On Computing Minimum Wheeler DFA from Their Language

Ruben Becker
Ca' Foscari University of Venice, Italy

Davide Cenzato
Ca' Foscari University of Venice, Italy

Nicola Prezza
Ca' Foscari University of Venice, Italy

Daniel Puttini
Ca' Foscari University of Venice, Italy

Abstract

Wheeler automata have recently emerged as a powerful generalization of the Burrows-Wheeler Transform, enabling optimal linear-time pattern matching on compressed labeled graphs – a task that is otherwise computationally hard. Consequently, when an automaton recognizes a Wheeler language (i.e., it is equivalent to some Wheeler automaton), computing its minimum equivalent Wheeler DFA is a powerful indexing strategy. This problem is particularly relevant in computational pangenomics, where pangenome graphs frequently recognize Wheeler languages.

However, constructing the minimum Wheeler DFA for a Wheeler language has remained a computational bottleneck. The problem is known to be PSPACE-hard for nondeterministic inputs. When the input is a DFA, state-of-the-art solutions forced a compromise: they were either fast but limited to acyclic DFAs (Alanko et al., SODA 2020) or capable of handling general topologies but prohibitively slow (D'Agostino et al., TCS 2023). In this work, we bridge this gap with the first algorithm solving the problem for general DFAs in near-optimal, linearithmic output-sensitive time. By matching the efficiency of acyclic-only solutions while retaining full generality, our approach improves upon the previous general solution by at least a quadratic factor. We demonstrate the practical impact of our algorithm on real-world pangenome graphs; our tool achieves a processing throughput of over 10^5 transitions per second on a standard workstation, enabling the construction of a provably optimal pattern matching data structure in such applications.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Information systems → Data compression

Keywords and phrases Wheeler Automata, Minimum DFA, Pangenomics, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.132

Related Version *Full Version*: `https://arxiv.org/abs/2607.07563` [4]

Funding Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) 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.

1 Introduction

Pattern matching lies at the heart of computer science, but while we have mastered it on strings, much work remains to be done on labeled graphs. As Equi et al. [10] proved, pattern matching on general labeled graphs can likely not be solved in strongly subquadratic time; as a result, even basic pattern-matching-related tasks can quickly become prohibitively expensive

depending on graph topology and size. This algorithmic bottleneck has long slowed down progress in data-intensive fields like bioinformatics, where pangenomics requires aligning millions of sequencing reads against complex graph-structured collections of genomes.

Wheeler automata, introduced by Gagie et al. [12], side-step the lower bound of Equi et al. [10] by imposing a specific structural constraint: informally speaking, their states can be totally sorted according to the co-lexicographic order of the strings labeling the paths reaching them. In the (simpler) case of Wheeler DFA, this means that $u < v$ in Wheeler order for any two states u, v if and only if all (possibly, left-infinite) strings labeling paths entering in u are co-lexicographically smaller than those labeling paths entering in v. This seemingly simple ordering property unlocks a "best-of-both-worlds" scenario: it generalizes the celebrated Burrows-Wheeler Transform (BWT) [5] from linear strings to complex graphs, enabling pattern matching in optimal constant time per character – just as if the graph were a simple string. Furthermore, Wheeler automata can be encoded in a constant number of bits per transition – again, as if they were simple strings – and every Wheeler NFA can be converted into a Wheeler DFA of at most twice the size in polynomial time [2]. The survey [7] gives an overview of the myriad virtues of Wheeler automata and languages.

However, a critical gap remains. A regular language is called a *Wheeler language* if it can be recognized by *some* Wheeler NFA. But knowing a language is Wheeler is one thing; actually constructing the minimum Wheeler automaton for it is another. This led Alanko et al. [2] to introduce a natural optimization problem:

▶ **Problem 1.** *Given the minimum DFA $\mathcal{D}$ accepting a Wheeler language $\mathcal{L}$, find the minimum Wheeler DFA $\mathcal{D}_w$ accepting $\mathcal{L}$.*

We emphasize that Problem 1 is fundamentally different from that of finding the minimum Wheeler DFA being equivalent to a given *Wheeler* DFA, a problem admitting a linear-time solution [1]. Furthermore, while (i) deciding if a given NFA is Wheeler and (ii) deciding whether the *language* of a given NFA is Wheeler are both NP-hard problems [13, 8], we can check if the language of a DFA is Wheeler in quadratic time [3]. This places Problem 1 in a fascinating "sweet spot": it is tractable enough to determine whether a solution exists [3], yet complex enough that the optimal target automaton ($\mathcal{D}_w$) might be exponentially larger than the input ($\mathcal{D}$) [2]. This means that the ultimate goal for this problem is an algorithm whose running time scales favorably with the size of the input plus the solution it produces.

State of the Art. Let n and m denote the number of states and transitions of the input *minimum* DFA $\mathcal{D}$, and let n_w and m_w denote the corresponding counts for the output Wheeler DFA $\mathcal{D}_w$ in Problem 1 (in particular, $n - 1 \leq m \leq m_w$ and $n_w - 1 \leq m_w$ hold). Alanko et al. [2] showed how to solve the problem in $O(m_w \log m_w)$ time, though their solution was restricted to acyclic DFA. Subsequently, D'Agostino et al. [9] proposed the first algorithm capable of handling arbitrary DFA. However, while their method solves the general case, it requires $O(n^3 n_w + n^2 \sigma n_w \log n_w)$ time (where σ is the alphabet size), making it computationally prohibitive for large inputs and thus primarily of theoretical interest.

Our Contributions and Techniques. In this paper, we establish the following main result:

▶ **Theorem 2.** *Given a minimum DFA $\mathcal{D}$ accepting a Wheeler language $\mathcal{L}$, the minimum Wheeler DFA $\mathcal{D}_w$ for $\mathcal{L}$ – having m_w transitions – can be computed in $O(m_w \log m_w)$ time.*

As discussed below, the algorithm behind Theorem 2 terminates if and only if the input language $\mathcal{L}(\mathcal{D})$ is Wheeler. However, combining the above theorem with the quadratic-time algorithm of Becker et al. [3] for checking whether the language of a given DFA is Wheeler, we obtain an algorithm that terminates for *any input DFA* $\mathcal{D}$, either determining that $\mathcal{L}(\mathcal{D})$ is not Wheeler, or returning the corresponding minimum WDFA $\mathcal{D}_w$.

Theorem 2 effectively combines the strengths of both prior approaches: we match the near-optimal complexity of Alanko et al. [2] while also retaining the ability to process any input DFA, as in the algorithm by D'Agostino et al. [9] (but $\Omega(n^2)$ times faster).

Experimental results. As we demonstrate experimentally, in real-world pangenome graphs, m_w is often very close to n; this is due to the fact that in those applications the input DFA is very sparse and often not much smaller than the output WDFA. As a result, in those applications, our optimized implementation of the algorithm behind Theorem 2 outputs over 10^5 transitions per second on a standard workstation. These performances are comparable with those of the only other existing tool [14] able to convert a pangenome graph into a (possibly not minimal) equivalent Wheeler DFA.

Intuitive Description of our Results. The algorithm that allows us to derive Theorem 2 starts by sorting co-lexicographically any spanning tree of $\mathcal{D}$ rooted in the source. As observed by Alanko et al. [2], this yields a spanning Wheeler subgraph $\mathcal{D}_w$ of $\mathcal{D}$. Let Δ be a queue initialized with all transitions of $\mathcal{D}$ not included in $\mathcal{D}_w$. While $\Delta \neq \emptyset$, the algorithm inserts those transitions into $\mathcal{D}_w$, possibly splitting its states and pushing new transitions into Δ, in such a way that $\mathcal{D}_w$ remains a WDFA all the time and that $\mathcal{D}_w^{\Delta}$, i.e., $\mathcal{D}_w$ augmented with the transitions in Δ, remains equivalent to $\mathcal{D}$. We give a brief overview of how this is achieved.

Let (u, a, v) denote a transition (meaning that state v is reached from state u by reading label a). If $\Delta \neq \emptyset$, we extract the transition (u, a, v) at the top of Δ. If adding (u, a, v) to $\mathcal{D}_w$ does not violate any Wheeler axiom (Definition 4), then we perform such an addition and proceed with the next transition from Δ. Otherwise, let $\mathcal{I}_v$ denote the set of strings labeling paths entering in state v in the automaton $\mathcal{D}_w$ and either originating in the source or being left-infinite (i.e., originating in a loop). The Wheeler order $<$ on $\mathcal{D}_w$ can be equivalently characterized as follows: for any two states $u \neq v$, it holds $u < v$ if and only if $\mathcal{I}_u \prec \mathcal{I}_v$, meaning that all strings in $\mathcal{I}_u$ are co-lexicographically smaller than those in $\mathcal{I}_v$. The fact that introducing (u, a, v) violates a Wheeler axiom, means that there exists a state $p \neq v$ such that, after including the new transition, either (1) the co-lexicographic range of $\mathcal{I}_p$ is contained in that of $\mathcal{I}_v$, (2) the two ranges intersect but none is contained in the other.

Both cases can be resolved by (possibly) copying states of $\mathcal{D}_w$ and changing the destinations of some transitions; here we comment only on case (1) as it is simpler, but a similar technique applies to case (2). See also Figure 1 for a simple example only involving case (1). The axiom violation of case (1) can be solved by splitting the set $\mathcal{I}_v$ into two disjoint sets $\mathcal{I}_{v'}$ and $\mathcal{I}_{v''}$ such that $\mathcal{I}_{v'} \prec \mathcal{I}_p \prec \mathcal{I}_{v''}$. On $\mathcal{D}_w$, this can be achieved by creating a new state v''. The incoming transitions of v stay unchanged, while the new transition (u, a, v) is renamed into (u, a, v'') and inserted in $\mathcal{D}_w$. This way, either $v < p < v''$ or $v'' < p < v$ hold in Wheeler order. Finally, for each outgoing transition (v, c, q) of v, we insert in Δ a new transition (v'', c, q). As a result, v'' becomes Myhill-Nerode equivalent to v in the full automaton $\mathcal{D}_w^{\Delta}$. On the other hand, an algorithm invariant makes sure that no two adjacent states in Wheeler order are simultaneously Myhill-Nerode equivalent and have the same incoming label; as discussed below, this will be important for correctness and completeness.

The algorithm terminates when Δ becomes empty. Since each step may potentially add transitions to $\mathcal{D}_w^\Delta$, termination of the algorithm is not trivial. As a matter of fact, we prove that the algorithm terminates if and only if $\mathcal{L}(\mathcal{D})$ is Wheeler; in that case, the algorithm's output $\mathcal{D}_w$ is the minimum Wheeler DFA recognizing $\mathcal{L}(\mathcal{D})$. It is not hard to explain intuitively why this holds true. We leverage on the fact that, as proved by Alanko et al. in [2], a trimmed [1] DFA is the minimum WDFA for its language if and only if adjacent states in Wheeler order sharing the same incoming label (unique by the Wheeler axioms), are not Myhill-Nerode equivalent. Since[2] (i) we start from the minimum DFA for the language, (ii) our algorithm never inserts state v near state p (in Wheeler order) with v being Myhill-Nerode equivalent to p and sharing its incoming letter, (iii) the Wheeler axioms hold true at every step on the transitions of $\mathcal{D}_w$, (iv) every state is reachable from the source in $\mathcal{D}_w$, (v) every state can reach a final state in $\mathcal{D}_w^\Delta$, and (vi) $\mathcal{D}_w^\Delta$ recognizes $\mathcal{L}(\mathcal{D})$ at every step, Alanko et al.'s characterization [2] allows us to conclude that, if the algorithm terminates, then $\mathcal{D}_w$ is indeed the minimum Wheeler DFA recognizing $\mathcal{L}(\mathcal{D})$. The claimed complexity follows from the fact that $\mathcal{D}_w$ is stored using a dynamic data structure supporting updates in logarithmic time.

To prove that, if $\mathcal{L}(\mathcal{D})$ is Wheeler, then the algorithm terminates, consider the minimum WDFA $\mathcal{M}$ of $\mathcal{L}(\mathcal{D})$, and let $n_\mathcal{M}$ be the number of its states. If, for a contradiction, the algorithm does not terminate, then Δ is replenished infinitely often. In turn, this implies that the algorithm creates new states of $\mathcal{D}_w$ infinitely often. Upon creating the $(n_\mathcal{M}+1)$-th state, we pick a string α_i labeling a path from the source state to the i-th state (in Wheeler order) of $\mathcal{D}_w$, for all $i \in \{0, \ldots, n_\mathcal{M}\}$ (this is possible by Property (iv) above). By Properties (v-vi), strings α_i must also belong to the prefix closure of $\mathcal{L}(\mathcal{D})$, thereby they must label paths from the source to states $w_0, \ldots, w_{n_\mathcal{M}}$ of $\mathcal{M}$. We finally argue that those states must be distinct, otherwise Property (ii) would imply that $\mathcal{L}(\mathcal{M}) \neq \mathcal{L}(\mathcal{D})$. This means that $\mathcal{M}$ has $n_\mathcal{M}+1$ states, a contradiction.

