Computing the LCP Array of a Labeled Graph

Jarno N. Alanko
University of Helsinki, Finland

Davide Cenzato
Ca' Foscari University of Venice, Italy

Nicola Cotumaccio
University of Helsinki, Finland

Sung-Hwan Kim
Ca' Foscari University of Venice, Italy

Giovanni Manzini
University of Pisa, Italy

Nicola Prezza
Ca' Foscari University of Venice, Italy

Abstract

The LCP array is an important tool in stringology, allowing to speed up pattern matching algorithms and enabling compact representations of the suffix tree. Recently, Conte *et al.* [DCC 2023] and Cotumaccio *et al.* [SPIRE 2023] extended the definition of this array to Wheeler DFAs and, ultimately, to arbitrary labeled graphs, proving that it can be used to efficiently solve matching statistics queries on the graph's paths. In this paper, we provide the first efficient algorithm building the LCP array of a directed labeled graph with n nodes and m edges labeled over an alphabet of size σ. The first step is to transform the input graph G into a deterministic Wheeler pseudoforest G_{is} with $O(n)$ edges encoding the lexicographically- smallest and largest strings entering in each node of the original graph. Using state-of-the-art algorithms, this step runs in $O(\min\{m \log n, m + n^2\})$ time on arbitrary labeled graphs, and in $O(m)$ time on Wheeler DFAs. The LCP array of G stores the longest common prefixes between those strings, i.e. it can easily be derived from the LCP array of G_{is}. After arguing that the natural generalization of a compact-space LCP-construction algorithm by Beller *et al.* [J. Discrete Algorithms 2013] runs in time $\Omega(n\sigma)$ on pseudoforests, we present a new algorithm based on dynamic range stabbing building the LCP array of G_{is} in $O(n \log \sigma)$ time and $O(n \log \sigma)$ bits of working space. Combined with our reduction, we obtain the first efficient algorithm to build the LCP array of an arbitrary labeled graph. An implementation of our algorithm is publicly available at `https://github.com/regindex/Labeled-Graph-LCP`.

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

Keywords and phrases LCP array, Wheeler automata, prefix sorting, pattern matching, sorting

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.1

Related Version *Full Version*: `https://arxiv.org/abs/2404.14235`

Supplementary Material *Software*: `https://github.com/regindex/Labeled-Graph-LCP`

Funding *Jarno Alanko, Nicola Cotumaccio*: Funded by the Helsinki Institute for Information Technology (HIIT).
Davide Cenzato, Sung-Hwan Kim, Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

© Jarno N. Alanko, Davide Cenzato, Nicola Cotumaccio, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza;
licensed under Creative Commons License CC-BY 4.0

Giovanni Manzini: Funded by the Italian Ministry of Health, POS 2014-2020, project ID T4-AN-07, CUP I53C22001300001, by INdAM-GNCS Project CUP E53C23001670001 and by PNRR ECS00000017 Tuscany Health Ecosystem, Spoke 6 CUP I53C22000780001.

1 Introduction

The LCP array of a string – storing the lengths of the longest common prefixes of lexicographic-adjacent string suffixes – is a data structure introduced by Manber and Myers in [23] that proved very useful in tasks such as speeding up pattern matching queries with suffix arrays [23] and representing more compactly the suffix tree [1]. As a more recent application of this array, Boucher *et al.* augmented the BOSS representation of a de Bruijn graph with a generalization of the LCP array that supports the navigation of the underlying variable-order de Bruijn graph [7]. Recently, Conte *et al.* [9] and Cotumaccio [14] extended this structure to Wheeler DFAs [19] – deterministic edge-labeled graphs admitting a total order of their nodes being compatible with the co-lexicographic order of the strings labeling source-to-node paths – and, ultimately, arbitrary edge-labeled graphs. The main idea when generalizing the LCP array to a labeled graph, is to collect the lexicographic smallest $\inf_u$ and largest $\sup_u$ string entering each node u (following edges backwards, starting from u), sorting them lexicographically, and computing the lengths of the longest common prefixes between lexicographically-adjacent strings in this sorted list [9]. As shown by Conte *et al.* [9] and Cotumaccio *et al.* [14, 16], such a data structure can be used to efficiently find Maximal Exact Matches (MEMs) on the graph's paths and to speed up the navigation of variable-order de Bruijn graphs.

Importantly, [9, 14, 16] did not discuss efficient algorithms for building the LCP array of a labeled graph. The goal of our paper is to design such algorithms.

Overview of our contributions

Let G be a directed labeled graph with n nodes and m edges labeled over an alphabet of cardinality σ. After introducing the main definitions and notation in Section 2, in Section 3 we describe our main three-steps pipeline for computing the LCP array of G: (1) *Pre-processing*. Using the algorithms described in [2, 5, 12] we transform G into a particular deterministic *Wheeler pseudoforest* G_{is} of size $O(n)$, that is, a deterministic Wheeler graph [19] whose nodes have in-degree equal to 1. Graph G_{is} compactly encodes the lexicographically- smallest and largest strings $\inf_u$ and $\sup_u$ (Definition 4) leaving backwards (i.e. following reversed edges) every node u of G, by means of the unique path entering the node. Using state-of-the-art algorithms, this step runs in $O(m)$ time and $O(m)$ words of space if G is a Wheeler semi-DFA [2], and in $O(\min\{m + n^2, m \log n\})$ time and $O(m)$ words of space on arbitrary deterministic graphs [5, 12]. (2) *LCP computation*: we describe a new compact-space algorithm computing the LCP array of G_{is} (see below for more details); (3) *Post-processing*: we turn the LCP array of G_{is} into the LCP array of the original graph G, in $O(m)$ time and $O(m)$ words of space.

