On Prefix-Sorting Finite Automata*

Jarno Alanko¹, Alberto Policriti², and Nicola Prezza³**

¹ University of Helsinki, Finland, jarno.alanko@helsinki.fi
² University of Udine, Italy, alberto.policriti@uniud.it
³ University of Pisa, Italy, nicola.prezza@di.unipi.it

Abstract. Indexing strings via prefix (or suffix) sorting is, arguably, one of the most successful algorithmic techniques developed in the last decades. String indexes allow solving efficiently a large number of problems, including counting and locating occurrences of a pattern in the indexed string. Can indexing be extended to languages? In this paper, we approach the problem by combining techniques from string processing (specifically, prefix-sorting) and automata theory (specifically, DFA minimization). Our main contributions are algorithms that, given a finite language represented either explicitly as a set of strings or implicitly as an acyclic DFA, generate the minimum accepting DFA that can be prefix-sorted and thus indexed for linear-time pattern matching queries. In order to achieve this result we use the recent notion of Wheeler graph [Gagie et al., TCS 2017], which extends naturally the concept of prefix sorting to labeled graphs. The main obstacle in following our approach is that, while several structures including strings, trees, and de Bruijn graphs can always be prefix-sorted, the situation on general graphs is more complicated: not all graphs enjoy this property, and a recent result shows that the problem of identifying them is NP-complete even for acyclic NFAs. In addition, NFA minimization is a notoriously hard problem (even in the acyclic case). We work around these difficulties by showing that these problems become tractable on DFAs. We start with three polynomial-time algorithms to recognize and prefix-sort Wheeler finite automata: a quadratic algorithm to sort a proper superset of the Wheeler DFAs (WDFAs), a $O(n \log n)$-time online algorithm to sort acyclic WDFAs, and an optimal linear-time offline algorithm to sort general WDFAs. We then provide a minimization algorithm that generates the smallest WDFA recognizing the same language of any input WDFA. The algorithm runs in optimal linear time in the acyclic case, and in $O(n \log n)$ time in the general case. Finally, we combine our techniques and provide a near-optimal algorithm that converts any acyclic DFA into the smallest equivalent WDFA. This contribution is also a big step towards a complete solution to the well-studied problem of indexing graphs for linear-time pattern matching queries: our algorithm essentially solves the deterministic-acyclic case with a solution of provably-minimum size.

* We wish to thank Travis Gagie for introducing us to the problem and for stimulating discussions.
** Corresponding author. Supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.
1 Introduction

Prefix-sorting is the process of ordering the positions of a string in the co-lexicographic order of their corresponding prefixes\(^4\). Once this step has been performed, several problems on strings become much easier to solve: for example, substrings can be located efficiently in the string without the need to read all of its characters. Given the versatility of this tool, it is natural trying to generalize it to more complex objects such as edge-labeled trees and graphs. For example, a procedure for lexicographically-sorting the states of a finite-state automaton could be useful to speed up subsequent membership queries in its accepting language or its substring/suffix closure. The newborn theory of Wheeler graphs [10] provides such a generalization. Intuitively, a labeled graph is Wheeler if and only if its nodes can be co-lexicographically sorted consistently in a total order according to their incoming paths (i.e. a node \(u\) precedes a node \(v\) iff all paths reaching \(u\) are co-lexicographycally smaller than those reaching \(v\)). As a consequence, Wheeler graphs admit indexes for linear-time exact pattern matching queries (also known as path queries). Wheeler graphs generalize several lexicographically-sorted structures studied throughout the past decades: indexes on strings [9, 21, 31], sets of strings [23], trees [8], de Bruijn graphs [4], variable-order de Bruijn graphs [29], wavelet trees [13], wavelet matrices [3]. These efforts are part of a more general wave of interest (dating as far back as 27 years ago [22]) towards techniques aimed at solving pattern matching on labeled graphs [1, 5, 6, 8, 10, 17, 22, 27, 29, 30]. As discussed above, existing graph-indexing solutions can only deal with simple labeled graphs. The problem of indexing general (or even just acyclic) graphs with a solution of provably-minimum size remains unsolved. Unfortunately, not all graphs enjoy the Wheeler property and, as Gibney and Thankachan [12] have recently shown, the problem of recognizing (and sorting) them turns out to be NP-complete even when the graph is acyclic (this includes, in particular, acyclic NFAs). Even worse, not all regular languages admit an accepting Wheeler finite automaton: the set of Wheeler languages is a proper superset of the finite languages and a proper subset of the regular languages [10]. Even when an index is not used, exact pattern matching on graphs is hard: Equi et al. [5, 6] have recently shown that any solution to the problem requires at least quadratic time (under the Orthogonal Vectors hypothesis), even on acyclic DFAs. In particular, this implies that converting an acyclic DFA into an equivalent Wheeler DFA cannot be done in less than quadratic time in the worst case.

The remaining open questions, therefore, are: which class of graphs admits polynomial-time prefix-sorting procedures? Can we efficiently build the minimum prefix-sortable finite-state automaton that accepts a given regular language? Note that, in order to answer the latter question, it makes sense to first focus on finite languages (which always admit a solution) and on DFAs, considering the hardness of the NFA-minimization [14, 20] and Wheeler NFAs recognition [12] problems (both hard also in the acyclic case). These questions are also of practical relevance: as shown in [29], acyclic DFAs recognizing pan-genomes (i.e. known variations in the reference genome of a population) can be turned into equivalent Wheeler DFAs of the same expected asymptotic size. While the authors do not find the minimum such automaton, their theoretical analysis (as well as experimental evaluation) suggests that the graph-indexing problem is tractable in real-case scenarios.

\(^4\) Usually, the lexicographic order is used to sort string suffixes. In this paper, we use the symmetric co-lexicographic order of the string's prefixes, and extend the concept to labeled graphs.
1.1 Our Contributions

Let $d$-NFA denote the class of NFAs with at most $d$ equally-labeled transitions leaving any state. In this paper we provide five main contributions, divided in sorting (1-3) and minimization (4-5):

1. We show that the problem of recognizing and sorting Wheeler $d$-NFAs is in P for $d \leq 2$. A recent result from Gibney and Thankachan [12] shows that the problem is NP-complete for $d \geq 5$. Our result almost completes the picture, the remaining open cases being $d = 3$ and $d = 4$.
2. We provide an online incremental algorithm that, when fed with an acyclic Wheeler DFA’s nodes in any topological order, can dynamically compute the co-lexicographic rank of each new incoming node among those already processed with just logarithmic delay.
3. We show that the problem of recognizing and sorting Wheeler DFAs can be solved in optimal linear time with an offline algorithm.
4. Given a Wheeler DFA $A$ of size $n$, we show how to compute, in $O(n \log n)$ time, the smallest Wheeler DFA recognizing the same language as $A$. If $A$ is acyclic, running time drops to $O(n)$.
5. Given any acyclic DFA $A$ of size $n$, we show how to compute, in $O(n + m \log m)$ time, the smallest Wheeler DFA $A'$, of size $m$, recognizing the same language as $A$.

The paper is structured in a top-down fashion to make it accessible also to non-experts. We start in Section 2 with result 1, a polynomial-time algorithm for recognizing and sorting Wheeler 2-NFAs. The algorithm is based on a reduction to 2-SAT; we expose it first since in our opinion it is useful to get an intuitive overview of the problem. Our algorithms 2-3 for prefix-sorting Wheeler DFAs (Sections 3 and 4) are described using intuitive high-level operations (e.g. manipulation of dynamic sequences and spanning tree computation) which do not require prior data-structures knowledge. All technical data-structures details showing how to implement those operations are given separately in Appendix F. These results generalize to labeled graphs existing prefix-sorting algorithms on strings [24] and labeled trees [8] that have been previously described in the literature. Contributions 4 and 5 (Section 5) combine our sorting algorithms 2-3 with DFA minimization techniques to solve the following problem: to compute, given a finite language represented either explicitly by a set of strings or implicitly by an acyclic DFA, the smallest accepting Wheeler DFA. Running time is optimal (linear) if the language is provided as a set of strings (using contribution 4), or a logarithmic multiplicative factor from the optimal if the input is an acyclic DFA (using contribution 5). Note that this can be interpreted as a technique to index arbitrary deterministic acyclic graphs. While we do not provide a lower bound stating that the space of our index is minimum, we note that all known fast indexes on strings, sets of strings, trees, and variable-order de Bruijn graphs are Wheeler graphs [10]. In this sense, any index on acyclic graphs improving our solution would probably require techniques radically different than those developed in the last decades to solve the indexing problem (virtually any known full-text index uses suffix/prefix sorting, including those based on LZ77 [19], run-length BWT [11], and grammars [2]). This section requires knowledge of basic automata-theory concepts such as the Myhill-Nerode equivalence and DFA minimization. Finally, in Appendix A we show how to index languages recognized by Wheeler finite automata using simple data structures.

1.2 Definitions

We start by giving a definition of finite-state automata that captures, to some extent, the amount of nondeterminism of the automaton. A $d$-NFA is a nondeterministic finite state automaton that
has at most \( d \) transitions with the same label leaving each state. Note that 1-NFAs correspond to DFAs, while \( \infty \)-NFAs correspond to general NFAs. Our definition is graph-theoretic: rather than defining a transition function, we treat automata as edge-labeled directed graphs with a source and a set of accepting states. This will be useful later when extending the notion of prefix sorting to labeled graphs.

**Definition 1.** A \( d \)-NFA is a quintuple \( A = (V, E, F, s, \Sigma) \), where \( V \) is a set of states (or vertices), \( \Sigma \) is the alphabet (or set of labels), \( E \subseteq V \times V \times \Sigma \) is a set of directed labeled edges, \( F \subseteq V \) is a set of accepting states, and \( s \in V \) is a start state (or source). Moreover require that \( s \in V \) is the only node with in-degree zero and that for each \( u \in V \) and \( a \in \Sigma \), \(|\{(u, v, a) \in E\}| \leq d\).

We denote with \( \sigma \) the cardinality of the alphabet: \( \sigma = |\Sigma| \). The notation \( L(A) \) indicates the language accepted by \( A \), i.e. the set of all strings labeling paths from \( s \) to an accepting state. We assume that each state either is \( s \) or is reachable from \( s \). Otherwise, any state that cannot be reached from \( s \) can be removed without changing \( L(A) \). Note that we allow states with incomplete transition function, i.e. such that the set of labels of their outgoing edges does not coincide with \( \Sigma \). If state \( s \) misses outgoing label \( a \), then any computation following label \( a \) from \( s \) is considered as non-accepting. In a standard NFA definition, this would be equivalent to having an outgoing edge labeled \( a \) to a universal non-accepting node (a sink). We call a \( d \)-NFA acyclic when the graph \((V, E)\) does not have cycles. We say that a \( d \)-NFA is input-consistent if, for every \( v \in V \), all incoming edges of \( v \) have the same label. If the \( d \)-NFA is input-consistent, we indicate with \( \lambda(v) \), \( v \in V \), the label of the incoming edges of \( v \). For the source, we take \( \lambda(s) = \# \notin \Sigma \), where \( \# \) is a character lexicographically smaller than all characters in \( \Sigma \). On DFAs, we denote by \( \text{succ}_u(v) \), with \( u \in V \) and \( a \in \Sigma \), the unique successor of \( v \) by label \( a \), when it exists. We define the size of an automaton to be the number of its edges. The notion of Wheeler graph generalizes in a natural way the concept of co-lexicographic sorting to labeled graphs:

**Definition 2 (Wheeler Graph).** A triple \( G = (V, E, \Sigma) \), where \( V \) is a set of vertices and \( E \subseteq V \times V \times \Sigma \) is a set of edges labeled from an alphabet \( \Sigma \), is called a Wheeler graph if there is a total ordering \( < \) on \( V \) such that vertices with in-degree 0 precede those with positive in-degree and for any two edges \((u_1, v_1, a_1), (u_2, v_2, a_2)\) we have

\[
\begin{align*}
(i) \quad & a_1 < a_2 \Rightarrow v_1 < v_2, \\
(ii) \quad & (a_1 = a_2) \land (u_1 < u_2) \Rightarrow v_1 \leq v_2,
\end{align*}
\]

where \( < \) denotes the ordering on \( \Sigma \).