2 Preliminaries

We denote by $[N]$ the set of integers $\{1, 2, \ldots, N\}$. We recall that a *total order* is a binary relation over a set U that is reflexive, transitive, antisymmetric, and strongly connected. For a total order $\preceq$ on a set U, if $x \preceq y$ and $x \neq y$ for two elements $x, y \in U$, we write $x \prec y$. Let Σ be a finite alphabet of size σ equipped with a total order $\preceq$. Without loss of generality we assume $\Sigma = [\sigma]$ to be the first σ integers with the natural total order. Symbol Σ^* denotes the set of all finite strings (including the empty string ϵ) that can be formed by concatenating symbols from Σ. With a slight abuse of notation, we use $\preceq$ also to denote the *co-lexicographic (or co-lex) order* between strings that is defined (recursively) as follows: For two strings $\alpha, \beta \in \Sigma^*$, we have $\alpha \preceq \beta$ if (i) $\alpha = \epsilon$ or if (ii) $\alpha = \alpha' a$ and $\beta = \beta' b$ with $a, b \in \Sigma$ and $\alpha', \beta' \in \Sigma^*$ such that $a \prec b$ or $a = b \wedge \alpha' \preceq \beta'$.

▶ **Definition 3** (DFA). *A* Deterministic Finite Automaton (DFA) $\mathcal{A}$ *is a 5-tuple* $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$*, where* Q *is a finite set of* states*,* Σ *is a finite alphabet,* $\delta : Q \times \Sigma \to Q$ *is the* transition function*,* $q_0 \in Q$ *is the* initial state*, and* $F \subseteq Q$ *is the set of* final states*.*

With a slight abuse of notation, we sometimes consider δ as the set of triples $\{(u, a, v) \in Q \times \Sigma \times Q : v = \delta(u, a)\}$. We extend the domain of the transition function to strings as usual: for $a \in \Sigma$, $\alpha \in \Sigma^*$, and $q \in Q$ we define $\delta(q, \epsilon) = q$ and $\delta(q, a\alpha) = \delta(\delta(q, a), \alpha)$.

[1] Meaning that every state is reachable from the source and can reach at least one final state.
[2] As a matter of fact, these properties are a simplified description of the algorithm's invariants.

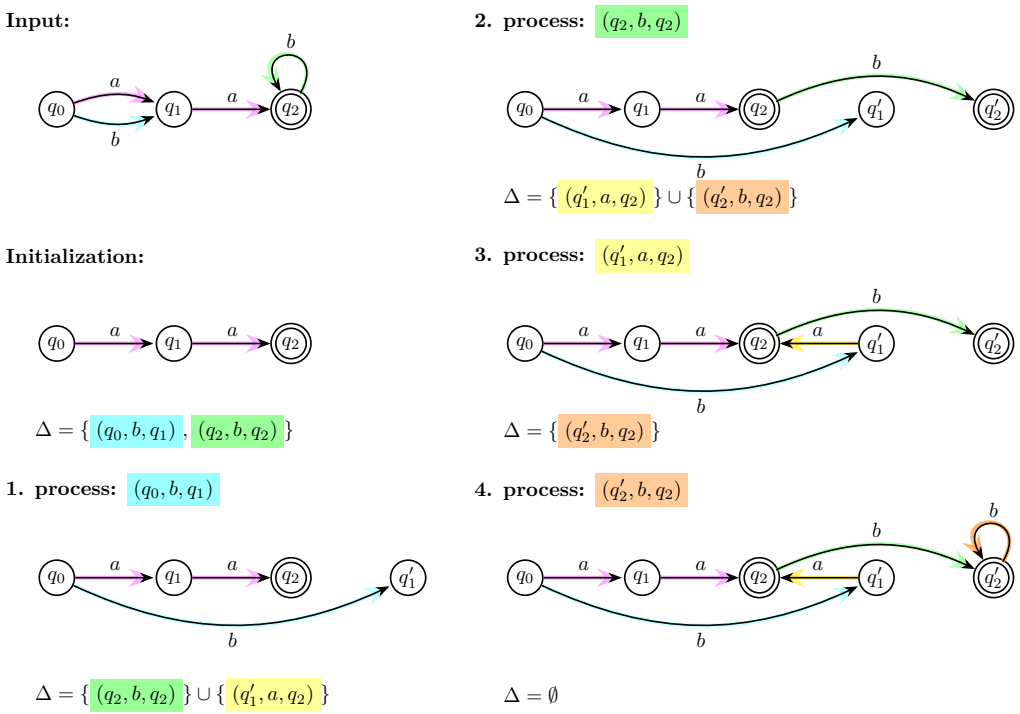

Figure 1 The (non-Wheeler) DFA $\mathcal{D}$ (top left) is minimum for the Wheeler language $\mathcal{L} = (a|b)ab^*$. **Initialization.** We start by choosing a spanning tree $\mathcal{D}_w$ rooted at the source, here containing the two pink transitions (q_0, a, q_1) and (q_1, a, q_2). The remaining two transitions (q_0, b, q_1) and (q_2, b, q_2) are pushed in the queue Δ. At any step, we denote with $\mathcal{D}_w^\Delta$ the union of $\mathcal{D}_w$ and Δ (at the beginning, $\mathcal{D}_w^\Delta = \mathcal{D}$). **First step.** The algorithm pops the transition (q_0, b, q_1) from Δ and processes it. The introduction of (q_0, b, q_1) in $\mathcal{D}_w$ would add the string b to the set of strings $\mathcal{I}_{q_1}$ entering in state q_1 in the sub-automaton $\mathcal{D}_w$ and the string ba to $\mathcal{I}_{q_2}$; as a result, the co-lexicographic ranges of $\mathcal{I}_{q_1} = \{a, b\}$ and $\mathcal{I}_{q_2} = \{aa, ba\}$ would intersect ($a \prec aa \prec ba \prec b$). Hence, the transition is instead introduced pointing to a new state q_1' that acts as a copy of state q_1. We now have $\mathcal{I}_{q_1} = \{a\}$ and $\mathcal{I}_{q_1'} = \{b\}$ and hence both these sets do not intersect co-lexicographically with $\mathcal{I}_{q_2} = \{aa\}$. As q_1 and q_1' need to be Myhill-Nerode equivalent in the full automaton $\mathcal{D}_w^\Delta$, the algorithm needs to add corresponding transitions (q_1', c, q) for all outgoing transitions (q_1, c, q) of q_1 in $\mathcal{D}_w^\Delta$ to the set of transitions Δ to be processed. In this particular case, the algorithm adds (q_1', a, q_2) to Δ. **Second step.** Next, the algorithm pops the transition (q_2, b, q_2) from Δ and processes it. This leads to the creation of a new state q_2'; this is required since otherwise $\mathcal{I}_{q_1'}$ and $\mathcal{I}_{q_2}$ would intersect co-lexicographically. The transition (q_2', b, q_2) is then added to Δ in order to make q_2 and q_2' Myhill-Nerode equivalent. **Third and fourth steps.** Finally, the transitions (q_1', a, q_2) and (q_2', b, q_2) from Δ can be simply added to $\mathcal{D}_w$ without breaking the Wheeler order, hence Δ becomes empty and the algorithm outputs the minimum Wheeler DFA $\mathcal{D}_w$ (bottom right) accepting $\mathcal{L}$.

For a state $q \in Q$, we denote with $I_\mathcal{A}(q)$ the set of strings [3] *reaching* q from the initial state in $\mathcal{A}$, i.e., $I_\mathcal{A}(q) := \{\alpha \in \Sigma^* : \delta(q_0, \alpha) = q\}$. The *language* $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ *recognized by* $\mathcal{A}$ is the set of strings reaching a final state from the initial state, that is $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in F} I_\mathcal{A}(q)$. With $\mathrm{Pref}(\mathcal{L}(\mathcal{A}))$ we denote the set of prefixes of words in $\mathcal{L}(\mathcal{A})$.

[3] The set $I_\mathcal{A}(q)$ has not to be confused with the set $\mathcal{I}_q$ mentioned in the introduction, which is instead the union of $I_{\mathcal{D}_w}(q)$ and all left-infinite strings reaching state q in $\mathcal{D}_w$

A DFA is *accessible* if $I_q \neq \emptyset$ for all states, i.e., every state is reachable by a directed path from the source state. A DFA is instead *co-accessible*, if every state can reach a final state. A DFA that is both accessible and co-accessible is called *trimmed*.

Two strings α, β are *Myhill-Nerode equivalent* (MN) in $\mathcal{A}$, denoted by $\alpha \equiv_{\mathcal{A}} \beta$, if for all $\gamma \in \Sigma^*$, it holds $\alpha \cdot \gamma \in \mathcal{L}(\mathcal{A})$ if and only if $\beta \cdot \gamma \in \mathcal{L}(\mathcal{A})$ [15, 17]. This equivalence relation naturally extends to states: two states u and v of $\mathcal{A}$ are Myhill-Nerode equivalent, denoted by $u \equiv_{\mathcal{A}} v$, if for all $\gamma \in \Sigma^*$, it holds that the state $\delta(u, \gamma)$ is final if and only if $\delta(v, \gamma)$ is. We omit the subscript from $\equiv_{\mathcal{A}}$ if it is clear from the context. The equivalence classes with respect to the Myhill-Nerode equivalence relation (in both variants: strings/states) correspond exactly to the states of the unique minimum DFA that accepts the same language [15, 17].

Wheeler automata have been introduced by Gagie et al. [12] as a tool extending the celebrated Burrows-Wheeler transform [5] from strings to labeled graphs. In this paper, we consider only deterministic Wheeler automata. In the following definition, we assume that the initial state q_0 is the only one with no incoming transitions (this will simplify our description and, as we show below, comes with no loss of generality).

▶ **Definition 4** (Wheeler DFA [12]). *A Wheeler DFA (WDFA) is a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ such that q_0 is the unique state with no incoming transitions. In addition, the automaton must admit a (unique) total order $\leq$ on Q, called the Wheeler order, where q_0 is the minimum element. This order is defined such that for any two transitions (u, a, u') and (v, b, v'), the following Wheeler Properties (a.k.a.* Wheeler Axioms*) hold:*
W1. *If $a \prec b$, then $u' < v'$.*
W2. *If $a = b$ and $u < v$, then $u' \leq v'$.*

The fact that WDFA admit a *unique* Wheeler order is known [2] and significantly simplifies the problem considered in this paper (Wheeler NFA, on the other hand, may admit multiple Wheeler orders). We notice that Property W 1 implies that a WDFA $\mathcal{A}$ is always *input-consistent*, meaning that for all $u, v \in Q$ and $a, b \in \Sigma$, if $\delta(u, a) = \delta(v, b)$, then $a = b$. On WDFA, we denote by $\lambda_{\mathcal{A}} : Q \to \Sigma$ the function that returns the unique incoming label of a state, that is, for a state $v \in Q \setminus \{q_0\}$ we define $\lambda_{\mathcal{A}}(v) = a$ if and only if for all $u \in Q$ with $\delta(u, b) = v$ it holds that $b = a$. For the initial state, we define $\lambda_{\mathcal{A}}(q_0) = \# \notin \Sigma$, where $\#$ is an artificial letter strictly smaller than all characters in Σ according to $\preceq$. When $\mathcal{A}$ is clear from the context, we drop the subscript and simply write λ. The assumption that q_0 has no incoming transitions is without loss of generality as one can always modify any regular language $\mathcal{L}$ into $\$\mathcal{L}$ for a new character $\$$ not belonging to the original alphabet of $\mathcal{L}$. As a result, any DFA $\mathcal{A}$ recognizing $\mathcal{L}$ can be converted into one recognizing $\$\mathcal{L}$ by just adding to it a new initial state q_0' without incoming transitions and a transition $(q_0', \$, q_0)$. It is not hard to show that this transformation also preserves Wheelerness, if $\mathcal{A}$ was Wheeler.

