees, and consequently, to pseudoforests. In HLD, every internal node u has a unique *heavy* edge (u, v), where v is the child of u that has the largest subtree. The heavy paths, formed by grouping heavy edges together, are stored in contiguous memory locations. All the other edges from u to its children are designated as *light* edges. This ensures that any node-to-root path encounters at most $\mathcal{O}(\log n)$ light edges. To extend this logic, we partition a pseudotree into its unique cycle and a set of disjoint subtrees. The cycle acts as a circular root; each node in the cycle serves as the root of the subtree containing all the nodes reachable via paths that do not traverse cycle edges. We linearize the cycle and apply HLD to each subtree independently, where the root of each subtree has a pointer to its position in the linearized cycle. Since following heavy edges cause $\mathcal{O}(\frac{m}{B})$ cache misses, and we can encounter at most $\mathcal{O}(\log n)$ light edges, we deduce that the total number of cache misses for this operation is $\mathcal{O}(\frac{m}{B} + \log n)$. Therefore, the overall complexity of a single call to the oracle can cause at most $\mathcal{O}((\frac{m}{B} + \log n) \log n)$ cache misses, since SM executes this operation for every iteration of the binary search over the interval $[1, 2n]$. This completes the proof of Corollary 1.3, because in Algorithm 2 we call SM only once and both HLD require $\mathcal{O}(n)$ RAM words of space to be stored (every pseudoforest has n nodes and n edges). Consequently, since Algorithm 1 calls the oracle SM at most $\mathcal{O}(\log m)$ times, the total number of cache misses from Line 1 to Line 13 is bounded by $\mathcal{O}((\frac{m}{B} + \log n) \log n \log m)$, which consequently completes the proof of Lemma 1.1.

Part 2, the outgoing transition. In the second phase of Algorithm 1, corresponding to Line 20, we have to determine whether there exists a transition labeled $a \in \Sigma$ leaving a state within the range $[u_i, u_j]$, where $\{u_i, u_{i+1}, \ldots, u_j\} = T(\beta)$. To this end, we utilize the Wheeler NFA representation proposed by Gagie et al. [12], which stores outgoing transitions via two sequences: OUT, containing edge labels ordered by the Wheeler rank of their source states, and OUT_DEG, a bitvector encoding the out-degrees in unary (see Figure 1). By augmenting the bitvector OUT_DEG with auxiliary data structures, we can support constant-time rank [24] and select [25] operations in $n + |\delta| + o(n + |\delta|)$ bits. Moreover, by representing OUT with alphabet partitioning [26, Theorem 3.2], we can encode the sequence using $\mathcal{O}(|\delta| \log \sigma)$ bits, while supporting rank operations in $\mathcal{O}(\log \log \sigma)$. Since we are assuming that the alphabet is effective, and since Wheeler DFAs are input-consistent, we conclude that $\sigma \leq n$. Consequently, the I/O complexity of this operation is dominated by the complexity of the

previous part. As shown by Gagie et al. [12], these operations are sufficient to verify the existence of a transition labeled a from the set $T(\beta)$.

Part 3, navigating the automaton. The third component, which is used to check the final segment γ, is a representation of the original DFA. In this third part, to decrease the number of cache misses, we aim to exploit the *sparseness* of the input automaton $\mathcal{D}$. Indeed, in applications like bioinformatics, researchers showed that pangenome graphs may be particularly sparse and linear, which means that most of the nodes have just one outgoing edge [18]. Therefore, to achieve a cache-friendly solution, we employ a *path-compression* over the input DFA. Path compression collapses sequences of states having exactly one outgoing edge into a contiguous location of the memory. We call each of these collapsed sequences a *unary path*, whose traversal can be executed in a cache-friendly manner through a sequential scan. Formally, we store contiguously in memory the maximal sequences of states $\hat{u}_1, \hat{u}_2, \ldots, \hat{u}_p$ satisfying: (i) the states $\hat{u}_1, \hat{u}_2, \ldots \hat{u}_{p-1}$ have out-degree equal to 1, (ii) the states $\hat{u}_2, \hat{u}_3, \ldots \hat{u}_p$ have in-degree equal to 1, and (iii) for every $i \in [p-1]$, it holds that $\hat{u}_{i+1} \in \delta_a(\hat{u}_i)$, for some $a \in \Sigma$. Figure 1 shows all the maximal unary paths for a given DFA $\mathcal{D}$.