As far as step (2) is concerned, we turn our attention to the algorithm of Beller *et al.* [6], working on strings in $O(n \log \sigma)$ time and $O(n \log \sigma)$ bits of working space on top of the LCP array (which can be streamed to the output in the form of pairs $(i, LCP[i])$ sorted by their second component). The idea of this algorithm is to keep a queue of suffix array intervals, initially filled with the interval $[1, n]$. After extracting from the queue an interval $[i, j]$ corresponding to all suffixes prefixed by some string α, the algorithm retrieves all *distinct* characters $c_1, \dots, c_k$ in the Burrows-Wheeler transform [8] interval $BWT[i, j]$ and, by means of backward searching [18], retrieves the intervals of strings $c_1 \cdot \alpha, \dots, c_k \cdot \alpha$, writing an LCP entry at the beginning of each of those intervals (unless that LCP entry was not already filled, in which case the procedure does not recurse on the corresponding interval). Recently, Alanko *et al.* [3] generalized this algorithm to the BWT of the infimum sub-graph of a de Bruijn graph (also known as the SBWT [4]). In Section 4 we first show a pseudoforest on which the natural generalization of Beller *et al.*'s algorithm [6] performs $\Omega(n\sigma)$ steps of forward search; this implies that, as a function of σ, this algorithm is *exponentially* slower on graphs than it is on strings. Motivated by this fact, we revisit the algorithm. For simplicity, in this paragraph, we sketch the algorithm on strings; see Section 4 for the generalization to graphs. Rather than working with a queue of suffix array intervals, we maintain a queue of LCP indexes $i \in [1, n]$. We furthermore keep a dynamic range-stabbing data structure on all the open intervals $(l, r]$ such that $BWT[l] = BWT[r] = c$ for some character c, and no other occurrence of c appears in $BWT[l, r]$. When processing position i with $LCP[i] = \ell$, we remove the intervals $(l, r]$ stabbed by i, and use them to derive new LCP entries of value $\ell + 1$ (and new positions to be inserted in the queue) by backward-stepping from $BWT[l]$ and $BWT[r]$. On deterministic Wheeler pseudoforests, our algorithm runs in $O(n \log \sigma)$ time and uses $O(n \log \sigma)$ bits of working space (the LCP array can be streamed to output as in the case of Beller *et al.* [6]).

Putting everything together (preprocessing, LCP of G_{is}, and post-processing), we prove (see Section 2 for all definitions):

▶ **Theorem 1.** *Given a labeled graph G with n nodes and m edges labeled over alphabet $[\sigma]$, with $\sigma \leq m^{O(1)}$, we can compute the LCP array of G in $O(m)$ words of space and $O(n \log \sigma + \min\{m \log n, m + n^2\})$ time. If G is a Wheeler semi-DFA, the running time reduces to $O(n \log \sigma + m)$.*

If the input graph G is a Wheeler pseudoforest with all strings $\inf_u$ being distinct, represented compactly as an FM-index of a Wheeler graph [19], we can do even better: our algorithm terminates in $O(n \log \sigma)$ time while using just $O(n \log \sigma)$ bits of working space (Lemma 15).

We implemented our algorithm computing the LCP array of G_{is} (Algorithm 1) and made it publicly available at `https://github.com/regindex/Labeled-Graph-LCP`.

Due to space constraints, some proofs can be found in the full version of the paper.

2 Preliminaries

We work with directed edge-labeled graphs $G = (V, E)$ on a fixed ordered alphabet $\Sigma = [\sigma] = \{1, \ldots, \sigma\}$, where $E \subseteq V \times V \times \Sigma$ and, without loss of generality, $V = [n]$ for some integer $n > 0$. Symbol $n = |V|$ denotes therefore the number of nodes of G. With $m = |E|$ we denote the number of edges of G. Without loss of generality, we assume that there are no nodes with both in-degree and out-degree equal to zero (such nodes can easily be treated separately in the problem we consider in this paper). In particular, this implies that $n \in O(m)$. We require that the alphabet's size is polynomial in the input's size: $\sigma \leq m^{O(1)}$. Notation in_u and out_u, for $u \in V$, indicates the in-degree and out-degree of u, respectively. We say that G is *deterministic* if $(u, v', a), (u, v'', b) \in E \Rightarrow a \neq b$ whenever $v' \neq v''$. We say that any node $u \in V$ with $\text{in}_u = 0$ is a *source*, that G is a *semi-DFA* if G is deterministic, has exactly one source, and all nodes are reachable from the source, and that G is a *pseudoforest* if and only if $\text{in}_u = 1$ for all $u \in V$. If $G = (V, E)$ is a pseudoforest, $\lambda(u) \in \Sigma$, for $u \in V$, denotes the character labeling the unique edge entering v. We say that two labeled graphs $G = (V, E), G' = (V', E')$ on the same alphabet Σ are *isomorphic* if and only if there exists a bijection $\phi : V \to V'$ preserving edges and labels: for every $u, v \in V, u', v' \in V', a \in \Sigma$, $(u, v, a) \in E$ if and only if $(\phi(u), \phi(v), a) \in E'$.

If $\alpha = c_1c_2\cdots c_n \in \Sigma^*$ is a finite string, the notation $\overleftarrow{\alpha} = c_nc_{n-1}\cdots c_1$ indicates α reversed. The notation Σ^ω indicates the set of *omega strings*, that is, right-infinite strings of the form $c_1c_2c_3\ldots$, with $c_i \in \Sigma$ for all $i \in \mathbb{N}^{>0}$. As usual, Σ^* and Σ^+ denote the sets of finite (possibly empty) strings and the set of nonempty finite strings from Σ, respectively. For $\alpha = c_1c_2\cdots \in \Sigma^\omega \cup \Sigma^*$, $\alpha[i\ldots]$ indicates the suffix $c_ic_{i+1}\ldots$ of α. If $\alpha, \beta \in \Sigma^\omega \cup \Sigma^*$, we write $\alpha \prec \beta$ to indicate that α is lexicographically smaller than β (similarly for $\preceq$: lexicographically smaller than or equal to). Symbol ϵ denotes the empty string, and it holds $\epsilon \prec \alpha$ for all $\alpha, \beta \in \Sigma^\omega \cup \Sigma^+$. Given a set $S \subseteq \Sigma^\omega \cup \Sigma^*$ and a string $\alpha \in \Sigma^\omega \cup \Sigma^*$, notation $S \prec \alpha$ indicates $(\forall \beta \in S)(\beta \prec \alpha)$ (similarly for $\alpha \prec S$, $S \preceq \alpha$, and $\alpha \preceq S$).

If $G = (V, E)$ is deterministic, given $u \in V$ we denote with $\mathsf{OUTL}(u)$ the sorted string of characters labeling outgoing edges from u: $\mathsf{OUTL}(u) = c_1 \ldots c_k$ if and only if $(\forall j \in [k])(\exists v \in V)\big((u, v, c_j) \in E\big)$, with $c_1 \prec c_2 \cdots \prec c_k$. We write $c \in \mathsf{OUTL}(u)$, with $c \in \Sigma$, as a shorthand for $c \in \{\mathsf{OUTL}(u)[1], \ldots, \mathsf{OUTL}(u)[k]\}$. Since we assume that G is deterministic when we use this notation, it holds $\mathrm{out}_u = |\mathsf{OUTL}(u)|$.