The following lemma states that the Wheeler order of the states is consistent with the co-lexicographic order of the strings reaching states.

▶ **Lemma 5** (Lemma 3 in [6]). *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a WDFA with Wheeler order $\leq$ and let $u, v \in Q$ and $\alpha \in I_{\mathcal{A}}(u)$ as well as $\beta \in I_{\mathcal{A}}(v)$. Then $u < v$ implies $\alpha \prec \beta$.*

Alanko et al. [2] prove the following *Myhill-Nerode theorem* for Wheeler languages.

▶ **Theorem 6** (Minimum Wheeler DFA, Theorem 4.2 in [2]). *Let $\mathcal{D}_w$ be a trimmed Wheeler DFA with Wheeler order $\leq$ and states $q_0 < q_1 < \cdots < q_t$. Furthermore, let $\equiv$ be the Myhill-Nerode equivalence relation among its states. Then, $\mathcal{D}_w$ is the minimum Wheeler DFA recognizing $\mathcal{L}(\mathcal{D}_w)$ if and only if $q_{i-1} \equiv q_i$ implies $\lambda(q_{i-1}) \neq \lambda(q_i)$ for every $i \in [t]$.*

Note that Theorem 4.2 in the paper Alanko et al. [2] does not explicitly mention the assumption that $\mathcal{D}_w$ needs to be trimmed, instead this necessary assumption is globally assumed (in Section 1.2) for all automata in their paper.

3 Formal Description of the Algorithm

Algorithm 1 is our solution to Problem 1. In this section, we describe all the details of the algorithm, while in the next section, we prove its correctness, completeness, and runtime. For convenience, Table 1 summarizes the meaning of all symbols used in the algorithm and its analysis.

Table 1 Cheatsheet of symbols and functions used in Algorithm 1.

$\mathcal{D} = (Q = [n], \Sigma, \delta, q_0, F)$	Input DFA. If $\delta(u, a)$ is not defined, we write $\delta(u, a) = \bot$. We let $\delta(\bot, a) = \bot$.
$\mathcal{D}_w = (Q_w, \Sigma, \delta_w, q_0, F_w)$	Intermediate and output WDFA. At the beginning, $Q_w = Q$, $F_w = F$, and $\delta_w \subseteq \delta$ is a spanning tree of $\mathcal{D}$ rooted in q_0. Invariants: $\mathcal{D}_w$ is Wheeler and accessible.
$\lambda(u)$	Shorthand for $\lambda_{\mathcal{D}_w}(u)$: label of incoming transitions of $u \in Q_w$.
Δ	Queue with transitions to be added to $\mathcal{D}_w$.
$\mathcal{D}_w^\Delta = (Q_w, \Sigma, \delta_w \cup \Delta, q_0, F_w)$	$\mathcal{D}_w$ augmented with Δ. Initially, $\mathcal{D}_w^\Delta = \mathcal{D}$ is the input. At the end, $\mathcal{D}_w^\Delta = \mathcal{D}_w$ is the algorithm's output. Invariants: $\mathcal{L}(\mathcal{D}_w^\Delta) = \mathcal{L}(\mathcal{D})$ and $\mathcal{D}_w^\Delta$ is deterministic and co-accessible.
$\equiv$	Myhill-Nerode equivalence relation on $\mathcal{D}_w^\Delta$.
$\cong$	Equivalence relation on $\mathcal{D}_w^\Delta$ computed by the algorithm. Invariant: $\cong$ and $\equiv$ are equal.
LEX	Wheeler order (permutation) of $Q_w = \{1, \ldots, \lvert Q_w \rvert\}$.
LEX^{-1}	Inverse of LEX. We define $\text{LEX}^{-1}[\bot] = 0$.
$\texttt{pred}(u, a)$	Largest state $v < u$ in Wheeler (LEX) order such that $\delta_w(v, a) \neq \bot$; $\texttt{pred}(u, a) = \bot$ if no such state exists.
$\texttt{succ}(u, a)$	minimum state $v > u$ in Wheeler (LEX) order such that $\delta_w(v, a) \neq \bot$; $\texttt{succ}(u, a) = \bot$ if no such state exists.
$\texttt{insert}(u, i)$	Inserts u at position i in LEX, i.e. $\text{LEX} \coloneqq \text{LEX}[\ldots, i-1]\ u\ \text{LEX}[i, \ldots]$.
$\texttt{copy}(q, T, j)$	Creates a $\cong$-equivalent copy q' of q and inserts it in position $\text{LEX}[j]$. Outgoing transitions of q belonging to Δ (resp. δ_w) are copied on q' and inserted in Δ (resp. T).

Let us recall that, given a minimum DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ for a Wheeler language $\mathcal{L}$, the goal is to construct the minimum Wheeler DFA $\mathcal{D}_w$ accepting $\mathcal{L}$. We assume that states are represented by integers: $Q = [n]$, and that the special symbol $\bot$ (non-existing state) corresponds to integer $\bot = 0$. Algorithm 1 is composed of two functions: function `MinWheeler` (the main algorithm) and the auxiliary function `copy`. We start from the former.

Initialization

Function `MinWheeler` begins in Line 10 by calling `SortedSpanningTree(`$\mathcal{D}$`,`q_0`)`. This function (1) computes a spanning tree of $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ rooted in q_0 (e.g. via a DFS visit) and (2) sorts Q according to the colexicographic order of the strings labeling the source-to-nodes paths in the spanning tree. Here, the linear-time algorithm [11, Theorem 2] can be used.

The result is returned as a pair (δ_w, LEX), where $\delta_w \subseteq \delta$ is the set of transitions forming the spanning tree and $\mathrm{LEX}[1, n]$ is the permutation of $Q = [n]$ containing the states Q sorted by the colexicographic order of the source-to-state strings in the spanning tree. In other words, LEX encodes the Wheeler order of the Wheeler subgraph $\mathcal{D}_w = (Q, \Sigma, \delta_w, q_0, F_w)$, where F_w is initially set to be equal to F (Line 11). At any point of the execution, LEX will be a permutation, and we will indicate with $\mathrm{LEX}^{-1}[q]$ the position i such that $\mathrm{LEX}[i] = q$. As a special case, we define $\mathrm{LEX}^{-1}[\perp] = 0$, where $\perp$ indicates a non-existing state.

In Line 12, the algorithm initializes the queue Δ to contain all transitions in $\delta \setminus \delta_w$. At any step of the algorithm's execution, we denote with $\mathcal{D}_w^{\Delta} = (Q, \Sigma, \delta_w \cup \Delta, q_0, F_w)$ the union between $\mathcal{D}_w$ and the transitions in Δ. Note that, at this point of the execution, $\mathcal{D}_w^{\Delta} = \mathcal{D}$.

The algorithm stores an internal representation of the Myhill-Nerode equivalence relation between states of $\mathcal{D}_w^{\Delta}$ as an integer label indicated with $[u]_{\cong}$, associated to each state u; Line 13 initializes $[u]_{\cong} := u$ to be equal to the original Myhill-Nerode equivalence relation $\equiv$ on $\mathcal{D}$ (recall that $\mathcal{D}$ is minimum, so each state is a class of $\equiv$); note that $[\perp]_{\cong} := \perp \notin Q$. Observe that in our pseudocode we use a symbol ($\cong$) different from $\equiv$ in order to distinguish the relation computed by the algorithm with the "real" Myhill-Nerode relation; ultimately, we will prove that $\cong$ and $\equiv$ are the same relation on $\mathcal{D}_w^{\Delta}$ before and after every iteration of the `while` loop, but this distinction will be useful in our proofs. We emphasize that $\equiv$ and $\cong$ are always to be read as equivalence relations on $\mathcal{D}_w^{\Delta}$ (and not $\mathcal{D}_w$).

`while` loop

Identifying the predecessor and successor of u by letter a. The `while` loop begins by popping a transition (u, a, v) from Δ (Line 15). Then, Line 16 (resp. 17) identifies the largest (resp. smallest) state $\texttt{pred}(u, a) < u$ (resp. $\texttt{succ}(u, a) > u$) in Wheeler order (i.e., in LEX) having an outgoing transition labeled with letter a in $\mathcal{D}_w$, that is, such that $p = \delta_w(\texttt{pred}(u, a), a)$ (resp. $s = \delta_w(\texttt{succ}(u, a), a)$) is defined. If no such state exists, then $p = \perp$ (resp. $s = \perp$).

We claim that, by determinism and the Wheeler axioms, if both p and s exist then no state lies strictly between them in Wheeler order; in particular, either $p = s$ or p and s are adjacent in LEX. Indeed, since $\mathcal{D}_w$ is deterministic, the only a-labeled transition leaving u is (u, a, v), which at this point belongs to Δ rather than to δ_w; hence u has *no* outgoing a-transition in $\mathcal{D}_w$. By definition of $\texttt{pred}(u, a)$ and $\texttt{succ}(u, a)$, the only state that could lie strictly between them in Wheeler order and carry an outgoing a-transition is u itself, which carries none. Thus $\texttt{pred}(u, a)$ and $\texttt{succ}(u, a)$ are *consecutive* among the states with an outgoing a-transition in $\mathcal{D}_w$. Now, Wheeler axiom W1 forces the targets of a-labeled transitions to form a contiguous range of LEX, while W2 makes the map sending each such source to its a-target monotone (non-decreasing) and onto this range. Two consecutive sources are therefore mapped to two targets with no state of the range strictly between them, which is precisely the claim for $p = \delta_w(\texttt{pred}(u, a), a)$ and $s = \delta_w(\texttt{succ}(u, a), a)$.

In order to satisfy the Wheeler axioms, the edge (u, v) should point to a state immediately after p (if it exists) and before s (if it exists).

Next, we describe the function `copy(`q, T, j`)` since understanding its behavior is important for the subsequent steps of the algorithm.

Function `copy(`q, T, j`)`. In Line 2, the function creates a copy q' of q initialized to be the next available integer $|\,\mathrm{LEX}\,| + 1$ (at any point, states are consecutive integers: $\{1, \ldots, |\,\mathrm{LEX}\,|\}$), inserts it in position j of LEX (Line 3), copies the "final" status of q and its Myhill-Nerode equivalence into q' (Lines 4 and 5), and copies the outgoing transitions of

q into q' (Lines 6 and 7). More in detail, Line 6 copies each outgoing transition (q, c, x) of q belonging to Δ, inserting the copy (q', c, x) in Δ. Line 7, on the other hand, takes care of the outgoing transitions of q belonging to δ_w. For each such transition $(q, c, x) \in \delta_w$, the new transition (q', c, x) is inserted in the set T passed *by reference* to the function; this detail is important since the function will be called once by binding T to δ_w, and once by binding it to Δ. We can now go back to describing the three cases that can occur in Lines 18-25 of function `MinWheeler`.

Case 1: $v \cong p$ or $v \cong s$. In other words, the `if` condition at line 18 succeeds. This case is the simplest: Lines 19 and 26 insert the transition (u, a, v') into δ_w, where v' is the state among p and s being $\cong$-equivalent to v (it could be $v = v'$). This case does not require creating new states, since adding the transition (u, a, v') (equivalent to (u, a, v) from a language perspective) to $\mathcal{D}_w$ does not violate any Wheeler axiom (we will prove this later).

Case 2: $p \ncong v \ncong s$ and $p = s \neq \bot$. In other words, the `if` condition at line 18 does not succeed and the `if` condition at line 21 succeeds. We cannot simply move (u, a, v) from Δ to δ_w: Wheeler axiom W2 would require $p < v < s$, which in this case is impossible since $v \neq p = s$. This issue can be solved by first splitting $p = s$ into two distinct $\cong$-equivalent states $p \neq s$. This is precisely the role of Line 22, which overwrites variable s with a brand new state with the same outgoing transitions as p by calling function `copy`. Importantly, this call to `copy` treats differently the outgoing transitions of p belonging to Δ and those belonging to δ_w. For each $(p, c, x) \in \Delta$ of the former type, a new transition (s, c, x) is inserted into Δ. For each $(p, c, x) \in \delta_w$ of the latter type, a new transition (s, c, x) is inserted into δ_w. This behavior is achieved by binding the formal parameter T of function `copy` to the actual parameter δ_w. As we will prove later, this update of δ_w does not violate any Wheeler axiom and is crucial for the completeness of our algorithm. At this point, Line 23 replaces every transition $(x, a, p) \in \delta_w$ such that $u < x$ (in Wheeler order), with transition (x, a, s). In other words, we "move" the destinations of all in-transitions of p coming from a state x larger than u to s. As we will prove later, this preserves the Wheeler axioms and the language.

