Regular Languages meet Prefix Sorting *

Jarno Alanko† Giovanna D'Agostino‡ Alberto Policriti§ Nicola Prezza¶

Abstract

Indexing strings via prefix (or suffix) sorting is, arguably, one of the most successful algorithmic techniques developed in the last decades. Can indexing be extended to languages? The main contribution of this paper is to initiate the study of the sub-class of regular languages accepted by an automaton whose states can be prefix-sorted. Starting from the recent notion of *Wheeler graph* [Gagie et al., TCS 2017]—which extends naturally the concept of prefix sorting to labeled graphs—we investigate the properties of *Wheeler languages*, that is, regular languages admitting an accepting Wheeler finite automaton. We first characterize this family as the natural extension of regular languages endowed with the co-lexicographic ordering: the sorted prefixes of strings belonging to a Wheeler language are partitioned into a *finite* number of co-lexicographic *intervals*, each formed by elements from a single Myhill-Nerode equivalence class. We proceed by proving several results related to Wheeler automata: (i) We show that every Wheeler NFA (WNFA) with n states admits an equivalent Wheeler DFA (WDFA) with at most $2n - 1 - |\Sigma|$ states (Σ being the alphabet) that can be computed in $O(n^3)$ time. (ii) We describe a quadratic algorithm to prefix-sort a proper superset of the WDFAs, a $O(n \log n)$-time *online* algorithm to sort acyclic WDFAs, and an optimal linear-time offline algorithm to sort general WDFAs. (iii) We provide a minimization theorem that characterizes the smallest WDFA recognizing the same language of any input WDFA. The corresponding constructive algorithm runs in optimal linear time in the acyclic case, and in $O(n \log n)$ time in the general case. (iv) We show how to compute the smallest WDFA equivalent to *any acyclic DFA* in nearly-optimal time. Our contributions imply new results of independent interest. Contributions (i-iii) provide a new class of NFAs for which the minimization problem can be approximated within a constant factor in polynomial time. Contribution (iv) provides a provably minimum-size solution for the well-studied problem of indexing deterministic-acyclic graphs for linear-time pattern matching queries.

*Partially supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

†University of Helsinki, Finland. jarno.alanko@helsinki.fi

‡University of Udine, Italy. giovanna.dagostino@uniud.it

§University of Udine, Italy. alberto.policriti@uniud.it

¶University of Pisa, Italy. nicola.prezza@di.unipi.it

1 Introduction

Prefix-sorting is the process of ordering the positions of a string in the co-lexicographic order of their corresponding prefixes[1]. Once this step has been performed, several problems on strings become much easier to solve: for example, substrings can be located efficiently in the string without the need to read all of its characters. Given the versatility of this tool, it is natural trying to generalize it to more complex objects such as edge-labeled trees and graphs. For example, a procedure for lexicographically-sorting the states of a finite-state automaton could be useful to speed up subsequent membership queries in its accepting language or its substring/suffix closure; as shown by Backurs and Indyk [2], membership and pattern matching problems on regular languages are hard in the general case. The newborn theory of *Wheeler graphs* [10] provides such a generalization. Intuitively, a labeled graph is Wheeler if and only if its nodes can be co-lexicographically sorted in a total order, i.e. pairwise-distinct nodes are ordered according to (i) their incoming labels or, when those labels are equal, (ii) their predecessors. As a consequence, Wheeler graphs admit indexes for linear-time exact pattern matching queries (also known as *path queries*). Wheeler graphs generalize several lexicographically-sorted structures studied throughout the past decades: indexes on strings [9, 22, 31], sets of strings [24], trees [8], de Bruijn graphs [5], variable-order de Bruijn graphs [29], wavelet trees [14], wavelet matrices [4]. These efforts are part of a more general wave of interest (dating as far back as 27 years ago [23]) towards techniques aimed at solving pattern matching on labeled graphs [1, 6, 8, 10, 18, 23, 27, 29, 30]. As discussed above, existing graph-indexing solutions can only deal with simple labeled graphs. The problem of indexing general (or even just acyclic) graphs with a solution of provably-minimum size remains unsolved. Unfortunately, not all graphs are Wheeler and, as Gibney and Thankachan [12] have recently shown, the problem of recognizing (and sorting) them turns out to be NP-complete even when the graph is acyclic (this in-

[1] Usually, the lexicographic order is used to sort string suffixes. In this paper, we use the symmetric co-lexicographic order of the string's prefixes, and extend the concept to labeled graphs.

cludes, in particular, acyclic NFAs). Even worse, not all regular languages admit an accepting Wheeler finite automaton: the set of *Wheeler languages* is a proper superset of the finite languages and a proper subset of the regular languages [10]. Even when an index is not used, exact pattern matching on graphs is hard: Equi et al. [6] have recently shown that any solution to the problem requires at least quadratic time (under the Orthogonal Vectors hypothesis), even on acyclic DFAs. In particular, this implies that converting an acyclic DFA into an equivalent Wheeler DFA cannot be done in less than quadratic time in the worst case.

The remaining open questions, therefore, are: what are the properties of Wheeler languages? Which class of automata admits polynomial-time prefix-sorting procedures? Can we efficiently build the smallest (that is, with the minimum number of states) prefix-sortable finite-state automaton that accepts a given regular language? These questions are also of practical relevance: as shown in [29], acyclic DFAs recognizing *pan-genomes* (i.e. known variations in the reference genome of a population) can be turned into equivalent WDFAs of the same expected asymptotic size. While the authors do not find the minimum such automaton, their theoretical analysis (as well as experimental evaluation) suggests that the graph-indexing problem is tractable in some real-case scenarios.

1.1 Our Contributions In this paper we provide the following contributions:

1. We show that Wheeler languages are the natural version of regular languages endowed with the co-lexicographic ordering: when sorted, the prefixes of strings belonging to a Wheeler language are partitioned into a *finite* number of *intervals*, each formed by elements from a single Myhill-Nerode equivalence class. In regular languages, those *intervals* are replaced with general *sets*.

2. We show that every Wheeler NFA (WNFA) with n states admits an equivalent Wheeler DFA (WDFA) with at most $2n - 1 - |\Sigma|$ states (Σ being the alphabet) that can be computed in $O(n^3)$ time. This is in sharp contrast with general NFAs (where the blow-up could be exponential).

3. Let d-NFA denote the class of NFAs with at most d equally-labeled edges leaving any state. We show that the problem of recognizing and sorting Wheeler d-NFAs is in P for $d \leq 2$. A recent result from Gibney and Thankachan [12] shows that the problem is NP-complete for $d \geq 5$. Our result almost completes the picture, the remaining open cases being $d = 3$ and $d = 4$.

4. We provide an online incremental algorithm that, when fed with an acyclic Wheeler DFA's nodes in any topological order, can dynamically compute the co-lexicographic rank of each new incoming node among those already processed with just logarithmic delay.

5. We improve the running time of (4) to linear in the offline setting for arbitrary WDFAs.

6. Given a Wheeler DFA $\mathcal{A}$ of size n, we show how to compute, in $O(n \log n)$ time, the smallest Wheeler DFA recognizing the same language as $\mathcal{A}$. If $\mathcal{A}$ is acyclic, running time drops to $O(n)$.

7. Given *any acyclic DFA* $\mathcal{A}$ of size n, we show how to compute, in $O(n + m \log m)$ time, the smallest Wheeler DFA $\mathcal{A}'$, of size m, recognizing the same language as $\mathcal{A}$.

We start in Section 2 with results **1** and **2**: a Myhill-Nerode theorem for Wheeler languages and a linear conversion from WNFAs to WDFAs. Result **1** shows that Wheeler languages are precisely those admitting a "finite interplay" between the co-lexicographic ordering and the Myhill-Nerode equivalence relation (i.e. the relation characterizing general DFAs). Result **2** implies that the WNFA minimization problem admits a polynomial-time 2-approximation. We remark that the NFA minimization problem is notoriously hard (even to approximate within $o(n)$-factor) in the general case [13, 15, 21]. In Section 3 we describe results **3-5**: polynomial-time algorithms for recognizing and sorting Wheeler 2-NFAs. These results generalize to labeled graphs existing prefix-sorting algorithms on strings [25] and labeled trees [8] that have been previously described in the literature. Combined with contribution **2**, our algorithms can be used to index any Wheeler NFA at the price of a moderate linear blow-up (i.e. by just a factor of 2) in the number of states. This result expands the universe of known regular languages for which membership and pattern matching problems can be solved efficiently [2]. Contributions **6** and **7** (Section 4) combine our sorting algorithms **3-5** with DFA minimization techniques to solve the following problem: to compute, given a finite language represented either explicitly by a set of strings or implicitly by an acyclic DFA, the smallest accepting WDFA. Note that this can be interpreted as a technique to index arbitrary deterministic acyclic graphs using the smallest prefix-sortable equivalent automaton.

1.2 Definitions We start by giving a definition of finite-state automata that captures, to some extent, the amount of nondeterminism of the automaton. A d-NFA

is a nondeterministic finite state automaton that has at most d transitions with the same label leaving each state. Note that 1-NFAs correspond to DFAs, while ∞-NFAs correspond to NFAs.

DEFINITION 1.1. *A d-NFA is a quintuple $\mathcal{A} = (V, E, F, s, \Sigma)$, where V is a set of states (or vertices), Σ is the alphabet (or set of labels), $E \subseteq V \times V \times \Sigma$ is a set of directed labeled edges, $F \subseteq V$ is a set of accepting states, and $s \in V$ is a start state (or source). We moreover require that $s \in V$ is the only node with in-degree zero and that for each $u \in V$ and $a \in \Sigma$, $|\{(u, v, a) \in E\}| \leq d$.*

In this paper we work on NFAs without epsilon-transitions. We denote $\sigma = |\Sigma|$. The notation $\mathcal{L}(\mathcal{A})$ indicates the language accepted by $\mathcal{A}$, i.e. the set of all strings labeling paths from s to an accepting state. We assume that each state either is s or is reachable from s. We furthermore require that, for every state v, there exists a path connecting v to some final state. These requirements are not restrictive, since any state that does not satisfy them can be removed without changing $\mathcal{L}(\mathcal{A})$. In particular, note that we allow states with incomplete transition function, i.e. such that the set of labels of their outgoing edges does not coincide with Σ. If state s misses outgoing label a, then any computation following label a from s is considered as non-accepting. In a standard NFA definition, this would be equivalent to having an outgoing edge labeled a to a universal non-accepting node (a sink). We call a d-NFA *acyclic* when the graph (V, E) does not have cycles. We say that a d-NFA is *input-consistent* if, for every $v \in V$, all incoming edges of v have the same label. If the d-NFA is input-consistent, we indicate with $\lambda(v)$, $v \in V$, the label of the incoming edges of v. For the source, we take $\lambda(s) = \# \notin \Sigma$. We will assume that characters in $\Sigma \cup \{\#\}$ are totally ordered by $\prec$ and that $\#$ is minimum. We extend $\prec$ to Σ^* co-lexicographically, still denoting it by $\prec$. On DFAs, we denote by $succ_a(u)$, with $u \in V$ and $a \in \Sigma$, the unique successor of v by label a, when it exists. We define the *size* of an automaton to be the number of its edges.

The notion of *Wheeler graph* generalizes in a natural way the concept of co-lexicographic sorting to labeled graphs:

DEFINITION 1.2. (WHEELER GRAPH) *A triple $G = (V, E, \Sigma)$, where V is a set of vertices and $E \subseteq V \times V \times \Sigma$ is a set of labeled edges, is called a* Wheeler graph *if there is a total ordering $<$ on V such that vertices with in-degree 0 precede those with positive in-degree and for any two edges $(u_1, v_1, a_1), (u_2, v_2, a_2)$ we have (i) $a_1 \prec a_2 \rightarrow v_1 < v_2$, and (ii) $(a_1 = a_2) \wedge (u_1 < u_2) \rightarrow v_1 \leq v_2$.*

Note that the above definition generalizes naturally the concept of prefix-sorting from strings to graphs: two nodes (resp. string prefixes) can be ordered either by looking at their incoming labels (resp. last characters) or, if the labels are equal, by looking at their predecessors (resp. previous prefixes). Considering that, differently from strings and trees, a graph's node can have multiple predecessors, it should be clear that there could exist graphs whose nodes cannot be sorted due to conflicting predecessors: not all labeled graphs enjoy the Wheeler properties. We call a total order of nodes satisfying Definition 1.2 a *Wheeler order* of the nodes and we write *WDFA, WNFA* as a shortcut for *Wheeler DFA, Wheeler NFA*. By property (i), input-consistence is a necessary condition for a graph to be Wheeler. An important property of Wheeler graphs is *path coherence*:

DEFINITION 1.3. (PATH COHERENCE [10]) *An edge-labeled directed graph G is path coherent if there is a total order of the nodes such that for any consecutive range $[i, j]$ of nodes and string α, the nodes reachable from those in $[i, j]$ in $|\alpha|$ steps by following edges whose labels form α when concatenated, themselves form a consecutive range.*

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

2 Wheeler Languages

In this section we collect our basic results on regular languages accepted by automata whose transition relation is a Wheeler graph: Wheeler languages.