Wheeler graphs were introduced by Gagie *et al.* [19]:

▶ **Definition 2.** *Let $G = (V, E)$ be an edge-labeled graph, and let $<$ be a strict total order on V. For every $u, v \in V$, let $u \le v$ indicate $u < v \vee u = v$. We say that $<$ is a* Wheeler order *for G if and only if:*

1. *(Axiom 1) For every $u, v \in V$, if $\mathrm{in}_u = 0$ and $\mathrm{in}_v > 0$ then $u < v$.*
2. *(Axiom 2) For every $(u', u, a), (v', v, b) \in E$, if $u < v$, then $a \preceq b$.*
3. *(Axiom 3) For every $(u', u, a), (v', v, a) \in E$, if $u < v$, then $u' \le v'$.*

A graph $G = (V, E)$ is Wheeler if it admits at least one Wheeler order.

Let $u, v \in V$, $\alpha \in \Sigma^+$, and $c \in \Sigma$. We write $u \overset{\alpha}{\rightsquigarrow} v$ to indicate that there exists a path from u to v labeled with string α (that is, we can go from u to v by following edges whose labels, when concatenated, yield α), and write $u \xrightarrow{c} v$ as an abbreviation for $(u, v, c) \in E$.

We use the symbol I_u to denote the set of strings obtained starting from node u and following edges backwards. This process either stops at a node with in-degree 0 (thereby producing a finite string), or continues indefinitely (thereby producing an omega-string). Notice that this notation differs from [2], where edges are followed forwards; we use this slight variation since, as seen below, it is a more natural way to define the LCP array of a graph. More formally:

▶ **Definition 3.** *Let $G = (V, E)$ be a labeled graph. For $u \in V$, $I^\omega_u \subseteq \Sigma^\omega$ denotes the set of* omega-strings *leaving backwards node u:*

$$I^\omega_u = \{c_1c_2\ldots \in \Sigma^\omega \;:\; (\exists v_1, v_2, \cdots \in V)(v_1 \xrightarrow{c_1} u \wedge (\forall i \ge 2)(v_i \xrightarrow{c_i} v_{i-1}))\}.$$

The symbol $I_u \subseteq \Sigma^\omega \cup \Sigma^*$ denotes instead the set of all *strings leaving backwards u:*

$$I_u = I^\omega_u \cup \{\overleftarrow{\alpha} \in \Sigma^+ \;:\; (\exists v \in V)(\mathrm{in}_v = 0 \wedge v \overset{\alpha}{\rightsquigarrow} u)\};$$

If $\mathrm{in}_u = 0$, we define $I_u = \{\epsilon\}$.

▶ **Definition 4** (Infimum and supremum strings [22])**.** *Let $G = (V, E)$ and $u \in V$. The infimum string $\inf^G_u = \inf I_u$ and the supremum string $\sup^G_u = \sup I_u$ relative to G are defined as:*

$$\inf\nolimits^G_u = \gamma \in \Sigma^* \cup \Sigma^\omega \;\; s.t. \;\; (\forall \beta \in \Sigma^* \cup \Sigma^\omega)(\beta \preceq I_u \rightarrow \beta \preceq \gamma \preceq I_u)$$
$$\sup\nolimits^G_u = \gamma \in \Sigma^* \cup \Sigma^\omega \;\; s.t. \;\; (\forall \beta \in \Sigma^* \cup \Sigma^\omega)(I_u \preceq \beta \rightarrow I_u \preceq \gamma \preceq \beta)$$

When G will be clear from the context, we will drop the superscript and simply write $\inf_u$ *and* $\sup_u$.

Cotumaccio [14] defines the LCP of a labeled graph as follows:

▶ **Definition 5.** *Let $G = (V, E)$ be a labeled graph. Let $\gamma_1 \preceq \gamma_2 \preceq \cdots \preceq \gamma_{2n}$ be the lexicographically-sorted strings* $\inf_u$ *and* $\sup_u$*, for all $u \in V$. The* LCP *array* $\mathsf{LCP}_G[2, 2n]$ *of G is defined as* $\mathsf{LCP}_G[i] = \mathsf{lcp}(\gamma_{i-1}, \gamma_i)$*, where* $\mathsf{lcp}(\gamma_{i-1}, \gamma_i)$ *is the length of the longest common prefix between γ_{i-1} and γ_i.*

In the above definition, observe that $\mathsf{LCP}_G[i] = \infty$ if and only if $\gamma_{i-1} = \gamma_i$ and $\gamma_{i-1}, \gamma_i \in \Sigma^\omega$. We are interested in this definition since, as shown in [14], it allows computing matching statistics on arbitrary labeled graphs.

If $G = (V, E)$ is a pseudoforest, then I_u is a singleton for every $u \in V$ and the above definition can be simplified since $\inf_u = \sup_u$ for every $u \in V$. In this paper we will consider the particular case of pseudoforests for which all the $\inf_u (= \sup_u)$ are distinct. In this particular case, we define the *reduced LCP array* as follows:

▶ **Definition 6.** *Let $G = (V, E)$ be a labeled pseudoforest such that $\inf_u \neq \inf_v$ for all $u \neq v \in V$. Let $\gamma^*_1 \prec \gamma^*_2 \prec \cdots \prec \gamma^*_n$ be the lexicographically-sorted strings $\inf_u$, for all $u \in V$. The* reduced LCP array $\mathsf{LCP}^*_G = \mathsf{LCP}^*_G[2, n]$ *of G is defined as* $\mathsf{LCP}^*_G[i] = \mathsf{lcp}(\gamma^*_{i-1}, \gamma^*_i)$*.*

Since pseudoforests will play an important role in our algorithms, we proceed by proving a useful property of the reduced LCP array of a pseudoforest.

▶ **Lemma 7.** *Let $G = (V, E)$ be a labeled pseudoforest such that $\inf_u \neq \inf_v$ for all $u \neq v \in V$. Let $1 \leq k < \infty$ be such that $\mathsf{LCP}^*_G[i] = k$ for some $2 \leq i \leq n$. Then, there exists $2 \leq i' \leq n$ such that $\mathsf{LCP}^*_G[i'] = k - 1$.*

Proof. Let $\gamma^*_1 \prec \gamma^*_2 \prec \cdots \prec \gamma^*_n$ be the lexicographically-sorted strings $\inf_u$, for all $u \in V$. For all $i \in [n]$ let $1 \leq p(i) \leq n$ be the unique integer such that $\gamma^*_{p(i)} = \gamma^*_i[2\ldots]$. Such integer always exists and it is unique, since the γ^*_j's are all the strings leaving each node of G (following edges backwards).