We now have two adjacent (in Wheeler order) states $p < s$ being Myhill-Nerode equivalent, i.e. $p \cong s$. We are left to insert a new state $v' \cong v$ between them and add transition (u, a, v') to δ_w. This is precisely what Lines 24, 25, and 26 do.

First, note that symbol λ in Line 24 indicates $\lambda_{\mathcal{D}_w}$; the subscript omission does not create ambiguity, since this notation is defined only on WDFA, and $\mathcal{D}_w$ is the only WDFA here. Line 24, in this particular case (p and s both exist), simplifies to $i \coloneqq 1 + \text{LEX}^{-1}[p]$ $(= \text{LEX}^{-1}[s])$ and therefore computes the position of s in LEX. To see this, observe that $\{j \ : \ \lambda(\text{LEX}[j]) \prec a\}$ is the set of indices in LEX of all states with incoming transitions (in δ_w) labeled by characters strictly smaller than a, hence by Wheeler axiom W1 they precede p (reached by a) in LEX; as a result, the max operator returns $\text{LEX}^{-1}[p]$. Line 25, with a call to `copy`, creates a new state v', places it between p and s (so that $p < v' < s$ are adjacent), and copies the outgoing transitions of v into v', inserting them into Δ (in this case, we cannot add transitions to δ_w as they could violate some Wheeler axiom; inserting them in Δ ensures that they will be processed later). This is achieved by binding the formal parameter T of `copy` to the actual parameter Δ. Finally, Line 26 adds (u, a, v') to δ_w.

Case 3: $p \ncong v \ncong s$ and $(p \neq s \vee s = \bot)$. In other words, both the `if` conditions at lines 18 and 21 do not succeed. Since $p \ncong v \ncong s$, we must place a $\cong$-equivalent copy v' of v immediately after p, if such a state exists (and before s, if it exists; note that, if both

p and s exist, they must be adjacent in Wheeler order). If p does not exist, then v' has to become the first state in Wheeler order such that $\lambda(v') = a$. In both cases ($p \neq \bot$ or $p = \bot$), Line 24 finds the position i where v' has to be inserted: if $p \neq \bot$ then $\mathrm{LEX}^{-1}[p]$ is larger than all indices in $\{j \ : \ \lambda(\mathrm{LEX}[j]) \prec a\}$, thereby Line 24 correctly computes $i := 1 + \mathrm{LEX}^{-1}[p]$; on the other hand, if $p = \bot$ then $\mathrm{LEX}^{-1}[p] = \mathrm{LEX}^{-1}[\bot] = 0$ and Line 24 computes $i := 1 + \max(\{j \ : \ \lambda(\mathrm{LEX}[j]) \prec a\} \cup \{0\})$. This means that $i - 1$ is the position of the last state in LEX with incoming letter smaller than a or, if all states in LEX have incoming letter larger than or equal to a, then $i - 1 = 0$. Thus, position i is where the first state with incoming letter a should be inserted in LEX according to Wheeler Axiom W1.

After that, Line 25, with a call to `copy`, creates a new state v', inserts it in position i of LEX, and copies the outgoing transitions of v into v', inserting them into Δ. Finally, Line 26 adds transition (u, a, v') to δ_w.

Algorithm 1 `MinWheeler`.

Input : minimum DFA $\mathcal{D}$ accepting Wheeler language $\mathcal{L}(\mathcal{D})$
Output : minimum Wheeler DFA $\mathcal{D}_w$ such that $\mathcal{L}(\mathcal{D}_w) = \mathcal{L}(\mathcal{D})$

1 **Function** `copy`(q,T,j)**:** `// T (transitions set) passed by reference`
2 $q' := |\,\mathrm{LEX}\,| + 1$ `// LEX, `F_w`, `Δ`, `δ_w`: global variables`
3 `insert`(q', j)
4 **if** $q \in F_w$ **then** $F_w := F_w \cup \{q'\}$
5 $[q']_{\cong} := [q]_{\cong}$
6 $\Delta := \Delta \cup \{(q', c, x) \ : \ (q, c, x) \in \Delta\}$
7 $T := T \cup \{(q', c, x) \ : \ (q, c, x) \in \delta_w\}$
8 **return** q'
9 **Function** `MinWheeler`$(\mathcal{D} = (Q, \Sigma, \delta, q_0, F))$**:**
10 $(\delta_w, \mathrm{LEX}) := \mathtt{SortedSpanningTree}(\mathcal{D}, q_0)$
11 $F_w := F$
12 $\Delta := \delta \setminus \delta_w$
13 **foreach** $u \in Q \cup \{\bot\}$ **do** $[u]_{\cong} := u$ `//` $[u]_{\cong}$ `is an integer associated with` u
14 **while** $\Delta \neq \emptyset$ **do**
15 $(u, a, v) := \Delta.\mathrm{pop}()$
16 $p := \delta_w(\mathtt{pred}(u, a), a)$ `// if` $\mathtt{pred}(u, a) = \bot$ `then` $p = \bot$ `(same for` s`).`
17 $s := \delta_w(\mathtt{succ}(u, a), a)$
18 **if** $[v]_{\cong} \in \{[p]_{\cong}, [s]_{\cong}\}$ **then**
19 **let** $v' \in \{p, s\}$ **be such that** $v \cong v'$ `// could be` $v' = v$
20 **else**
21 **if** $p = s \neq \bot$ **then**
22 $s := \mathtt{copy}(p, \delta_w, \mathrm{LEX}^{-1}[p] + 1)$
23 **foreach** $(x, a, p) \in \delta_w$ s.t. $u < x$ **do** $\delta_w := (\delta_w \setminus \{(x, a, p)\}) \cup \{(x, a, s)\}$
24 $i := 1 + \max\left(\{j \ : \ \lambda(\mathrm{LEX}[j]) \prec a\} \cup \{\mathrm{LEX}^{-1}[p]\}\right)$ `//` $\mathrm{LEX}^{-1}[\bot] = 0$
25 $v' := \mathtt{copy}(v, \Delta, i)$
26 $\delta_w := \delta_w \cup \{(u, a, v')\}$
27 **return** $\mathcal{D}_w = (\{1, \ldots, |\,\mathrm{LEX}\,|\}, \Sigma, \delta_w, q_0, F_w)$

4 Analysis: Correctness, Completeness, and Runtime

In this section we prove the correctness, completeness, and the bound on the runtime of Algorithm 1. In what follows, we fix an input DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ that is minimum for the Wheeler language $\mathcal{L}$ that it accepts and denote with $\mathcal{D}_w$ the output DFA of `MinWheeler`. Our goal is to prove the following theorem from the introduction, restated here.

▶ **Theorem 2.** *Given a minimum DFA $\mathcal{D}$ accepting a Wheeler language $\mathcal{L}$, the minimum Wheeler DFA $\mathcal{D}_w$ for $\mathcal{L}$ – having m_w transitions – can be computed in $O(m_w \log m_w)$ time.*

The proof of Theorem 2 relies on three main results. In Lemma 8 we show that the algorithm always terminates (assuming that $\mathcal{L}$ is a Wheeler language). By Lemma 9, the output automaton $\mathcal{D}_w$ is deterministic, recognizes the same language $\mathcal{L}$, and is the minimum Wheeler DFA that accepts $\mathcal{L}$. Finally at the end of this, we show how to implement `MinWheeler` using dynamic data structures achieving the above claimed running time.

To facilitate the proofs of completeness and correctness, we first state the invariants maintained by the algorithm at the beginning of each iteration of the while loop. Due to space limitations, the proof of the lemma is deferred to Appendix A.

▶ **Lemma 7** (Invariants). *At any step of Algorithm 1, let us denote:*
- $\mathcal{D}_w = (Q_w = \mathrm{LEX}, \Sigma, \delta_w, q_0, F_w)$,
- $\mathcal{D}_w^\Delta = (Q_w = \mathrm{LEX}, \Sigma, \delta_w \cup \Delta, q_0, F_w)$, *and*
- $u \cong v$, *with* $u, v \in Q_w$, *if and only if* $[u]_\cong = [v]_\cong$.

The following invariants hold before and after every iteration of the `while` *loop of Algorithm 1:*

(1) ***Determinism:*** *$\mathcal{D}_w^\Delta$ is deterministic.*
(2) ***Accessibility:*** *The automaton $\mathcal{D}_w$ is accessible.*
(3) ***Wheeler Order:*** *$\mathcal{D}_w$ is Wheeler and* LEX *encodes its Wheeler order.*
(4) ***MN Equivalence:*** *$\equiv$ and $\cong$ are the same equivalence relation in the automaton $\mathcal{D}_w^\Delta$.*
(5) ***Language:*** *$\mathcal{L}(\mathcal{D}_w^\Delta) = \mathcal{L}(\mathcal{D})$.*
(6) ***Wheeler-minimality:*** *For any $i \in \{1, \ldots, |\,\mathrm{LEX}\,| - 1\}$, if $\mathrm{LEX}[i] \equiv \mathrm{LEX}[i+1]$ in $\mathcal{D}_w^\Delta$, then $\lambda_{\mathcal{D}_w}(\mathrm{LEX}[i]) \neq \lambda_{\mathcal{D}_w}(\mathrm{LEX}[i+1])$.*
(7) ***Co-accessibility:*** *The automaton $\mathcal{D}_w^\Delta$ is co-accessible.*

Completeness

Using the above invariants, we now prove that the algorithm always terminates for a Wheeler language $\mathcal{L}(\mathcal{D})$ in input.

▶ **Lemma 8** (Completeness). *Algorithm 1 terminates for every input DFA $\mathcal{D}$ that is minimum for a Wheeler language $\mathcal{L}(\mathcal{D})$.*

Proof. Let $\mathcal{M} = (Q_\mathcal{M}, \Sigma, \delta_\mathcal{M}, q^0_\mathcal{M}, F_\mathcal{M})$ be the unique minimum Wheeler DFA recognizing $\mathcal{L}(\mathcal{D})$, and let $n_\mathcal{M}$ be the number of its states. Assume, for a contradiction, that the algorithm does not terminate. Since each iteration of the `while` loop in line 15 removes a transition from Δ, non-termination implies that Δ is replenished infinitely via calls to `copy()`. Because each such call adds a new state to LEX, the number of states $|\,\mathrm{LEX}\,|$ must eventually exceed $n_\mathcal{M}$. Consider the iteration when $|\,\mathrm{LEX}\,| = n_\mathcal{M} + 1$ and let $\mathrm{LEX} = [v_0, v_1, \ldots, v_{n_\mathcal{M}}]$. By Invariant 2, each state v_i is reachable from q_0 in $\mathcal{D}_w$, implying that the set $I_{\mathcal{D}_w}(v_i) = \{\alpha \in \Sigma^* : \delta_w(q_0, \alpha) = v_i\}$ is non-empty for every $i \in \{0, \ldots, n_\mathcal{M}\}$. Hence, there exist $\alpha_i \in I_{\mathcal{D}_w}(v_i)$ for every $i \in \{0, \ldots, n_\mathcal{M}\}$. Furthermore, by Invariant 3 the DFA $\mathcal{D}_w$ is Wheeler and thus Lemma 5 yields that $\alpha_i \prec \alpha_j$ for all $i < j$. As $\mathcal{D}_w^\Delta$ is co-accessible due to Invariant 7, it

follows that $\alpha_i \in \mathrm{Pref}(\mathcal{L}(\mathcal{D}_w^\Delta))$ for all $i \in \{0, \ldots, n_\mathcal{M}\}$. As $\mathcal{L}(\mathcal{D}_w^\Delta) = \mathcal{L}(\mathcal{D})$ due to Invariant 5, we furthermore have $\alpha_i \in \mathrm{Pref}(\mathcal{L}(\mathcal{D}))$. Hence also the minimum Wheeler DFA $\mathcal{M}$ for $\mathcal{L}(\mathcal{D})$ has to contain states $w_0, \ldots, w_{n_\mathcal{M}}$ such that $\alpha_i \in I_\mathcal{M}(w_i)$. Furthermore $\alpha_0 \prec \ldots \prec \alpha_{n_w}$ and Lemma 5 imply that $w_0 \leq_\mathcal{M} \ldots \leq_\mathcal{M} w_{n_\mathcal{M}}$, where $\leq_\mathcal{M}$ is the unique Wheeler order on $\mathcal{M}$. As $\leq_\mathcal{M}$ is a total order and $\mathcal{M}$ has $n_\mathcal{M}$ states, we must have $w_{i-1} = w_i$ for some $i \in [n_\mathcal{M}]$. Wheeler axiom W1 implies that $\mathcal{M}$ is input consistent and as $\delta_\mathcal{M}(q^0_\mathcal{M}, \alpha_{i-1}) = \delta_\mathcal{M}(q^0_\mathcal{M}, \alpha_i)$, the two strings α_{i-1} and α_i have to end with the same letter. As also $\delta_w(q_0, \alpha_{i-1}) = v_{i-1}$ and $\delta_w(q_0, \alpha_i) = v_i$, this implies $\lambda_{\mathcal{D}_w}(v_{i-1}) = \lambda_{\mathcal{D}_w}(v_i)$. Invariant 6 now implies that $v_{i-1} \not\equiv v_i$ in $\mathcal{D}_w^\Delta$. Hence there exists $\alpha \in \Sigma^*$ such that $|\{\alpha_{i-1}\alpha, \alpha_i\alpha\} \cap \mathcal{L}(\mathcal{D}_w^\Delta)| = 1$. Notice however that $\delta_\mathcal{M}(q_0, \alpha_{i-1}\alpha) = \delta_\mathcal{M}(q_0, \alpha_i\alpha)$ and hence $|\{\alpha_{i-1}\alpha, \alpha_i\alpha\} \cap \mathcal{L}(\mathcal{M})| \in \{0, 2\}$. As $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{D}_w^\Delta)$, this is a contradiction. Hence, the algorithm terminates. ◀

Correctness

Having established termination, we prove that the resulting automaton is indeed the minimum Wheeler DFA for the target language.