DEFINITION 2.1. *Let Σ be a finite set. A language $\mathcal{L}$ is* Wheeler *if and only if $\mathcal{L} = \mathcal{L}(\mathcal{A})$ for a Wheeler NFA $\mathcal{A}$.*

Let us begin with some basic notation. Given a language $\mathcal{L} \subseteq \Sigma^*$ we denote by $Pref(\mathcal{L})$, $Suff(\mathcal{L})$, and $Fact(\mathcal{L})$ the set of prefixes, suffixes, and factors of strings in $\mathcal{L}$, respectively. More formally: $Pref(\mathcal{L}) = \{\alpha : \exists \beta \in \Sigma^* \ \alpha\beta \in \mathcal{L}\}$, $Suff(\mathcal{L}) = \{\beta : \exists \alpha \in \Sigma^* \ \alpha\beta \in \mathcal{L}\}$, $Fact(\mathcal{L}) = \{\alpha : \exists \beta, \gamma \in \Sigma^* \ \gamma\alpha\beta \in \mathcal{L}\}$. Given two states u, v of an NFA $\mathcal{A}$, we denote by $u \rightsquigarrow v$ a path from u to v in $\mathcal{A}$.

DEFINITION 2.2. *If $\mathcal{A} = (V, E, F, s, \Sigma)$ is an NFA, $u \in V$, and $\alpha \in \text{Pref}(\mathcal{L}(\mathcal{A}))$, we define:*

1. $V_\alpha = \{v \mid \alpha \text{ labels } s \rightsquigarrow v\}$,
2. $\text{Pref}(\mathcal{L}(\mathcal{A}))_u := \{\alpha \in \text{Pref}(\mathcal{L}(\mathcal{A})) : \alpha \text{ labels } s \rightsquigarrow u\}$.

Clearly, from the above definition it follows that (i) $V_\alpha \subseteq V$, (ii) $Pref(\mathcal{L}(\mathcal{A}))_u \subseteq Pref(\mathcal{L}(\mathcal{A}))$, and (iii) $u \in V_\alpha$ if and only if $\alpha \in Pref(\mathcal{L}(\mathcal{A}))_u$.

The prefix/suffix property introduced below is going to be crucial to determine the Wheeler ordering among states—when such an ordering exists.

DEFINITION 2.3. *Consider a linear order* $(L, <)$.

1. *An* interval *in* $(L, <)$ *is a* $I \subseteq L$ *such that* $(\forall x, x' \in I)(\forall y \in L)(x < y < x' \rightarrow y \in I)$.
2. *Given* I, J *intervals in* $(L, <)$ *and* $I \subseteq J$, *then:*
   - I *is a* prefix *of* J *if* $(\forall x \in I)(\forall y \in J \backslash I)(x < y)$
   - I *is a* suffix *of* J *if* $(\forall y \in J \backslash I)(\forall x \in I)(y < x)$.
3. *A family* $\mathcal{C}$ *of distinct non-empty intervals in* $(L, <)$ *is said to have the* prefix/suffix property *if, for all* $I, J \in \mathcal{C}$ *such that* $I \subseteq J$, I *is either a prefix or a suffix of* J.

The following lemma will allow us to bound (linearly) the blow-up of the number of states taking place when moving from a WNFA to a WDFA.

LEMMA 2.1. *Let* $(L, <)$ *be a finite linear order of cardinality* $|L| = n$, *and let* $\mathcal{C}$ *be a* prefix/suffix *family of non-empty intervals in* $(L, <)$. *Then:*

1. $|\mathcal{C}| \leq 2n - 1$.
2. *The upper bound is tight: for every* n, *there exists a prefix/suffix family of size* $2n - 1$.

Proof. Let us order the elements of L by the relation $<$, and let us denote by $L[i]$ the i-th element in the ordering. The notation $L[i, j]$, with $j \geq i$, denotes the interval $\{L[k] : i \leq k \leq j\}$. In particular, $L[1, n] = L$.

By assumption, $\mathcal{C}$ is a family of intervals over L enjoying the prefix/suffix property. We say that an interval $I \in \mathcal{C}$ is *maximal* if I is not the prefix nor the suffix of any other interval in $\mathcal{C}$. We say that an interval $I \in \mathcal{C}$ is *prefix* (resp. *suffix*) if I is the *proper* prefix (resp. suffix) of a maximal interval of $\mathcal{C}$. Note that, by this definition, intervals of $\mathcal{C}$ are either maximal or prefix/suffix. Note also that there could be elements of $\mathcal{C}$ being both prefix and suffix intervals.

(1) We first prove that (1.i) $\mathcal{C}$ contains at most n prefix intervals, then (1.ii) slightly improve this bound to $n - 1$, and finally (1.iii) show that the sum between the number of maximal and suffix intervals is at most n. To prove (1.i), we show that every $L[j]$, $1 \leq j \leq n$, can be the largest element of at most one prefix interval. In turn, this is shown by considering the prefixes of any two pairwise distinct maximal intervals of $\mathcal{C}$. Consider two distinct maximal intervals $I = L[i, j]$ and $J = L[i', j']$. If I and J do not overlap (i.e. $j < i'$ or $j' < i$), then the property is trivially true: if I' and J' are prefixes of I and J, respectively, then $\max(I') \neq \max(J')$. Consider now the case where I and J overlap. Without loss of generality, we can assume $i < i' \leq j < j'$ (the strict inequalities follow from the fact that, by maximality, it cannot be $i = i'$ or $j = j'$). Assume, for contradiction, that $\mathcal{C}$ contains two intervals $L[i, j'']$ and $L[i', j'']$ such that $i' \leq j'' < j$, i.e. $L[i, j'']$ and $L[i', j'']$ are (proper) prefixes of I and J, respectively, that share their largest element j''. Then, we have $i < i' \leq j'' < j$: interval $L[i', j'']$ is strictly contained inside $L[i, j] \in \mathcal{C}$ (i.e. $L[i', j''] \subset L[i, j]$) and it is not a prefix nor a suffix of it. This is forbidden by the definition of prefix/suffix family. From this contradiction we deduce that any two distinct prefix intervals $I', J' \in \mathcal{C}$ satisfy $\max(I') \neq \max(J')$, which implies that $\mathcal{C}$ contains at most n prefix intervals.

To improve the above bound to $n - 1$ and prove (1.ii), consider the rightmost maximal interval $I = L[i, j]$, i.e. the one having largest j. We show that j cannot be the maximum element of any prefix interval. Assume, for contradiction, that such a prefix interval $K = L[i', j]$ exists. Then, the corresponding maximal interval $J = L[i', j']$ of which K is a proper prefix satisfies $j' > j$. This contradicts the fact that I is the rightmost maximal interval.

The next step is to prove (1.iii), i.e. that the sum between the number of maximal and suffix intervals is at most n. We proceed by induction on the number M of maximal intervals. If $M = 1$, then the unique maximal interval $I = L[i, j]$ contains at most $j - i$ suffix intervals. In total, $\mathcal{C}$ contains at most $1 + (j - i) \leq n$ maximal and suffix intervals. For $M > 1$, consider the maximal interval $I = L[i, j]$ with minimum j (call it the "leftmost"). Now, consider the immediate maximal "successor" $J = L[i', j']$ of I, i.e. the maximal interval with the smallest endpoint $j' \geq j$. Clearly, such j' satisfies $j' > j$, otherwise J would be a suffix of I (contradicting maximality of J). Note that it must also be the case that $i' > i$: if $i = i'$, then I would be a prefix of J (contradicting maximality of I); on the other hand, if $i' < i$ then I would be strictly contained in J, contradicting the definition of prefix/suffix family. We are left with two cases:

(a) $i \leq j < i' \leq j'$. In this case, I and J are disjoint. As seen above, I contributes to at most one maximal interval (I itself) and $j - i$ suffix intervals. In total, I contributes to at most $j - i + 1$ maximal and suffix intervals. We are left to count the number of maximal and suffix intervals in the remaining portion of the linear order $L[i', ..., n]$. Note that there are no other intervals to be considered: if $L[i'', j'']$ is a maximal interval in $\mathcal{C}$, different from I, J, then $j'' > j'$ and hence $i'' > i'$ or $L[i'', j'']$ would contain J. The portion

$L[i', ..., n]$ contains $M-1$ maximal intervals, so we can apply the inductive hypothesis and obtain that this segment contains at most $n-i'+1$ maximal and suffix intervals. In total, we have that $L[1, ..., n]$ contains at most $(j-i+1)+(n-i'+1)$ maximal and suffix intervals. Since $i' > j$ and $i \geq 1$, this quantity is at most n.

(b) $i < i' \leq j < j'$. Denote by $k = i'-i$ the number of L's elements belonging to $I \setminus J$. Then, $\mathcal{C}$ can contain at most k proper suffixes of I: $L[i+1, j]$, $L[i+2, j]$, ..., $L[i', j]$. All other suffixes of I are strictly contained inside J, and cannot belong to $\mathcal{C}$ due to the prefix/suffix property. Actually, one of those suffixes, $L[i', j]$, is a prefix of J so it has already been counted above in points (1.i) and (1.ii). We are left with $k-1$ suffixes to take into account, plus the maximal interval I itself: in total, $k = i'-i$ maximal and suffix intervals. As noted above, all remaining maximal and suffix intervals of $\mathcal{C}$ to take into account are those contained in $L[i', n]$. Since $L[i', n]$ contains $M-1$ maximal intervals, we can apply the inductive hypothesis and deduce that it contains at most $n-i'+1$ maximal and suffix intervals. In total, $L[1, n]$ contains therefore at most $(i'-i)+(n-i'+1) \leq n$ maximal and suffix intervals. This concludes the proof of the upper bound $|\mathcal{C}| \leq 2n-1$.

(2) Consider the prefix/suffix family containing just one maximal interval and all its proper prefixes and suffixes: $\mathcal{C} = \{L[1, n], L[1, 1], \ldots, L[1, n-1], L[2, n], \ldots, L[n, n]\}$. This family satisfies $|\mathcal{C}| = 2n-1$. ∎

Definition 2.4. *Let $\mathcal{C}$ be a family of non-empty intervals of a linear order $(L, <)$ having the prefix/suffix property. Let $<^i$ (or simply $<$) be the binary relation over $\mathcal{C}$ defined by*

$$I <^i J \quad \text{if and only if} \quad (\exists x \in I)(\forall y \in J)(x < y) \vee (\exists y \in J)(\forall x \in I)(x < y).$$

The following lemma is easily proved.

Lemma 2.2. *Let $\mathcal{C}$ be a family of non-empty intervals of a linear order $(L, <)$ having the prefix/suffix property, then $(\mathcal{C}, <^i)$ is a linear order.*

Proof. We just prove transitivity when $I <^i J$ and $J <^i K$ are witnessed by $x_0 \in I$ satisfying $(\forall y \in J)(x_0 < y)$, and $z_0 \in K$ satisfying $(\forall y \in J)(y < z_0)$, respectively (the other cases are similar). We claim that $z_0 > x$, for all $x \in I$. Suppose, for contradiction, that there exists $x_1 \in I$ with $z_0 \leq x_1$; then, from $x_0 < y < z_0 \leq x_1$ for all $y \in J$ and the fact that I is an interval, it follows that $z_0 \in I$, $J \subseteq I$, so that, by prefix/suffix property of $\mathcal{C}$, J is either a prefix or a suffix of I. Since $x_0 < y$ for all $y \in J$, we see that J must be a suffix of I. Knowing that $z_0 \in I$, this implies $z_0 \in J$. A contradiction. ∎

Note that whenever the linear order $(L, <)$ is finite, any non-empty interval I has minimum m_I and maximum M_I. In this special case, the above order $<^i$ can be equivalently described on a family having the prefix/suffix property, by: $I <^i J$ if and only if $(m_I < m_J) \vee [(m_I = m_J) \wedge (M_I < M_J)]$.

We now have the basics to start our study of Wheeler languages. In this section, we use $V, E, F, s, \Sigma, <$ to denote the set of states, edges, final states, initial state, alphabet, and Wheeler order of a generic WNFA. The key property of path-coherence will be re-proved below—in Lemma 2.4—, together with what we may call a sort of its "dual", that is, the the set of strings read while reaching a given state is an interval. More precisely, if $\mathcal{A}$ is a WNFA, $u \in V$, and $\alpha \in \Sigma^*$, we have that V_α is an interval (I_α, from now on) in $(V, <)$, $\mathit{Pref}(\mathcal{L}(\mathcal{A}))_u$ is an interval (I_u, from now on), in $(\mathit{Pref}(\mathcal{L}(\mathcal{A})), \prec)$ and

$$\alpha \in I_u \text{ if and only if } u \in I_\alpha.$$

Preliminary to our result are the following lemmas, exploiting the interval-structure of both Wheeler languages and automata.

Lemma 2.3. *If $\mathcal{A}$ is a WNFA, $u, v \in V$ are states, and $\alpha, \beta \in \mathit{Pref}(\mathcal{L}(\mathcal{A}))$, then:*

1. *if $\alpha \in I_u, \beta \in I_v$, and $\{\alpha, \beta\} \not\subseteq I_v \cap I_u$, then $\alpha \prec \beta$ if and only if $u < v$;*
2. *if $u \in I_\alpha, v \in I_\beta$, and $\{u, v\} \not\subseteq I_\beta \cap I_\alpha$, then $\alpha \prec \beta$ if and only if $u < v$.*

Proof. (1) Suppose $\alpha \in I_u, \beta \in I_v$ and $\{\alpha, \beta\} \not\subseteq I_v \cap I_u$. From this we have that $\alpha \in I_u \setminus I_v$ or $\beta \in I_v \setminus I_u$, hence $u \neq v$ and $\alpha \neq \beta$ follows.