Since $1 \leq k < \infty$, then $\gamma^*_{i-1}[1] = \gamma^*_i[1]$ and $\gamma^*_{p(i-1)} = \gamma^*_{i-1}[2\ldots] \prec \gamma^*_i[2\ldots] = \gamma^*_{p(i)}$, which implies $p(i-1) < p(i)$. As a consequence, $k = \mathsf{LCP}^*_G[i] = \mathsf{lcp}(\gamma^*_{i-1}, \gamma^*_i) = 1 + \mathsf{lcp}(\gamma^*_{p(i-1)}, \gamma^*_{p(i)}) = 1 + \min_{p(i-1)+1 \leq t \leq p(i)} \mathsf{lcp}(\gamma^*_{t-1}, \gamma^*_t) = 1 + \min_{p(i-1)+1 \leq t \leq p(i)} \mathsf{LCP}^*_G[t]$, so there exists $p(i-1) + 1 \leq i' \leq p(i)$ such that $k = 1 + \mathsf{LCP}^*_G[i']$, or equivalently, $\mathsf{LCP}^*_G[i'] = k - 1$. ◀

3 The pipeline: computing LCP_G from G

As mentioned in Section 1, we reduce the computation of LCP_G to three steps: (1) a pre-processing phase building a Wheeler pseudoforest G_{is}, (2) the computation of the reduced LCP array of G_{is}, and (3) a post-processing phase yielding LCP_G. Steps (1) and (3) mainly use existing results from the literature and we illustrate them in this section. Our main contribution is step (2) for which in Section 4 we describe a new algorithm.

3.1 Pre-processing

Let $G = (V, E)$ be the input graph. As the first step of our pre-processing phase, we augment $\Sigma \leftarrow \Sigma \cup \{\#\}$ with a new symbol $\#(= 0)$ lexicographically-smaller than all symbols in the original alphabet $\{1, \ldots, \sigma\}$, and add a self-loop $u \xrightarrow{\#} u$ to all nodes $u \in V$ such that $\mathrm{in}_u = 0$. This will simplify our subsequent steps as now all strings leaving (backwards) any node belong to Σ^ω; from now on, we will therefore work with graphs with no sources.

Consider the set $IS = \{\inf_u, \sup_u : u \in V\}$, and let $N = |IS| \leq 2n$. Note that N could be strictly smaller than $2n$ since some nodes may share the same infimum/supremum string. Moreover, by the definition of infima and suprema strings, for each $\alpha \in IS$ it holds that $\alpha[2\dots] \in IS$: this is true because, if α is the infimum of I_u, then u has a predecessor v such that (i) $v \stackrel{\alpha[1]}{\to} u$ and (ii) $\alpha[2\dots]$ is the infimum of I_v (the same holds for suprema strings). Let us give the following definition.

▶ **Definition 8.** *Given $G = (V, E)$ with no sources, let $IS = \{\inf_u, \sup_u : u \in V\}$. We denote with $G'_{is} = (IS, E'_{is})$ the labeled graph with edge set $E'_{is} = \{(\alpha[2\dots], \alpha, \alpha[1]) : \alpha \in IS\}$.*

Observe that (i) each node $\alpha \in IS$ of G'_{is} has exactly one incoming edge, and (ii) G'_{is} is deterministic since $\alpha \stackrel{a}{\to} \beta$ and $\alpha \stackrel{a}{\to} \beta'$ imply $\beta = a \cdot \alpha = \beta'$. In other words, G'_{is} is a deterministic pseudo-forest. In addition, $\inf_u \neq \inf_v$ for every $u \neq v \in IS$. Let us prove that G'_{is} is a Wheeler graph.

▶ **Lemma 9.** *The lexicographic order $\prec$ on the nodes of G'_{is} is a Wheeler order.*

Proof. We prove that $\prec$ satisfies the three axioms of Definition 2.

(Axiom 1). This axiom holds trivially, since G'_{is} has no nodes with in-degree 0.

(Axiom 2). Let $(\alpha', \alpha, \alpha[1]), (\beta', \beta, \beta[1]) \in E'_{is}$, with $\alpha \prec \beta$. Then, by definition of the lexicographic order $\prec$, it holds $\alpha[1] \preceq \beta[1]$.

(Axiom 3). Let $(\alpha[2\dots], \alpha, a), (\beta[2\dots], \beta, a) \in E'_{is}$, with $a = \alpha[1] = \beta[1]$ and $\alpha \prec \beta$. Then, by the definition of $\prec$ it holds $\alpha[2\dots] \prec \beta[2\dots]$. ◀

Importantly, we remark that our subsequent algorithms will only require the topology and edge labels of G'_{is} to work correctly. In other words, any graph *isomorphic* to G'_{is} will work, and it will not be needed to compute *explicitly* the set IS (an impractical task, since IS contains omega-strings). Figure 1 shows an example of labeled graph G (with sources) pre-processed to remove sources and converted to such a graph isomorphic to G'_{is}. We can compute such a graph using recent results in the literature:

▶ **Theorem 10.** *Let $G = (V, E)$ be a labeled graph with no sources and alphabet of size $\sigma \leq m^{O(1)}$. Let moreover $V_i = \{u_i : u \in V\}$ and $V_s = \{u_s : u \in V\}$ be two duplicates of V. Then, we can compute a graph $G_{is} = ([N], E_{is})$ being isomorphic to $G'_{is} = (IS, E'_{is})$ (Definition 8), together with a function $\mathsf{map} : V_i \cup V_s \to [N]$ such that:*

1. *for every $u \in V$, $\inf^G_u = \inf^{G_{is}}_{\mathsf{map}(u_i)}$ and $\sup^G_u = \sup^{G_{is}}_{\mathsf{map}(u_s)}$, and*
2. *the total order $<$ on the integers coincides with the Wheeler order of G_{is}. In particular, for all $i, j \in [N]$, $i < j$ if and only if $\inf^{G_{is}}_i = \sup^{G_{is}}_i \prec \inf^{G_{is}}_j = \sup^{G_{is}}_j$.*

Function map is returned as an array of $2n = 2|V|$ words, so that it can be evaluated in constant time. G_{is} and map can be computed from G in $O(m + n^2)$ time [12] or in $O(m \log n)$ time [5]. If G is a Wheeler semi-DFA, the running time reduces to $O(m)$ [2]. All these algorithms use $O(m)$ words of working space.