Note that the above definition generalizes naturally the concept of prefix-sorting from strings to graphs: two nodes (resp. string prefixes) can be ordered either looking at their incoming labels (resp. last characters) or, if the labels are equal, by looking at their predecessors (resp. previous prefixes). Since that, differently from strings and trees, a graph’s node can have multiple predecessors, it should be clear that there could exist graphs with nodes that cannot be sorted due to conflicting predecessors: not all labeled graphs enjoy the Wheeler property. We call a total order of nodes satisfying Definition 2 a Wheeler order of the nodes and we write \( WDFA \) as a shortcut for Wheeler DFA. By property (i), input-consistency is a necessary condition for a graph to be Wheeler. We note that an input consistent graph is, essentially, a state-labeled graph: since each state has only one distinct incoming label, we can move the label from the incoming edges to the state itself. While we believe this interpretation is more intuitive, we preferred to stick to the original definition given
in [10] and consider edge-labeled graphs instead. An important property of Wheeler graphs is path coherence:

**Definition 3 (Path coherence [10]).** An edge-labeled directed graph \( G \) is path coherent if there is a total order of the nodes such that for any consecutive range \([i, j]\) of nodes and string \( \alpha \), the nodes reachable from those in \([i, j]\) in \(|\alpha|\) steps by following edges whose labels form \( \alpha \) when concatenated, themselves form a consecutive range.

A Wheeler graph is path coherent with respect to any Wheeler order of the nodes [10].

## 2 Recognizing Wheeler 2-NFAs is in P

We show a reduction from the problem of recognizing Wheeler 2-NFAs to 2-SAT. The reduction introduces only a polynomial number of boolean variables and can be computed in polynomial time; since 2-SAT is in P, this implies that Wheeler 2-NFA recognition is in P. In the next sections we discuss faster algorithms for DFAs. We decided to present first our reduction to 2-SAT since we believe it gives a clearer understanding of the problem. For space constraints, here we only give an overview of the proof. For the full formal proof, see Appendix C.

**Theorem 1.** Let \( A = (V, E, F, s, \Sigma) \) be a 2-NFA. In \( O(|E|^2) \) time we can:
1. Decide whether \( A \) is a Wheeler graph, and
2. If \( A \) is a Wheeler graph, return a node ordering satisfying the Wheeler graph definition.

**Proof (Sketch).** We first check in linear time if the graph is input-consistent. If it is, we provide a reduction of the recognition problem to 2-SAT. It is actually easy to express the Wheeler properties (i)-(ii) with 2-SAT clauses. We create a variable \( x_{u<v} \) for each pair of states. Then:

(a) For each \( u, v \), if \( \lambda(u) < \lambda(v) \) then we add the unary clause \( x_{u<v} \).
(b) For each \( u \neq v \), if \( \lambda(u) = \lambda(v) = a \), then for every pair \( u' \neq v' \) such that \((u', u, a) \in E \) and \((v', v, a) \in E \) we add the clause \( x_{u'v'} \rightarrow x_{uv} \).

Such a formula, however, could generate a node ordering that is not total. To make the ordering total, we need to express antisymmetry, completeness (connex), and transitivity. The first two properties are easily expressible using only two literals per clause:

1. **Antisymmetry.** For every pair \( u \neq v \), add the clause \( x_{u<v} \rightarrow \neg x_{v<u} \).
2. **Completeness.** For every pair \( u \neq v \), add the clause \( x_{u<v} \lor x_{v<u} \).

Transitivity, however, requires 3-SAT clauses on general graphs. The core of the full formal proof in Appendix C is to show that, on input-consistent 2-NFAs, transitivity automatically “propagates” from the source to all nodes and does not require additional clauses. We show this property by induction on the length of the shortest path connecting each node with the source \( s \).

Gibney and Thankachan [12] have recently shown that the problem of recognizing Wheeler \( d \)-NFAs is NP-complete for \( d \geq 5 \). Theorem 1 almost completes the picture, the remaining open cases being \( d = 3 \) and \( d = 4 \).

It is tempting trying to generalize the above solution to general NFAs by simulating arbitrary degree-\( d \) nondeterminism using binary trees: a node with \( d \) equally-labeled outgoing edges could be expanded to a binary tree with \( d \) leaves (brining down the degree of nondeterminism to 2). Unfortunately, while this solution works for transitivity (which is successfully propagated from the source), it could make the graph non-Wheeler: the topology of those trees cannot be arbitrary and must satisfy the co-lexicographic ordering of the nodes, i.e. the solution we are trying to compute.
3 Sorting Acyclic Wheeler DFAs Online

In this section we present an online algorithm that solves the problem considered in Theorem 1 in $O(|E| \log |V|)$ time when the graph is an acyclic DFA. The algorithm is online in the following sense. We assume that the nodes, together with their incoming labeled edges, are provided to the algorithm in any valid topological ordering: when a new node $v$ arrives together with its incoming labeled edges $(u_1, v, a), \ldots, (u_k, v, a)$, then $u_1, \ldots, u_k$ have already been seen in the past node sequence. At any step, we maintain a prefix-sorted list of the current nodes, which is updated when a new node is added. When a new incoming node falsifies the Wheeler property, we detect this event, report it, and stop the computation. In Section 5 we will modify this algorithm so that, instead of failing on non-Wheeler graphs, it computes the smallest Wheeler DFA equivalent to the input acyclic DFA.

We first show that, on Wheeler DFAs, there is a unique admissible ordering. For space constraints, here we only give a sketch of the proof, which is reported in Appendix D.

**Lemma 1.** Let $\mathcal{A}$ be a Wheeler DFA, $\prec$ be the node ordering satisfying the Wheeler property, and $\prec$ be the co-lexicographic order among strings. For any two nodes $u \neq v$, the following holds: $\alpha_u \prec \alpha_v$ for all string pairs $\alpha_u, \alpha_v$ labeling paths $s \sim u$ and $s \sim v$ if and only if $u \prec v$.

**Proof (Sketch).** Consider two strings $\alpha_u$ and $\alpha_v$ labeling two paths from the source to $u$ and $v$. Since the automaton is deterministic and $u \neq v$, it must be the case that $\alpha_u \neq \alpha_v$. At this point, the proof works by induction on the length of $\alpha_u$ and $\alpha_v$, using the Wheeler properties at the base case where $\alpha_u$ and $\alpha_v$ end with two distinct characters (or one of the two nodes is the source). □

Lemma 1 has two important consequences: on DFAs, (i) we can use any paths connecting $s$ with two nodes $u \neq v$ to decide their co-lexicographic order, and (ii) if it exists, the total ordering of the nodes is unique.

We now give an intuitive overview of our algorithm, which is reported in all details (pseudocode and data structures) in Appendices E and F. Figure 1 shows how our dynamic structures evolve while processing states in topological order. We keep three dynamic sequences: IN, LEX, and OUT. Sequence LEX stores, in co-lexicographic order, the states that have already been processed. Sequences IN and OUT store strings and are synchronized with LEX: let $IN[v], v \in LEX$, be the entry of IN corresponding to the position where $v$ is stored in LEX (similar for OUT). $IN[v]$ stores the labels of all $v$'s incoming edges. Similarly, $OUT[v]$ stores the labels of all $v$'s outgoing edges that lead to nodes in LEX. The key idea of our online step is the following. Suppose we are about to process node $v$. Let $u_1 < \ldots < u_k \in LEX$ be the sorted predecessors of $v$, and let $b = \lambda(v)$ be the label of $v$'s incoming edges. Note that (1) no nodes other than $u_1, \ldots, u_k$ between $u_1$ and $u_k$ in LEX can have outgoing edges labeled $b$, and (2) there cannot be nodes $w_1, w_2$, with $w_1 < w_1 < u_k < w_2$, such that $\text{succ}_b(w_1) = \text{succ}_b(w_2)$. This can be seen as follows. In case (1), let $u_1 < w < u_k$ be such that $\text{succ}_b(w) = v'$, for some $v' \neq v$. Then, by wheeler property (ii) nodes $v$ and $v'$ cannot be ordered. Case (2) is analogous, with $v' = \text{succ}_b(w_1) = \text{succ}_b(w_2)$. These events can be detected by opportunely marking (reserving) with alphabet characters (in this case, $b$) intervals in OUT. We call these two events inconsistencies of type 1 and 2, respectively. The left-hand and right-hand sides of Figure 1 show these events. If an inconsistency occurs, then the graph cannot be Wheeler and we stop. Otherwise, let $w$ be the immediate predecessor of $u_1$ (in co-lexicographic order) in LEX having outgoing label $b$, and let $\text{succ}_b(w) = z$. Then, Wheeler property (ii) implies that $v$ must be the immediate successor of $z$ in LEX (or the first node with incoming label $b$ if such an edge $(w, z, b)$ does not exist). After inserting $v$ in LEX, we also insert its incoming labels in $IN[v]$, insert
an empty string in $\text{OUT}[v]$ ($v$ does not have yet outgoing edges to nodes in $\text{LEX}$), and append $b$ to all strings $\text{OUT}[u_i]$, $i = 1, \ldots, k$. In Appendix F we show that all these operations can be implemented in logarithmic time using dynamic sequence data structures to represent $\text{IN}$, $\text{LEX}$, and $\text{OUT}$. Our algorithm is a generalization of a well-known algorithm for building online the Burrows-Wheeler transform of a text [24]. We obtain (for a full proof see Appendix G):

**Theorem 2.** Let $A = (V, E, F, s, \Sigma)$ be an acyclic DFA. Our algorithm either prefix-sorts the nodes of $A$ or returns $\text{FAIL}$ if such an ordering does not exist online with $O(\log |V|)$ delay per input edge.

![Fig. 1. Left: Inconsistency of type 1. The five tables show how arrays $\text{IN}$, $\text{LEX}$, and $\text{OUT}$ evolve during insertions of nodes $v_1, \ldots, v_5$ in topological order. Up to node $v_4$, the DAG is Wheeler. When inserting node $v_3$, we successfully reserve the interval $[v_1, v_2]$ with label 'a' (shown in red). From this point, no 'a's can be inserted inside the reserved interval. When inserting node $v_5$ (with incoming label 'b'), the co-lexicographically smallest and largest predecessors of $v_5$ are $v_3$ and $v_4$, respectively. This means we have to reserve the interval $[v_3, v_4]$ with label 'b' (shown in blue dashed line); however, this is not possible since there already is a 'b' (highlighted in blue) in $\text{OUT}[v_3, \ldots, v_4]$. Right: Inconsistency of type 2. Up to node $v_5$, the DAG is Wheeler. Note that we successfully reserve two intervals: $[v_1, v_2]$ (with letter 'a', red interval), and $[v_3, v_4]$ (with letter 'b', blue interval). When inserting node $v_4$ we do not need to reserve any interval since the node has only one predecessor. The conflict arises when inserting node $v_6$ (with incoming label 'b'). Since $v_3$ is a predecessor of $v_6$, we need to append 'b' in $\text{OUT}[v_6]$. However, this 'b' (underlined in the picture) falls inside a reserved interval for 'b' (in blue).