▶ **Lemma 9** (Correctness). *The automaton $\mathcal{D}_w$ returned by Algorithm 1 is the minimum Wheeler DFA recognizing $\mathcal{L}(\mathcal{D})$.*

Proof. Recall that $\mathcal{D}_w = (\mathrm{LEX}, \Sigma, \delta_w, q_0, F_w)$ and $\mathcal{D}_w^\Delta = (\mathrm{LEX}, \Sigma, \delta_w \cup \Delta, q_0, F_w)$. At termination, since Δ is empty, then the algorithm returns $\mathcal{D}_w = \mathcal{D}_w^\Delta$. Invariant 1 implies that $\mathcal{D}_w$ is deterministic and Invariants 2 and 7 imply that $\mathcal{D}_w = \mathcal{D}_w^\Delta$ is both accessible and co-accessible and thus trimmed. Invariant 5 furthermore implies that $\mathcal{L}(\mathcal{D}_w) = \mathcal{L}(\mathcal{D})$ and from Invariant 3 it follows that $\mathcal{D}_w$ is Wheeler and LEX encodes its Wheeler order. Hence, $\mathcal{D}_w$ is a trimmed WDFA that accepts $\mathcal{L}(\mathcal{D})$. Theorem 6 thus yields that $\mathcal{D}_w$ is the minimum Wheeler DFA for $\mathcal{L}(\mathcal{D})$ if and only if, for any two subsequent states u and v, in its Wheeler order (that is encoded by LEX), $u \equiv v$ implies $\lambda_{\mathcal{D}_w}(u) \neq \lambda_{\mathcal{D}_w}(v)$. This is exactly what Invariant 6 states and hence this concludes the proof. ◀

Runtime

We now describe the data structures employed by Algorithm 1 to construct the minimum Wheeler DFA $\mathcal{D}_w = (Q_w, \Sigma, \delta_w, p_0, F_w)$ within $O(m_w \log m_w)$ time.

The algorithm maintains the following data structures: a read-only representation of the input DFA $\mathcal{D}$ supporting navigation queries in time $O(\log m) \subseteq O(\log m_w)$ (an adjacency list representation where the child labeled $a \in \Sigma$ of a given node u can be found by binary search on the adjacency list of u); a queue Δ supporting push and pop operations in constant time; a dynamic set F_w (a self-balancing tree); a simple resizable array implementing $[\cdot]_\cong$; a dynamic data structure supporting updates and queries on $\mathcal{D}_w$ (this structure will include LEX, see below). We represent $\mathcal{D}_w$ using the same approach of Alanko et al. [2]. We sketch the overall idea next and give all implementation details at the end of this section.

The representation of $\mathcal{D}_w$ leverages on the following classic representation of Wheeler automata [12]: any WDFA $\mathcal{D}_w$ with n_w states and m_w transitions can be reconstructed from the four sequences, sorted in Wheeler order, of the nodes' names LEX, incoming labels $\lambda' = \lambda(\mathrm{LEX}[1]), \ldots, \lambda(\mathrm{LEX}[n_w])$ (i.e. $\lambda'[i]$ is the incoming label of $\mathrm{LEX}[i]$), in-degrees `IN` (i.e. `IN`$[i]$ is the in-degree of $LEX[i]$), and outgoing labels `OUT` (i.e. `OUT`$[i]$ is the string formed by all distinct characters labeling the out-going transitions of $LEX[i]$). From such sequences, it is possible to reconstruct the original WDFA by exploiting the following fact:

▶ **Lemma 10** ([12]). *Consider the following total orderings of the transitions δ_w of $\mathcal{D}_w$:*

1. *Sort the transitions (u, a, v) by the Wheeler order of their destinations (i.e. by $\mathrm{LEX}^{-1}[v]$), breaking ties by the Wheeler order of their sources (i.e. $\mathrm{LEX}^{-1}[u]$).*
2. *Sort the transitions (u, a, v) by the Wheeler order of their sources (i.e. $\mathrm{LEX}^{-1}[u]$), breaking ties by their label a.*

Then, for any $a \in \Sigma$, the relative order of transitions labeled a in the orderings is the same.

Suppose we now want to compute $\delta_w(u, a)$ on such a representation $(\mathrm{LEX}, \lambda', \texttt{IN}, \texttt{OUT})$ (an operation which suffices to reconstruct the WDFA). First, we locate the position $i = \mathrm{LEX}^{-1}[u]$ of u in Wheeler order. If $a \notin \texttt{OUT}[i]$ then $\delta_w(u, a) = \perp$ and we are done. Otherwise, we count the number k of nodes in $\mathrm{LEX}[1, \ldots, i]$ having an out-going edge labeled a. In other words, since $\mathcal{D}_w$ is deterministic, (u, v, a) is the k-th transition labeled with a in the total order (2) of Lemma 10. At this point, Lemma 10 tells us that (u, v, a) is the k-th transition labeled with a in the total order (1) as well. We can therefore identify v easily using λ', $\texttt{IN}$, and LEX.

Below we show how to dynamically maintain $(\mathrm{LEX}, \lambda', \texttt{IN}, \texttt{OUT})$ so that a wide range of updates and queries on those sequences can be performed in $O(\log m_w)$ time each. At this point, it is not hard to show that each of the updates and queries on $\mathcal{D}_w$ performed by Algorithm 1 can be implemented in $O(\log m_w)$ time by reducing them to updates and queries on $(\mathrm{LEX}, \lambda', \texttt{IN}, \texttt{OUT})$ (see below). The claimed complexity of Algorithm 1 follows immediately. Computing the spanning tree in Line 10, as well as performing the simple operations in Lines 11-13 takes linear $O(m) \subseteq O(m_w)$ time. At this point note that, in each iteration of the main `while` loop, at least one new transition is added to $\mathcal{D}_w$ (Line 26), while transitions are never deleted from $\mathcal{D}_w$. Each operation in the `while` loop takes $O(\log m_w)$ time (including the `foreach` operation at Line 23 – see below for details, all those renamings of transitions are performed implicitly with $O(1)$ updates to $\texttt{IN}$ and λ'), except the calls to `copy` in Lines 22 and 25. Each call to `copy` may insert several new transitions in Δ and new states and transitions in $\mathcal{D}_w$. The former (new transitions in Δ) will be processed in later iterations of the `while` loop and will be charged to the creation of a new transition of $\mathcal{D}_w$ each; the latter (new states and transitions in $\mathcal{D}_w$) amortize globally to $O(m_w \log m_w)$ time since states and transitions are never removed from $\mathcal{D}_w$.

Implementing Dynamic Data Structures

We describe how all queries and updates on $\mathcal{D}_w$ performed by Algorithm 1 are implemented via queries and updates on the representation $(\mathrm{LEX}, \lambda', \texttt{IN}, \texttt{OUT})$ of $\mathcal{D}_w$.

Data structures. LEX and λ' are implemented with the dynamic string data structure of Nekrich and Navarro [16], representing any sequence S over an integer alphabet in $O(|S|)$ words of space and supporting the following operations in $O(\log |S|)$ time:

- Access any element: $S[i]$;
- Replace any character: given symbol (integer) c and position i, set $S[i] \coloneqq c$;
- $S.\texttt{select}_c(i)$: position of the i-th symbol equal to c in S;
- Insert a new symbol in an arbitrary position of S.

Observe that, since LEX is always a permutation, operation $\mathrm{LEX}^{-1}[q]$ is then solved simply as $\mathrm{LEX}.\texttt{select}_q(1)$. $\texttt{IN}$ is represented using the dynamic searchable partial sum data structure of [18], using $O(|\texttt{IN}|)$ words of space and supporting the following operations in $O(\log |\texttt{IN}|)$:

- Partial sum: compute $\sum_{i=1}^{j} \texttt{IN}[i]$ for any given $j \in \{1, \ldots, |\texttt{IN}|\}$;

- $\texttt{IN}.\texttt{search}(k)$: given integer $k \geq 0$, return the minimum position j such that $\sum_{i=1}^{j} \texttt{IN}[i] \geq k$ (if any; otherwise, return $|\texttt{IN}| + 1$);
- Insert a new integer in an arbitrary position of $\texttt{IN}$;
- Update: given any integers $i \in \{1, \ldots, |\texttt{IN}|\}$ and Δ, update $\texttt{IN}[i] \coloneqq \texttt{IN}[i] + \Delta$.

Finally, $\texttt{OUT}$ is a sequence of sequences. Let $k = |\texttt{OUT}|$. We concatenate all those sequences in a dynamic string $\texttt{OUT}' = \texttt{OUT}[1]\texttt{OUT}[2] \cdots \texttt{OUT}[k]$ represented using the data structure of Nekrich and Navarro [16]. We also keep a dynamic bitvector B (again using the data structure of Nekrich and Navarro [16]) storing the length of those strings in unary. Letting $t_i = |\texttt{OUT}[i]|$, the bitvector is $B = 10^{t_1}10^{t_2}, \ldots, 10^{t_k}$. At this point, it is not hard to see that we can solve the following queries and updates in $O(\log |\texttt{OUT}|)$ time on $\texttt{OUT}$ by reducing them to queries and updates on $\texttt{OUT}'$ and B (we omit the details of such a classic reduction; see, e.g., [2]):

- $\texttt{OUT}.rank_c(i)$: given a character c and a position $i \in [k]$, count the number of symbols equal to c in the strings $\texttt{OUT}[1, \ldots, i]$.
- $\texttt{OUT}.select_c(i)$: given a character c and an integer $i \geq 1$, return the position j such that $\text{LEX}[j]$ is the i-th state in Wheeler order having an out-going transition labeled c. Return 0 if no such state exists.
- Given a character c and a position $i \in [k]$, append c to $\texttt{OUT}[i]$.

Since all manipulated sequences have length $O(m_w)$, all basic operations discussed above, as well as the more complex operations described below, cost $O(\log m_w)$ time.

Creating New States: $\texttt{insert}(q, j)$. This update easily translates to an insert operation in all four components $\text{LEX}, \lambda', \texttt{IN}, \texttt{OUT}$: we insert q in position j of LEX; we insert the special symbol # in position j of λ', signaling that q is created (temporarily) without incoming edges; we insert 0 in position j of $\texttt{IN}$, signaling that the in-degree of q is (temporarily) 0; we insert the empty string ϵ in position j of $\texttt{OUT}$, since q (temporarily) has no out-going edges.

Inserting Transitions Into δ_w. Adding individual transitions (u, a, v) to δ_w (Lines 7 and 26) translates to the following operations. We compute $i_u = \text{LEX}^{-1}[u]$ and $i_v = \text{LEX}^{-1}[v]$; append a to $\texttt{OUT}[i_u]$; increment $\texttt{IN}[i_v] \coloneqq \texttt{IN}[i_v] + 1$; and if $\lambda'[i_v] = \#$, we substitute $\lambda'[i_v] \coloneqq a$.