If $u = s$ or $v = s$, either α or β is the empty string ϵ and the result follows easily. Hence, we suppose $u \neq s \neq v$ and (hence) $\alpha \neq \epsilon \neq \beta$.

To see the left-to-right implication, assume $\alpha \prec \beta$: we prove that $u < v$ by induction on the maximum between $|\alpha|$ and $|\beta|$. If $|\alpha| = |\beta| = 1$, then the property follows from the Wheeler-(i). If $\max(|\alpha|, |\beta|) > 1$ and α and β end with different letters, then again the property follows from Wheeler-(i). Hence, we are just left with the case in which $\alpha = \alpha' e$ and $\beta = \beta' e$, with $e \in \Sigma$. If $\alpha \prec \beta$, then $\alpha' \prec \beta'$. Consider states u', v' such that $\alpha' \in I_{u'}, \beta' \in I_{v'}$, and $(u', u, e), (v', v, e) \in E$. Then $\alpha' \in I_{u'} \setminus I_{v'}$ or $\beta' \in I_{v'} \setminus I_{u'}$ because otherwise we would have $\alpha' \in I_{v'}$ and $\beta' \in I_{u'}$ which imply respectively $\alpha \in I_v$ and $\beta \in I_u$. By induction we have $u' < v'$ and therefore, by Wheeler-(ii), $u \leq v$. From $u \neq v$ it follows $u < v$.

Conversely, for the right-to-left implication, suppose $u < v$. Since $\alpha \neq \beta$, if it were $\beta \prec \alpha$ then, by the above, we would have $v < u$: a contradiction. Hence, $\alpha \prec \beta$ holds.

(2) By definition, $\alpha \in I_u$ if and only if $u \in I_\alpha$ and $\beta \in I_v$ if and only if $v \in I_\beta$. Hence, the hypothesis that $u \in I_\alpha, v \in I_\beta$ and $\{u, v\} \not\subseteq I_\beta \cap I_\alpha$, is equivalent to say that $\alpha \in I_u, \beta \in I_v$ and $\{\alpha, \beta\} \not\subseteq I_v \cap I_u$. Therefore, (2) follows from (1). ∎

Let $I_V = \{I_u : u \in V\}$ and $I_{Pref(\mathcal{L}(\mathcal{A}))} = \{I_\alpha : \alpha \in Pref(\mathcal{L}(\mathcal{A}))\}$.

LEMMA 2.4. *If $\mathcal{A}$ is a WNFA and $\mathcal{L} = \mathcal{L}(\mathcal{A})$, then:*

1. *for all $u \in V$, the set I_u is an interval in $(Pref(\mathcal{L}(\mathcal{A})), \prec)$;*
2. *I_V is a prefix/suffix family of intervals in $(Pref(\mathcal{L}(\mathcal{A})), \prec)$;*
3. *for all $\alpha \in Pref(\mathcal{L}(\mathcal{A}))$, the set I_α is an interval in $(V, <)$;*
4. *$I_{Pref(\mathcal{L}(\mathcal{A}))}$ is a prefix/suffix family of intervals in $(V, <)$.*

Proof. (1) Suppose $\alpha \prec \beta \prec \gamma$ with $\alpha, \gamma \in I_u$ and $\beta \in Pref(\mathcal{L}(\mathcal{A}))$; we want to prove that $\beta \in I_u$. From $\beta \in Pref(\mathcal{L}(\mathcal{A}))$ it follows that there exists a state v such that $\beta \in I_v$. Suppose, for contradiction, that $\beta \notin I_u$. Then $\beta \in I_v \setminus I_u$ and from $\alpha \prec \beta$ and Lemma 2.3, it follows $u < v$. Similarly, applying again Lemma 2.3, from $\beta \prec \gamma$ we have $v < u$, which is a contradiction.

(2) Suppose, for contradiction, that $I_u, I_v \in I_V$ are such that $I_u \subsetneq I_v$ and I_u is neither a prefix nor a suffix of I_v. In these hypotheses there must exist $\alpha, \alpha' \in I_v \setminus I_u$ and $\beta \in I_u$ such that $\alpha \prec \beta \prec \alpha'$. Lemma 2.3 implies $v < u < v$, which is a contradiction.

(3), (4) follow similarly from Lemma 2.3. ∎

From Lemma 2.2 it follows that both $(I_v, \prec^i)$ and $(I_{Pref(\mathcal{L}(\mathcal{A}))}, <^i)$ are linear orders. The link between such orders is made explicit below.

LEMMA 2.5. *Consider $I_u, I_v \in I_V$ and $I_\alpha, I_\beta \in I_{Pref(\mathcal{L}(\mathcal{A}))}$.*

1. *$I_u \prec^i I_v$ implies that $u < v$ and $u < v$ implies that $I_u \preceq^i I_v$*
2. *$I_\alpha <^i I_\beta$ implies that $\alpha \prec \beta$ and $\alpha \prec \beta$ implies that $I_\alpha \leq^i I_\beta$*

If $\mathcal{A}$ is a WNFA we can prove that the following interval construction—which is the analogous of the power-set construction for NFAs—allows determinization.

DEFINITION 2.5. *If $\mathcal{A}$ is a WNFA we define its (Wheeler)* determinization *as the automaton $\mathcal{A}^d = (V^d, E^d, F^d, s^d, <^d, \Sigma)$, where:*

- *$V^d = I_{Pref(\mathcal{L}(\mathcal{A}))}$;*
- *$s^d = I_\epsilon = \{s\}$*
- *$F^d = \{I_\alpha \mid \alpha \in \mathcal{L}(\mathcal{A})\}$;*
- *E^d is the set of triples $(I_\alpha, I_{\alpha e}, e)$, for all $e \in \Sigma$ and $\alpha e \in Pref(\mathcal{L}(\mathcal{A}))$;*
- *$<^d = <^i$.*

The bound proved in Lemma 2.1 can be slightly improved in the special case of prefix/suffix families corresponding to WNFA intervals.

LEMMA 2.6. (WNFA DETERMINIZATION) *If $\mathcal{A}$ is a WNFA with n states over an alphabet Σ, then $\mathcal{A}^d$ is a WDFA with at most $2n - 1 - |\Sigma|$ states, and $\mathcal{L}(\mathcal{A}^d) = \mathcal{L}(\mathcal{A})$.*

Proof. The verification that $\mathcal{L}(\mathcal{A}^d) = \mathcal{L}(\mathcal{A})$ follows the same lines of the proof in the classical regular case. We prove that $<^d$ is a Wheeler order on the states of the automaton $\mathcal{A}^d$. By Lemma 2.4, the set $V^d = I_{Pref(\mathcal{L}(\mathcal{A}))}$ of states of $\mathcal{A}^d$ is a prefix/suffix family of intervals, so that, by Lemma 2.2, $<^d$ is a linear order on V^d. Next, we check the Wheeler properties. The only vertex with in-degree 0 is I_ϵ, and it clearly precedes those with positive in-degree. For any two edges $(I_\alpha, I_{\alpha a_1}, a_1)$, $(I_\beta, I_{\beta a_2}, a_2)$ we have:

(i) if $a_1 \prec a_2$ then $\alpha a_1 \prec \beta a_2$, and from Lemma 2.5 it follows $I_{\alpha a_1} \leq^d I_{\beta a_2}$. Moreover, by the input consistency of $\mathcal{A}$, states in $I_{\alpha a_1}$ are a_1-states, while states in $I_{\beta a_2}$ are a_2-states; hence $I_{\alpha a_1} \neq I_{\beta a_2}$, so that $I_{\alpha a_1} <^d I_{\beta a_2}$ follows.

(ii) If $a = a_1 = a_2$ and $I_\alpha < I_\beta$, from Lemma 2.5 it follows $\alpha \prec \beta$, so that $\alpha a \prec \beta a$ and, using again Lemma 2.5, we obtain $I = I_{\alpha a} \leq^i I = I_{\beta a}$.

Finally, we prove that $|V^d| \leq 2n - 1 - |\Sigma|$. By the Wheeler properties, we know that the only interval in $I_{Pref(\mathcal{L}(\mathcal{A}))}$ containing the initial state s of the automaton $\mathcal{A}$ is $\{s\}$ and that the remaining intervals can be partitioned into $|\Sigma|$-classes, by looking at the letter labelling incoming edges. If $\Sigma = \{a_1, \ldots, a_k\}$, and, for every $i = 1, \ldots, k$, we denote by m_i the number of states of the automaton $\mathcal{A}$ whose incoming edges are labeled a_i, we have $\sum_{i=1}^k m_i = n - 1$. Using Lemma 2.1 we see that the intervals in V^d composed by a_i states are at most $2m_i - 1$, so that the total number of intervals in V^d is at most $1 + \sum_{i=1}^k (2m_i - 1) = 1 + 2(\sum_{i=1} m_i) - k = 1 + 2(n-1) - k = 2n - 1 - k = 2n - 1 - |\Sigma|$. ∎

LEMMA 2.7. (COMPUTING THE DETERMINIZATION) *If $\mathcal{A}$ is a WNFA with n states, then $\mathcal{A}^d$ can be computed in $O(n^3)$ time.*

Proof. We apply the standard powerset construction algorithm starting from the original WNFA $\mathcal{A}$. By Lemma 2.6, the powerset algorithm does not generate more than $2n-1-|\Sigma|$ distinct sets of states. Remember that such algorithm starts from the set $\{s\}$ containing the NFA's source and simulates a visit of the final DFA, whose states are represented as sets of states of the original NFA. At each step, the successor with label $a \in \Sigma$ of a set K is computed by calculating all the a-successors of states in K, and taking their union. In the worst case, $|K| = O(n)$ and each state in K has $O(n)$ a-successors. After having obtained the a-successor K' of K, we need to check if K' had already been visited. Since K''s cardinality is at most n, this operation takes $O(n)$ time using a standard dictionary (e.g. a hash table). Overall, we spend $O(n^2)$ time to simulate an edge traversal of the final DFA. By Lemma 2.6, we visit at most $O(n)$ distinct sets of states. Overall, the powerset algorithm's complexity is $O(n^3)$. ∎

Lemma 2.6 above— saying that we can restrict the automata recognizing Wheeler Languages to deterministic ones without an exponential blow up—marks a difference between the standard and the Wheeler case for regular languages and can be seen as the first step in the study of Wheeler Languages. Further differences can be observed. For example, the reader can check that the language $\mathcal{L}(\mathcal{A}) = \mathcal{L} = b^+a$ is accepted by *incomplete* WDFAs only.

The subsequent step to take in a theory of Wheeler Languages is a Myhill-Nerode like theorem for this class. To this end, we define:

DEFINITION 2.6. *Given a language $\mathcal{L} \subseteq \Sigma^*$, an equivalence relation $\sim$ over* $\mathrm{Pref}(\mathcal{L})$ *is:*

- right invariant*, when for all $\alpha, \beta \in \mathrm{Pref}(\mathcal{L})$ and $\gamma \in \Sigma^*$: if $\alpha \sim \beta$ and $\alpha\gamma \in \mathrm{Pref}(\mathcal{L})$, then $\beta\gamma \in \mathrm{Pref}(\mathcal{L})$ and $\alpha\gamma \sim \beta\gamma$;*
- convex *if $\sim$-classes are intervals of* $(\mathrm{Pref}(\mathcal{L}), \prec)$*;*
- input consistent *if all words belonging to the same $\sim$-class end with the same letter.*

Consider a language $\mathcal{L} \subseteq \Sigma^*$. The Myhill-Nerode equivalence $\equiv_{\mathcal{L}}$ among words in $Pref(L)$ is defined as

$$\alpha \equiv_{\mathcal{L}} \beta \text{ if and only if } (\forall \gamma \in \Sigma^*)(\alpha\gamma \in \mathcal{L} \Leftrightarrow \beta\gamma \in \mathcal{L}).$$

DEFINITION 2.7. *The input consistent, convex refinement $\equiv^c_{\mathcal{L}}$ of $\equiv_{\mathcal{L}}$ is defined as follows. For all $\alpha, \beta \in$ Pref($\mathcal{L}$), $\alpha \equiv^c_{\mathcal{L}} \beta$ is true if and only if all the following three conditions hold:*

(i) $\alpha \equiv_{\mathcal{L}} \beta$,

(ii) $end(\alpha) = end(\beta)$, *and*

(iii) $(\forall \gamma \in \mathrm{Pref}(\mathcal{L}))(\min\{\alpha,\beta\} \prec \gamma \prec \max\{\alpha,\beta\} \rightarrow \gamma \equiv_{\mathcal{L}} \alpha)$,

where $end(\alpha)$ is the final character of α when $\alpha \neq \epsilon$, and ϵ otherwise.