In the full version of the paper we discuss how Theorem 10 can be obtained using the results in [2, 5, 12] (which were originally delivered for a different purpose: computing the maximum co-lex order [11, 13, 15, 17]). Figure 1 shows an example of G_{is} (right) for a particular labeled graph G (left). Table 1 (right) shows the nodes $[N]$ of such a graph G_{is}, together with the strings entering in each node, sorted lexicographically, and their longest common prefix array $\mathsf{LCP}^*_{G_{is}}$. In Table 1 (left) we sort the duplicated nodes ($V_i \cup V_s$) of G by their infima (V_i) and suprema (V_s) strings, and show the mapping $\mathsf{map} : V_i \cup V_s \to [N]$ in the second and third columns.

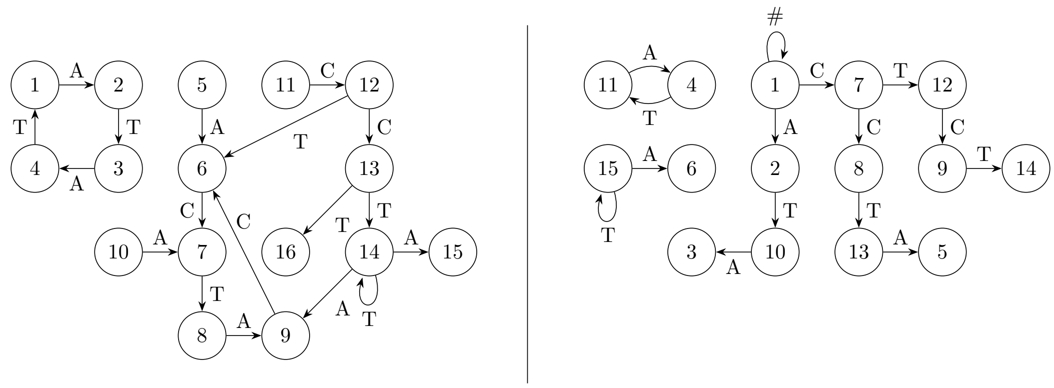

Figure 1 *Left*: a labeled graph G. *Right*: a graph G_{is} isomorphic to G'_{is} (Definition 8) satisfying Theorem 10. Note that in G_{is} the node numbering coincides with the Wheeler order.

Table 1 *Left:* lexicographically-sorted infima and suprema strings γ_j of graph $G = (V, E)$ of Figure 1, along with the nodes of V they reach (subscripted using the duplicates V_i and V_s of V to show whether the string is an infimum or a supremum), and array LCP_G. The second ($[N]$) and third ($V_i \cup V_s$) columns of the table show the mapping $\mathsf{map} : V_i \cup V_s \to [N]$. *Right:* The sorted infima (equivalently, suprema) γ^*_j of graph G_{is} of Figure 1, and the array $\mathsf{LCP}^*_{G_{is}}$.

j	$[N]$	$V_i \cup V_s$	γ_j	LCP_G	j	$[N]$	$V_i \cup V_s$	γ_j	LCP_G
1	1	10_i	ε	-	17	7	12_i	C	0
2	1	10_s	ε	0	18	7	12_s	C	1
3	1	11_i	ε	0	19	8	13_i	CC	1
4	1	11_s	ε	0	20	8	13_s	CC	2
5	1	5_i	ε	0	21	9	7_s	CTC	1
6	1	5_s	ε	0	22	10	8_i	TA	0
7	2	6_i	A	0	23	11	1_i	TATAT...	2
8	2	7_i	A	1	24	11	1_s	TATAT...	∞
9	3	9_i	ATA	1	25	11	3_i	TATAT...	∞
10	4	2_i	ATATA...	3	26	11	3_s	TATAT...	∞
11	4	2_s	ATATA...	∞	27	12	6_s	TC	1
12	4	4_i	ATATA...	∞	28	13	14_i	TCC	2
13	4	4_s	ATATA...	∞	29	13	16_i	TCC	3
14	5	15_i	ATCC	2	30	13	16_s	TCC	3
15	6	15_s	ATTTT...	2	31	14	8_s	TCTC	2
16	6	9_s	ATTTT...	∞	32	15	14_s	TTTTT...	1

$[N]$	γ^*_j	$\mathsf{LCP}^*_{G_{is}}$
1	#####...	-
2	A####...	0
3	ATA##...	1
4	ATATA...	3
5	ATCC#...	2
6	ATTTT...	2
7	C####...	0
8	CC###...	1
9	CTC##...	1
10	TA###...	0
11	TATAT...	2
12	TC###...	1
13	TCC##...	2
14	TCTC#...	2
15	TTTTT...	1

3.2 Post-processing

In Section 3.1 we have shown how to convert the input $G = (V, E)$ into a Wheeler pseudoforest $G_{is} = ([N], E_{is})$ encoding the infima and suprema strings of G. In Section 4 we will show how to compute the reduced $\mathsf{LCP}^*_{G_{is}}$ of G_{is}. Here, we discuss the last step of our pipeline, converting $\mathsf{LCP}^*_{G_{is}}$ into LCP_G.

Let $\gamma^*_1 \prec \gamma^*_2 \prec \cdots \prec \gamma^*_N$ be the sorted strings $\inf_j$, for $j \in [N]$, relative to graph G_{is}, and $\gamma_1 \preceq \gamma_2 \preceq \cdots \preceq \gamma_{2n}$ be the sorted strings $\inf_u, \sup_u$, for $u \in V$, relative to graph G. By construction (Section 3.1), the former sequence of strings is obtained from the latter by performing these two operations: (1) duplicates are removed, and (2) finite strings γ_i are turned into omega-strings of the form $\gamma_i \cdot \#^\omega$ (see Table 1, right table). This means that LCP_G is almost the same as $\mathsf{LCP}^*_{G_{is}}$, except in maximal intervals $\mathsf{LCP}_G[i+1, j]$ such that $\gamma_i = \gamma_{i+1} = \cdots = \gamma_j$. In those intervals, we have $\mathsf{LCP}_G[i+1, j] = |\gamma_i|$; notice that this value could be either finite (if $\gamma_i \in \Sigma^*$) or infinite (if $\gamma_i \in \Sigma^\omega$).