Evaluating Transition Function. Evaluating the destination v of $\delta_w(u, a)$ for any given state u and character a requires the following operations. We compute $i_u = \text{LEX}^{-1}[u]$; if $\texttt{OUT}[i_u]$ does not contain a (we can discover this with two simple rank queries on $\texttt{OUT}$), then we return $v = \perp$; otherwise, we compute the number t of states before u included (in Wheeler order) having an out-going transition labeled with a: $t = \texttt{OUT}.rank_a(i_u)$; we compute the cumulative in-degrees z of states with an incoming label strictly smaller than a: $z = \sum_{i=1}^{\lambda'.select_a(1)-1} \texttt{IN}[i]$; we obtain (by Lemma 10) the position i_v in Wheeler order of v: $i_v = \texttt{IN}.search(z + t)$; we return $v = \text{LEX}[i_v]$.

$\texttt{pred}(u, a)$ and $\texttt{succ}(u, a)$. We discuss only $\texttt{pred}(u, a)$, as $\texttt{succ}(u, a)$ is symmetric. To solve $\texttt{pred}(u, a)$ we proceed as follows. We retrieve $i_u = \text{LEX}^{-1}[u]$; we compute the number t of states before u excluded (in Wheeler order) having an out-going transition labeled with a: $t = \texttt{OUT}.rank_a(i_u - 1)$ (if $i_u = 1$ or $t = 0$, then we return $\perp$); finally, we return the t-th state in Wheeler order having an out-going transition labeled with a, that is, $\text{LEX}[\texttt{OUT}.select_a(t)]$.

Batch of Transitions Renamings (Line 23 of Algorithm). The goal of this operation is to change the destination of all incoming transitions $(x, a, p) \in \delta_w$ of p such that $u < x$ to s, i.e. renaming those transitions to (x, a, s). Even though this operation renames the destination of several transitions of $\mathcal{D}_w$, we can perform the whole batch of transitions with $O(1)$ updates to `IN` and λ'. This is possible thanks to (i) our WDFA representation, (ii) to the fact that those transitions originally end in the same state p, and (iii) to the fact that after the renaming they end up in two adjacent (in Wheeler order) states $p < s$.

Recall that, at this point of execution, the in-degree of s is 0. Let $i_p = \text{LEX}^{-1}[p]$ and $i_s = i_p + 1$. In particular, $\text{LEX}[i_s] = s$. Let $k = \texttt{IN}[i_p]$. Let d be the number of such transitions $(x, a, p) \in \delta_w$ whose destination has to be renamed to s; the value d can be found as follows. Let y be the largest state in Wheeler order such that $(y, a, p) \in \delta_w$, and let $i_y = \text{LEX}^{-1}[y]$. Let moreover $i_u = \text{LEX}^{-1}[u]$. Then, $d = \texttt{OUT}.rank_a(i_y) - \texttt{OUT}.rank_a(i_u)$. We are left to find i_y. Letting k being the cumulative number of incoming transitions entering in states less than or equal to p in Wheeler order, then $i_y = \texttt{OUT}.select_a(k)$. Finally, $k = \sum_{i=1}^{i_p} \texttt{IN}[i] - \sum_{i=1}^{f_a - 1} \texttt{IN}[i]$, where $\text{LEX}[f_a]$ is the first node in Wheeler order having an incoming transition labeled with a. Integer f_a can be maintained explicitly for every a using, for example, a self-balancing tree (even though this is not necessary as one can compute it with operations on `OUT` and `IN`).

We are only left to perform the actual update, which requires just three operations: (1) $\lambda'[i_s] \coloneqq a$, (2) $\texttt{IN}[i_p] \coloneqq \texttt{IN}[i_p] - d$, and (3) $\texttt{IN}[i_s] \coloneqq \texttt{IN}[i_s] + d = d$.

Operation at Line 24 of Algorithm. This operation boils down to finding the minimum alphabet's character x being larger than or equal to a and labeling some transition in δ_w (a simple self-balancing tree storing those characters can be used here). Then, $\lambda'.select_x(1) - 1$ is the position in LEX of the largest node in Wheeler order having an incoming transition labeled with a character strictly smaller than a.

5 Experiments

We implemented our minimum WDFA algorithm, `MinWheeler` (Algorithm 1), in C++, the source code is available at `https://github.com/regindex/Minimum-WDFA-Constructor`. To evaluate its performance, we compared it with GCSA [14], which, to the best of our knowledge, is the only other available tool capable of computing a Wheeler DFA (termed prefix-sorted automaton in the original work) for a Wheeler language, crucially without providing any minimality guarantee on the size of the output automata. GCSA provides a comprehensive pipeline that starts with a genomic variation file (.VCF) and a reference genome, and produces a final index that supports exact path-matching queries on the resulting pangenome graph. For a fair comparison with `MinWheeler`, we only ran the part of the GCSA pipeline that computes a WDFA from the input DFA, storing the pangenome to be indexed. For our test data, we employed the 23 automata used in the original GCSA benchmarks [19] (Table 2 in [14]), encoding the Finnish subset of frequent mutations from the dbSNP database relative to the human reference chromosomes. Since GCSA operates on the reverse-deterministic automaton, we feed `MinWheeler` the reversed input, so that the two tools compute Wheeler DFAs on the same orientation and their output sizes can be directly compared. 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.

Table 2 shows a summary of our results. From left to right, we report the reference chromosomes, followed by the numbers of nodes and edges for the corresponding input DFAs. We then show the wall-clock time and peak memory (RSS) achieved by the two competing software. Finally, we report the size of the minimum Wheeler DFA (WDFA) computed by

Table 2 Performance and other statistics. Comparison between GCSA and `MinWheeler` DFA construction. Time is expressed in minutes, Space in GB. % increase is calculated as the relative increase of nodes in the Wheeler DFA compared to the initial graph nodes.

Input DFA			GCSA		`MinWheeler`		Minimum WDFA Size		
Chrom.	Nodes	Edges	Time	Space	Time	Space	Nodes	Edges	% increase
Chr 1	250.2M	251.2M	26.1 min	17.1 GB	24.4 min	66.7 GB	273.6M	275.1M	9.43%
Chr 2	244.3M	245.3M	23.8 min	17.1 GB	31.3 min	63.1 GB	274.5M	276.4M	12.52%
Chr 3	198.9M	199.8M	–	–	–	–	>2200M	>2200M	>1100%
Chr 4	192.1M	193.0M	19.8 min	21.6 GB	31.1 min	49.7 GB	226.2M	228.4M	18.05%
Chr 5	181.7M	182.5M	18.2 min	13.2 GB	35.0 min	47.0 GB	216.2M	217.9M	19.19%
Chr 6	172.0M	172.8M	–	–	–	–	>2200M	>2200M	>1200%
Chr 7	159.9M	160.6M	15.6 min	11.6 GB	31.7 min	41.3 GB	188.9M	190.7M	18.44%
Chr 8	147.0M	147.7M	–	–	–	–	>2200M	>2200M	>1400%
Chr 9	141.8M	142.3M	14.0 min	9.7 GB	11.4 min	37.8 GB	153.2M	153.9M	8.10%
Chr 10	136.2M	136.8M	12.8 min	9.5 GB	13.9 min	35.2 GB	151.4M	152.3M	11.25%
Chr 11	135.6M	136.3M	–	–	–	–	>2200M	>2200M	>1600%
Chr 12	134.5M	135.1M	14.1 min	10.8 GB	33.0 min	34.8 GB	177.9M	180.9M	33.09%
Chr 13	115.6M	116.1M	11.2 min	8.1 GB	10.2 min	31.1 GB	127.0M	127.7M	9.93%
Chr 14	107.7M	108.8M	10.1 min	7.2 GB	7.3 min	28.8 GB	113.5M	113.8M	4.99%
Chr 15	103.0M	103.3M	9.8 min	7.1 GB	9.4 min	27.5 GB	112.9M	113.5M	9.74%
Chr 16	90.7M	91.1M	–	–	–	–	>2200M	>2200M	>2000%
Chr 17	81.5M	81.9M	58.9 min	117.2 GB	609.8 min	48.1 GB	1205.3M	1244.8M	1399.5%
Chr 18	78.4M	78.8M	–	–	241.5 min	29.0 GB	654.7M	700.8M	762.28%
Chr 19	59.4M	59.7M	5.4 min	4.5 GB	8.5 min	15.4 GB	70.0M	70.6M	18.05%
Chr 20	63.3M	63.6M	5.6 min	4.6 GB	6.1 min	16.4 GB	69.9M	70.3M	10.48%
Chr 21	48.3M	48.5M	4.1 min	3.4 GB	3.1 min	12.5 GB	51.6M	51.8M	6.82%
Chr 22	51.5M	51.7M	4.3 min	3.6 GB	3.4 min	13.9 GB	55.0M	55.2M	6.78%
Chr X	155.7M	156.1M	14.1 min	10.6 GB	14.3 min	40.2 GB	168.6M	169.3M	8.37%

`MinWheeler` and the resulting percentage increase in size. Dashes indicate cases where the computation did not finish, due to either exceeding the internal memory limit or reaching a 24-hour timeout. In those cases, we report the number of nodes and edges reached by the output WDFA at termination. On every chromosome where GCSA terminates, the WDFA in output has the same size as the minimum WDFA computed by `MinWheeler`, as the near-linear structure of pangenome graphs makes the number of state insertions needed for prefix-sortability essentially independent of the two algorithms. We observe that both software do not terminate when the resulting WDFAs are very large. This behavior is expected, as the algorithms are output-sensitive; thus, their running time scales with the size of the final automata. A clear example is Chromosome 16, where the WDFA under construction reached 2.2 billion nodes within the first 24 hours, resulting in a size increase of over 2000% relative to the input DFA. This indicates that, even if all input DFAs are acyclic and thus recognize a Wheeler language, the size of the minimum WDFA recognizing such a language is not necessarily small.

In terms of performance, `MinWheeler` generally requires more resources than GCSA; specifically, on 16 input instances, our software uses between 2 and 4 times more memory. However, thanks to the incremental construction, which processes one transition at a time, `MinWheeler` does not suffer from the memory peaks observed in the competitor. In all cases where our computation did not finish, it terminated due to reaching the timeout, whereas GCSA always failed due to exceeding the internal memory size. This is evident for chromosomes 17 and 18: for chromosome 17, our software reaches a significantly lower memory peak, while for chromosome 18, it successfully completes construction, whereas

GCSA fails. Regarding running time, our software has proved competitive with GCSA. In particular, GCSA was at least two times faster in only 3 cases; while `MinWheeler` showed better or equivalent running time in 10 other instances, resulting in an average throughput of more than 10^5 transitions/second. This is especially remarkable considering that `MinWheeler` relies on dynamic data structures to maintain the output WDFAs during construction.

In conclusion, we presented a proof-of-concept implementation of an algorithm that computes a minimum WDFA for a Wheeler language. Our experimental results show that our software is competitive with GCSA while providing the additional guarantee of minimum output size. Additionally, we proved more robust in corner cases, successfully completing the WDFA construction of one additional chromosome. This confirms that our algorithm is effective at computing minimum WDFAs for sparse graphs, unlocking the construction of provably optimal-size pattern-matching data structures for pangenomes. On the other hand, our implementation relies on dynamic data structures, which can limit practical performance on large automata. Future work will focus on developing optimized data structures to further improve performance for large-scale applications.

References

1 Jarno Alanko, Nicola Cotumaccio, and Nicola Prezza. Linear-time minimization of wheeler dfas. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2022, Snowbird, UT, USA, March 22-25, 2022*, pages 53–62. IEEE, 2022. doi:10.1109/DCC52660.2022.00013.

2 Jarno N. Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.

3 Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Alberto Policriti, and Nicola Prezza. Optimal wheeler language recognition. In *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 2023. doi:10.1007/978-3-031-43980-3_6.

4 Ruben Becker, Davide Cenzato, Nicola Prezza, and Daniel Puttini. On computing minimum Wheeler DFA from their language, 2026. arXiv:2607.07563.

5 Michael Burrows and David 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.

6 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on wheeler dfas. In *2023 Data Compression Conference (DCC)*, pages 150–159, 2023. doi:10.1109/DCC55655.2023.00023.

7 Nicola Cotumaccio, Giovanna D'Agostino, Daniel Gibney, Alberto Policriti, Nicola Prezza, and Sharma V. Thankachan. Wheeler Graphs and Wheeler Languages. In Paolo Ferragina, Travis Gagie, and Gonzalo Navarro, editors, *The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday*, volume 131 of *Open Access Series in Informatics (OASIcs)*, pages 12:1–12:28, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.Manzini.12.