Using the above results in this section we can prove:

THEOREM 2.1. *Given a language $\mathcal{L} \subseteq \Sigma^*$, the following are equivalent:*

1. *$\mathcal{L}$ is a Wheeler language (i.e. $\mathcal{L}$ is recognized by a WNFA).*
2. *$\equiv^c_{\mathcal{L}}$ has finite index.*
3. *$\mathcal{L}$ is a union of classes of a convex, input consistent, right invariant equivalence over Pref($\mathcal{L}$) of finite index.*
4. *$\mathcal{L}$ is recognized by a WDFA.*

Proof. (1) ⇒ (2). If $\mathcal{A}$ is a Wheeler NFA such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$, consider the following equivalence relation $\sim_{\mathcal{A}}$ over *Pref*($\mathcal{L}$):

$$\alpha \sim_{\mathcal{A}} \beta \quad \Leftrightarrow I_\alpha = I_\beta.$$

Using the fact that the I_α are intervals (see Lemma 2.4), and other properties of Wheeler automata, one can easily prove that the equivalence $\sim_{\mathcal{A}}$ is a refinement of $\equiv^c_{\mathcal{L}}$, so that each $\equiv^c_{\mathcal{L}}$-class is a union of $\sim_{\mathcal{A}}$-classes. Moreover, the equivalence $\sim_{\mathcal{A}}$ has finite index, bounded by the number of intervals I_α, hence $\equiv^c_{\mathcal{L}}$ has finite index as well.

(2) ⇒ (3). We prove that the relation $\equiv^c_{\mathcal{L}}$ is a convex, input consistent, right invariant equivalence, and that $\mathcal{L}$ is a union of $\equiv^c_{\mathcal{L}}$-classes; this last property is true simply because $\mathcal{L}$ is a union of $\equiv_{\mathcal{L}}$-classes and $\equiv^c_{\mathcal{L}}$ is a refinement of $\equiv_{\mathcal{L}}$. The fact that $\equiv^c_{\mathcal{L}}$ is convex and input consistent follows directly from its definition. We prove that $\equiv^c_{\mathcal{L}}$ is right invariant. Suppose $\alpha, \alpha', \gamma \in$ *Pref*($\mathcal{L}$) and $\alpha \equiv^c_{\mathcal{L}} \alpha'$. Note that if $\alpha\gamma \in$ *Pref*($\mathcal{L}$) then there exists $\nu \in \Sigma^*$ such that $\alpha\gamma\nu \in \mathcal{L}$, so that $\alpha'\gamma \in$ *Pref*($\mathcal{L}$) follows from $\alpha \equiv_{\mathcal{L}} \alpha'$. Hence, we are left to prove that $\alpha\gamma \equiv^c_{\mathcal{L}} \alpha'\gamma$. We easily prove the following:

- $\alpha\gamma \equiv_{\mathcal{L}} \alpha'\gamma$ (it follows from $\alpha \equiv_{\mathcal{L}} \alpha'$).
- If $\alpha\gamma \prec \beta' \prec \alpha'\gamma$, for $\beta' \in$ *Pref*($\mathcal{L}$), then $\beta' \equiv_{\mathcal{L}} \alpha\gamma$: from $\alpha\gamma \prec \beta' \prec \alpha'\gamma$ it follows that $\beta' = \beta\gamma$, for some $\beta \in$ *Pref*($\mathcal{L}$), and $\alpha \prec \beta \prec \alpha'$. Since α, α' belong to the same $\equiv^c_{\mathcal{L}}$ class, then $\beta \equiv_{\mathcal{L}} \alpha$, and $\beta\gamma \equiv_{\mathcal{L}} \alpha\gamma$ follows from the right invariance of $\equiv_{\mathcal{L}}$.

Since $\alpha\gamma, \beta\gamma$ end with the same letter, the previous points imply that $\alpha\gamma \equiv^c_{\mathcal{L}} \alpha'\gamma$ and $\equiv^c_{\mathcal{L}}$ is right invariant.

(3) $\Rightarrow$ (4). Suppose $\mathcal{L}$ is a union of classes of a convex, input consistent, right invariant equivalence relation $\sim$ of finite index. We build a WDFA $\mathcal{A}_\sim = (V_\sim, E_\sim, F_\sim, s_\sim, \Sigma, <_\sim)$ such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$ as follows:

- $V_\sim = \{[\alpha]_\sim \mid \alpha \in \mathit{Pref}(\mathcal{L})\}$;
- $s_\sim = [\epsilon]_\sim$ (note that, by input consistency, $[\epsilon]_\sim = \{\epsilon\}$);
- $(I, J, e) \in E_\sim$ if and only if $Ie \cap \mathit{Pref}(\mathcal{L}) \neq \emptyset$ and $Ie \subseteq J$, where $Ie = \{\alpha e \mid \alpha \in I\}$ (note that J, if existing, is unique because $\sim$-classes are disjoint);
- $F_\sim = \{I \mid I \subseteq \mathcal{L}\}$;
- $<_\sim = \prec^i$ (being pairwise disjoint and convex, the classes in $V_\sim$ form a prefix/suffix family of intervals of $(\mathit{Pref}(\mathcal{L}), \prec)$).

Note that all words in $\mathit{Pref}(\mathcal{L})$ label a computation in $\mathcal{A}_\sim$. We claim that, for all $\sim$-class I and $\alpha \in \mathit{Pref}(\mathcal{L})$:

$$\alpha \in I \quad \Leftrightarrow \quad s_\sim \rightsquigarrow I \text{ in } \mathcal{A}_\sim \text{ reading } \alpha.$$

We prove the implication from right to left by induction on the length of $\alpha \in \mathit{Pref}(\mathcal{L})$.

If $\alpha = \epsilon$ then the claim follows from the definition of $s_\sim$.

If $\alpha = \alpha' e \in \mathit{Pref}(\mathcal{L})$ with $e \in \Sigma$, then $\alpha' \in \mathit{Pref}(\mathcal{L})$. Then, if $K \in V_\sim$ is such that $s_\sim \rightsquigarrow K$ reading α' in $\mathcal{A}_\sim$, by induction we know that $\alpha' \in K$. Since $\alpha = \alpha' e \in Ke$, we have $Ke \cap \mathit{Pref}(\mathcal{L}) \neq \emptyset$; by right invariance of $\sim$ there exists a unique J such that $Ke \subseteq J$. From $\alpha = \alpha' e \in Ke \subseteq J$ it follows $\alpha \in J$, and also $J = I$, because $\mathcal{A}_\sim$ is a deterministic automaton and $s_\sim \rightsquigarrow I$, $s_\sim \rightsquigarrow J$, both by reading α.

In order to prove the implication from left to right of the claim, suppose $\alpha \in I$, and $J \in V_\sim$ is such that $s_\sim \rightsquigarrow J$ in $\mathcal{A}_\sim$ reading α. Then, by the first part of the proof of the claim we obtain $\alpha \in J$; since J and I are equivalence classes and $\alpha \in I \cap J$, it follows that $I = J$ and $s_\sim \rightsquigarrow I$ in $\mathcal{A}_\sim$ reading α.

From the above claim and the definition of $F_\sim$, it easily follows that $\mathcal{L}$ is the language recognised by $\mathcal{A}_\sim$.

We conclude by checking that $\mathcal{A}_\sim$ is Wheeler, proving the two Wheeler properties (i) and (ii) with respect to the linear order $(V_\sim, <_\sim)$.

To see Wheeler-(i) assume $e \prec e'$ with $e, e' \in \Sigma$. Consider $I, J \in V_\sim$ such that $(I, H, e) \in E_\sim$ and $(J, K, e') \in E_\sim$. We want to prove that $H <_\sim K$ (i.e. $H \prec^i K$). By definition of $E_\sim$, in our hypotheses there are $\alpha \in I$, $\alpha' \in J$ with $\alpha e \in H$ and $\alpha' e' \in K$. From $e \prec e'$ it follows $\alpha e \prec \alpha' e'$ and hence $H \preceq^i K$. To conclude observe that $H \prec^i K$ since all words in H end with e, while all words in K end with e'.

To see Wheeler-(ii) assume $I <_\sim J$ (i.e. $I \prec^i J$), $e \in \Sigma$, $(I, H, e) \in E_\sim$, and $(J, K, e) \in E_\sim$. In these hypotheses there are $\alpha \in I$, $\alpha' \in J$, with $\alpha e \in H$ and $\alpha' e \in K$. From $I \prec^i J$ and the fact that different classes are disjoint it follows $\alpha \prec \alpha'$; therefore, $\alpha e \prec \alpha' e$ and hence $H \preceq^i K$.

This ends the proof of the implication (3) $\Rightarrow$ (4).

(4) $\Rightarrow$ (1). Trivial. ∎

Note that the relation $\equiv^c_{\mathcal{L}}$ induces the minimum WDFA. This and other results on Wheeler Languages are going to be part of a companion paper of this one.

3 Sorting Wheeler Finite Automata

In this section we provide efficient algorithms to sort a relevant sub-class of the Wheeler automata. Combined with the results of the previous section, our algorithms can be used to index *any* WNFA.

3.1 Recognizing Wheeler 2-NFAs is in P We start with a reduction from the problem of recognizing Wheeler 2-NFAs to 2-SAT. The reduction introduces only a polynomial number of boolean variables and can be computed in polynomial time; since 2-SAT is in P, this implies that Wheeler 2-NFA recognition is in P.

THEOREM 3.1. *Let $\mathcal{A} = (V, E, F, s, \Sigma)$ be a 2-NFA. In $O(|E|^2)$ time we can:*

1. *Decide whether $\mathcal{A}$ is a Wheeler graph, and*
2. *If $\mathcal{A}$ is a Wheeler graph, return a node ordering satisfying the Wheeler graph definition.*

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

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

ensure that the order of nodes induced by the variables is total.

The following formulas ensure that the Wheeler properties are satisfied:

(a) For each u, v, if $\lambda(u) \prec \lambda(v)$ then we add the unary clause $x_{u<v}$.

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

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

The following formulas guarantee that the order is total. Note that we omit transitivity which, on a general graph, would require a 3-literals clause $(x_{u<v} \wedge x_{v<w}) \rightarrow x_{u<w}$ for each triple u, v, w. We will show that, if the graph is an input-consistent 2-NFA, then transitivity is satisfied "for free".

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

(2) *Completeness.* For every pair $u \neq v$, add the clause $x_{u<v} \vee x_{v<u}$.

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

We now show that on input-consistent 2-NFAs transitivity propagates from the source to all nodes. Consider a variable assignment that satisfies clauses (a),(b),(1), and (2) (if $\mathcal{A}$ is a Wheeler 2-NFA, then such an assignment exists by definition). Assume, moreover, that $x_{u<v}$ and $x_{v<w}$ are set true by the assignment, for three pairwise distinct nodes u, v, w. We want to show that also $x_{u<w}$ must be true.

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

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

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

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

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

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

We note that it is tempting to try to generalize the above solution to general NFAs by simulating arbitrary degree-d nondeterminism using binary trees: a node with d equally-labeled outgoing edges could be expanded to a binary tree with d leaves (bringing down the degree of nondeterminism to 2). Unfortunately, while this solution works for transitivity (which is successfully propagated from the source), it could make the graph

non-Wheeler: the topology of those trees cannot be arbitrary and must satisfy the co-lexicographic ordering of the nodes, i.e. the solution we are trying to compute.

Gibney and Thankachan [12] have recently shown that the problem of recognizing Wheeler d-NFAs is NP-complete for $d \geq 5$. Theorem 3.1 almost completes the picture, the remaining open cases being $d = 3$ and $d = 4$. We note that Theorem 3.1 combined with our determinization result of Section 2 does not break the problem's NP-completeness: in some cases, our determinization algorithm turns a non-Wheeler NFA into a WDFA, as shown in the example depicted in Figure 1.

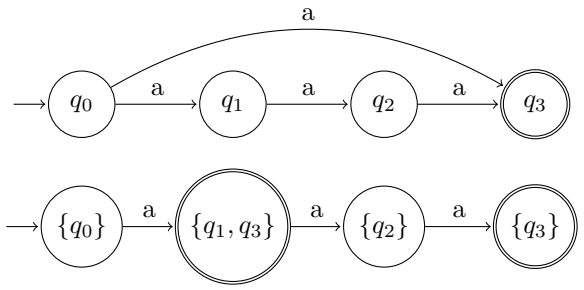

Figure 1: **Top**: a non-Wheeler NFA accepting the finite (Wheeler) language $\{a, aaa\}$. The automaton is clearly non-Wheeler, since states q_2 and q_3 cannot be ordered: looking at all edges but $q_0 \to q_3$, the only admissible ordering would be $q_0 < q_1 < q_2 < q_3$. However, $q_0 < q_1$ forces $q_3 < q_2$. **Bottom**: the equivalent DFA obtained after running the powerset construction algorithm. Note that this automaton is a Wheeler DFA, with Wheeler order $q_0 < q_1 < q_2 < q_3$. The example generalizes automatically to the infinite family of finite (Wheeler) languages $\{a, a^n\}$, for any $n \geq 3$.