As an example, consider the interval $\mathsf{LCP}_G[10, 13] = (3, \infty, \infty, \infty)$ in Tabe 1 (left). This interval corresponds to strings `ATATA...`, and corresponds to $\mathsf{LCP}^*_{G_{is}}[4] = 3$ (right). The first value $\mathsf{LCP}_G[10]$ is equal to $\mathsf{LCP}^*_{G_{is}}[4] = 3$, while the others, $\mathsf{LCP}_G[11, 13] = (\infty, \infty, \infty)$ are equal to the length (∞) of the omega-string `ATATA...`. A similar example not involving an omega string is $\mathsf{LCP}_G[7, 8] = (0, 1)$ (string `A`), corresponding to $\mathsf{LCP}^*_{G_{is}}[2] = 0$.

Given $\mathsf{LCP}^*_{G_{is}}$, G_{is}, and map (see Theorem 10) it is immediate to derive LCP_G in $O(m)$ worst-case time and $O(m)$ words of working space, as follows. First of all, we sort $V_i \cup V_s$ (the two duplicates of V, see Theorem 10) according to the order given by the integers $\mathsf{map}(x)$, for $x \in V_i \cup V_s$. Let $u^1, u^2, \ldots, u^{2n}$ be the corresponding sequence of sorted nodes, i.e. such that $\mathsf{map}(u^1) \le \mathsf{map}(u^2) \le \cdots \le \mathsf{map}(u^{2n})$. The second step is to identify the above-mentioned maximal intervals $\mathsf{LCP}_G[i + 1, j]$: these are precisely the maximal intervals such that $\mathsf{map}(u^i) = \mathsf{map}(u^{i+1}) = \cdots = \mathsf{map}(u^j)$. For each such interval, we set $\mathsf{LCP}_G[i] = \mathsf{LCP}^*_{G_{is}}[\mathsf{map}(u^i)]$ and $\mathsf{LCP}_G[i + 1] = \mathsf{LCP}_G[i + 2] = \ldots \mathsf{LCP}_G[j] = |\gamma_i|$. In order to compute the length $|\gamma_i|$, observe that $\gamma_i = \gamma^*_{\mathsf{map}(u^i)}$ if $\gamma_i \in \Sigma^\omega$, and $\gamma_i \cdot \#^\omega = \gamma^*_{\mathsf{map}(u^i)}$ if $\gamma_i \in \Sigma^*$. Then, a simple DFS visit of G_{is} starting from nodes u with a self-loop $u \xrightarrow{\#} u$ reveals whether we are in the former or latter case, and allows computing the length (DFS depth) of γ_i in the latter.

4 Computing the LCP array of G_{is}

In view of Sections 3.1 and 3.2, the core task to solve in order to compute LCP_G is the computation of $\mathsf{LCP}^*_{G_{is}}$. To make notation lighter, in this section we denote the input graph with symbol $G = (V, E)$ and assume it is a deterministic Wheeler pseudoforest such that $\inf_u \ne \inf_v$ for all $u \ne v \in V$. In the rest of the section, n and m denote the number of nodes and edges of G. The goal of our algorithms is to compute LCP^*_G.

To solve this problem, we first focused our attention on the algorithm of Beller *et al.* [6], computing the LCP array from the Burrows-Wheeler Transform [8] of the input string in $(1 + o(1)) \cdot n \log \sigma + O(n)$ bits of working space (including the indexed BWT and excluding the LCP array, which however can be streamed to output in order of increasing LCP values) and $O(n \log \sigma)$ time. However, as we briefly show in Figure 2, we realized that the natural generalization of this algorithm to pseudoforests runs in $\Omega(n\sigma)$ time in the worst case. Motivated by this fact, in this section we re-design the algorithm by resorting to the dynamic interval stabbing problem (e.g., see [24]), achieving running time $O(n \log \sigma)$ on deterministic Wheeler pseudoforests. This will require designing a novel dynamic range-stabbing data structure that could be of independent interest.

Let $v_1 < v_2 < \cdots < v_n$ denote the Wheeler order of G. We define:

▶ **Definition 11** (bridge). *For $1 \le l < r \le n$ and $c \in \Sigma$, a triplet (l, r, c) is said to be a* bridge *of G if and only if (i) both v_l and v_r have an outgoing edge labeled with c and (ii) for every k such that $l < k < r$, v_k does not have any outgoing edge labeled with c.*

Consider two consecutive nodes (in the Wheeler order) v_{i-1} and v_i with the same incoming label $c = \lambda(v_{i-1}) = \lambda(v_i)$ for some $1 < i \le n$. On deterministic pseudoforests there is a one-to-one correspondence between the set of such node pairs and the set of bridges:

▶ **Lemma 12.** *The following bijection exists between the set of bridges and the set of pairs (v_{i-1}, v_i) such that $\lambda(v_{i-1}) = \lambda(v_i)$. Let $\lambda(v_{i-1}) = \lambda(v_i) = c$, and let v_l and v_r be the nodes such that $v_l \xrightarrow{c} v_{i-1}$ and $v_r \xrightarrow{c} v_i$. Then, (l, r, c) is a bridge. Conversely, for every bridge (l, r, c), letting v_h and v_i be the unique nodes such that $v_l \xrightarrow{c} v_h$ and $v_r \xrightarrow{c} v_i$, we have $h = i - 1$.*

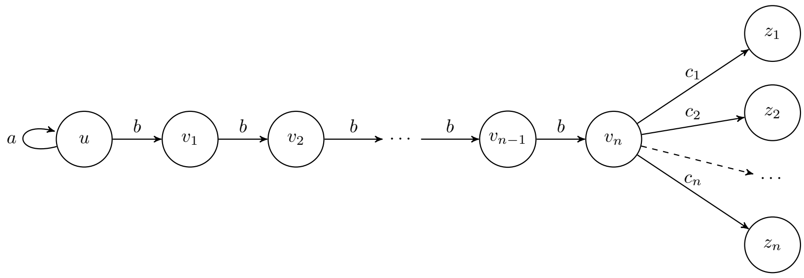

Figure 2 A Wheeler pseudoforest with $V = \{u < v_1 < \cdots < v_n < z_1 < \cdots < z_n\}$ and Wheeler order $<$ where the number of forward search steps in the natural generalization of Beller *et al.*'s algorithm is $\Theta(n\sigma)$. The range of nodes (BWT interval) reached by a path suffixed by b^i, for $1 \le i \le n$, is $[v_i, v_n]$. The algorithm right-extends (via *forward search*) each of these ranges by all the n characters $c_1, \ldots, c_n$; when extending with c_i, we obtain the unary range $[z_i]$ of nodes reached by a path suffixed by $b^i c_i$. This means that Beller *et al.*'s algorithm will perform in total $\Theta(n\sigma) = \Theta(n^2)$ forward search steps. Intuitively, our solution to this problem will be to extend *just one* of those ranges by $c_1, \ldots, c_n$. We achieve this by resorting to a dynamic range stabbing data structure.