4 Sorting Wheeler DFAs in Linear Time

In this section, we show that in the offline setting we can improve upon the result of the previous section: Wheeler DFAs can be recognized and sorted (offline) in linear time. We first need to show that we can check the Wheeler property in linear time given a candidate total ordering. See Appendix H for the full proof.

**Lemma 2.** Given an input-consistent edge-labeled graph $G = (V, E, \Sigma)$ and a permutation of $V$ sorted by a total order $<$ on $V$, we can check whether $<$ satisfies the Wheeler properties in optimal $O(|V| + |E|)$ time.
Proof (Sketch). This can be achieved easily by radix-sorting edges \((u,v,a)\) represented as triples \((a,i_u,i_v)\), where \(i_u\) and \(i_v\) are the Wheeler ranks of the source and the destination, and checking the Wheeler properties with one scan of the sorted triples.

We now use Lemmas 1 and 2 to prove the following (see Appendix I for the full proof):

**Theorem 3.** Let \(A = (V,E,F,s,\Sigma)\) be a DFA. In \(O(|V| + |E|)\) time we can:

1. Decide whether \(A\) is a Wheeler graph, and
2. If \(A\) is a Wheeler graph, return a node ordering satisfying the Wheeler graph definition.

Proof (Sketch). By Lemma 1, if \(A\) is a Wheeler graph then we can use the strings labeling any two paths \(s \rightsquigarrow u\) and \(s \rightsquigarrow v\) to decide the order of any two nodes \(u\) and \(v\). We build a spanning tree of \(A\) rooted in \(s\) and prefix-sort it using [8, Thm 2]. Finally, we verify correctness using Lemma 2.  

We note that the above strategy cannot be used to sort Wheeler NFAs, since the spanning tree could connect \(s\) with several distinct nodes using the same labeled path; this would prevent us from finding the order of those nodes using the spanning tree as support.

## 5 Wheeler DFA Minimization

We are now ready to use the algorithms of the previous sections to prove our main results: (i) a minimization algorithm for Wheeler DFAs (Theorem 4) and (ii) a near-optimal algorithm generating the minimum Wheeler acyclic DFA equivalent to any given acyclic DFA (Theorem 6).

Let \(\equiv\) be an equivalence relation over the states \(V\) of an automaton \(A = (V,E,F,s,\Sigma)\). The quotient automaton is defined as \(A/\equiv = (V/\equiv,E/\equiv,F/\equiv,[s]_\equiv,\Sigma)\), where \(E/\equiv = \{([u]_\equiv,[v]_\equiv,c) : (u,v,c) \in E\}\). The symbol \(\approx\) denotes the Myhill-Nerode equivalence among states \([25]\): \(u \approx v\), with \(u,v \in V\), if and only if, for any string \(\alpha\), we reach a final state by following the path labeled \(\alpha\) from \(u\) if and only if the same holds for \(v\). The goal of any DFA-minimization algorithm is to find \(\equiv\), which is the, provably existing and unique, coarsest (i.e. largest classes) equivalence relation stable with respect to the initial partition in final/non-final states. To abbreviate, we will simply say “coarsest equivalence relation” instead of “coarsest equivalence relation stable with respect to an initial partition”.

In our case, assuming that \(A\) is Wheeler, we want to find the coarsest equivalence relation \(\equiv_w\) finer than \(\approx\), such that \(A/\equiv_w\) is Wheeler. Our Algorithm 1 achieves precisely this goal: we start with \(\approx\) and then refine it preserving stability with respect to characters, while also ensuring that the resulting equivalence classes can be ordered consistently with the Wheeler constraints.

We show (full proof in Appendix J):

**Theorem 4.** Let \(A\) be a WDFA. The automaton \(A/\equiv_w\) returned by Algorithm 1 is the minimum WDFA recognizing \(L(A)\).

Proof (Sketch). Consider the infinite automata tree \(T\) recognizing \(L(A)\) obtained by “unraveling” \(A\). The minimum WDFA accepting \(L(A)\) can be obtained by minimizing \(T\) without breaking the Wheeler properties. By the same arguments of Lemma 1, the co-lexicographic order of \(T\)’s nodes is unique. Moreover, by the Wheeler properties we can only collapse states of \(T\) that are adjacent in co-lexicographic order. The next step is to prove that all of \(T\)’s nodes that are collapsed in \(A\), must also be collapsed in the minimum WDFA. These observations imply that the minimum WDFA is obtained by collapsing maximal runs of \(\approx\)-equivalent states of \(A\) that are adjacent in co-lexicographic order and with the same incoming label. This is precisely what Algorithm 1 does.  

\(\square\)
Algorithm 1: WheelerMinimization($A$)

**input**: Wheeler DFA $A$

**output**: Minimum Wheeler DFA $A'$ such that $L(A) = L(A')$

1. Compute the Myhill-Nerode equivalence $\approx$ among states of $A$.
2. Prefix-sort $A$'s states, obtaining the ordering $v_1 < v_2 < \cdots < v_n$.
3. Compute a new relation $\equiv_w$ defined as follows. Insert in the same equivalence class all maximal runs $v_i < v_{i+1} < \cdots < v_{i+t}$ such that:
   (a) $v_i \approx v_{i+1} \approx \cdots \approx v_{i+t}$
   (b) $\lambda(v_i) = \lambda(v_{i+1}) = \cdots = \lambda(v_{i+t})$.
4. Return $A/\equiv_w$.

Note that uniqueness of the minimum WDFA follows from Lemma 1 (uniqueness of the Wheeler order) and Algorithm 1. Note also that, in the automaton output by Algorithm 1, adjacent states in co-lexicographic order are distinct by the relation $\approx$ unless their incoming labels are different (in which case they might be equivalent). It follows that if a sorted Wheeler DFA does not have this property, then it is not minimum (otherwise Algorithm 1 would collapse some of its states). Conversely, if a Wheeler DFA has this property, then Algorithm 1 does not collapse any state, i.e. the automaton is already of minimum size. We therefore obtain the following characterization:

**Theorem 5 (Minimum WDFA).** Let $A$ be a Wheeler DFA, let $v_1 < v_2 < \cdots < v_t$ be its co-lexicographically ordered states, and let $\approx$ be the Myhill-Nerode equivalence among them. $A$ is the minimum Wheeler DFA recognizing $L(A)$ if and only if the following holds: for every $1 \leq i < t$, if $v_i \approx v_{i+1}$ then $\lambda(v_i) \neq \lambda(v_{i+1})$.

Theorem 4 implies the following corollaries.

**Corollary 1.** Given a WDFA $A$ of size $n$, in $O(n \log n)$ time we can build the minimum WDFA recognizing $L(A)$.

**Proof.** We run Algorithm 1 computing $\approx$ with Hopcroft’s algorithm [15] ($O(n \log n)$ time), and prefix-sorting $A$ with Theorem 3 ($O(n)$ time). Note that we can check $u \approx v$ in constant time by representing the equivalence relation as a vector $EQ[v] = [v]_{\approx}$, where we choose $V = \{1, \ldots, |V|\}$ and where $[v]_{\approx}$ is any representative of the equivalence class of $v$ (e.g., the smallest one, which we can identify in linear time by radix-sorting equivalent states). Then, $u \approx v$ if and only if $EQ[u] = EQ[v]$. Using this structure, the runs of Algorithm 1 can easily be identified in linear time. \qed

**Corollary 2.** Given an acyclic WDFA $A$ of size $n$, in $O(n)$ time we can build the minimum acyclic WDFA recognizing $L(A)$.

**Proof.** We run Algorithm 1 computing $\approx$ with Revuz’s algorithm [28] ($O(n)$ time), prefix-sorting $A$ with Theorem 3 ($O(n)$ time), and testing $u \approx v$ in constant time as done in Corollary 1. \qed

Note that Corollary 2 implies that we can, in optimal linear time, build the minimum WDFA $A/\equiv_w$ recognizing any input finite language $L$ represented as a set of strings: we build the tree DFA accepting $L$ and apply Corollary 2. The corollary can be applied since trees are always Wheeler [8, 10]. In the next subsection we treat the (more interesting) case where $L$ is represented by a DFA. Note that this result could already be achieved by unraveling the DFA into a tree and minimizing it using Corollary 2. However, the intermediate tree could be exponentially larger than the output: this algorithm is not output-sensitive.
5.1 Acyclic DFAs to Smallest Equivalent WDFAs

We show how to build the smallest acyclic Wheeler DFA equivalent to any acyclic DFA in output-sensitive time. Let \( \mathcal{A} = (V, E, F, s, \Sigma) \) be an acyclic DFA. We first minimize \( \mathcal{A} \) using Revuz's algorithm \[28\] and obtain the equivalent minimum acyclic DFA \( \mathcal{A}_1 = \mathcal{A}/\approx = (V_1, E_1, F_1, s_1, \Sigma) \). Let us denote \( |V_1| = t \). The idea is to run a modified version of the online Algorithm 3 on \( \mathcal{A}_1 \). The difference is that now we will solve (not just detect) violations to the Wheeler properties without changing the accepting language.

The next step is to topologically-sort \( \mathcal{A}_1 \)'s states (e.g. using Kahn’s algorithm \[18\]). At this point, we modify \( \mathcal{A}_1 \) in \( t \) steps by processing its states in topological order. This defines a sequence of automata \( \mathcal{A}_1, \mathcal{A}_1, \ldots, \mathcal{A}_t \). At each step, the states of \( \mathcal{A}_i \) are partitioned in two sets:

− those not yet processed: \( N_i = \{ v_{i+1}, v_{i+2}, \ldots, v_t \} \), and
− the remaining states \( V_i - N_i \), sorted by a total ordering \( < \) in a sequence \( \text{LEX}_i \).

At the beginning, \( N_1 = \{ v_2, \ldots, v_t \} \) and \( \text{LEX}_1 = s \). Note that \( N_t = \emptyset \) (i.e. at the end we will have processed all states). At each step \( i \), we maintain the following invariants:

1. \( \mathcal{L}(\mathcal{A}_i) = \mathcal{L}(\mathcal{A}_1) \).
2. States in \( \text{LEX}_i \) are sorted by a total order \( < \) that does not violate the Wheeler properties among states in \( \text{LEX}_i \) itself: in Definition 2, we require \( u_1, u_2, v_1, v_2 \in \text{LEX}_i \).
3. for each \( j = 1, \ldots, |\text{LEX}_i| - 1 \), if \( \text{LEX}_i[j] \approx \text{LEX}_i[j + 1] \) then \( \lambda(\text{LEX}_i[j]) \neq \lambda(\text{LEX}_i[j + 1]) \).

Invariant 1 implies \( \mathcal{L}(\mathcal{A}_i) = \mathcal{L}(\mathcal{A}) \). Since \( N_t = \emptyset \) and \( \text{LEX}_i \) contains all \( \mathcal{A}_i \)'s states, invariant 2 implies that \( \mathcal{A}_t \) is Wheeler (note that intermediate automata \( \mathcal{A}_i \), with \( 1 < i < t \) might be non-Wheeler).