We now describe more efficient algorithms for the deterministic case. First, note that Lemma 2.3 implies that Wheeler DFAs admit a unique admissible ordering (this follows from the fact that, on WDFAs, $\{\alpha, \beta\} \not\subseteq I_v \cap I_u$ always holds):

Corollary 3.1. *Let $\mathcal{A}$ be a Wheeler DFA, $<$ be the node ordering satisfying the Wheeler properties, and $\prec$ be the co-lexicographic order among strings. For any two nodes $u \neq v$, the following holds: $\alpha_u \prec \alpha_v$ for all string pairs α_u, α_v labeling paths $s \rightsquigarrow u$ and $s \rightsquigarrow v$ if and only if $u < v$.*

Corollary 3.1 has two important consequences: on DFAs, (i) we can use *any* paths connecting s with two nodes $u \neq v$ to decide their co-lexicographic order, and (ii) if it exists, the total ordering of the nodes is unique. We will extensively use these properties in the next sections.

3.2 Sorting Acyclic WDFAs Online In this section we describe an online algorithm that solves the problem considered in Theorem 3.1 in $O(|E| \log |V|)$ time when the graph is an acyclic DFA. The algorithm is online in the following sense. We assume that the nodes, together with their incoming labeled edges, are provided to the algorithm in any valid topological ordering. At any step, we maintain a prefix-sorted list of the current nodes, which is updated when a new node is added. When a new node v arrives together with its incoming labeled edges $(u_1, v, a), \ldots, (u_k, v, a)$, then $u_1, \ldots, u_k$ have already been seen in the past node sequence and can be used to decide the co-lexicographic rank of v. If v falsifies the Wheeler properties, we detect this event, report it, and stop the computation. Our algorithm is an extension of an existing one that builds online the Burrows-Wheeler transform of a string [25]. In Section 4 we will modify this algorithm so that, instead of failing on non-Wheeler graphs, it computes the smallest Wheeler DFA equivalent to the input acyclic DFA.

Algorithm 3.1 initializes all variables used by our procedure and implements Kahn's topological-sorting algorithm [19]. Every time a new node is appended to the topological ordering, we call Algorithm 3.2—our actual online algorithm—to update also the co-lexicographic ordering. This step also checks if the new node and its incoming edges falsify the Wheeler properties. We use the following structures (indices start from 1):

- `LEX` is a dynamic sequence of distinct nodes $v_1, \ldots, v_k \in V$ supporting the following operations:
  1. `LEX[i]` returns v_i.
  2. $\texttt{LEX}^{-1}[\texttt{v}]$, with $v \in$ `LEX`, returns the index i such that $\texttt{LEX}[i] = v$.
  3. `LEX.insert(v, i)` inserts node v between $\texttt{LEX}[\texttt{i} - 1]$ and $\texttt{LEX}[\texttt{i}]$. If $i = 1$, v is inserted at the beginning of the sequence. This operation increases the sequence's length by one.

- `IN` and `OUT` are dynamic sequences of strings, i.e. sequences $\alpha_1, \ldots, \alpha_k$, where $\alpha_i \in \Sigma^*$ (note that α_i could be the empty string ϵ). To make our pseudocode more readable, we index `IN` and `OUT` by nodes of `LEX` (these three arrays will be synchronized). Let $\texttt{T} = \alpha_1, \alpha_2, \ldots, \alpha_k$, with $\texttt{T} \in \{\texttt{IN}, \texttt{OUT}\}$. Both arrays support the following operation:
  4. `T.insert`$(\alpha, \texttt{v})$, where $\alpha \in \Sigma^*$ and $v \in$ `LEX`: insert α between $\alpha_{\texttt{LEX}^{-1}[v]-1}$ and $\alpha_{\texttt{LEX}^{-1}[v]}$. If $\texttt{LEX}^{-1}[\texttt{v}] = 1$, then α is inserted at the

beginning of T. This operation increases the sequence's length by one.

Sequence OUT supports these additional operations:

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

In our algorithm, sequence IN will always be partitioned in at most $t \leq \sigma + 1$ sub-sequences $\texttt{IN} = \alpha_1^{c_1}, \ldots, \alpha_{k_{c_1}}^{c_1}, \alpha_1^{c_2}, \ldots, \alpha_{k_{c_2}}^{c_2}, \ldots, \alpha_1^{c_t}, \ldots, \alpha_{k_{c_t}}^{c_t}$, where each α_i^c contains only character c and $c_1 \prec c_2 \prec \cdots \prec c_t$. We define an additional operation on IN:

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

Figure 2 shows how our dynamic structures evolve while processing states in topological order. In Subsection 3.2.1 we discuss data structures implementing the above operations in $O(\log k)$ time, k being the sequence's length. Intuitively, these three dynamic sequences have the following meaning: LEX will contain the co-lexicographically-ordered sequence of nodes. IN[v] and OUT[v], with $v \in$ LEX, will contain the labels of the incoming and outgoing edges of v, respectively. To keep the three sequences synchronized, when inserting v in LEX we will also need to update the other two sequences so that IN[v] $= c^t$, where t is the number of incoming edges, labeled c, of v, and OUT[v] $= \epsilon$, since v does not have yet outgoing edges. OUT[v] will (possibly) be updated later, when new nodes adjacent to v will arrive in the topological order. Our representation is equivalent to that used in [29] to represent the GCSA data structure. Intuitively, OUT is a generalized version of the well-known Burrows-Wheeler transform (except that we sort prefixes in co-lexicographic order instead of suffixes in lexicographic order). If the graph is a path (i.e. a string) then OUT is precisely the BWT of the reversed path.

We proceed with a discussion of the pseudocode. In Lines 1-16 of Algorithm 3.1 we initialize all variables and data structures. Let $u \in V$. The variable u.in memorizes the number of incoming edges in u; we will use this counter to implement Kahn's topological sorting procedure. u.label is the label of all incoming edges of u, or # if $u = s$. IN, LEX, and OUT are initialized as empty dynamic sequences. Lines 16-28 implement Kahn's topological sorting algorithm [19]. Each time a new node u is appended to the order, we call our online procedure update(u), implemented in Algorithm 3.2. Algorithm 3.2 works as follows. Assume that we have already sorted $v_1, \ldots, v_k$, that LEX contains the nodes' permutation reflecting their co-lexicographic order, and that IN[v_i] and OUT[v_i] contain the incoming and outgoing labels for each $i = 1, \ldots, k$ in the subgraph induced by $v_1, \ldots, v_k$. When a new node u arrives in topological order, all its t predecessors are in LEX. Let $b =$ u.label be the incoming label of u. We find the co-lexicographically smallest v_{min} and largest v_{max} predecessors of u (using function LEX^{-1} on all u's predecessors). In our pseudocode, if $u = s$ then $v_{min} = v_{max} =$ NULL. To keep the Wheeler properties true, note that there cannot be b's in the range OUT[v_{min}..v_{max}]: if there are, since we will append b to OUT[v_{min}] and OUT[v_{max}], there will be three nodes $v_{min} < v' < v_{max}$ such that $(v_{min}, u, b), (v', u', b), (v_{max}, u, b) \in E$ for some u'. Then, by Wheeler property (ii), this would imply that $u < u' < u$, a contradiction. We therefore check this event using function contains (note: this function can be easily implemented using two calls to rank). If b's are present, then the graph is no longer Wheeler: such an event is shown in Figure 2, left-hand side (where $u = v_5$). Otherwise, the number j of b's before v_{min} (which is equal to the number of b's before v_{max}) tells us the co-lexicographic rank i of u (similarly to the standard string-BWT, we obtain this number by adding j to the starting position of b's in IN), and we can mark (reserve) range OUT[v_{min}..v_{max}] with letter b using function reserve. Such an event is shown in Figure 2, left-hand side, when inserting, e.g., node v_3. At this point, we may have an additional inconsistency falsifying the Wheeler properties in the case that one

of the predecessors v_i of u falls inside a reserved range for b (reserved by a node other than u): this happens, for example, when inserting v_6 in Figure 2, right-hand side. This check requires calling function `is_reserved`. If all tests succeed, we insert u in position i of `LEX` and we update `IN` and `OUT` by inserting b^t at the i-th position in `IN` (i.e. the position corresponding to u) and by appending b at the end of each $\texttt{OUT}[\texttt{v}_\texttt{i}]$ for each predecessor v_i of u.

ALGORITHM 3.1. `sort(G)`

```
1: for u ∈ V do
2:    u.in ← 0
3:    u.label ← NULL
4: end for
5: s.label ← #
6: for (u, v, a) ∈ E do
7:    v.in ← v.in + 1
8:    if v.label ≠ NULL and v.label ≠ a then
9:       return FAIL {Cannot be Wheeler graph}
10:   end if
11:   v.label ← a
12: end for
13: IN ← new_dyn_sequence(Σ*){Sequence of strings}
14: LEX ← new_dyn_sequence(V){Sequence of nodes}
15: OUT ← new_dyn_sequence(Σ*){Sequence of strings}
16: S ← {s}{Set of nodes with no incoming edges}
17: while S ≠ ∅ do
18:   u ← S.pop(){Extract any u ∈ S}
19:   update(u){Call to Algorithm 3.2. If this fails,
      return FAIL.}
20:   for (u, v, a) ∈ E do
21:      v.in ← v.in − 1
22:      if v.in = 0 then
23:         S ← S ∪ {v}
24:      end if
25:   end for
26: end while
27: if ∃ v ∈ V : v.in > 0 then
28:   return FAIL{cycle found!}
29: end if
30: return LEX
```

ALGORITHM 3.2. `update(u)`

```
1: v_min ← min_pred(u){co-lexicographically-smallest
   predecessor}
2: v_max ← max_pred(u){co-lexicographically-largest
   predecessor}
3: if u ≠ s then
4:    if OUT[v_min, ..., v_max].contains(u.label) then
5:       return FAIL{Inconsistency of type 1}
6:    else
7:       for (v, u, a) ∈ E do
8:          if OUT.is_reserved(v, u.label) then
9:             return FAIL{Inconsistency of type 2}
10:         else
11:            OUT[v].append(u.label)
12:         end if
13:      end for
14:      OUT.reserve(v_min, v_max, u.label){Reserve
         [v_min, v_max] with u.label}
15:   end if
16:   i ← IN.start(u.label) + OUT.rank(u.label, v_min)
17:   LEX.insert(u, i)
18:   p ← |pred(u)|{Number of predecessors of u}
19: else
20:   LEX.insert(u, 1)
21:   p ← 1{Number of predecessors of u}
22: end if
23: IN.insert(u.label^p, u){Insert p times u.label}
24: OUT.insert(ϵ, u){u does not have successors yet}
```

THEOREM 3.2. *Let $\mathcal{A} = (V, E, F, s, \Sigma)$ be an acyclic DFA. There exists an algorithm that either prefix-sorts the nodes of $\mathcal{A}$ or returns* `FAIL` *if such an ordering does not exist online with $O(\log |V|)$ delay per input edge.*

Proof. In Subsection 3.2.1 we show that all operations on the dynamic sequences can be implemented in logarithmic time. Correctness follows from the fact that we always check that the Wheeler properties are maintained true. To prove completeness, note that at each step we place u between two nodes v_1 and v_2 in array `LEX` only if the smallest u's predecessor is larger than the largest v_1's predecessor, and if the largest u's predecessor is smaller than the smallest v_2's predecessor. This is the only possible choice we can make in order to satisfy $w_{v_1} \prec w_u \prec w_{v_2}$ for all strings labeling paths $s \rightsquigarrow v_1$, $s \rightsquigarrow u$, and $s \rightsquigarrow v_2$ and to obtain, by Corollary 3.1, the only possible correct ordering of the nodes. It follows that, if the new node v does not falsify the Wheeler properties, then we are computing its co-lexicographic rank correctly. ∎

3.2.1 Data Structure Details In this subsection we show how to implement operations **1-10** used by Algorithms 3.1 and 3.2 using state-of-the-art data structures. At the core of `LEX` and `OUT` stands the dynamic sequence representation of Navarro and Nekrich [25]. This structure supports insertions, access, rank, and select in $O(\log n)$ worst-case time, n being the sequence's length. The space usage is bounded by $nH_0 + o(n \log \sigma) + O(\sigma \log n)$ bits, where H_0 is the zero-th order entropy of the sequence. Sequence `IN` will instead be represented using a dynamic partial sum data structure, e.g. a balanced binary tree or a Fenwick tree [7], and a dynamic bitvector. All details follow.