8 Giovanna D'Agostino, Davide Martincigh, and Alberto Policriti. Ordering regular languages: a danger zone. *arXiv preprint arXiv:2106.00315*, 2021. arXiv:2106.00315.

9 Giovanna D'Agostino, Davide Martincigh, and Alberto Policriti. Ordering regular languages and automata: Complexity. *Theor. Comput. Sci.*, 949:113709, 2023. doi:10.1016/j.tcs.2023.113709.

10 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, Patras, Greece, July 9-12, 2019*, volume 132 of *LIPIcs*, pages 55:1–55:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.55.

11 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.

12 Travis Gagie, Giovanni Manzini, Jouni Sirén, Marinella Sciortino, and Sebastian Wild. Wheeler graphs: A framework for bwt-based data structures. *Theor. Comput. Sci.*, 698:67–78, 2017. doi:10.1016/j.tcs.2017.06.016.

13 Daniel Gibney and Sharma V. Thankachan. On the hardness and inapproximability of recognizing wheeler graphs. In *27th Annual European Symposium on Algorithms, ESA 2019, Munich/Garching, Germany, September 9-11, 2019*, volume 144 of *LIPIcs*, pages 51:1–51:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.51.

14 Veli Mäkinen, Niko Välimäki, and Jouni Sirén. Indexing graphs for path queries with applications in genome research. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 11(2):375–388, 2014. Data: https://www.cs.helsinki.fi/group/gsa/1000gen-FIN-WholeGenome/. doi:10.1109/TCBB.2013.2297101.

15 John Myhill. Finite automata and the representation of events. *WADD Technical Report*, 57:112–137, 1957.

16 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014. doi:10.1137/130908245.

17 Anil Nerode. Linear automaton transformations. *Proc. Am. Math. Soc.*, 9(4):541–544, 1958. doi:10.1090/S0002-9939-1958-0135681-9.

18 Nicola Prezza. A framework of dynamic data structures for string processing. In *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.11.

19 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.

A Proof of Lemma 7

In order to prove Lemma 7, we need the following three simple statements whose proof can be found in the full version of the article [4].

▶ **Lemma 11.** *Let $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ be a DFA.*

(1) *If we modify $\mathcal{D}$ by replacing a transition $\delta(u, a) = v$ with $\delta(u, a) = v'$, for any $v' \equiv v$, then the equivalence relation $\equiv$ and the language $\mathcal{L}(\mathcal{D})$ remain unchanged.*

(2) *Let $u \in Q$. If we modify $\mathcal{D}$ by adding a new state u' to Q, adding a new transition $\delta(u', a) = v$ for every transition $\delta(u, a) = v$, and making u' final if and only if u is final, then the language $\mathcal{L}(\mathcal{D})$ does not change, and $u \equiv u'$ in the modified DFA. Apart from the addition of u', the relation $\equiv$ does not change.*

(3) *Assume $\mathcal{D}$ to be co-accessible and let $q, q' \in Q$ be distinct such that $q \equiv q'$. If we modify $\mathcal{D}$ by replacing a transition (p, a, q) with a transition (p, a, q'), then $\mathcal{D}$ stays co-accessible.*

We are now ready to prove Lemma 7. We start by arguing that the invariants hold initially and then argue that they are maintained during each iteration.

Initialization. Invariants 1, 3, and 5 hold trivially since $\mathcal{D}_w^\Delta = \mathcal{D}$ and $\mathcal{D}_w$ is by construction Wheeler with Wheeler order encoded by LEX. For Invariants 2 and 7, note that by minimality of $\mathcal{D}$, every state in $\mathcal{D}$ is reachable from q_0 and can reach a final state. As initially $\mathcal{D}_w^\Delta = \mathcal{D}$ this shows co-accessibility of $\mathcal{D}_w^\Delta$. Moreover, $\mathcal{D}_w$ is a spanning out-tree of $\mathcal{D}$ rooted in q_0, hence $\mathcal{D}_w$ is accessible. For Invariants 4 and 6, observe that $\mathcal{D} = \mathcal{D}_w^\Delta$ is minimum and no two distinct states of $\mathcal{D}_w^\Delta$ are Myhill-Nerode equivalent. This immediately implies Invariant 6 and also implies that Line 13 sets $\cong$ to be equal to $\equiv$, thus also Invariant 4 holds.

Maintenance. Every iteration of the `while` loop of Algorithm 1 removes a transition (u, a, v) from Δ. As in the algorithm, we let $p \coloneqq \delta(\texttt{pred}(u, a), a)$ and $s \coloneqq \delta(\texttt{succ}(u, a), a)$. The algorithm then modifies $\mathcal{D}_w$ and $\mathcal{D}_w^\Delta$ in one of the following three ways, depending on the branches taken in the two `if` statements in the while loop. We show that the invariants are maintained in each of the three cases. Note that when proving that an Invariant k is maintained we only use Invariants l such that $l < k$.

Case 1: $v \cong p$ or $v \cong s$. Hence, the `if` condition at line 18 holds. The algorithm's actions:

(i) The transition (u, a, v) is removed from Δ.

(ii) The transition (u, a, v') is added to δ_w, where $v' \in \{p, s\}$ is such that $v' \cong v$.

Invariant 1. The invariant is trivially maintained.

Invariant 2. Only Action (ii) modifies $\mathcal{D}_w$. As it adds a transition between existing states, the invariant is trivially maintained.

Invariant 3. By its definition, v' has at least one incoming transition in $\mathcal{D}_w$, hence $v' \neq q_0$ and Invariant 2 implies that q_0 is still the only state with in-degree equal to zero in $\mathcal{D}_w$.
(W1) Action (ii) adds (u, a, v') to δ_w and, by definition of v', $\lambda_{\mathcal{D}_w}(v') = a$ before the action. Hence $\lambda_{\mathcal{D}_w}$ does not change. Since LEX is also unchanged, q_0 still precedes all states in LEX after the action and the Wheeler property W1 is maintained.
(W2) The actions only touch transitions labeled with a, so we can ignore those labeled with other letters. Action (ii) adds (u, a, v') to δ_w, so we have to verify that for all $x \in Q_w$ such that $\delta_w(x, a) = x'$ is defined, $x < u$ implies $x' \leq v'$ (the case "$u < x$ implies $v' \leq x'$" is completely symmetric and we do not treat it). Let x be such that $\delta_w(x, a) = x'$ is defined and $x < u$; if no such state exists, then we are done. We therefore assume that x exists. Note that $x < u$ implies that $\texttt{pred}(u, a)$ exists, therefore $p \coloneqq \delta(\texttt{pred}(u, a), a)$ is defined as well. If s is defined, then $\texttt{pred}(u, a) < \texttt{succ}(u, a)$ and property W2 (before the action) imply that $p \leq s$. Two cases can happen. (i) If $x = \texttt{pred}(u, a)$, then $x' = p \leq s$ (if s is defined; otherwise, $x' = p$) and we are done since $v' \in \{s, p\}$ implies that $x' \leq v'$. (ii) If $x \neq \texttt{pred}(u, a)$, then it must be $x < \texttt{pred}(u, a)$ by the definitions of $\texttt{pred}(u, a)$ and x. Property W2 (before the action) implies $x' = \delta(x, a) \leq \delta(\texttt{pred}(u, a), a) = p$. Also in this case we are done: $v' \in \{s, p\}$ and (if s exists) $p \leq s$ implies $x' \leq v'$.

Invariants 4 and 5. By assumption, before applying the actions $\equiv$ and $\cong$ are the same equivalence relation. After applying the actions, by Lemma 11 (1) the equivalence relation $\equiv$ does not change. Since the actions do not modify $\cong$ nor the language, we conclude that $\equiv$ and $\cong$ are still the same relation, hence Invariants 4 and 5 are maintained.

Invariant 6. By Invariant 3, the automaton $\mathcal{D}_w$ is Wheeler and hence input-consistent after the actions. Since the actions only add one transition to δ_w, we conclude that $\lambda_{\mathcal{D}_w}(v)$ and $\lambda_{\mathcal{D}_w}(v')$ do not change. No other state's incoming transitions are modified, so $\lambda_{\mathcal{D}_w}$ does not change. By Invariant 4, the relation $\equiv$ does not change through the actions either. Additionally, the actions do not modify LEX. Hence Invariant 6 is maintained.

Invariant 7. Actions (i) and (ii) replace the transition (u, a, v) in $\mathcal{D}_w^\Delta$ with the transition (u, a, v'), where $v \cong v'$. By Invariant 4, it holds that $v \equiv v'$ and hence Lemma 11 (3) yields that $\mathcal{D}_w^\Delta$ is still co-accessible after this transformation.

Case 2: $p \ncong v \ncong s$ and $p = s \neq \bot$. In other words, the `if` condition at line 18 does not succeed and the `if` condition at line 21 succeeds. Summary of the algorithm's actions:

- **(i)** The transition (u, a, v) is removed from Δ.
- **(ii)** The state $p = s$ is "split" into two distinct states $p \neq s$ with the same $\cong$-class and same "final" status. This state s is put in LEX in the position immediately following p. For each transition $(p, c, x) \in \Delta$, a new transition (s, c, x) is inserted into Δ. For each transition $(p, c, x) \in \delta_w$, a new transition (s, c, x) is inserted into δ_w.
- **(iii)** Every $(x, a, p) \in \delta_w$ with $u < x$ (in Wheeler order), is replaced (in δ_w) with (x, a, s).
- **(iv)** A new state v' is inserted between p and s (so that $p < v' < s$ are adjacent in LEX), then every outgoing transition $(v, c, x) \in \Delta \cup \delta_w$ of v is copied as (v', c, x) into Δ, and the $\cong$-class of v' and its "final" status are set to be equal to those of v.
- **(v)** The transition (u, a, v') is inserted into δ_w.

Invariant 1. The new states s and v' get the same outgoing transitions as p and v, respectively. No other state gets a new outgoing label in $\mathcal{D}_w^\Delta$, hence the invariant is maintained.

Invariant 2. We show that all states q in $\mathcal{D}_w$ are reachable from q_0 in $\mathcal{D}_w$ after the actions by distinguishing the following three cases: (1) q is a previously existing state with $q \neq p$, (2) $q \in \{p, s\}$, and (3) $q = v'$.

For (1), let $q \neq p$ be a previously existing state different from p and let P be a directed path from q_0 to q in $\mathcal{D}_w$ before the actions. Such a path exists as by assumption every state is reachable from q_0 in $\mathcal{D}_w$ before the actions. Now, note that the only transitions removed from δ_w are the transitions (x, a, p) such that $u < x$ through Action (iii). Hence, if P does not contain any such transition, the path P still exists after the actions and nothing is to be shown. Thus, assume P to contain some transition (x, a, p) and let y be the successor of p on the path P, say through a transition (p, c, y). Action (iii) inserts the transition (x, a, s) into δ_w and Action (ii) inserts the transition (s, c, y) into δ_w. Hence the state q is now reachable from q_0 through the path P' that is obtained by replacing such transitions $(x, a, p), (p, c, y)$ with $(x, a, s), (s, c, y)$. This shows (1). For (2), notice that $\texttt{pred}(u, a) \neq \texttt{succ}(u, a)$ and thus we cannot have $\texttt{pred}(u, a) = p = \texttt{succ}(u, a)$. We assume that $p \neq \texttt{pred}(u, a)$, the case $p \neq \texttt{succ}(u, a)$ is symmetric. In this case $\texttt{pred}(u, a)$ is a previously existing state different from p and by (1) $\texttt{pred}(u, a)$ is reachable from q_0 and then so is p via the transition $(\texttt{pred}(u, a), a, p)$. It remains to argue that s is reachable from q_0. Notice that the transition $(\texttt{succ}(u, a), a, s)$ is contained in δ_w after Action (iii) and thus it suffices to argue that $\texttt{succ}(u, a)$ is reachable from q_0. If $\texttt{succ}(u, a) = p$ this follows from the previous argument, if $\texttt{succ}(u, a) \neq p$, it follows from (1). For (3), note that the new state v' is reachable from q_0 via a path to the state u (that exists due to (1) and (2)) and the transition (u, a, v') that is inserted through Action (v).

Invariant 3. Clearly, q_0 does not get a new incoming transition in δ_w as the only states that get new in-transitions in δ_w are v' and s. Furthermore, by Invariant 2, all states are reachable from q_0 and hence all states different from q_0 have an in-transition. We conclude that q_0 is still the only state without incoming transitions in $\mathcal{D}_w$.