Finally, invariant 3 and Theorem 5 imply that \( \mathcal{A}_t \) is the minimum WDFA accepting \( \mathcal{L}(\mathcal{A}_t) \). As a result, \( \mathcal{A}_t = \mathcal{A}/\approx \). We describe all the details of our algorithm in Appendix K for space constraints; here we give an overview of the procedure. The idea is to process states in topological order as done in Section 3. This time, however, we also solve inconsistencies of type 1 and 2 among nodes in \( \text{LEX}_i \cup \{v_{i+1}\} \) by splitting nodes in \( \approx \)-equivalent copies. Here, splitting means creating two or more copies of a state \( v \) in such a way that (i) each copy duplicates all \( v \)'s outgoing edges, (ii) \( v \)'s incoming edges are distributed (not duplicated) among the copies, and (iii) each copy is a final state if and only if \( v \) is a final state. Figure 2 shows how the inconsistency-resolution process is carried out when processing a node \( v \) in order to insert it in \( \text{LEX}_i \). We may split either \( v \) (in the figure, \( v \) is split in \( v_1, v_2, v_3 \)), to solve inconsistencies of type 1, or some \( a \)-successor (in the figure, \( z_3 \)) of at least two nodes (in the figure, \( w_3, w_4 \)) that are separated in co-lexicographic order by some \( v \)'s predecessor (in the figure, \( w_3 < w_2 < w_3 < w_4 \)), to solve inconsistencies of type 2. We can make sure that the processed node \( v \) has all edges labeled with the same character by first splitting it into one node per distinct incoming label, and then processing separately each resulting node; let \( \{c_1, \ldots, c_k\} \) be the set of \( v \)'s incoming labels. We replace \( v \) with \( k \) states \( v_{c_1}, \ldots, v_{c_k} \) such that \( v_c \) has only the incoming edges labeled \( c \) of \( v \). We make each \( v_{c_1}, \ldots, v_{c_k} \) final if and only if \( v \) is final. We moreover replace each outgoing edge \( (v, u, c) \) of \( v \) with \( k \) edges \( (v_{c_1}, u, c), \ldots, (v_{c_k}, u, c) \). First, note that the splitting process creates \( \approx \)-equivalent nodes, therefore the accepted language never changes (invariant 1 stays true). Moreover, since the states of \( \mathcal{A} \) have already been collapsed by the equivalence \( \approx \), after inserting nodes (or their copies) in \( \text{LEX}_i \) we never create runs of length greater than one of \( \approx \)-equivalent states with equal incoming labels (invariant 3 stays true). As a result, we incrementally build the minimum WDFA \( \mathcal{A}/\approx \) recognizing \( \mathcal{L}(\mathcal{A}) \). Since our algorithm never deletes edges, the running time is bounded by the output’s size (which could nevertheless be much larger — or smaller — than \( \mathcal{A} \)). We obtain:
Fig. 2. Inconsistency resolution. Nodes are ordered left-to-right by the total ordering $<$ (except $v$ in the top part of the figure). Top: we are trying to insert $v$ in $\text{LEX}$, but this violates the Wheeler properties (edges’ destinations are not ordered as the sources, no matter where we insert $v$). Bottom: we solve the inconsistencies by splitting $v$ in three equivalent nodes $v_1 \approx v_2 \approx v_3$ and $z_3$ in two equivalent nodes $z'_3 \approx z''_3$. Note that (i) the splitting procedure induces naturally an ordering of the nodes that satisfies the Wheeler properties, and (ii) after splitting, no two adjacent states with the same incoming label are equivalent by $\approx$. By Theorem 5, this is the minimum way of splitting nodes. For simplicity, in the figure nodes $z_1, \ldots, z_4$ do not coincide with any node $w_1, \ldots, w_5$ or $u_1, \ldots, u_4$. This may not necessarily be the case. In our full proof in Appendix K we show that our procedure is correct even when this happens.

**Theorem 6.** Given an acyclic DFA $A$ of size $n$, we can build and prefix-sort the minimum acyclic WDFA, of size $m$, recognizing $\mathcal{L}(A)$ in $O(n + m \log m)$ time.

Theorem 6 solves the problem of indexing deterministic DAGs for linear-time pattern matching queries in nearly-optimal time with a solution of minimum size. Combined with the hardness result of Equi et al. [6] it also implies that, under the Orthogonal Vectors hypothesis, the minimum WDFA has size $\Omega(n^2 / \log n)$ in the worst case (even though we believe the blow-up could be exponential in the worst case). Another consequence of Theorem 6 is that we can also index languages recognized by acyclic NFAs: we use the powerset construction to build an equivalent DFA, and then convert it to the minimum acyclic WDFA using Theorem 6. In general however, the smallest DFA recognizing a language $\mathcal{L}$ might be exponentially larger than an NFA recognizing $\mathcal{L}$. In contrast, here we show that for any Wheeler NFA $A$, there exists an equivalent DFA of quadratic size. This means that, if $\mathcal{L}(A)$ admits a WDFA of polynomial size (in the size of $A$), then our indexing algorithm $\text{WNFA} \rightarrow \text{DFA} \rightarrow \text{WDFA}$ runs in polynomial time.

**Theorem 7.** For any Wheeler NFA with $n$ nodes, there exists an equivalent DFA with at most $n(n+1)/2$ nodes.

**Proof.** By path coherence (Definition 3), for any string $\alpha$ the set of states at the ends of paths labeled by $\alpha$ form a consecutive range in the Wheeler order. This means that if we use the standard powerset construction to determinize a WNFA, the reachable states of the DFA correspond to consecutive ranges of states of the WNFA. The reachable part of the DFA has a number of states that is at most equal to the number of possible consecutive ranges in $[1, n]$, i.e. $n(n+1)/2$. $\square$
A Indexing Wheeler Automata

We show that Wheeler automata can be indexed in order to support fast membership queries in their accepting language or in its substring/suffix closure. Let \( \mathcal{A} \) be any Wheeler NFA. We first remove all states that do not lead to a final state. This preserves the accepted language, the total ordering, and the Wheeler property. We then use our algorithms to prefix-sort the automaton in polynomial time (if the automaton is a 2-NFA), and build a (generalized) FM-index on the graph as described in [10]. We mark in a bitvector \( B[1..|V|] \) supporting constant-time rank queries [16] all accepting states of the Wheeler NFA in our array \( \text{LEX} \) containing the states in co-lexicographic order. To check membership of a word \( w \), we search the word \( \#w \) and get a range \( \text{LEX}[L,R] \) of all states reachable from the root by a path labeled \( w \). At this point, \( w \) is accepted if and only if \( B[L,R] \) contains at least one bit set (constant time using rank on \( B \)). Note that this procedure works in \( O(w \log \sigma) \) time also if the automaton is nondeterministic. If we search for \( w \) instead of \( \#w \), then we get the range of states reachable by a path labeled \( uw \), for any \( u \in \Sigma^* \). This range is non-empty if and only if \( w \) belongs to the substring closure of \( \mathcal{L}(\mathcal{A}) \). Finally, if we search a word \( w \) and get a range \( \text{LEX}[L,R] \), then \( w \) is in the suffix closure of \( \mathcal{L}(\mathcal{A}) \) if and only if \( B[L,R] \) contains at least one bit set.

B Conclusions and Future Extensions

In this paper, we have solved the problem of indexing finite languages with prefix-sortable DFAs of minimum size. Our work leaves several intriguing lines of research (some of which will be explored in a future extension of this work). First of all, is the problem of recognizing Wheeler languages (encoded, e.g. as regular expressions) decidable? We believe that the answer to this question is positive: the Wheelerness of a regular language seems to translate into particular constraints (that can be verified in bounded time) on the topology of its minimum accepting DFA. Once a regular language has been classified as Wheeler, can we build the minimum accepting Wheeler DFA? Also in this case, we believe that the task can be solved by iterating conflict-resolution (Section 5.1) from the minimum DFA until the process converges to the minimum Wheeler DFA. What about Wheeler NFAs? In this case, our techniques cannot be applied directly since they rely on automata minimization, and the problem of minimizing NFAs (even acyclic [14]) is notoriously hard. Moreover, the problem of sorting Wheeler NFAs is NP-complete [12]. However, we believe that two things can be done. Despite the WNFA sorting problem being NP-complete, we believe it is possible to prefix-sort in polynomial time an equivalent minimized (not necessarily minimum) version of the WNFA. Moreover, since Wheeler graphs can be viewed as node-labeled graphs (we noted this in Section 1.2), we believe that Paige and Tarjan’s algorithm for the single-relation coarsest partition problem [26] can be used to find a minimized (though not minimum) version of WNFA, using the nodes’ labels as initial partition for the algorithm.

C Proof of Theorem 1

Proof. We can assume, without loss of generality, that \( \mathcal{A} \) is input-consistent, since checking this property takes linear time. If \( \mathcal{A} \) is not input-consistent, then it is not Wheeler. We show a reduction of problem 1 to 2-SAT, which can be solved in linear time using Aspvall, Plass, and Tarjan’s (APT) algorithm based on strongly connected components computation. The reduction introduces \( O(|V|^2) \)
variables and $O(|E|^2)$ clauses, hence the final running time will be $O(|E|^2)$. Moreover, since a satisfying assignment to our boolean variables will be sufficient to define a total order of the nodes, APT will essentially solve also problem 2.

For every pair $u \neq v$ of nodes we introduce a variable $x_{u<v}$ which, if true, indicates that $u$ must precede $v$ in the ordering. We now describe a 2-SAT CNF formula whose clauses are divided in two types: clauses of the former type ensure that the Wheeler graph property is satisfied, while clauses of the second type ensure that the order of nodes induced by the variables is total.

The following formulas ensure that the Wheeler property is satisfied:

(a) For each $u, v$, if $\lambda(u) < \lambda(v)$ then we add the unary clause $x_{u<v}$.
(b) For each $u \neq v$, if $\lambda(u) = \lambda(v) = a$, then for every pair $u' \neq v'$ such that $(u', u, a) \in E$ and $(v', v, a) \in E$ we add the clause $x_{u'<v'} \rightarrow x_{u<v}$.

There are at most $|V|^2 \leq |E|^2$ clauses of type (a) and at most $|E|^2$ clauses of type (b).

The following formulas guarantee that the order is total. Note that we omit transitivity which, on a general graph, would require a 3-literals clause ($\lambda$, if $\lambda(u) \neq \lambda(v)$ forces a contradiction by a (1)-clause. In all cases (i.1)-(i.3) we have that $x_{u<v}$.

(1) **Antisymmetry.** For every pair $u \neq v$, add clause $x_{u<v} \rightarrow \neg x_{v<u}$.
(2) **Completeness.** For every pair $u \neq v$, add the clause $x_{u<v} \lor x_{v<u}$.

There are at most $O(|V|^2) = O(|E|^2)$ clauses of types (1) and (2).

We now show that the order is total. Note that we omit transitivity which, on a general graph, would require a 3-literals clause ($\lambda$, if $\lambda(u) \neq \lambda(v)$ forces a contradiction by a (1)-clause. In all cases (i.1)-(i.3) we have that $x_{u<v}$.