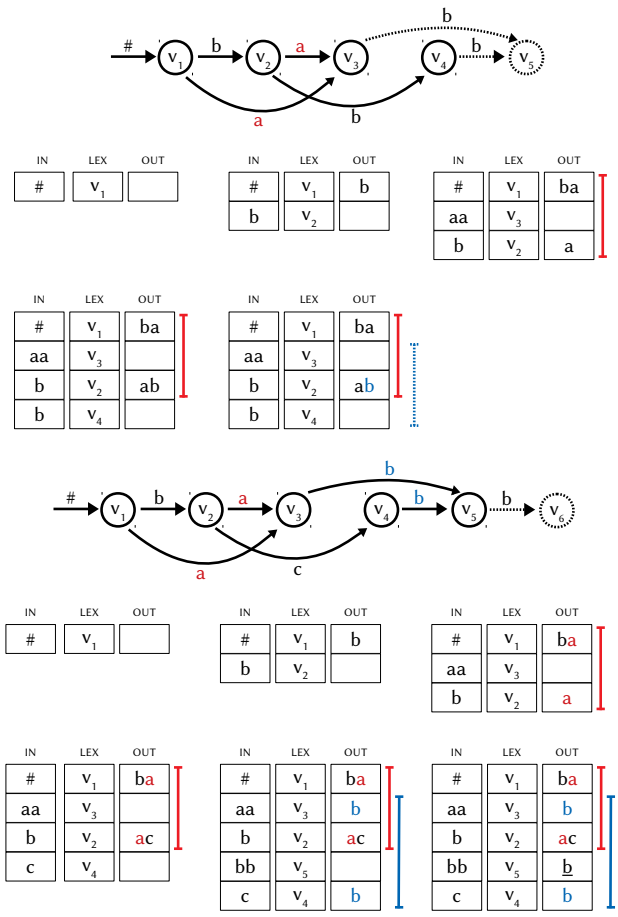

Figure 2: **Top**: Inconsistency of type 1. The five tables show how arrays `IN`, `LEX`, and `OUT` evolve during insertions of nodes $v_1, \ldots, v_5$ in topological order. Up to node v_4, the Directed Acyclic Graph (DAG) is Wheeler. When inserting node v_3, we successfully reserve the interval $[v_1, v_2]$ with label 'a' (shown in red). From this point, no 'a's can be inserted inside the reserved interval. When inserting node v_5 (with incoming label 'b'), the co-lexicographically smallest and largest predecessors of v_5 are v_3 and v_4, respectively. This means we have to reserve the interval $[v_3, v_4]$ with label 'b' (shown in blue dashed line); however, this is not possible since there already is a 'b' (highlighted in blue) in $\texttt{OUT}[v_3, \ldots, v_4]$. **Bottom**: Inconsistency of type 2. Up to node v_5, the DAG is Wheeler. Note that we successfully reserve two intervals: $[v_1, v_2]$ (with letter 'a', red interval), and $[v_3, v_4]$ (with letter 'b', blue interval). When inserting node v_4 we do not need to reserve any interval since the node has only one predecessor. The confict arises when inserting node v_6 (with incoming label 'b'). Since v_5 is a predecessor of v_6, we need to append 'b' in $\texttt{OUT}[\texttt{v5}]$. However, this 'b' (underlined in the picture) falls inside a reserved interval for 'b' (in blue).

Sequence `LEX` is stored with Navarro and Nekrich's dynamic sequence representation [25]. Operations **1-3** are directly supported on the representation. Operation **2** is simply a $select_v(1)$ (i.e. the position of the first v).

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

Operations **8-9** require an additional dynamic sequence of parentheses $\texttt{PAR}[1..n]$ on alphabet $\{(_c \ : \ c \in \Sigma\} \cup \{)_c \ : \ c \in \Sigma\} \cup \{\square\}$. Every time a new character is inserted at position i in `out`, we also insert $\square$ at position i in `PAR`. When $\texttt{OUT.reserve(u, v, c)}$ is called (i.e. operation **8**), let i_v and i_u be the positions in `out` corresponding to the two occurrences of character c in $\texttt{OUT}[\texttt{u}]$ and $\texttt{OUT}[\texttt{u}]$, respectively (remember that the automaton is deterministic, so these positions are unique). These positions can easily be computed in $O(\log n)$ time using *select* and *rank* operations on `out` and $\texttt{B}_{\texttt{out}}$. Then, we replace $\texttt{PAR}[i_u]$ and $\texttt{PAR}[i_v]$ with characters $(_\texttt{c}$ and $)_\texttt{c}$, respectively (replacing a character requires a deletion followed by an insertion). Note that reserved intervals for a fixed character do not overlap, so this parentheses representation permits to unambiguously reconstruct the structure of the intervals. At this point, operation **9** is implemented as follows. Let i_v be the position in `out` corresponding to the first character in $\texttt{OUT}[v]$. This position can be computed in $O(\log n)$ time with two *select* operations on `LEX` and $\texttt{B}_{\texttt{out}}$. Then, $\texttt{OUT.is\_reserved(v, c)}$ returns true if and only if $\texttt{PAR.rank}_{(_\texttt{c}}(\texttt{i}_\texttt{v}) > \texttt{PAR.rank}_{)_\texttt{c}}(\texttt{i}_\texttt{v})$, i.e. if we did not close all opening parentheses $(_\texttt{c}$ before position i_v (note that does not make any difference if i_v is the first or last position in $\texttt{OUT}[v]$, since when we call this operation $\texttt{OUT}[v]$ does not contain characters equal to c).

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

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

Fenwick trees [7] support the above operations within this time bound. Bitvector $\texttt{B}_{\texttt{IN}}[\texttt{1..n}]$ contains the bit sequence $110^{t_{v_2}-1}10^{t_{v_3}-1}\ldots10^{t_{v_k}-1}$, where t_{v_i} is the number of predecessors of v_i in the current sequence $\texttt{LEX} = v_1, \ldots, v_k$ of sorted nodes (note that v_1 is always the source s). Assume, for simplicity, that

$\Sigma = [1, \sigma + 1]$, where # corresponds to 1 (this is not restrictive, as we can map the alphabet to this range at the beginning of the computation). At the beginning, the partial sum is initialized as $\texttt{IN}[\texttt{c}] = 0$ for all c. Operation **10**, $\texttt{IN.start}(\texttt{c})$, with $c \neq \#$, is implemented as $\texttt{PS.ps}(\texttt{c} - 1) + 1$. If $c = \#$, the operation returns 1. Operation **4**, $\texttt{IN.insert}(\texttt{c}^\texttt{p}, \texttt{u})$, is implemented as $\texttt{PS}[\texttt{c}] \leftarrow \texttt{PS}[\texttt{c}] + p$ followed by $\texttt{B}_\texttt{IN}.\texttt{insert}(10^{\texttt{p}-1}, \texttt{i}_\texttt{u})$ (i.e. p calls to *insert* on the dynamic bitvector at position i_u), where i_u is the position of the j-th bit set in $\texttt{B}_\texttt{IN}$ (a *select* operation) and $j = \texttt{LEX}^{-1}[\texttt{u}]$, or $i_u = n + 1$ if $\texttt{B}_\texttt{IN}$ has $j - 1$ bits set (note that, when we call $\texttt{IN.insert}(\texttt{c}^\texttt{p}, \texttt{u})$, node u has already been inserted in LEX).

3.3 Sorting Wheeler DFAs in Linear Time To conclude the section, we show that in the offline setting we can improve upon the previous result. We first need the following lemma:

LEMMA 3.1. *Given an input-consistent edge-labeled graph $G = (V, E, \Sigma)$ and a permutation of V sorted by a total order $<$ on V, we can check whether $<$ satisfies the Wheeler properties in optimal $O(|V| + |E|)$ time.*

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

Let L denote the sorted list of edges. We claim that the given order $<$ satisfies the Wheeler properties (Definition 1.2) if and only if for all pairs of consecutive edges $(a_i, u_i, v_i), (a_{i+1}, u_{i+1}, v_{i+1})$ in L, we have $(a_i = a_{i+1}) \rightarrow v_i \leq v_{i+1}$ and $(a_i \neq a_{i+1}) \rightarrow v_i < v_{i+1}$. Clearly this can be checked in time $O(|E|)$ with one scan over L. We now argue the correctness of this algorithm.

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

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

THEOREM 3.3. *Let $\mathcal{A} = (V, E, F, s, \Sigma)$ be a DFA. In $O(|V| + |E|)$ time we can:*

1. *Decide whether $\mathcal{A}$ is a Wheeler graph, and*
2. *If $\mathcal{A}$ is a Wheeler graph, return a node ordering satisfying the Wheeler graph definition.*

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

By Corollary 3.1, if $\mathcal{A}$ is a Wheeler graph then we can use the strings that label *any* two paths $s \rightsquigarrow u$ and $s \rightsquigarrow v$ to decide the order of any two nodes u and v. We can therefore sort V according to the paths spelled by $\mathcal{T}$; by Corollary 3.1, if $\mathcal{A}$ is Wheeler then we obtain the correct (unique) ordering. To prefix-sort $\mathcal{T}$, we compute its XBW transform [2] in $O(|V|)$ time [8, Thm 2]. The array containing the lexicographically-sorted nodes (i.e. the prefix array of $\mathcal{T}$) can easily be obtained from the XBW transform using, e.g. the partial rank counters defined in the proof of Lemma 3.1 to navigate the tree (this is analogous to repeatedly applying function LF on the BWT in order to obtain the suffix array). At this point, we check that the resulting node order satisfies the Wheeler properties using Lemma 3.1. If this is this case, then the above-computed prefix array contains the prefix-sorted nodes of $\mathcal{A}$. ∎

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

4 Wheeler DFA Minimization

We are now ready to use the algorithms of the previous sections to prove our main algorithmic results: (i) a minimization algorithm for WDFAs (Theorem 4.1) and (ii) a near-optimal algorithm generating the minimum acyclic WDFA equivalent to any input acyclic DFA (Theorem 4.3).

Let $\equiv$ be an equivalence relation over the states V of an automaton $\mathcal{A} = (V, E, F, s, \Sigma)$. The *quotient automaton* is defined as $\mathcal{A}/_\equiv = (V/_\equiv, E/_\equiv, F/_\equiv, [s]_\equiv, \Sigma)$, where $E/_\equiv = \{([u]_\equiv, [v]_\equiv, c) : (\exists u' \in [u]_\equiv, v' \in [v]_\equiv)((u', v', c) \in E)\}$, and $F/_\equiv = \{[f] : f \in F\}$. In general, $\mathcal{A}/_\equiv$ could be a NFA (even if $\mathcal{A}$ is a DFA).

[2] note: this requires mapping the labels of $\mathcal{T}$ to alphabet $\Sigma' \subseteq [1, |V|]$ while preserving their lexicographic ordering. Since we assume that the original alphabet's size does not exceed $|E|^{O(1)} = |V|^{O(1)}$, this step can be performed in linear time by radix-sorting the labels.

The particular equivalence relation we define in the next paragraph will guarantee that $\mathcal{A}/_{\equiv}$ is a WDFA, provided that $\mathcal{A}$ is a WDFA. The symbol $\approx$ denotes the Myhill-Nerode equivalence among states [26]: $u \approx v$, with $u, v \in V$, if and only if, for any string α, we reach a final state by following the path labeled α from u if and only if the same holds for v. Note that this is the "state" version of the relation $\equiv_{\mathcal{L}}$ given in Section 2 (which instead is defined among strings). The goal of any DFA-minimization algorithm is to find $\approx$, which is the, provably existing and unique, coarsest (i.e. largest classes) equivalence relation stable with respect to the initial partition in final/non-final states. To abbreviate, we will simply say "coarsest equivalence relation" instead of "coarsest equivalence relation stable with respect to an initial partition".

In our case, assuming that $\mathcal{A}$ is Wheeler, we want to find the (unique as proved below) coarsest equivalence relation $\equiv_w$ finer than $\approx$, such that $\mathcal{A}/_{\equiv_w}$ is Wheeler. Our Algorithm 4.1 achieves precisely this goal: we start with $\approx$ and then refine it preserving stability with respect to characters, while also ensuring that the resulting equivalence classes can be ordered consistently with the Wheeler constraints. Again, it can be proved that $\equiv_w$ is the "state" version of the relation $\equiv^c_{\mathcal{L}}$ given in Section 2. For the purposes of the following results, we do not need to prove the connection between the two relations and we keep a distinct notation to avoid confusion.

Algorithm 4.1. (Wheeler Minimization)
Input: Wheeler DFA $\mathcal{A}$
Output: Minimum WDFA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$

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

We show:

Theorem 4.1. *Let $\mathcal{A}$ be a WDFA. The automaton $\mathcal{A}/_{\equiv_w}$ returned by Algorithm 4.1 is the minimum WDFA recognizing $\mathcal{L}(\mathcal{A})$.*

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

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

Consider now the process of minimizing $\mathcal{T}$ by collapsing states in equivalence classes in such a way that (i) the quotient automaton is finite, (ii) the accepting language of the quotient DFA is the same as that of $\mathcal{T}$ and (iii) the quotient DFA is Wheeler. By the existence of $\mathcal{A}$, there exists such a partition (not necessarily the coarsest): the one putting u^i and u^j in the same equivalence class if and only if $u^i, u^j \in L^u$, for some $u \in V$ (in this case, $\mathcal{A}$ itself is the resulting quotient automaton). Call $\equiv$ the relation among states of $\mathcal{T}$ yielding the *smallest* such WDFA $\mathcal{A}/_{\equiv}$. By definition, $\mathcal{A}/_{\equiv}$ is the smallest WDFA recognizing $\mathcal{L}(\mathcal{A})$. Our claim is that $\equiv = \equiv_w$, i.e. that Algorithm 4.1 returns this automaton. We observe that:

1. $u^i \approx u^j$ for any $u^i, u^j \in L^u$ and all $u \in V$. Otherwise, assume for a contradiction that there exists a string α leading to an accepting state from u^i but not from u^j. By construction of $\mathcal{T}$, u^i and u^j are $\approx$-equivalent to u: this leads to a contradiction, since the state reached from u with label α cannot be both accepting and not accepting.
2. Since $\mathcal{A}$ is a Wheeler DFA, Corollary 3.1 applied to $\mathcal{A}$ tells us that, for any two nodes $u < v \in V$, all strings labeling paths from the root of $\mathcal{T}$ to nodes in L^u are co-lexicographically smaller than those labeling paths from the root of $\mathcal{T}$ to nodes in L^v. We express this fact using the notation $L^u < L^v$.
3. Since $\mathcal{A}$ is Wheeler, then each $u \in V$ has only one distinct incoming label and $\lambda(u^j) = \lambda(u)$ for all $u^j \in L^u$.