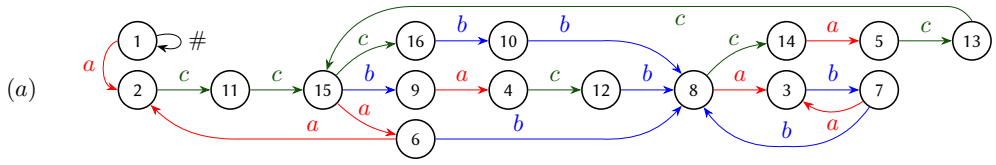
To navigate between unary paths, we use the following data structure. Let v be the last state of an arbitrary unary path, and let out_v be its out-degree. We use a minimal perfect hash function $h_v : \Sigma \to [out_v]$ and a hash table H of size out_v, storing at position $h_v(a)$ the pointer to the state z such that $z \in \delta_a(v)$ (if any). By using the minimal perfect hash function described in the book of Ferragina [27, Theorem 8.8], we can evaluate the functions h_v in $O(1)$, therefore we have $\mathcal{O}(d)$ cache misses in the worst-case required to move from a unary path to another, where d is the number of unary paths we encounter while traversing $\mathcal{D}$ using the string γ. Moreover, since each unary path is stored in contiguous positions of the memory, traversing the unary paths can trigger at most $\mathcal{O}(\frac{|\gamma|}{B})$ cache misses, which becomes $\mathcal{O}(\frac{m}{B})$ since $|\gamma| < m$. Therefore, this data structure allows us to execute Algorithm 1 from Line 21 to Line 23 triggering at most $\mathcal{O}(d + \frac{m}{B})$ cache misses. Despite the fact that the parameter d is $\Theta(m)$ in the worst-case on arbitrary DFAs, in Section 5 we experimentally observe that on pangenome graphs, due to their linear structure, is significantly smaller.

Concerning the space complexity, the minimal perfect hash function h_v described by Ferragina uses $\mathcal{O}(|out_v|)$ RAM words [27, Theorem 8.8]. Thus, the total space complexity becomes $\mathcal{O}(|\mathcal{D}|)$ RAM words, where $|\mathcal{D}|$ is the input dimension (number of transitions plus number of states), when we account for all the states, as well as the other components of $\mathcal{D}$. Therefore, by accounting the three parts, we obtain an overall number of cache misses given by $\mathcal{O}((\frac{m}{B} + \log n) \log n \log m + d)$, while the space complexity is $\mathcal{O}(|\mathcal{D}|)$ RAM words, which completes the proof of Theorem 1.2.

5. Experiments

We implemented our novel indexing technique based on the Graph Suffix Array (Algorithm 1), in C++ and made the source code available at https://github.com/regindex/Gr-SA/tree/ictcs-paper. To assess its performance, we compared it with the classical forward search algorithm for BWT-based indexes. Since to the best of our knowledge there exists no public code of this latter algorithm, we also provided an implementation of the forward search on C++. We ran our algorithm on Wheeler pangenome graphs constructed from the dbSNP database [28]. We use pangenome graphs constructed from the genomic data related to the chromosomes 21 and 14. All experiments were run on a server equipped with an Intel Xeon W-2245 CPU (3.90 GHz, 8 cores) and 128 GB of RAM, running Ubuntu 18.04 LTS 64-bit.

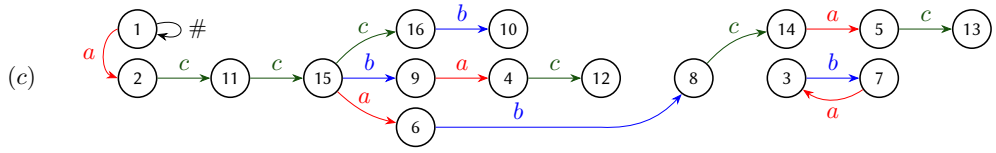
Implementation details. Our practical implementation differs from the description provided in Section 4 in the following aspects. (i) Our solution is optimized for alphabets Σ of size 4, i.e., $\Sigma = \{A, C, G, T\}$. Thus, we do not use minimal perfect hash functions to navigate between unary paths, but we store the outgoing transitions of the last state v in the unary path through contiguous pairs (a, u), where a is the character label, and u is a pointer to the destination state. Scanning this sequence of pairs (a, u) will be fast since $|out(v)| \leq 4$. (ii) To save memory space, we do not store explicitly