Consider a directed shortest-path tree $T$ with root $s$ of $A$. Since we assume that each state is reachable from $s$, $T$ must exist and must contain all nodes of $A$. Let $d_v$ be the length of a shortest directed path connecting $s$ to $v$. By definition of $T$, the path connecting $s$ to $v$ in $T$ has length $d_v$, with $d_s = 0$. We proceed by induction on $k = \max\{d_u, d_v, d_w\}$. The case $k = 0$ is trivial, since there are no triples of pairwise distinct nodes in $\{u : d_u \leq 0\}$ (this set contains just $s$). Take now a general $k > 0$. We consider two main cases:

(i) $|\{\lambda(u), \lambda(v), \lambda(w)\}| > 1$. Then, since $x_{u<v}$ and $x_{v<w}$, for some $a < b < c \in \Sigma$ either: (i.1) $\lambda(u) = a$, $\lambda(v) = b$, $\lambda(w) = c$, or (i.2) $\lambda(u) = a$, $\lambda(v) = a$, $\lambda(w) = b$, or (i.3) $\lambda(u) = a$, $\lambda(v) = b$, $\lambda(w) = b$. Any other choice would force one of the variables $x_{v<u}$, $x_{w<v}$ to be true (by an (a)-clause), forcing a contradiction by a (1)-clause. In all cases (i.1)-(i.3) we have that $\lambda(u) < \lambda(w)$, therefore $x_{u<w}$ must be true by (a).

(ii) $\lambda(u) = \lambda(v) = \lambda(w) = a$ for some $a \in \Sigma$. Assume, moreover, that $u, v, w$ are distinct by assumption. Let $u', v', w'$ be the parents of $u, v, w$, respectively, in $T$. Note that $u', v', w'$ cannot be the same vertex, since $u, v, w$ are distinct and every node has at most two outgoing edges with the same label. We therefore consider two sub-cases:

(ii.1) $\{|u', v', w'|\} = 2$. We first show that $u' = w' \neq v'$ generates a contradiction. Since $x_{u<v}$ and $x_{w<v}$ are true and $u' \neq v'$ and $v' \neq w'$ hold, $x_{u'<v'}$ and $x_{v'<w'}$ must be true: otherwise, by (b), would imply that $x_{v<u}$ and $x_{w<v}$ are true, which generates a contradiction. Now, $u' = w'$ means that $x_{v'<w'}$ and $x_{v'<w'}$ have the same truth value; since $x_{u'<v'}$ and $x_{v'<w'}$ cannot be both true, we have a contradiction. We are therefore left with the case $u' = v' \neq w'$ (or $v' \neq u'$, which is symmetric).
Remember that we assumed $x_{u<v}$ and $x_{u<w}$ are true. Hence, $x_{u' v}$ must be true; otherwise, by (b), the truth of $x_{w'<v'}$ would imply that $x_{w<v}$ is true, which generates a contradiction. Since $x_{v'u'<w'} = x_{w'<u'}$ is true, by (b) we conclude that also $x_{u<w}$ must be true.

(ii.2) $u', v', w'$ are pairwise distinct. We show that $x_{u'<v'}$ and $x_{v'<w'}$ must be true. Suppose, for contradiction, that $x_{u'<v'}$ is false (the proof is symmetric for $x_{v'<w'}$). Then, by (2), $x_{v'<u'}$ is true. But then, by (b) it must be the case that $x_{v<u}$ is true. Since we are assuming that $x_{u<v}$ is true, this introduces a contradiction by (1). Therefore, we conclude that $x_{u'<v'}$ and $x_{v'<w'}$ are true for the (pairwise distinct) parents $u', v', w'$ of $u, v, w$ in $T$. Now, by definition of the shortest-path tree $T$ it must be the case that $d_{u'} = d_u - 1$, $d_{v'} = d_v - 1$, and $d_{w'} = d_w - 1$ as $u', v', w'$ are the parents of $u, v, w$ in $T$. As a consequence, $\max\{d_{u'}, d_{v'}, d_{w'}\} = k - 1$. We can therefore apply the inductive hypothesis and conclude that $x_{u'<v'}$ is true. But then, by (b) we conclude that $x_{u<w}$ must also be true.

From the above proof correctness follows: if $A$ is an input-consistent 2-NFA and there exists a truth assignment satisfying the formula, then the assignment induces a total ordering of the nodes satisfying the Wheeler property. Conversely, the algorithm is clearly complete: if $A$ is a Wheeler 2-NFA, then there exists a total ordering of the nodes satisfying the Wheeler properties. This defines a truth assignment of the variables that satisfies our 2-SAT formula.

\[\square\]

D Proof of Lemma 1

Proof. ($\Rightarrow$) Assume $\alpha_u < \alpha_v$ for two paths connecting $s$ to $u$ and $s$ to $v$, respectively (with $u \neq v$). Since the paths start in $s$, we may assume that the first character in the two strings $\alpha_u$ and $\alpha_v$ is $\#$. We prove that $u < v$ by induction on $k = \min\{|\alpha_u|, |\alpha_v|\}$. The base case corresponds to $k = 1$: in this case, $u$ must coincide with the source $s$ and therefore $u < v$. Let now $k > 1$, and let $c_u$ and $c_v$ be the last character of $\alpha_u$ and $\alpha_v$, respectively. Since $\alpha_u < \alpha_v$, we can have two cases. Case (1): $c_u < c_v$. Then, $u < v$ by Wheeler property (i). Case (2): $c_u = c_v$. Then, let $u'$ and $v'$ be the predecessors of $u$ and $v$ on the two paths, respectively. Since $c_u = c_v$ and $\alpha_u < \alpha_v$, it must be the case that $\alpha_u[1..|\alpha_u| - 1] < \alpha_v[1..|\alpha_v| - 1]$. Then, we can apply the inductive hypothesis and obtain that $u' < v'$. But then, it must be the case that $u < v$ by Wheeler property (ii).

($\Leftarrow$) Assume $u < v$. We want to show that $\alpha_u < \alpha_v$ for all strings $\alpha_u$ and $\alpha_v$ labeling paths from $s$ to $u$ and from $s$ to $v$, respectively. Assume, for contradiction, that there exists two such paths such that $\alpha_u \preceq \alpha_v$. Since the DFA is deterministic and $u \neq v$, it cannot be the case that $\alpha_v = \alpha_u$ (a sequence of labels leads to a unique node from the source $s$). Then, by the above proof, we deduce that $v < u$. This contradicts the fact that $u < v$, being $<$ a total order of the nodes (required by the Wheeler property).

E Sorting WDFAs Online

Algorithm 2 initializes all variables used by our procedure and implements Kahn’s topological-sorting algorithm [18]. Every time a new node is appended to the topological ordering, we call Algorithm 3—our actual online algorithm—to update also the co-lexicographic ordering. This step also checks if the new node and its incoming edges falsify the Wheeler property. We use the following structures (indices start from 1):

- **LEX** is a dynamic sequence of distinct nodes $v_1, \ldots, v_k \in V$ supporting the following operations:
  1. $\text{LEX}[i]$ returns $v_i$.  

13
2. \(\text{LEX}^{-1}[v]\), with \(v \in \text{LEX}\), returns the index \(i\) such that \(\text{LEX}[i] = v\).
3. \(\text{LEX}.\text{insert}(v, i)\) inserts node \(v\) between \(\text{LEX}[i-1]\) and \(\text{LEX}[i]\). If \(i = 1\), \(v\) is inserted at the beginning of the sequence. This operation increases the sequence’s length by one.

- \(\text{IN}\) and \(\text{OUT}\) are dynamic sequences of strings, i.e. sequences \(\alpha_1, \ldots, \alpha_k\), where \(\alpha_i \in \Sigma^*\) (note that \(\alpha_i\) could be the empty string \(\epsilon\)). To make our pseudocode more readable, we index \(\text{IN}\) and \(\text{OUT}\) by nodes of \(\text{LEX}\) (these three arrays will be synchronized). Let \(T = \alpha_1, \alpha_2, \ldots, \alpha_k\), with \(T \in \{\text{IN}, \text{OUT}\}\). Both arrays support the following operation:
4. \(\text{T}\.\text{insert}(\alpha, v)\), where \(\alpha \in \Sigma^*\) and \(v \in \text{LEX}\): insert \(\alpha\) between \(\alpha_{\text{LEX}^{-1}[v] - 1}\) and \(\alpha_{\text{LEX}^{-1}[v]}\). If \(\text{LEX}^{-1}[v] = 1\), then \(\alpha\) is inserted at the beginning of \(T\). This operation increases the sequence’s length by one.

Sequence \(\text{OUT}\) supports these additional operations:
5. \(\text{OUT}[v]\), with \(v \in \text{LEX}\), returns \(\alpha_{\text{LEX}^{-1}[v]}\).
6. \(\text{OUT}.\text{append}(\alpha, v)\), where \(\alpha \in \Sigma^*\) and \(v \in \text{LEX}\): append the string \(\alpha\) at the end of the string \(\text{OUT}[v]\), i.e. replace \(\text{OUT}[v] \leftarrow \text{OUT}[v] \cdot \alpha\). Note that this operation does not increase \(\text{OUT}\)’s length.
7. \(\text{OUT}.\text{rank}(c, u)\), with \(u \in \text{LEX}\) and \(c \in \Sigma\): return the number of characters equal to \(c\) in all strings \(\text{OUT}[v]\), with \(v = \text{LEX}[1], \text{LEX}[2], \ldots, \text{LEX}[\text{LEX}^{-1}[u]]\).
8. \(\text{OUT}.\text{reserve}(u, v, c)\), with \(u, v \in \text{LEX}\) and \(c \in \Sigma\): from the moment this operation is called, the sequence \(\alpha_{\text{LEX}^{-1}[u]}, \ldots, \alpha_{\text{LEX}^{-1}[v]}\) is marked with label \(c\). Note that inserting new elements inside \(\alpha_{\text{LEX}^{-1}[u]}, \ldots, \alpha_{\text{LEX}^{-1}[v]}\) will increase the length of the reserved sequence.
9. \(\text{OUT}.\text{is}\_\text{reserved}(v, c)\), with \(v \in \text{LEX}\) and \(c \in \Sigma\): return \(\text{TRUE}\) iff \(\alpha_{\text{LEX}^{-1}[v]}\) falls inside a sequence that has been marked (reserved) with character \(c\).

In our algorithm, sequence \(\text{IN}\) will always be partitioned in at most \(t \leq \sigma + 1\) sub-sequences \(\text{IN} = \alpha_1^{c_1}, \ldots, \alpha_{k_1}^{c_1}, \alpha_1^{c_2}, \ldots, \alpha_{k_2}^{c_2}, \ldots, \alpha_1^{c_t}, \ldots, \alpha_{k_t}^{c_t}\), where each \(\alpha_i^{c_i}\) contains only character \(c\) and \(c_1 < c_2 < \cdots < c_t\). We define an additional operation on \(\text{IN}\):

10. \(\text{IN}.\text{start}(c)\), with \(c \in \Sigma\), returns the largest integer \(j \geq 1\) such that all characters in \(\text{IN}[v]\) are strictly smaller than \(c\), for all \(v = \text{LEX}[1], \ldots, \text{LEX}[j-1]\).

In Appendix F we discuss data structures implementing the above operations in \(O(\log k)\) time, \(k\) being the sequence’s length. Intuitively, these three dynamic sequences have the following meaning: \(\text{LEX}\) will contain the co-lexicographically-ordered sequence of nodes. \(\text{IN}[v]\) and \(\text{OUT}[v]\), with \(v \in \text{LEX}\), will contain the labels of the incoming and outgoing edges of \(v\), respectively. To keep the three sequences synchronized, when inserting \(v\) in \(\text{LEX}\) we will also need to update the other two sequences so that \(\text{IN}[v] = c_i\), where \(t\) is the number of incoming edges, labeled \(c\), of \(v\), and \(\text{OUT}[v] = \epsilon\), since \(v\) does not have yet outgoing edges. \(\text{OUT}[v]\) will (possibly) be updated later, when new nodes adjacent to \(v\) will arrive in the topological order. Our representation is equivalent to that used in [29] to represent the GCSA data structure. Intuitively, \(\text{OUT}\) is a generalized version of the well-known Burrows-Wheeler transform (except that we sort prefixes in co-lexicographic order instead of suffixes in lexicographic order). If the graph is a path (i.e. a string) then \(\text{OUT}\) is precisely the BWT of the reversed path.