By the above properties, $u^i \equiv u^j$ for all $u^i, u^j \in L^u$, $u \in V$. To see this, note that, by property **1**, those states are all equivalent by relation $\approx$. Moreover, properties **2-3** combined with Corollary 3.1 imply that,

by grouping states in each L^u, we cannot break any Wheeler property. It follows that $\equiv$ must group those states, being the coarsest partition finer than $\approx$ with these two properties. Let us indicate with $L^u \equiv L^v$ the fact that $u^i \equiv v^j$ for all $u^i \in L^u,\ v^j \in L^v$.

Suppose now, for a contradiction, that there exist $L^u < L^v < L^w$ with $L^u \equiv L^w \not\equiv L^v$. Then, by Corollary 3.1, $L^u < L^v$ implies that, in the quotient automaton, states $[L^w]_\equiv = [L^u]_\equiv$ and $[L^v]_\equiv$ are reachable from the source by two paths α and β, respectively, with $\alpha \prec \beta$. Conversely, $L^v < L^w$ implies that states $[L^v]_\equiv$ and $[L^w]_\equiv$ are reachable from the source by two paths α' and β', respectively, with $\alpha' \prec \beta'$. Then, by Corollary 3.1 we cannot define a total order on $\mathcal{A}/_\equiv$'s states, i.e. $\mathcal{A}/_\equiv$ is not Wheeler.

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

Note that uniqueness of the minimum WDFA follows from Corollary 3.1 (uniqueness of the Wheeler order) and Algorithm 4.1. Note also that, in the automaton output by Algorithm 4.1, adjacent states in co-lexicographic order are distinct by the relation $\approx$ unless their incoming labels are different (in which case they might be equivalent). It follows that if a sorted Wheeler DFA does not have this property, then it is not minimum (otherwise Algorithm 4.1 would collapse some of its states). Conversely, If a Wheeler DFA has this property, then Algorithm 4.1 does not collapse any state, i.e. the automaton is already of minimum size. We therefore obtain the following characterization:

THEOREM 4.2. (MINIMUM WDFA) *Let $\mathcal{A}$ be a Wheeler DFA, let $v_1 < v_2 < \cdots < v_t$ be its co-lexicographically ordered states, and let $\approx$ be the Myhill-Nerode equivalence among them. $\mathcal{A}$ is the minimum Wheeler DFA recognizing $\mathcal{L}(\mathcal{A})$ if and only if the following holds: for every $1 \leq i < t$, if $v_i \approx v_{i+1}$ then $\lambda(v_i) \neq \lambda(v_{i+1})$.*

Theorem 4.1 implies the following corollaries.

COROLLARY 4.1. *Given a WDFA $\mathcal{A}$ of size n, in $O(n \log n)$ time we can build the minimum WDFA recognizing $\mathcal{L}(\mathcal{A})$.*

Proof. We run Algorithm 4.1 computing $\approx$ with Hopcroft's algorithm [16] ($O(n \log n)$ time), and prefix-sorting $\mathcal{A}$ with Theorem 3.3 ($O(n)$ time). Note that we can check $u \approx v$ in constant time by representing the equivalence relation as a vector $EQ[v] = [v]_\approx$, where we choose $V = \{1, \ldots, |V|\}$ and where $[v]_\approx$ is any representative of the equivalence class of v (e.g., the smallest one, which we can identify in linear time by radix-sorting equivalent states). Then, $u \approx v$ if and only if $EQ[u] = EQ[v]$. Using this structure, the runs of Algorithm 4.1 can easily be identified in linear time. ∎

COROLLARY 4.2. *Given an acyclic WDFA $\mathcal{A}$ of size n, in $O(n)$ time we can build the minimum acyclic WDFA recognizing $\mathcal{L}(\mathcal{A})$.*

Proof. We run Algorithm 4.1 computing $\approx$ with Revuz's algorithm [28] ($O(n)$ time), prefix-sorting $\mathcal{A}$ with Theorem 3.3 ($O(n)$ time), and testing $u \approx v$ in constant time as done in Corollary 4.1. ∎

Note that Corollary 4.2 implies that we can, in *optimal linear* time, build the minimum WDFA $\mathcal{A}/_{\equiv_w}$ recognizing *any* input finite language $\mathcal{L}$ represented as a set of strings: we build the tree DFA accepting $\mathcal{L}$ and apply Corollary 4.2. The corollary can be applied since trees are always Wheeler [8, 10]. In the next subsection we treat the (more interesting) case where $\mathcal{L}$ is represented by a DFA. Note that this result could already be achieved by unraveling the DFA into a tree and minimizing it using Corollary 4.2. However, the intermediate tree could be exponentially larger than the output.

4.1 Acyclic DFAs to Smallest Equivalent WDFAs We show how to build the smallest acyclic Wheeler DFA equivalent to any acyclic DFA in output-sensitive time. While in the literature there already exists a technique (the so-called *GCSA*) to convert DFAs into equivalent WDFAs [29], we stress out that such approach is not guaranteed to find the *minimum* such automaton. Let $\mathcal{A} = (V, E, F, s, \Sigma)$ be an acyclic DFA. We first minimize $\mathcal{A}$ using Revuz's algorithm [28] and obtain the equivalent minimum acyclic DFA $\mathcal{A}_1 = \mathcal{A}/_\approx = (V_1, E_1, F_1, s_1, \Sigma)$. Let us denote $|V_1| = t$. The idea is to run a modified version of the online Algorithm 3.2 on $\mathcal{A}_1$. The difference is that now we will *solve* (not just detect) violations to the Wheeler properties without changing the accepting language. The next step is to topologically-sort $\mathcal{A}_1$'s states (e.g. using Kahn's algorithm [19]). At this point, we modify $\mathcal{A}_1$ in t steps by processing its states in topological order. This defines a sequence of automata $\mathcal{A}_1, \mathcal{A}_1, \ldots, \mathcal{A}_t$. At each step, the states of $\mathcal{A}_i$ are partitioned in two sets:

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

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

1. $\mathcal{L}(\mathcal{A}_i) = \mathcal{L}(\mathcal{A}_1)$.
2. States in $\texttt{LEX}_i$ are sorted by a total order $<$ that does not violate the Wheeler properties among states in $\texttt{LEX}_i$ itself: in Definition 1.2, we require $u_1, u_2, v_1, v_2 \in \texttt{LEX}_i$.
3. for each $j = 1, \ldots, |\texttt{LEX}_i| - 1$, if $\texttt{LEX}_i[j] \approx \texttt{LEX}_i[j+1]$ then $\lambda(\texttt{LEX}_i[j]) \neq \lambda(\texttt{LEX}_i[j+1])$.

Invariant **1** implies $\mathcal{L}(\mathcal{A}_t) = \mathcal{L}(\mathcal{A})$. Since $N_t = \emptyset$ and $\texttt{LEX}_t$ contains all $\mathcal{A}_t$'s states, invariant **2** implies that $\mathcal{A}_t$ is Wheeler (note that intermediate automata $\mathcal{A}_i$, with $1 < i < t$ might be non-Wheeler). Finally, invariant **3** and Theorem 4.2 imply that $\mathcal{A}_t$ is the minimum WDFA accepting $\mathcal{L}(\mathcal{A}_t)$. As a result, $\mathcal{A}_t = \mathcal{A}/_{\equiv_w}$.

We describe an online step of our algorithm. Assume we successfully built $\mathcal{A}_i$, with $i < t$, and we are about to process v_{i+1} in order to build $\mathcal{A}_{i+1}$. Let $\{c_1, \ldots, c_k\}$ be the labels of incoming v_{i+1}'s edges. We first replace (split) v_{i+1} by k equivalent states $v_{i+1}^{c_1} \approx \cdots \approx v_{i+1}^{c_k}$: each $v_{i+1}^{c_k}$ (i) is accepting if and only if v_{i+1} is accepting, (ii) keeps only the incoming edges of v_{i+1} labeled c_i, and (iii) it duplicates all its outgoing edges: we replace each (v_{i+1}, u, c) with the edges $(v_{i+1}^{c_1}, u, c), \ldots, (v_{i+1}^{c_k}, u, c)$. Note that all the newly-created edges must be present in the final automaton $\mathcal{A}_t$ since the states $v_{i+1}^{c_1}, \ldots, v_{i+1}^{c_k}$ cannot be collapsed back by $\equiv_w$ (as they have different incoming labels); it follows that in this step we are not creating more edges than necessary.

We now insert separately $v_{i+1}^{c_1}, \ldots, v_{i+1}^{c_k}$ in $\texttt{LEX}_i$ in any order as follows. The procedure is the same for all those vertices, therefore we may simply assume we are about to process a node v with all incoming edges labeled with the same character a. Let $u_1 < \cdots < u_k$ be the predecessors of v in the graph; note that those nodes must belong to $\texttt{LEX}_i$ (since we are processing states in topological order), therefore their order $<$ is well-defined. We now must detect and solve inconsistencies of type 1 and 2 as defined in the proof of Theorem 3.2.

We start with inconsistencies of type 1: there already are nodes $w_i \notin \{u_1, \ldots, u_k\}$ with outgoing edges labeled a inside the range $[u_1, u_k]$. This breaks the sequence $u_1 < \cdots < u_k$ into q sub-intervals $[u_{i_j}, u'_{i_j}]$, $j = 1, \ldots, q$, that do not contain nodes with outgoing label a different than those in $\{u_1, \ldots, u_k\}$. The range has therefore the following form, where we denote with w_i and w'_i all nodes not in $\{u_1, \ldots, u_k\}$ with outgoing edges labeled a and we highlight in bold the runs $[u_{i_j}, u'_{i_j}]$: $w_1 < \mathbf{u_{i_1}} \leq \cdots \leq \mathbf{u'_{i_1}} < w_2 \leq \cdots \leq w'_2 < \mathbf{u_{i_2}} \leq \cdots \leq \mathbf{u'_{i_2}} < \cdots < \mathbf{u_{i_q}} \leq \cdots \leq \mathbf{u'_{i_q}} < w_{q+1}$, where $u_{i_1} = u_1$, $u'_{i_q} = u_k$, and $w_1 < u_1$, $w_{q+1} > u_k$ are the rightmost and leftmost states with an outgoing edge labeled a, respectively (if they exist). The top part of Figure 3 depicts this situation, where $k = 4$ and $u_1, \ldots, u_4$ are clustered in $q = 3$ runs: $w_1 < \mathbf{u_1} < w_2 < w_3 < \mathbf{u_2} < \mathbf{u_3} < w_4 < \mathbf{u_4} < w_5$. We solve the inconsistencies of type 1 by splitting v in (i.e. replacing it with) q equivalent nodes: $v_1 \approx \cdots \approx v_q$. Each v_j is final if and only if v is final, duplicates all v's outgoing edges (as seen above), and keeps only incoming edges from v's predecessors inside the corresponding run $[u_{i_j}, u'_{i_j}]$. This is depicted in the bottom part of Figure 3: v has been split into the three equivalent nodes $v_1 \approx v_2 \approx v_3$.

Inconsistencies of type 2 are solved similarly by splitting a-successors of $w_1, \ldots, w_{q+1}$ that belong to $\texttt{LEX}_i$ when necessary. Let $\texttt{LEX}_i \cap \{succ_a(w_1), \ldots, succ_a(w_{q+1})\} = \{z_1 < \cdots < z_{q'}\}$ be the a-successors of $w_1, \ldots, w_{q+1}$ in $\texttt{LEX}_i$. Note that it might be the case that $q' < q + 1$. Note also that some of the nodes z_i might belong to $\{u_1, \ldots, u_k\} \cup \{w_1, \ldots, w_{q+1}\}$. We have an inconsistency of type 2 (among nodes in $\texttt{LEX}_i$) whenever $succ_a(w_i) = succ_a(w_{i+1}) = z_e$, for some $1 \leq e \leq q'$, and there exist some u_j such that $w_i < u_j < w_{i+1}$. In this case, we split $z_e = succ_a(w_i) = succ_a(w_{i+1})$ in two equivalent nodes $z'_e \approx z''_e$ ordered as $z'_e < succ_a(u_j) < z''_e$. This cannot contradict the Wheeler properties (even if $z_i \in \{u_1, \ldots, u_k\} \cup \{w_1, \ldots, w_{q+1}\}$), since $succ_a(u_j)$ is one of the copies of v (or v itself if v has not been splitted in the previous step) and has therefore no successors in the current automaton. The process of fixing inconsistencies of type 2 is shown in Figure 3: nodes w_3 and w_4 are separated by u_2, u_3 as $w_3 < u_2 < u_3 < w_4$. In this case, $succ_a(w_3) = succ_a(w_4) = z_3$, and we split z_3 in the two equivalent nodes z'_3 and z''_3. Note also that we only need to check those w_i that immediately precede or follow a predecessor of v (i.e. $w_1, w_2, w'_2, \ldots, w_{q+1}$): those nodes are at most $O(k)$, where k is the number of v's predecessors.