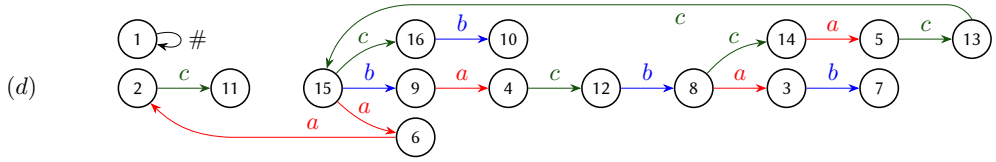


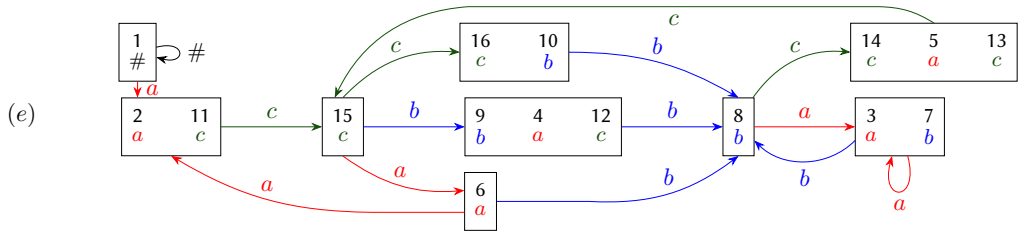
(b)

OUT_DEG = 0 0 1 0 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1

OUT = # a c b c c a b a b a c a b c a b c b







1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16	
inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup	inf	sup
#	#	a	a	a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b	c	c	c	c	c	c	c	c	c	c	c	c
#	#	#	a	b	b	b	b	c	c	c	c	a	a	a	c	c	c	c	c	a	a	a	a	a	a	b	b	c	c	c	c
#	#	#	c	a	c	c	c	b	b	c	c	b	b	c	a	c	c	c	c	#	a	b	b	c	c	a	c	a	a	c	c
#	#	#	c	b	a	c	c	a	c	a	a	a	c	c	b	a	a	c	c	#	c	c	c	b	b	c	a	#	c	a	a
#	#	#	a	a	b	a	a	c	a	#	c	b	a	a	c	#	c	a	a	#	c	c	c	a	c	c	b	#	b	#	c
#	#	#	c	b	c	#	c	c	b	#	b	a	b	#	c	#	b	#	c	#	a	a	a	c	a	a	c	#	c	#	b
#	#	#	b	a	c	#	b	a	c	#	c	b	c	#	a	#	c	#	b	#	c	#	c	c	b	#	c	#	a	#	c
#	#	#	c	b	a	#	c	#	c	#	a	a	c	#	c	#	a	#	c	#	b	#	b	a	c	#	a	#	b	#	a

Figure 1: The figure shows: (a) a DFA $\mathcal{D}$, (b) the sequences OUT_DEG and OUT representing the outgoing transitions of $\mathcal{D}$, (c) the infimum pseudoforest of $\mathcal{D}$, (d) the supremum pseudoforest of $\mathcal{D}$, and (e) the DFA $\mathcal{D}$ partitioned into its maximal unary paths. The tables shows a suffix of the infimum and supremum strings in $\mathcal{D}$.

the infimum and supremum pseudoforests. Instead, we use the information contained in the plain representation of the DFA to read the infimum and supremum string when we call the oracle SM. The graph is linearized in such a way to form a HLD on the infimum pseudoforest. Since most of the nodes have only one incoming edge, this linearization will decrease the number of cache misses also when we read supremum strings. (iii) We store in a table T all the intervals $]i, j[$ returned by the oracle $\textsc{sm}(\beta')$ for every string β' of length k. This parameter k is chosen is such a way that the table T occupies at most 40% of the total space used by the other components of the index. When we call $\textsc{sm}(\beta)$, we fetch the integers i and j from $T[\beta']$, where β' is the suffix of β of length k. This allows us to run the binary search of Line 1 on the restricted interval $[i, j]$, instead of $[1, 2n]$. All these heuristics result in a significant improvement of the time and space performance of our algorithm.

Experiments description. We conduct two types of experiments. In the first experiment we empirically measure the value d, representing the number of encountered unary paths in the navigation step. To do so, we perform 10^4 locate queries for every analyzed length of the pattern, and we measure the