We proceed with a discussion of the pseudocode. In Lines 2-12 of Algorithm 2 we initialize all variables and data structures. Let \(u \in V\). The variable \(u\.\text{in}\) memorizes the number of incoming
edges in \( u \); we will use this counter to implement Kahn’s topological sorting procedure. \( \texttt{u.label} \) is the label of all incoming edges of \( u \), or \# if \( u = s \). \( \texttt{IN}, \texttt{LEX}, \) and \( \texttt{OUT} \) are initialized as empty dynamic sequences. Lines 13-22 implement Kahn’s topological sorting algorithm [18]. Each time a new node \( u \) is appended to the order, we call our online procedure \texttt{update(u)}, implemented in Algorithm 3. Algorithm 3 works as follows. Assume that we have already sorted \( v_1, \ldots, v_k \), that \( \texttt{LEX} \) contains the nodes’ permutation reflecting their co-lexicographic order, and that \( \texttt{IN}[v_1] \) and \( \texttt{OUT}[v_1] \) contain the incoming and outgoing labels for each \( i = 1, \ldots, k \) in the sub-graph induced by \( v_1, \ldots, v_k \). When a new node \( u \) arrives in topological order, all its \( t \) predecessors are in \( \texttt{LEX} \). Let \( b = \texttt{u.label} \) be the incoming label of \( u \). We find the co-lexicographically smallest \( v_{\text{min}} \) and largest \( v_{\text{max}} \) predecessors of \( u \) (using function \( \texttt{LEX}^{-1} \) on all \( u \)’s predecessors). In our pseudocode, if \( u = s \) then \( v_{\text{min}} = v_{\text{max}} = \texttt{NULL} \). To keep the Wheeler property true, note that there cannot be \( b \)’s in the range \( \texttt{OUT}[v_{\text{min}}..v_{\text{max}}] \); if there are, since we will append \( b \) to \( \texttt{OUT}[v_{\text{min}}] \) and \( \texttt{OUT}[v_{\text{max}}] \), there will be three nodes \( v_{\text{min}} < v' < v_{\text{max}} \) such that \((v_{\text{min}}, u, b), (v', u', b), (v_{\text{max}}, u, b) \in E \) for some \( u' \). Then, by Wheeler property (ii), this would imply that \( u < u' < u \), a contradiction. We therefore check this event using function \texttt{contains} (note: this function can be easily implemented using two calls to \texttt{rank}). If \( b \)’s are present, then the graph is no longer Wheeler: such an event is shown in Figure 1, left-hand side (where \( u = v_5 \)). Otherwise, the number \( j \) of \( b \)’s before \( v_{\text{min}} \) (which is equal to the number of \( b \)’s before \( v_{\text{max}} \)) tells us the co-lexicographic rank \( i \) of \( u \) (similarly to the standard string-BWT, we obtain this number by adding \( j \) to the starting position of \( b \)’s in \( \texttt{IN} \)), and we can mark (reserve) range \( \texttt{OUT}[v_{\text{min}}..v_{\text{max}}] \) with letter \( b \) using function \texttt{reserve}. Such an event is shown in Figure 1, left-hand side, when inserting, e.g., node \( v_3 \). At this point, we may have an additional inconsistency falsifying the Wheeler property in the case that one of the predecessors \( v_i \) of \( u \) falls inside a reserved range for \( b \) (reserved by a node other than \( u \)): this happens, for example, when inserting \( v_6 \) in Figure 1, right-hand side. This check requires calling function \texttt{isReserved}. If all tests succeed, we insert \( u \) in position \( i \) of \( \texttt{LEX} \) and we update \( \texttt{IN} \) and \( \texttt{OUT} \) by inserting \( b' \) at the \( i \)-th position in \( \texttt{IN} \) (i.e. the position corresponding to \( u \)) and by appending \( b \) at the end of each \( \texttt{OUT}[v_i] \) for each predecessor \( v_i \) of \( u \).

F  Data Structure Details

In this section we show how to implement operations \texttt{1-10} used by Algorithms 2 and 3 using state-of-the-art data structures. At the core of \( \texttt{LEX} \) and \( \texttt{OUT} \) stands the dynamic sequence representation of Navarro and Nekrich [24]. This structure supports insertions, access, rank, and select in \( O(\log n) \) worst-case time, \( n \) being the sequence’s length. The space usage is bounded by \( nH_0 + o(n \log \sigma) + O(\sigma \log n) \) bits, where \( H_0 \) is the zero-th order entropy of the sequence. Sequence \( \texttt{IN} \) will instead be represented using a dynamic partial sum data structure, e.g. a balanced binary tree or a Fenwick tree [7], and a dynamic bitvector. All details follow.

Sequence \( \texttt{LEX} \) is stored with Navarro and Nekrich’s dynamic sequence representation [24]. Operations \texttt{1-3} are directly supported on the representation. Operation \texttt{2} is simply a \( \texttt{select}_v(1) \) (i.e. the position of the first \( v \)).

Sequence \( \texttt{OUT} \) is stored using a dynamic sequence \( \texttt{OUT} \) and a bitvector, both represented with Navarro and Nekrich’s dynamic sequence. The idea is to store all the strings \( \texttt{OUT}[1], \ldots, \texttt{OUT}[\texttt{OUT}] \) concatenated in a single sequence \( \texttt{OUT} \), and mark the beginning of those strings with a bit set in a dynamic bitvector \( \texttt{B}_{\texttt{OUT}}[1..n] \), were \( n = |\texttt{OUT}| \). Clearly, operations \texttt{4-7} on \( \texttt{OUT} \) can be simulated with a constant number of operations (\texttt{insert, access, rank, select}) on \( \texttt{OUT} \) and \( \texttt{B}_{\texttt{OUT}} \).
Algorithm 2: sort(G)

input : Labeled DAG G = (V, E, s, Σ)
output: A permutation of V reflecting the co-lexicographic ordering of the nodes, or FAIL if such an ordering does not exist.

for u ∈ V do
    u.in ← 0;
    u.label ← NULL;

s.label ← #;

for (u, v, a) ∈ E do
    v.in ← v.in + 1;
    if v.label ≠ NULL and v.label ≠ a then
        return FAIL; /* Cannot be Wheeler graph */
    v.label ← a;

IN ← new_dyn_sequence(Σ'); /* Sequence of strings */
LEX ← new_dyn_sequence(V); /* Sequence of nodes */
OUT ← new_dyn_sequence(Σ'); /* Sequence of strings */
S ← {s}; /* Set of nodes with no incoming edges */

while S ≠ ∅ do
    u ← S.pop(); /* Extract any u ∈ S */
    update(u); /* Call to Algorithm 3. If this fails, return FAIL. */
    for (u, v, a) ∈ E do
        v.in ← v.in − 1;
        if v.in = 0 then
            S ← S ∪ {v};

if ∃ v ∈ V : v.in > 0 then
    return FAIL; /* cycle found! */

return LEX;

Operations 8-9 require an additional dynamic sequence of parentheses PAR[1..n] on alphabet {(c : c ∈ Σ) ∪ {□} : c ∈ Σ} ∪ {□}. Every time a new character is inserted at position i in out, we also insert □ at position i in PAR. When OUT.reserve(u, v, c) is called (i.e. operation 8), let i_u and i_v be the positions in out corresponding to the two occurrences of character c in OUT[u] and OUT[v], respectively (remember that the automaton is deterministic, so these positions are unique). These positions can easily be computed in O(log n) time using select and rank operations on out and B_out. Then, we replace PAR[i_u] and PAR[i_v] with characters (c and ), respectively (replacing a character requires a deletion followed by an insertion). Note that reserved intervals for a fixed character do not overlap, so this parentheses representation permits to unambiguously reconstruct the structure of the intervals. At this point, operation 9 is implemented as follows. Let i_v be the position in out corresponding to the first character in OUT[v]. This position can be computed in O(log n) time with two select operations on LEX and B_out. Then, OUT.is_reserved(v, c) returns true if and only if PAR.rank_c(i_v) > PAR.rank_c(i_v), i.e. if we did not close all opening parentheses
Algorithm 3: update(u)

\begin{algorithm}
\begin{algorithmic}
  \State \textbf{input} : Node $u$
  \State \textbf{behavior:} Inserts $u$ at the right place in the co-lexicographic ordering LEX of the nodes already processed, or returns FAIL if a conflict is detected.

  \begin{align*}
    &v_{\text{min}} \leftarrow \text{min_pred}(u); \quad \text{/* co-lexicographically-smallest predecessor */} \\
    &v_{\text{max}} \leftarrow \text{max_pred}(u); \quad \text{/* co-lexicographically-largest predecessor */} \\
    &\text{if } u \neq s \text{ then} \\
    &\quad \text{if OUT}[v_{\text{min}}, \ldots, v_{\text{max}}].\text{contains}(u.\text{label}) \text{ then} \\
    &\quad \quad \text{return FAIL; } \quad \text{/* Inconsistency of type 1 */} \\
    &\quad \text{else} \\
    &\quad \quad \text{for } (v, u, a) \in E \text{ do} \\
    &\quad \quad \quad \text{if OUT.is_reserved}(v.\text{label}) \text{ then} \\
    &\quad \quad \quad \quad \text{return FAIL; } \quad \text{/* Inconsistency of type 2 */} \\
    &\quad \quad \text{else} \\
    &\quad \quad \quad \quad OUT[v].\text{append}(u.\text{label}); \\
    &\quad \quad \text{OUT.reserve}(v_{\text{min}}, v_{\text{max}}, u.\text{label}); \quad \text{/* Reserve } [v_{\text{min}}, v_{\text{max}}] \text{ with } u.\text{label } */ \\
    &\quad i \leftarrow \text{IN.start}(u.\text{label}) + \text{OUT.rank}(u.\text{label}, v_{\text{min}}); \\
    &\quad \text{LEX.insert}(u, i); \\
    &\quad p \leftarrow |\text{pred}(u)|; \quad \text{/* Number of predecessors of } u \text{ */} \\
    &\text{else} \\
    &\quad \text{LEX.insert}(u, 1); \\
    &\quad p \leftarrow 1; \quad \text{/* Number of predecessors of } u \text{ */} \\
    &\quad \text{IN.insert}(u.\text{label}^p, u); \quad \text{/* Insert } p \text{ times } u.\text{label } */ \\
    &\quad \text{OUT.insert}(c, u); \quad \text{/* } u \text{ does not have successors yet } */
  \end{align*}
\end{algorithmic}
\end{algorithm}

($c$ before position $i_v$ (note that does not make any difference if $i_v$ is the first or last position in OUT[v], since when we call this operation OUT[v] does not contain characters equal to $c$).

To conclude, IN is represented with a dynamic bitvector $B_{IN}[1..n]$ and a partial sum PS[1..σ + 1] supporting the following operations in $O(\log \sigma)$ time:

- **partial sum**: $\text{PS.ps}(i) = \sum_{j=1}^{i} \text{PS}[j]$.
- **update**: $\text{PS}[i] \leftarrow \text{PS}[i] + \delta$.