Proof. Given such nodes v_{i-1} and v_i, v_l and v_r are unambiguously determined because every node has exactly one incoming edge. To see that $(l, r, \lambda(v_i))$ is a bridge, notice that for every edge $(u, u', \lambda(v_i))$, if $u' < v_{i-1}$, then by Axiom 3 and determinism $u < v_l$, and if $v_i < u'$, then similarly $v_r < u$, so for every k such that $l < k < r$, v_k does not have an outgoing edge labeled $\lambda(v_i)$. For the reverse implication, notice that $h \neq i$ because every node has exactly one incoming edge, and it cannot be $i < h$ otherwise from Axiom 3 we would obtain $v_r < v_l$. Hence $h \le i - 1$. If we had $h < i - 1$, then by Axiom 2 the unique edge entering v_{i-1} should be labeled c, and from Axiom 3 and determinism its start node k should satisfy $l < k < r$, so (l, r, c) would not be a bridge. ◀

The intuition behind our algorithm is as follows. Given a bridge (l, r, c), suppose that $\mathsf{lcp}(\gamma_l^*, \gamma_r^*) = d \ge 0$. Let v_{i-1} and v_i be the nodes such that $v_l \xrightarrow{c} v_{i-1}$ and $v_r \xrightarrow{c} v_i$ (see Lemma 12). These nodes can be obtained from v_l and v_r by one forward search step. Then, we have $\gamma_{i-1}^* = c\gamma_l^*$ and $\gamma_i^* = c\gamma_r^*$, thus $\mathsf{lcp}(\gamma_{i-1}^*, \gamma_i^*) = \mathsf{lcp}(\gamma_l^*, \gamma_r^*) + 1 = d + 1$. By Lemma 9, the Wheeler order $v_1 < \cdots < v_n$ corresponds to the lexicographic order of the node's incoming strings $\gamma_1^* \prec \cdots \prec \gamma_n^*$, hence $\mathsf{lcp}(\gamma_l^*, \gamma_r^*) = \min_{j \in (l,r]} \mathsf{lcp}(\gamma_{j-1}^*, \gamma_j^*) = \min_{j \in (l,r]} \mathsf{LCP}_G^*[j]$. Therefore, the minimum value $d = \min_{j \in (l,r]} \mathsf{LCP}_G^*[j]$ within the left-open interval $(l, r]$ corresponding to a bridge (l, r, c) yields $\mathsf{LCP}_G^*[i] = \mathsf{lcp}(\gamma_{i-1}^*, \gamma_i^*) = d + 1$. This observation stands at the core of our algorithm: if we compute LCP values in nondecreasing order, then the position $j_{min} = \mathrm{argmin}_{j \in (l,r]} \mathsf{LCP}_G^*[j]$ $(1 < j_{min} \le n)$ of the first (smallest) generated LCP value inside $(l, r]$ "stabs" interval $(l, r]$. This yields $\mathsf{LCP}_G^*[i] = \mathsf{LCP}_G^*[j_{min}] + 1$. After this interval stabbing query, bridge (l, r, c) is removed from the set of bridges since the resulting LCP value $\mathsf{LCP}_G^*[i]$ has been correctly computed once for all.

Our procedure for computing LCP_G^* is formalized in Algorithm 1. The algorithm takes as input a deterministic Wheeler pseudoforest G represented as an FM index (Lemma 13 below) and outputs all pairs $(i, \mathsf{LCP}_G^*[i])$, one by one in a streaming fashion, in nondecreasing order of their second component (i.e. LCP value). This is useful in space-efficient applications where one cannot afford storing the whole LCP array in $n \log n$ bits, see for example [26].

▶ **Lemma 13.** *A deterministic Wheeler pseudoforest G can be represented succinctly in $(1 + o(1)) \cdot n \log \sigma + O(n)$ bits of space with a data structure (FM index of a Wheeler graph [19]) supporting the following queries in $O(\log \sigma)$ time:*

1. $G.\mathsf{forward\_step}(i, c)$: *given* $1 < i \leq n$ *and a character* $c \in \Sigma$*, let* $k \geq i$ *be the smallest integer such that* v_k *has an outgoing edge labeled with* c*. This query returns the integer* i' *such that* $v_k \xrightarrow{c} v_{i'}$*, or* $\perp$ *if such* k *does not exist.*
2. $\mathsf{OUTL}(v_i)[j]$: *given a node* v_i *and an index* $j \in [\sigma]$*, return the* j*-th outgoing label of* v_i*. Return* $\perp$ *if* $j > \mathrm{out}_{v_i}$*.*
3. $\lambda(v_i)$: *given* $i \in [n]$*, return the incoming label of* v_i*.*

Proof. Following [19], we represent G as a triple $(\mathsf{C}, \mathsf{OUT}, \mathsf{L}) \in \{0, \ldots, n\}^\sigma \times \{0, 1\}^{2n} \times [\sigma]^n$ defined as:

- $\mathsf{OUT} = 0^{\mathrm{out}_{v_1}}1 \cdot 0^{\mathrm{out}_{v_2}}1 \cdots 0^{\mathrm{out}_{v_n}}1$ is the concatenation of the outdegrees of nodes $v_1, \ldots, v_n$, written in unary,
- $\mathsf{L} = \mathsf{OUTL}(v_1) \cdot \mathsf{OUTL}(v_2) \cdots \mathsf{OUTL}(v_n)$ is the concatenation of all the labels of the node's outgoing edges in Wheeler order, and
- $\mathsf{C}[c] = |\{u \in V \ : \ \lambda(u) \prec c\}|$, $c \in \Sigma$, denotes the number of nodes whose incoming edge is labelled with a character c' such that $c' \prec c$. Importantly, C is not actually stored explicitly; as we show below, any $\mathsf{C}[c]$ can be retrieved from L in $O(\log \sigma)$ time.