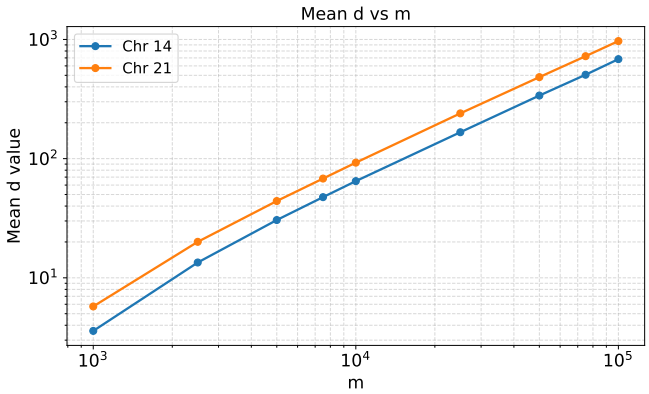


Figure 2: Mean d (number of traversed unary paths) measured during locate(α) versus length m (log-log scale). The mean is computed across 10^4 random queries, where each string α of length m is sampled from the graph.

number of unary path traversed in the third phase of the algorithm. The results are shown in Figure 2. As we can observe, due to their sparseness, the typical values of d are on both graphs about two orders of magnitude smaller compared to the pattern length. This shows that the number of cache misses triggered during the third phase of Algorithm 1 (i.e., from Line 21 to Line 23) does not have a significant impact on the algorithm performance. Next, we compare the time and space efficiency of our index (Gr-SA) with respect to the original forward search (Fw.) algorithm for Wheeler graphs (Fw.) [12]. We considered only patterns appearing in the graphs and we reported the average time to process a single character of those patterns. The results are summarized in Table 1.

Dataset	WDFA Properties		Index Size			Queries		Execution Time		
	n	$\lvert\delta\rvert$	Fw.	Gr-SA	Blowup	m	$m/\lvert\beta\rvert$	Fw.	Gr-SA	Speedup
Chr 21	3.85×10^7	3.88×10^7	38.1 MB	528.9 MB	$\times 13.9$	10^3	0.58	968.5 ns	9.4 ns	$\times 103$
						10^4	0.09	976.8 ns	2.4 ns	$\times 410$
Chr 14	9.44×10^7	9.47×10^7	93.2 MB	1374.7 MB	$\times 14.8$	10^3	0.70	1199.0 ns	10.0 ns	$\times 119$
						10^4	0.30	1192.8 ns	2.4 ns	$\times 501$

Table 1
The table shows the space usage and the average query time used by the forward search (Fw) and the Graph Suffix Array (Gr-SA) on the pangenomes built from chromosomes 21 and 14. Specifically, the table shows: (i) the number or size of the pangenome graph; (ii) the space usage for both indexes; (iii) the length m of the patterns and the average ratio of m to the length of the longest prefix appearing at least twice in the graph; (iv) the average execution time per character of the pattern.

Acknowledgments

We would like to thank Daniel Puttini for providing us the pangenome graphs employed in the experimental section of the paper, and Davide Cenzato for fruitful discussions on the implementation details. All authors are funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biology 10 (2009) R25. doi:10.1186/gb-2009-10-3-r25.

[2] H. Li, N. Homer, A survey of sequence alignment algorithms for next-generation sequencing, Briefings Bioinform. 11 (2010) 473–483. doi:10.1093/BIB/BBQ015.

[3] D. E. Knuth, J. H. Morris, Jr, V. R. Pratt, Fast pattern matching in strings, SIAM journal on computing 6 (1977) 323–350.

[4] P. Weiner, Linear Pattern Matching Algorithms, in: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973, IEEE Computer Society, 1973, pp. 1–11. doi:10.1109/SWAT.1973.13.

[5] E. M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, J. ACM 23 (1976) 262–272. doi:10.1145/321941.321946.

[6] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, in: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '90, Society for Industrial and Applied Mathematics, USA, 1990, p. 319–327.

[7] M. Burrows, D. J. Wheeler, A Block-sorting Lossless Data Compression Algorithm, Technical Report 124, Digital Equipment Corporation, 1994. URL: https://www.hp.com/hpinfo/ex-labs/research/src/digital_src_research_report_1994_124.pdf.

[8] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (2005) 552–581. URL: https://doi.org/10.1145/1082036.1082039. doi:10.1145/1082036.1082039.

[9] R. Williams, A new algorithm for optimal 2-constraint satisfaction and its implications, Theor. Comput. Sci. 348 (2005) 357–365. doi:10.1016/J.TCS.2005.09.023.

[10] M. Equi, V. Mäkinen, A. I. Tomescu, Graphs Cannot Be Indexed in Polynomial Time for Sub-quadratic Time String Matching, Unless SETH Fails, in: T. Bures, R. Dondi, J. Gamper, G. Guerrini, T. Jurdzinski, C. Pahl, F. Sikora, P. W. H. Wong (Eds.), Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2021, Bolzano-Bozen, Italy, January 25-29, 2021, Proceedings, Lecture Notes in Computer Science, Springer, 2021, pp. 608–622. doi:10.1007/978-3-030-67731-2_44.