Fenwick trees [7] support the above operations within this time bound. Bitvector $B_{IN}[1..n]$ contains the bit sequence $110^{t_{v_2}-1}10^{t_{v_3}-1}\ldots10^{t_{v_k}-1}$, where $t_{v_i}$ is the number of predecessors of $v_i$ in the current sequence LEX = $v_1, \ldots, v_k$ of sorted nodes (note that $v_1$ is always the source $s$). Assume, for simplicity, that $\Sigma = [1, \sigma + 1]$, where $\#$ corresponds to 1 (this is not restrictive, as we can map the alphabet to this range at the beginning of the computation). At the beginning, the partial sum is initialized as $\text{IN}[c] = 0$ for all $c$. Operation 10, IN.start(c), with $c \neq \#$, is implemented as $\text{PS.ps}(c - 1) + 1$. If $c = \#$, the operation returns 1. Operation 4, IN.insert(c^p, u), is implemented as $\text{PS}[c] \leftarrow \text{PS}[c] + p$ followed by $B_{IN}.\text{insert}(10^{p-1}, i_u)$ (i.e. $p$ calls to insert on the dynamic bitvector at position $i_u$), where $i_u$ is the position of the $j$-th bit set in $B_{IN}$ (a select operation) and $j = \text{LEX}^{-1}[u]$, or $i_u = n + 1$ if $B_{IN}$ has $j - 1$ bits set (note that, when we call IN.insert(c^p, u), node $u$ has already been inserted in LEX).
G Proof of Theorem 2

In Appendix F we show that all operations on the dynamic sequences can be implemented in logarithmic time. Correctness follows from the fact that we always check that the Wheeler properties are maintained true. To prove completeness, note that at each step we place \( u \) between two nodes \( v_1 \) and \( v_2 \) in array \( \text{LEX} \) only if the smallest \( u \)'s predecessor is larger than the largest \( v_1 \)'s predecessor, and if the largest \( u \)'s predecessor is smaller than the smallest \( v_2 \)'s predecessor. This is the only possible choice we can make in order to satisfy \( w_{v_1} < w_u < w_{v_2} \) for all strings labeling paths \( s \leadsto v_1, s \leadsto u, \) and \( s \leadsto v_2 \) and to obtain, by Lemma 1, the only possible correct ordering of the nodes. It follows that, if the new node \( v \) does not falsify the Wheeler property, then we are computing its co-lexicographic rank correctly.

H Proof of Lemma 2

Proof. First, we sort edges by label, with ties broken by origin, and further ties broken by destination. This can be achieved in time \( O(|E| + |V|) \) by radix sorting the edges represented as triples \((a, u, v)\), where \( a \) is the label, and \( u \) and \( v \) respectively are the ranks of the source and destination nodes in the given order \(<\).

Let \( L \) denote the sorted list of edges. We claim that the given order \(<\) satisfies the Wheeler properties (Definition 2) if and only if for all pairs of consecutive edges \((a_i, u_i, v_i), (a_{i+1}, u_{i+1}, v_{i+1})\) in \( L \), we have \((a_i = a_{i+1}) \rightarrow v_i \leq v_{i+1} \) and \((a_i \neq a_{i+1}) \rightarrow v_i < v_{i+1} \). Clearly this can be checked in time \( O(|E|) \) with one scan over \( L \). We now argue the correctness of this algorithm.

Wheeler property (ii) is equivalent to the condition that when all edges labeled by some character \( a \in \Sigma \) are sorted by source with ties broken by destination, the sequence of destinations is monotonically increasing, which is expressed by the condition \((a_i = a_{i+1}) \rightarrow v_i \leq v_{i+1} \).

Wheeler property (i) is equivalent to the condition that for all pairs of characters \( a, b \in \Sigma \) such that \( b \) is a successor of \( a \) in the order of \( \Sigma \), denoting by \( v_a \) the largest node with an incoming \( a \)-edge, and by \( v_b \) the smallest node with an incoming \( b \)-edge, we have \( v_a < v_b \). If Wheeler property (ii) holds, then destinations \( v_a \) and \( v_b \) are consecutive in \( L \) because the list is sorted primarily by label and destinations are monotonically increasing for each label. Hence checking for \((a_i \neq a_{i+1}) \rightarrow v_i < v_{i+1} \) verifies Wheeler property (i) given that Wheeler property (ii) holds. \( \square \)

I Proof of Theorem 3

Proof. In \( O(|V| + |E|) \) time we build a directed spanning tree \( \mathcal{T} \) of \( \mathcal{A} \) with root \( s \) (e.g. its directed shortest-path tree with root \( s \)). Note that this is always possible since we assume that all states are reachable from \( s \).

By Lemma 1, if \( \mathcal{A} \) is a Wheeler graph then we can use the strings labeling any two paths \( s \leadsto u \) and \( s \leadsto v \) to decide the order of any two nodes \( u \) and \( v \). We can therefore sort \( V \) according to the paths spelled by \( \mathcal{T} \); by Lemma 1, if \( \mathcal{A} \) is Wheeler then we obtain the correct (unique) ordering. To prefix-sort \( \mathcal{T} \), we compute its XBW transform \(^5\) in \( O(|V|) \) time [8, Thm 2]. The array containing the lexicographically-sorted nodes (i.e. the prefix array of \( \mathcal{T} \)) can easily be obtained from the XBW transform.

\(^5\) Note: this requires mapping the labels of \( \mathcal{T} \) to alphabet \( \Sigma' \subseteq [1, |V|] \) while preserving their lexicographic ordering. Since we assume that the original alphabet’s size does not exceed \(|E|^{O(1)} = |V|^{O(1)}\), this step can be performed in linear time by radix-sorting the labels.
transform using, e.g., the partial rank counters defined in the proof of Lemma 2 to navigate the tree (this is analogous to repeatedly applying function LF on the BWT in order to obtain the suffix array). At this point, we check that the resulting node order satisfies the Wheeler properties using Lemma 2. If this is the case, then the above-computed prefix array contains the prefix-sorted nodes of \( A \).

\( \square \)

### J Proof of Theorem 4

**Proof.** Let \( A = (\mathcal{V}, \mathcal{E}, \mathcal{F}, s, \Sigma) \). Consider the (possibly infinite) DFA \( \mathcal{T} \) that is a tree and that is equivalent to \( A \) in the following sense: \( \mathcal{T} \) is the (unique) tree obtained by “unraveling” \( A \), i.e., the tree containing all words in \( \mathcal{L}(A) \) such that each path labeled with such a word leads to an accepting state. Clearly, \( \mathcal{T} \) is a (possibly infinite) deterministic automaton recognizing \( \mathcal{L}(A) \): a string \( \alpha \) leads to a final state in \( A \) if and only if it does in \( \mathcal{T} \).

Let \( L^u = \{u^1, u^2, \ldots, u^k\} \) be the (possibly infinite) set of nodes of \( \mathcal{T} \) reached by following, from its root, all the paths labeled \( \alpha \) for each \( \alpha \) labeling a path \( s \prec u \) connecting \( s \) with \( u \) in \( A \). Note that each state \( u \) of \( A \) can be identified by the set \( L^u \) of states of \( \mathcal{T} \); this allows us to extend \( \equiv_w \) to the states of \( \mathcal{T} \) as follows: \( u^i \equiv_w u^j \) for all \( u^i, u^j \in L^u \), \( u \in \mathcal{V} \), and \( u^i \equiv_w v^j \) for \( u^i \in L^u \), \( v^j \in L^v \) if and only if \( u \equiv_w v \).

Consider now the process of minimizing \( \mathcal{T} \) by collapsing states in equivalence classes in such a way that (i) the quotient automaton is finite, (ii) the accepting language of the quotient DFA is the same as that of \( \mathcal{T} \) and (iii) the quotient DFA is Wheeler. By the existence of \( A \), there exists such a partition (not necessarily the coarsest): the one putting \( u^i \) and \( v^j \) in the same equivalence class if and only if \( u^i, v^j \in L^u \), for some \( u \in \mathcal{V} \) (in this case, \( A \) itself is the resulting quotient automaton). Call \( \equiv \) the relation among states of \( \mathcal{T} \) yielding the *smallest* such WDFA \( A/\equiv \). By definition, \( A/\equiv \) is the smallest WDFA recognizing \( \mathcal{L}(A) \). Our claim is that \( \equiv \equiv_w \), i.e., that Algorithm 1 returns this automaton.

We observe that:

1. \( u^i \approx u^j \) for any \( u^i, u^j \in L^u \) and all \( u \in \mathcal{V} \). Otherwise, assume for a contradiction that there exists a string \( \alpha \) leading to an accepting state from \( u^i \) but not from \( u^j \). By construction of \( \mathcal{T} \), \( u^i \) and \( u^j \) are \( \approx \)-equivalent to \( u \); this leads to a contradiction, since the state reached from \( u \) with label \( \alpha \) cannot be both accepting and not accepting.

2. Since \( A \) is a Wheeler DFA, Lemma 1 applied to \( A \) tells us that, for any two nodes \( u < v \in \mathcal{V} \), all strings labeling paths from the root of \( \mathcal{T} \) to nodes in \( L^u \) are co-lexicographically smaller than those labeling paths from the root of \( \mathcal{T} \) to nodes in \( L^v \). We express this fact using the notation \( L^u < L^v \).

3. Since \( A \) is Wheeler, then each \( u \in \mathcal{V} \) has only one distinct incoming label and \( \lambda(u^j) = \lambda(u) \) for all \( u^j \in L^u \).

By the above properties, \( u^i \equiv u^j \) for all \( u^i, u^j \in L^u \), \( u \in \mathcal{V} \). To see this, note that, by property 1, those states are all equivalent by relation \( \equiv \). Moreover, properties 2-3 combined with Lemma 1 imply that, by grouping states in each \( L^u \), we cannot break any Wheeler property. It follows that \( \equiv \equiv_w \) must group those states, being the coarsest partition finer than \( \approx \) with these two properties. Let us indicate with \( L^u \equiv L^v \) the fact that \( u^i \equiv v^j \) for all \( u^i \in L^u \), \( v^j \in L^v \).

Suppose now, for a contradiction, that there exist \( L^u < L^v < L^w \) with \( L^u \equiv L^w \neq L^v \). Then, by Lemma 1, \( L^u < L^v \) implies that, in the quotient automaton, states \( [L^u]_\equiv \) and \( [L^v]_\equiv \)
\[ [L^v]_\equiv \text{ are reachable from the source by two paths } \alpha \text{ and } \beta, \text{ respectively, with } \alpha < \beta. \text{ Conversely, } L^v < L^w \text{ implies that states } [L^v]_\equiv \text{ and } [L^w]_\equiv \text{ are reachable from the source by two paths } \alpha' \text{ and } \beta', \text{ respectively, with } \alpha' < \beta'. \text{ Then, by Lemma 1 we cannot define a total order on } A/\equiv \text{’s states, i.e. } A/\equiv \text{ is not Wheeler. } \]

By all the above observations, we conclude that \( \equiv \text{ must (i) group only equivalent states by } \approx, \text{ (ii) group only states with the same incoming label, (iii) group all states inside each } L^v, \text{ and (iv) group only states in adjacent sets } L^u, L^v \text{ in the co-lexicographic order. By its definition, the relation } \equiv_w \text{ induces the coarsest partition that satisfies (i)-(iv), therefore we conclude that } \equiv = \equiv_w. \]

**K Converting DFAs to minimum WDFAs**

We describe of an online step of our algorithm. Assume we successfully built \( A_i \), with \( i < t \), and we are about to process \( v_{i+1} \) in order to build \( A_{i+1} \). Let \( \{c_1, \ldots, c_k\} \) be the labels of incoming \( v_{i+1} \)'s edges. We first split (split) \( v_{i+1} \) by \( k \) equivalent states \( v_{i+1}^{c_1} \approx \cdots \approx v_{i+1}^{c_k} \): each \( v_{i+1}^{c_k} \) (i) is accepting if and only if \( v_{i+1} \) is accepting, (ii) keeps only the incoming edges of \( v_{i+1} \) labeled \( c_i \), and (iii) duplicates all its outgoing edges: we replace each \( (v_{i+1}, u, c) \) with the edges \( (v_{i+1}^{c_1}, u, c), \ldots, (v_{i+1}^{c_k}, u, c) \).