(W1) The actions remove some (but not all) of the incoming transitions of p, hence $\lambda_{\mathcal{D}_w}(p)$ does not change. The two new states $v' < s$ are inserted immediately after p and get new incoming transitions with label a. We conclude that W1 is maintained by the actions.

(W2) The actions only touch transitions labeled with a, so we can ignore those labeled with other letters. There are pre-existing transitions labeled with a – we call them of type (o), and three types of newly inserted transitions labeled with a that the algorithm inserts through the actions: (a) Action (ii) inserts a transition (s, a, x) into δ_w for every transition (p, a, x) in δ_w, (b) Action (iii) inserts a transition (x, a, s) into δ_w for every transition (x, a, p) in δ_w with $u < x$, (c) Action (v) inserts the transition (u, a, v') into δ_w. Let (y, a, y') and (z, a, z') be two transitions after the actions, with $y < z$. We need to prove that $y' \leq z'$. We distinguish cases based on the types (a, b, c, or o) of the two transitions. We denote these cases with (t1/t2), where (t1) is the type of transition (y, a, y') and (t2) the type of transition (z, a, z'). This results in a total of 16 cases.
Out of those 16 cases, 4 cases ((a/a), (b/c), (c/a), (c/c)) cannot occur (e.g., because there is a single transition of type (c) or because they contradict a previous assumption on the order, e.g., $p < u$ implies that (c/a) does not occur. Due to space limitations we omit the case distinction here and point the interested reader to the full version [4].

Invariants 4 and 5. By the invariants, $\cong$ and $\equiv$ are the same relation, and $\mathcal{L}(\mathcal{D}_w^\Delta) = \mathcal{L}(\mathcal{D})$ before the actions. Since we modify the structure of $\mathcal{D}_w^\Delta$ by performing only actions mentioned in Lemma 11 (1) and (2), after the actions $s \equiv p$ and $v' \equiv v$ hold, the relation $\equiv$ does not change apart from the addition of s and v', and the language of $\mathcal{D}_w^\Delta$ does not change. Since Action (ii) sets $[s]_\cong := [p]_\cong$ and Action (iv) sets $[v']_\cong := [v]_\cong$, we also conclude that $\cong$ and $\equiv$ are the same relation after the actions.

Invariant 6. Let u' be a state adjacent to $p = s$ before the actions. By the invariant, if $u' \equiv p = s$, then $\lambda_{\mathcal{D}_w}(u') \neq \lambda_{\mathcal{D}_w}(p)$. The actions split p into two $\equiv$-equivalent adjacent states $p \neq s$ (variable s is renamed) with $\lambda_{\mathcal{D}_w}(p) = \lambda_{\mathcal{D}_w}(s)$, and a new state $v' \equiv v$ is inserted between them: $p < v' < s$. State u' is now adjacent to either p or s; in the former case, it still holds that if $u' \equiv p$, then $\lambda_{\mathcal{D}_w}(u') \neq \lambda_{\mathcal{D}_w}(p)$. The latter case is analogous. Finally, $v' \equiv v \not\equiv p \equiv s$ implies that the invariant is also maintained between v' and its neighbors p, s.

Invariant 7. Let $\mathcal{D}_w^\Delta$ be the DFA before the actions were executed. Consider now the DFA that results from $\mathcal{D}_w^\Delta$ with only Action (ii) executed (in particular this DFA still contains the transition (u, a, v)). As all out-transitions of p in $\mathcal{D}_w^\Delta$ are copied as out-transitions to s, it follows that s can reach the same final state in this DFA as p. Now consider the DFA where in addition Action (iii) was executed. We have $p \cong s$ and by Invariant 4 $p \equiv s$. Lemma 11 (3) implies that after each replacement of a transition (x, a, p) with a transition (x, a, s), the DFA remains co-accessible. Now consider the DFA where in addition Action (iv) was executed. As all out-transitions of v in $\mathcal{D}_w^\Delta$ are copied as out-transitions to v', it follows that v' can reach the same final state in this DFA as v. Finally, consider the DFA after all actions, i.e., assume that in addition the replacement of transition (u, a, v) with transition (u, a, v') was executed through Actions (i) and (v). Note that we have $v \cong v'$ and by Invariant 4 $v \equiv v'$. Lemma 11 (3) then again implies that the DFA after these two actions is co-accessible.

Case 3: $p \ncong v \ncong s$ and $(p \neq s \vee s = \bot)$. In other words, both the `if` conditions at lines 18 and 21 do not succeed. Summary of the algorithm's actions:

(i) The transition (u, a, v) is removed from Δ.

(ii) A new state v' is created and inserted at position i in LEX, where the value of i depends on whether $p = \bot$: if $p \neq \bot$ then $i = 1 + \text{LEX}^{-1}[p]$, else $i := 1 + \max(\{j \ : \ \lambda(\text{LEX}[j]) \prec a\} \cup \{0\})$, i.e., $i - 1$ is the position of the last state in LEX with incoming letter strictly

smaller than a or 0 if all states in LEX have an incoming letter at least a. Then every outgoing transition $(v, c, x) \in \Delta \cup \delta_w$ of v is copied as (v', c, x) into Δ, and the $\cong$-class of v' and its "final" status are set to be equal to those of v.

(iii) The transition (u, a, v') is inserted into δ_w.

Invariant 1. The node v' that is inserted by Action (ii) inherits its out-transitions from v and hence determinism is maintained at v'. Besides this change Action (iii) inserts (u, a, v'), but as Action (i) removes (u, a, v) from Δ, also this change maintains determinism of $\mathcal{D}_w^\Delta$.

Invariant 2. By the invariant, all states that exist before the actions are reachable from q_0 in $\mathcal{D}_w$. These states are still reachable after the actions as no transitions are removed from δ_w. The newly inserted state v' is reachable from u by the (u, a, v') that is inserted into δ_w and u is a state existing previous to the actions and thus reachable as argued before.

Invariant 3. Clearly, q_0 does not get a new incoming transition in δ_w as the only state that gets a new incoming transitions in δ_w is the new state v'. Furthermore, the new state v' does get an incoming transition (u, a, v') (Action (iii)) and no other changes are made to δ_w. We conclude that q_0 is still the only state without incoming transitions in $\mathcal{D}_w$.

(W1) The actions do not remove any transitions and hence $\lambda_{\mathcal{D}_w}(x)$ does not change for any previously existing state x. The position i in LEX of the new state v' depends on whether $p = \bot$. If $p \neq \bot$, Action (ii) sets $i = 1 + \text{LEX}^{-1}[p]$ and thus v' gets inserted immediately after p with $\lambda_{\mathcal{D}_w}(p) = a$. If $p = \bot$, $i - 1$ is the position of the last state in LEX with incoming letter strictly smaller than a or 0 if all states in LEX have an incoming letter at least a. We conclude that W1 is maintained by the actions.

(W2) The actions only touch transitions labeled with a, so we can ignore those labeled with other letters. There is a single newly inserted transition, namely Action (iii) inserts the transition (u, a, v') into δ_w. Let (y, a, y') and (z, a, z') be two transitions after the actions, with $y < z$. We need to prove that $y' \leq z'$. There are two cases (I) both (y, a, y') and (z, a, z') have already existed before the actions, (II) exactly one out of the two transitions is the newly inserted transition (u, a, v').

In case (I), there is nothing to show as the relative order of y, z, y', z' in LEX are not changed. In case (II), let us assume that (y, a, y') is newly inserted (the other case is symmetric), i.e., $y = u$, $y' = v'$, and $u < z$. We have to show that $v' \leq z'$. As (z, a, z') was a previously existing transition and $u < z$, it follows that $\texttt{succ}(u, a) \neq \bot \neq s$. Moreover, by Action (ii), it follows that v' is inserted in LEX before s, i.e., $v' < s$. Furthermore, by definition $\texttt{succ}(u, a) \leq z$. Now, if (1) $\texttt{succ}(u, a) = z$, then $s = z'$ (by determinism) and as v' is inserted in LEX before s, $v' < s = z'$. If however (2) $\texttt{succ}(u, a) < z$, together with the existence of the two transitions $(\texttt{succ}(u, a), a, s)$ and (z, a, z') before the actions we get $s \leq z'$. The fact that v' is inserted in LEX before s yields $v \leq z'$.

Invariants 4 and 5. Before the actions, by Invariant 4 it holds that $\equiv$ and $\cong$ are the same relation. Action (ii) creates a state v' with the same outgoing transitions as v and with its same "final" status. By Lemma 11 (2), $v' \equiv v$ and the relation $\equiv$ remains unchanged, except for the addition of v'. As Action (ii) sets $[v']_\cong = [v]_\cong$, this results in $\equiv$ and $\cong$ being identical again. Finally, by Lemma 11 (1), after the replacement of transition $(u, a, v) \in \Delta$ (by Action (i)) with transition $(u, a, v') \in \delta_w$ (through Action (iii)), the equivalence relation $\equiv$ and the language $\mathcal{L}(\mathcal{D}_w^\Delta)$ do not change. This proves that both invariants are maintained.

Invariant 6. Through the actions, all previously existing states maintain their $\equiv$-equivalence class in $\mathcal{D}_w^\Delta$ (as established above) and their incoming labels in $\mathcal{D}_w$. Action (ii) inserts the new state v' at position i, where i is as described in the action above. We thus only need to check that the property holds (1) between positions $i-1$ and i (for i with $2 \leq i \leq |\,\text{LEX}\,|$)

as well as (2) between positions i and $i+1$ (for i with $1 \leq i \leq |\,\mathrm{LEX}\,| - 1$). We start with statement (1) and consider two cases depending on whether $p = \bot$. If $p \neq \bot$, we have that $\mathrm{LEX}[i-1] = p$. The statement then follows from the fact that $[p]_{\cong} \neq [v]_{\cong} = [v']_{\cong}$ in this case and $\cong$ and $\equiv$ are the same equivalence relation (Invariant 4). Now assume that $p = \bot$. Then by Action (ii), $i-1$ is the position of the last state in LEX with incoming letter strictly smaller than a or 0 if all states in LEX have an incoming letter at least a. As $i \geq 2$, the case $i-1 = 0$ cannot occur, otherwise we have $\lambda_{\mathcal{D}_w}(\mathrm{LEX}[i-1]) \neq \lambda_{\mathcal{D}_w}(\mathrm{LEX}[i])$ and the statement holds. For (2) we consider two cases depending on whether $s = \bot$. If $s \neq \bot$, we have that $\mathrm{LEX}[i+1] = s$. The statement then follows from the fact that $[s]_{\cong} \neq [v]_{\cong} = [v']_{\cong}$ in this case and $\cong$ and $\equiv$ are the same equivalence relation (Invariant 4). Now assume that $s = \bot$. We are left to show that $\lambda_{\Delta_w}(\mathrm{LEX}[i+1]) \neq a$ in this case. We do so by arguing that $\mathrm{LEX}[i+1]$ cannot have an in-transition on letter a. As $s = \bot$, we have $\mathtt{succ}(u, a) = \bot$ and thus there is no state x with $u < x$ that has an outgoing transition with letter a. Hence an in-transition at $\mathrm{LEX}[i+1]$ would have to come from a state x with $x \leq u$. Wheeler axiom (W2) (using Invariant 3) and the existence of $(u, a, v') \in \delta_w$ imply that there is no state x with $x < u$ such that $(x, a, \mathrm{LEX}[i+1])$. Finally, due to determinism (Invariant 1) there is no transition $(u, a, \mathrm{LEX}[i+1])$. Hence $\lambda_{\Delta_w}(\mathrm{LEX}[i+1]) \neq a = \lambda_{\Delta_w}(\mathrm{LEX}[i])$ and this completes the proof.

Invariant 7. Let $\mathcal{D}_w^{\Delta}$ be the DFA before the actions were executed. Consider now the DFA that results from $\mathcal{D}_w^{\Delta}$ with only Action (ii) executed (in particular this DFA still contains the transition (u, a, v)). As all out-transitions of v in $\mathcal{D}_w^{\Delta}$ are copied as out-transitions to v', it follows that v' can reach the same final state in this DFA as v. Now consider the DFA after all actions, i.e., assume that in addition the replacement of transition (u, a, v) with transition (u, a, v') was executed through Actions (i) and (iii). Note that we have $v \cong v'$ and by Invariant 4 $v \equiv v'$. Lemma 11 (3) then implies that the DFA after these two actions is co-accessible.