As shown in Figure 3 (bottom), after solving the inconsistencies of type 1 and 2 the nodes in $\texttt{LEX}_{i+1}$ are again range-consistent: the a-successors of any (sorted) range of nodes form themselves a (sorted) range. Moreover, the splitting process defines unambiguously a total ordering of the new nodes among those already in $\texttt{LEX}_i$, which can be therefore updated to $\texttt{LEX}_{i+1}$ by inserting those nodes at the right place: to insert a node v' in $\texttt{LEX}_i$, let u' be its a-predecessor: $succ_a(u') = v'$. Let moreover $u'' < u'$ be the rightmost node preceding u' (in

$\texttt{LEX}_i$) having an outgoing edge labeled a, and let v'' be its a-successor: $succ_a(u'') = v''$. By range-consistency, node v' has to be inserted immediately after v'' in $\texttt{LEX}_i$. If such a node u'' does not exist (i.e. u' is the leftmost node in $\texttt{LEX}_i$ having an outgoing edge labeled a), then v' has to be inserted in $\texttt{LEX}_i$ so that it becomes the first node with incoming edges labeled a (i.e. in the position immediately following the rightmost node v'' with incoming label a', where a' is the lexicographically-largest character such that $a' \prec a$, or at the first position in $\texttt{LEX}_i$ if such a character a' does not exist). This shows that invariant **2** is maintained: the Wheeler properties are kept true among nodes in $\texttt{LEX}_{i+1}$. It is also clear that we do not insert $\approx$-equivalent adjacent states with the same incoming label (see Figure 3: by construction, the newly-inserted nodes $v_1, z'_3, v_2, z''_3, v_3$ are non-equivalent to their neighbors), i.e. invariant **3** is maintained. Finally, the accepted language does not change since the splitting process generates $\approx$-equivalent nodes: also invariant **1** stays true.

Note that the minimization process on the original acyclic DFA $\mathcal{A}$ takes linear time. After that, we only insert edges/nodes in the minimum output WDFA: never delete. It follows that the number of performed operations is equal to the output's size. The final automaton could be either smaller or up to exponentially-larger than $\mathcal{A}$. We note that all the discussed operations can be easily implemented in logarithmic time using the data structures discussed in Section 3.2.1: finding the q runs of states $[u_{i_j}, u'_{i_j}]$, as well as finding the $O(k)$ states w_i, requires executing a constant number of *rank* operations on sequence $\texttt{OUT}$ and *start* operations on $\texttt{IN}$ for each predecessor of v. Nodes can be inserted at the right position in sequence $\texttt{LEX}$ exactly as done in Algorithm 3.2 (by also updating $\texttt{IN}$ and $\texttt{OUT}$). Finally, the graph can be dynamically updated (i.e. splitting nodes) and queried (i.e. navigation) by keeping it as a dynamic adjacency list: since we can spend logarithmic time per edge, we can store the graph as a self-balancing tree associating nodes to their predecessors and successors (also kept as self-balancing trees). This structure supports all updates and queries on the graph in logarithmic time. It follows that the overall procedure terminates in $O(n + m \log m)$ time, n and m being the input and output's sizes, respectively.

THEOREM 4.3. *Given an acyclic DFA $\mathcal{A}$ of size n, we can build and prefix-sort the minimum acyclic WDFA, of size m, recognizing $\mathcal{L}(\mathcal{A})$ in $O(n + m \log m)$ time.*

Theorem 4.3 solves the problem of indexing deterministic DAGs for linear-time pattern matching queries in nearly-optimal time with a solution of minimum size. This, combined with the hardness result of Equi et al. [6]

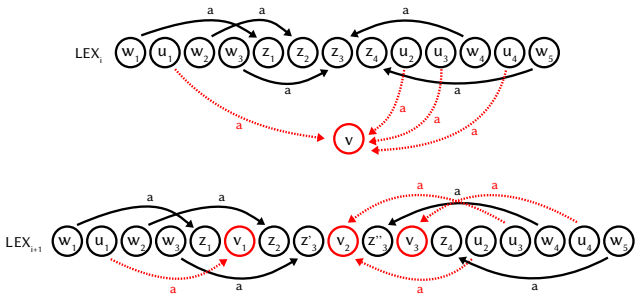

Figure 3: Inconsistency resolution. Nodes are ordered left-to-right by the total ordering $<$ (except v in the top part of the figure). **Top**: we are trying to insert v in $\texttt{LEX}_i$, but this violates the Wheeler properties (edges' destinations are not ordered as the sources, no matter where we insert v). **Bottom**: we solve the inconsistencies by splitting v in three equivalent nodes $v_1 \approx v_2 \approx v_3$ and z_3 in two equivalent nodes $z'_3 \approx z''_3$. Note that (i) the splitting procedure induces naturally an ordering of the nodes that satisfies the Wheeler properties, and (ii) after splitting, no two adjacent states with the same incoming label are equivalent by $\approx$. By Theorem 4.2, this is the minimum way of splitting nodes. For simplicity, in the figure nodes $z_1, \ldots, z_4$ do not coincide with any node $w_1, \ldots, w_5$ or $u_1, \ldots, u_4$. This may not necessarily be the case. In our detailed discussion we show that our procedure is correct even when this happens.

implies that, under the Orthogonal Vectors hypothesis, in the worst case the minimum WDFA has size $\Omega(n^{2-\epsilon})$ for any constant $\epsilon > 0$. We can do better: we now show that there exists a family of regular languages where the size of the smallest WDFA is exponential in the size of the smallest DFA. Consider the family of languages $L_1, L_2, \ldots$, where $L_m = \{c\alpha e \mid \alpha \in \{a,b\}^m\} \cup \{d\alpha f \mid \alpha \in \{a,b\}^m\}$. Figure 4 shows a DFA and the smallest WDFA for the language L_3. In general, we can build a DFA for L_m by generalizing the construction in the figure: the source node has outgoing edges labeled with c and d, followed by simple linear size "universal gadgets" capable of generating all binary strings of length m, with one gadget followed by an e and the other by an f. The two sink states are the only accepting states.

The smallest WDFA for L_m is an unraveling of the described DFA, such that all paths up to (but not including) the sinks end up in distinct nodes, i.e. the universal gadgets are replaced by full binary trees (see Figure 4). It is easy to see that the automaton is Wheeler as the only nodes that have multiple incoming paths are the sinks, and the sinks have unique labels.

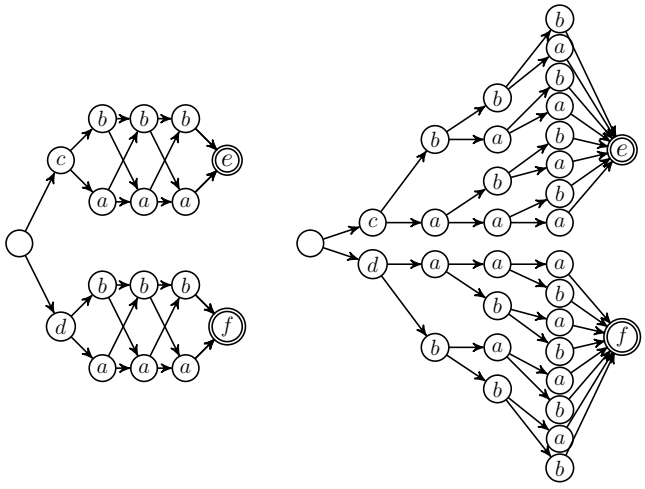

Figure 4: Left: a DFA recognizing L_3. Right: the minimum WDFA recognizing L_3. For clarity the labels are drawn on the nodes: the label of an edge is the label of the destination node.

To prove that this is the minimum WDFA, we need to check the condition of Theorem 4.2, i.e. that all colexicographically consecutive pairs of nodes with the same incoming label are Myhill-Nerode inequivalent. As labels c, d, e and f occur only once, it is enough to focus on nodes that have label a or b. Let $B_1, B_2, B_{2^{m+1}-1}$ be the colexicographically sorted sequence of all possible binary strings with lengths $1 \leq |B_i| \leq m$ from the alphabet $\{a, b\}$. Observe that the nodes with incoming label a and b correspond to path labels of the form cB_i and dB_i for all $1 \leq i \leq 2^{m+1} - 1$. The colexicographically sorted order of these path labels is:

$$cB_1 < dB_1 < cB_2 < dB_2 < \ldots < cB_{2^{m+1}-1} < dB_{2^{m+1}-1}$$

Here we can see that all consecutive pairs have a different first character: they therefore lead to a different sink in the construction and hence are not Myhill-Nerode equivalent. We therefore conclude that the automaton is the minimum WDFA. The DFA has $n = 4m+5$ states and the WDFA has $1 + 2^{m+2} = 1 + 2^{(n-5)/4+2}$ states, so we obtain the following result:

THEOREM 4.4. *The minimum WDFA equivalent to an acyclic DFA with n states has $\Omega(2^{n/4})$ states in the worst case.*

5 Indexing Wheeler Automata

We show that any Wheeler NFA can be efficiently indexed in order to support fast membership queries in its accepting language or in its substring/suffix closure. Let $\mathcal{A}$ be any Wheeler NFA. We first remove all states that do not lead to a final state. This preserves the accepted language, the total ordering, and the Wheeler properties. We then use our algorithms to convert the automaton to a WDFA, prefix-sort it in polynomial time, and build a (generalized) FM-index on the graph as described in [10]. The above transformations are only necessary if the Wheeler order of the input WNFA is unknown; otherwise, one can directly proceed with the following steps. We mark in a bitvector $B[1..|V|]$ supporting constant-time *rank* queries [17] all accepting states of the Wheeler NFA in our array `LEX` containing the states in co-lexicographic order. To check membership of a word w, we search the word $\#w$ and get a range $\texttt{LEX}[L, R]$ of all states reachable from the root by a path labeled w. At this point, w is accepted if and only if $B[L, R]$ contains at least one bit set (constant time using *rank* on B). Note that this procedure works in $O(w \log \sigma)$ time also if the original automaton is nondeterministic (this, in general, is not possible for general NFAs). If we search for w instead of $\#w$, then we get the range of states reachable by a path labeled uw, for any $u \in \Sigma^*$. This range is nonempty if and only if w belongs to the substring closure of $\mathcal{L}(\mathcal{A})$. Finally, if we search a word w and get a range $\texttt{LEX}[L, R]$, then w is in the suffix closure of $\mathcal{L}(\mathcal{A})$ if and only if $B[L, R]$ contains at least one bit set.

6 Conclusions and Future Extensions

In this paper, we have initiated the study of Wheeler languages, that is, regular languages that can be indexed via prefix-sorting techniques. On our way, we provided new results of independent interest: (i) we described a new class of NFAs for which the minimization problem can be approximated up to multiplicative factor 2 in polynomial time and that admit fast membership and pattern matching algorithms, and (ii) we solved the problem of indexing finite languages with prefix-sortable DFAs of minimum size. This paper leaves several intriguing lines of research that we have just started to explore. Is the problem of recognizing Wheeler languages decidable? In a future extension of this work we will give a positive answer to this question: the *Wheelerness* of a regular language translates into particular constraints on the topology of its minimum accepting DFA. Given a DFA for a Wheeler language, can we build the minimum accepting Wheeler DFA? In this case, we believe that the task can be solved by iterating conflict-resolution (Section 4.1) until the process converges to the minimum WDFA. We will also provide a technique to convert any WNFA into a (never larger) equivalent non-redundant automaton that can be prefix-sorted in polynomial time.

References

[1] Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99, 2000.

[2] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 457–466. IEEE, 2016.

[3] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 180–192, 2012.

[4] Francisco Claude, Gonzalo Navarro, and Alberto Ordónez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

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

[6] Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the Complexity of String Matching for Graphs. *46th International Colloquium on Automata, Languages and Programming*, 2019.

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

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

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

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

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

[12] Daniel Gibney and Sharma V. Thankachan. On the Hardness and Inapproximability of Recognizing Wheeler Graphs. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:16, Dagstuhl, Germany, 2019.

[13] Gregor Gramlich and Georg Schnitger. Minimizing NFA's and Regular Expressions. In Volker Diekert and Bruno Durand, editors, *STACS 2005*, pages 399–411, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

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

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

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

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

[18] Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the Complexity of Sequence to Graph Alignment. *BioRxiv*, page 522912, 2019.

[19] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[20] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[21] Andreas Malcher. Minimizing Finite Automata is Computationally Hard. *Theor. Comput. Sci.*, 327(3):375–390, November 2004.

[22] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[23] Udi Manber and Sun Wu. Approximate string matching with arbitrary costs for text and hypertext. In *Advances In Structural And Syntactic Pattern Recognition*, pages 22–33. World Scientific, 1992.

[24] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

[25] Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.

[26] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[27] Mikko Rautiainen and Tobias Marschall. Aligning sequences to general graphs in O (V+ mE) time. *bioRxiv*, page 216127, 2017.

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

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

[30] Kavya Vaddadi, Naveen Sivadasan, Kshitij Tayal, and Rajgopal Srinivasan. Sequence Alignment On Directed Graphs. *bioRxiv*, page 124941, 2017.

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