Note that all the newly-created edges must be present in the final automaton \( A_t \) since the states \( v_{i+1}^{c_1}, \ldots, v_{i+1}^{c_k} \) cannot be collapsed back by \( \equiv_w \) (as they have different incoming labels); it follows that in this step we are not creating more edges than necessary.

We now insert separately \( v_{i+1}^{c_1}, v_{i+1}^{c_2}, \ldots, v_{i+1}^{c_k} \) in \( \text{LEX}_i \) in any order as follows. The procedure is the same for all those vertices, therefore we may simply assume we are about to process a node \( v \) with all incoming edges labeled with the same character \( a \). Let \( u_1 < \cdots < u_k \) be the predecessors of \( v \) in the graph; note that those nodes must belong to \( \text{LEX}_i \) (since we are processing states in topological order), therefore their order \( < \) is well-defined. We now must detect and solve inconsistencies of type 1 and 2 as defined in Section 3 (see also Figure 1).

We start with inconsistency of type 1: there already are nodes \( w_i \notin \{u_1, \ldots, u_k\} \) with outgoing edges labeled \( a \) inside the range \([u_1, u_k]\). This breaks the sequence \( u_1 < \cdots < u_k \) into \( q \) sub-intervals \([u_{ij}, u'_{ij}], j = 1, \ldots, q\), that do not contain nodes with outgoing label \( a \) different than those in \( \{u_1, \ldots, u_k\}\). The range has therefore the following form, where we denote with \( w_i \) and \( w'_i \) all nodes not in \( \{u_1, \ldots, u_k\} \) with outgoing edges labeled \( a \) and we highlight in bold the runs \([u_{ij}, u'_{ij}]\):

\[
\begin{align*}
w_1 < u_{i_1} < \cdots < u'_{i_1} < w_2 < \cdots < u_{i_2} < \cdots < u'_{i_2} < \cdots < u_{i_q} < \cdots < u'_{i_1} < w_{q+1},
\end{align*}
\]

where \( u_{i_1} = u_1, u'_{i_1} = u_k, \) and \( w_1 < u_1, w_{q+1} > u_k \) are the rightmost and leftmost states with an outgoing edge labeled \( a \), respectively (if they exist). The top part of Figure 2 depicts this situation, where \( k = 4 \) and \( u_1, \ldots, u_4 \) are clustered in \( q = 3 \) runs: \( w_1 < u_1 < w_2 < u_3 < w_3 < u_4 < \ldots < w_5 \). We solve the inconsistencies of type 1 by splitting \( v \) in (i.e. replacing it with) \( q \) equivalent nodes: \( v_1 \approx \cdots \approx v_q \). Each \( v_j \) is final if and only if \( v \) is final, duplicates all \( v \)'s outgoing edges (as seen above), and keeps only incoming edges from \( v \)'s predecessors inside the corresponding run \([u_{ij}, u'_{ij}]\). This is depicted in the bottom part of Figure 2: \( v \) has been split into the three equivalent nodes \( v_1 \approx v_2 \approx v_3 \).

Inconsistencies of type 2 are solved similarly by splitting \( a \)-successors of \( w_1, \ldots, w_{q+1} \) that belong to \( \text{LEX}_i \) when necessary. Let \( \text{LEX}_i \cap \{\text{succ}_a(w_1), \ldots, \text{succ}_a(w_{q+1})\} = \{z_1 < \cdots < z_{q'}\} \) be the \( a \)-successors of \( w_1, \ldots, w_{q+1} \) in \( \text{LEX}_i \). Note that it might be the case that \( q' < q+1 \). Note also that some of the nodes \( z_i \) might belong to \( \{u_1, \ldots, u_k\} \cup \{w_1, \ldots, w_{q+1}\} \). We have an inconsistency of type 2 (among nodes in \( \text{LEX}_i \)) whenever \( \text{succ}_a(w_1) = \text{succ}_a(w_{i+1}) = z_e \), for some \( 1 \leq e \leq q' \), and
there exist some \( u_j \) such that \( w_i < u_j < w_{i+1} \). In this case, we split \( z_e = \text{succ}_a(w_i) = \text{succ}_a(w_{i+1}) \) in two equivalent nodes \( z'_e \approx z''_e \) ordered as \( z'_e < \text{succ}_a(u_j) < z''_e \). This cannot contradict the Wheeler properties (even if \( z_i \in \{u_1, \ldots, u_k\} \cup \{w_1, \ldots, w_{q+1}\} \)), since \( \text{succ}_a(u_j) \) is one of the copies of \( v \) (or \( v \) itself if \( v \) has not been split in the previous step) and has therefore no successors in the current automaton. The process of fixing inconsistencies of type 2 is shown in Figure 2: nodes \( w_3 \) and \( w_4 \) are separated by \( w_2, u_3 \) as \( w_3 < w_2 < u_3 < w_4 \). In this case, \( \text{succ}_a(w_3) = \text{succ}_a(w_4) = z_3 \), and we split \( z_3 \) in the two equivalent nodes \( z'_3 \) and \( z''_3 \). Note also that we only need to check those \( w_i \) that immediately precede or follow a predecessor of \( v \) (i.e. \( w_1, w_2, w'_2, \ldots, w_{q+1} \)): those nodes are at most \( O(k) \), where \( k \) is the number of \( v \)'s predecessors.

As shown in Figure 2 (bottom), after solving the inconsistencies of type 1 and 2 the nodes in \( \text{LEX}_{i+1} \) are again range-consistent: the \( a \)-successors of any (sorted) range of nodes form themselves a (sorted) range. Moreover, the splitting process defines unambiguously a total ordering of the new nodes among those already in \( \text{LEX}_i \), which can be therefore updated to \( \text{LEX}_{i+1} \) by inserting those nodes at the right place: to insert a node \( v' \) in \( \text{LEX}_i \), let \( u' \) be its \( a \)-predecessor: \( \text{succ}_a(u') = v' \). Let moreover \( u'' < u' \) be the rightmost node preceding \( u' \) (in \( \text{LEX}_i \)) having an outgoing edge labeled \( a \), and let \( v'' \) be its \( a \)-successor: \( \text{succ}_a(u'') = v'' \). By range-consistency, node \( v' \) has to be inserted immediately after \( v'' \) in \( \text{LEX}_i \). If such a node \( u'' \) does not exist (i.e. \( u' \) is the leftmost node in \( \text{LEX}_i \) having an outgoing edge labeled \( a \)), then \( v' \) has to be inserted in \( \text{LEX}_i \) so that it becomes the first node with incoming edges labeled \( a \) (i.e. in the position immediately following the rightmost node \( v'' \) with incoming label \( a' \), where \( a' \) is the lexicographically-largest character such that \( a' < a \), or at the first position in \( \text{LEX}_i \) if such a character \( a' \) does not exist). This shows that invariant 2 is maintained: the Wheeler properties are kept true among nodes in \( \text{LEX}_{i+1} \). It is also clear that we do not insert \( \approx \)-equivalent adjacent states with the same incoming label (see Figure 2: by construction, the newly-inserted nodes \( v_1, z'_3, v_2, z''_3, v_3 \) are non-equivalent to their neighbors), i.e. invariant 3 is maintained. Finally, the accepted language does not change since the splitting process generates \( \approx \)-equivalent nodes; also invariant 1 stays true.

Note that the minimization process on the original acyclic DFA \( \mathcal{A} \) takes linear time. After that, we only insert edges/nodes in the minimum output WDFA: never delete. It follows that the number of performed operations is equal to the output’s size. The final automaton could be either smaller or exponentially-larger than \( \mathcal{A} \). We note that all the discussed operations can be easily implemented in logarithmic time using the data structures discussed in Section F: finding the \( q \) runs of states \( \{u_{i_j}, u'_{i_j}\} \), as well as finding the \( O(k) \) states \( w_{i_j} \), requires executing a constant number of \( \text{rank} \) operations on sequence \( \text{OUT} \) and \( \text{start} \) operations on \( \text{IN} \) for each predecessor of \( v \). Nodes can be inserted at the right position in sequence \( \text{LEX} \) exactly as done in Algorithm 3 (by also updating \( \text{IN} \) and \( \text{OUT} \)). Finally, the graph can be dynamically updated (i.e. splitting nodes) and queried (i.e. navigation) by keeping it as a dynamic adjacency list: since we can spend logarithmic time per edge, we can store the graph as a self-balancing tree associating nodes to their predecessors and successors (also kept as self-balancing trees). This structure supports all updates and queries on the graph in logarithmic time. It follows that the overall procedure terminates in \( O(n + m \log m) \) time, \( n \) and \( m \) being the input and output’s sizes, respectively.

References

1. Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. Journal of Algorithms, 35(1):82–99, 2000.
2. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 7608, pages 180–192, 2012.

21
3. Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

4. Nicolaas Govert De Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49(49):758–764, 1946.

5. Massimo Equi, Roberto Grossi, and Veli Mäkinen. On the complexity of exact pattern matching in graphs: Binary strings and bounded degree. *arXiv preprint arXiv:1901.05264*, 2019.

6. Massimo Equi, Roberto Grossi, Alexandru I. Tomescu, and Veli Mäkinen. On the complexity of exact pattern matching in graphs: Determinism and zig-zag matching. *arXiv preprint arXiv:1902.03560*, 2019.

7. Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

8. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, 57(1):4, 2009.

9. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

10. Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical computer science*, 698:67–78, 2017.

11. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. Society for Industrial and Applied Mathematics, 2018.

12. Daniel Gibney and Sharma V. Thankachan. On the Hardness and Inapproximability of Recognizing Wheeler Graphs. *arXiv preprint arXiv:1902.01960*, 2019.

13. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

14. Hermann Gruber and Markus Holzer. Computational Complexity of NFA Minimization for Finite and Unary Languages. *LATA*, 8:261–272, 2007.

15. John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

16. Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1988.

17. Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence to graph alignment. *BioRxiv*, page 522912, 2019.

18. S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

19. Andreas Malcher. Minimizing finite automata is computationally hard. *Theor. Comput. Sci.*, 327(3):375–390, November 2004.

20. Peter Weiner. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

21. Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

22. Mikko Rautiainen and Tobias Marschall. Aligning sequences to general graphs in o (v+ me) time. *bioRxiv*, page 216127, 2017.

23. Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

24. Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 11(2):375–388, 2014.

25. Mikko Rautiainen and Tobias Marschall. Aligning sequences to general graphs in o (v+ me) time. *bioRxiv*, page 216127, 2017.

26. Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

27. Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 11(2):375–388, 2014.

28. Kayva Vaddadi, Naveen Sivadasan, Kshitij Tayal, and Rajgopal Srinivasan. Sequence alignment on directed graphs. *bioRxiv*, page 124941, 2017.

29. Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT’98. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.