[11] M. Equi, V. Mäkinen, A. I. Tomescu, R. Grossi, On the Complexity of String Matching for Graphs, ACM Trans. Algorithms 19 (2023) 21:1–21:25. doi:10.1145/3588334.

[12] T. Gagie, G. Manzini, J. Sirén, Wheeler graphs: A framework for BWT-based data structures, Theor. Comput. Sci. 698 (2017) 67–78. doi:10.1016/J.TCS.2017.06.016.

[13] T. C. P.-G. Consortium, Computational pan-genomics: status, promises and challenges, Briefings in Bioinformatics 19 (2018) 118–135. doi:10.1093/bib/bbw089.

[14] J. Sirén, N. Välimäki, V. Mäkinen, Indexing Graphs for Path Queries with Applications in Genome Research, IEEE ACM Trans. Comput. Biol. Bioinform. 11 (2014) 375–388. doi:10.1109/TCBB.2013.2297101.

[15] A. Aggarwal, S. Vitter, Jeffrey, The input/output complexity of sorting and related problems, Commun. ACM 31 (1988) 1116–1127. doi:10.1145/48529.48535.

[16] A. Conte, N. Cotumaccio, T. Gagie, G. Manzini, N. Prezza, M. Sciortino, Computing Matching Statistics on Wheeler DFAs, in: 2023 Data Compression Conference (DCC), 2023, pp. 150–159. doi:10.1109/DCC55655.2023.00023.

[17] S.-H. Kim, F. Olivares, N. Prezza, Faster Prefix-Sorting Algorithms for Deterministic Finite Automata, in: L. Bulteau, Z. Lipták (Eds.), 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023), volume 259 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 16:1–16:16. doi:10.4230/LIPIcs.CPM.2023.16.

[18] J. Sirén, E. Garrison, A. M. Novak, B. Paten, R. Durbin, Haplotype-aware graph indexes, Bioinformatics 36 (2020) 400–407. doi:10.1093/bioinformatics/btz575.

[19] R. Becker, N. Cotumaccio, S. Kim, N. Prezza, C. Tosoni, Encoding Co-Lex Orders of Finite-

State Automata in Linear Space, in: P. Bonizzoni, V. Mäkinen (Eds.), 36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, Milan, Italy, June 17-19, 2025, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 15:1–15:17. doi:10.4230/LIPICS.CPM.2025.15.

[20] D. Gibney, S. V. Thankachan, On the Hardness and Inapproximability of Recognizing Wheeler Graphs, in: M. A. Bender, O. Svensson, G. Herman (Eds.), 27th Annual European Symposium on Algorithms, ESA 2019, Munich/Garching, Germany, September 9-11, 2019, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 51:1–51:16. doi:10.4230/LIPICS.ESA.2019.51.

[21] J. Alanko, G. D'Agostino, A. Policriti, N. Prezza, Regular Languages meet Prefix Sorting, in: S. Chawla (Ed.), Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, SIAM, 2020, pp. 911–930. doi:10.1137/1.9781611975994.55.

[22] R. Grossi, J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract), in: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, STOC '00, Association for Computing Machinery, New York, NY, USA, 2000, p. 397–406. doi:10.1145/335305.335351.

[23] D. D. Sleator, R. E. Tarjan, A Data Structure for Dynamic Trees, J. Comput. Syst. Sci. 26 (1983) 362–391. doi:10.1016/0022-0000(83)90006-5.

[24] G. Jacobson, Space-efficient Static Trees and Graphs, in: 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989, IEEE Computer Society, 1989, pp. 549–554. doi:10.1109/SFCS.1989.63533.

[25] D. Clark, Compact Pat Trees, Phd thesis, University of Waterloo, 1996. URL: http://www.nlc-bnc.ca/obj/s4/f2/dsk3/ftp04/nq21335.pdf, section 2.2.2.

[26] D. Belazzougui, G. Navarro, Optimal Lower and Upper Bounds for Representing Sequences, ACM Trans. Algorithms 11 (2015) 31:1–31:21. doi:10.1145/2629339.

[27] P. Ferragina, Pearls of Algorithm Engineering, Cambridge University Press, 2023. doi:10.1017/9781009128933.

[28] University of Helsinki, DFAs encoding the finnish subset of frequent mutations from dbSNP (GCSA's experimental data), https://www.cs.helsinki.fi/group/gsa/1000gen-FIN-WholeGenome/, 2014. Accessed: 2026-02-04.