The only difference with [19] (where arbitrary Wheeler graphs are considered) is that a pseudoforest has n nodes and n edges, and all nodes have in-degree equal to 1. This simplifies the structure, since we do not need to store in-degrees. L is encoded with a wavelet tree [21] and OUT with a succinct bitvector data structure [27]. A root-to leaf traversal of the wavelet tree of L is sufficient to retrieve any $\mathsf{C}[c]$ in $O(\log \sigma)$ time at no additional space cost (see for example [28, Alg 3]). This representation uses $n \log \sigma + O(n)$ bits of space and supports the following operations in $O(\log \sigma)$ time: (1) random access to any of the arrays $\mathsf{C}, \mathsf{OUT}, \mathsf{L}$, (2) $\mathsf{L}.\mathsf{rank}_c(j)$, returning the number of occurrences of c in $\mathsf{L}[1, j]$, and (3) $\mathsf{OUT}.\mathsf{select}_1(j)$, returning the position of the j-th occurrence of bit 1 in bitvector OUT.

Using these queries, we can solve query (1) as follows: $G.\mathsf{forward\_step}(i, c) = \mathsf{C}[c] + \mathsf{L}.\mathsf{rank}_c(\mathsf{OUT}.\mathsf{select}_1(i-1) - (i-1)) + 1$. If $\mathsf{L}.\mathsf{rank}_c(\mathsf{OUT}.\mathsf{select}_1(i-1) - (i-1)) + 1$ exceeds the number of characters equal to c in L (we discover this in $O(\log \sigma)$ time using the wavelet tree on L), the query returns $\perp$. Query (2) is answered as follows: $\mathsf{OUTL}(v_i)[j] = \mathsf{L}[\mathsf{OUT}.\mathsf{select}_1(i-1) - (i-1) + j]$. If j exceeds the out-degree of v_i (we discover this in constant time using rank and select queries on OUT), the query returns $\perp$. Query (3) $\lambda(v_i)$ can be solved with a root-to-leaf visit of the wavelet tree of L, in $O(\log \sigma)$ time (range quantile queries, see [20]). ◀

We proceed by commenting the pseudocode and proving its correctness and complexity. In Line 3 of Algorithm 1, we compute the set $\mathcal{I}$ of bridges of the input graph using Lemma 14. Each bridge (l, r, c) will "survive" in $\mathcal{I}$ until any $\mathsf{LCP}^*_G[i]$ with $i \in (l, r]$ is computed. Set $\mathcal{I}$ is represented as a dynamic range stabbing data structure (Lemma 14 below) on the set of *character-labeled* intervals $\{(l, r]_c \ : \ (l, r, c) \text{ is a bridge}\}$, where notation $(l, r]_c$ indicates a left-open interval labeled with character c. We require this data structure to support interval stabbing and interval deletion queries. General solutions for this problem solving both queries in amortized $O(\log n / \log\log n)$ time exist [24]. While in the general case this is optimal, in our particular case observe that, by Definition 11, no more than σ intervals get stabbed by a given $i \in [n]$. We exploit this property to develop a more efficient (if $\log \sigma = o(\log n / \log\log n)$) dynamic range stabbing data structure (for the full proof, see the full version of the paper):

▶ **Lemma 14.** *Given a Wheeler pseudoforest* G *represented with the data structure of Lemma 13, in* $O(n \log \sigma)$ *time and* $O(n \log \sigma)$ *bits of working space we can build a* dynamic interval stabbing *data structure* $\mathcal{I}$ *of* $O(n \log \sigma)$ *bits representing the set of bridges of* G

Algorithm 1 Given a Wheeler pseudoforest G, compute LCP^*_G. In Line 8, $\mathcal{I}.\,\mathsf{stab\_and\_remove}(i)$ stabs and removes bridges from $\mathcal{I}$.

1: $\mathsf{LCP}^*_G \leftarrow$ Array $\mathsf{LCP}^*_G[2, n]$, with values initialized to ∞
2: $W \leftarrow \emptyset$ ▷ W: queue of integer pairs of the form $(i, \mathsf{LCP}^*_G[i])$
3: $\mathcal{I} \leftarrow \{(l, r, c) \in [n] \times [n] \times \Sigma \ : \ (l, r, c)$ is a bridge of $G\}$ ▷ $\mathcal{I}$: bridges of G
4: **for all** $1 < i \le n$ **such that** $\lambda(v_{i-1}) \prec \lambda(v_i)$: $W.\mathsf{push}(i, 0)$
5: **while** $W \neq \emptyset$ **do**
6: $\quad (i, d) \leftarrow W.\mathsf{pop}()$
7: $\quad$ **output** (i, d) ▷ Stream pair $(i, \mathsf{LCP}^*_G[i])$ to output
8: $\quad R \leftarrow \mathcal{I}.\,\mathsf{stab\_and\_remove}(i)$ ▷ $R \subseteq \Sigma$: set of labels of bridges stabbed and removed
9: $\quad$ **for** $c \in R$ **do**
10: $\qquad i' \leftarrow G.\mathsf{forward\_step}(i, c)$
11: $\qquad W.\mathsf{push}(i', d+1)$
12: $\quad$ **end for**
13: **end while**

(Definition 11) and answering the following query: $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$, where $i \in [n]$. Letting $S = \{(l, r, c) \in \mathcal{I} \ : \ l < i \le r\}$ be the set of stabbed bridges, this query performs the following two operations:

1. *it returns the set of characters $R = \{c \ : \ (l, r, c) \in S\}$ labeling bridges stabbed by i, and*
2. *it removes those bridges: $\mathcal{I} \leftarrow \mathcal{I} \setminus S$.*

Let $\ell = \sum_{(l,r,c)\in S}(r - l + 1)$ be the total length of the stabbed bridges. The query is solved in $O(\log\sigma + |R| + \ell/\sigma)$ time.

Proof (Sketch, see the full paper version for all the details). We divide the nodes $v_1, \ldots, v_n$ into non-overlapping blocks $v_{k\sigma}, \ldots, v_{(k+1)\sigma}$ of σ nodes each, for $k = 0, \ldots, n/\sigma - 1$ (assume σ divides n for simplicity). Let I be the set of labeled intervals $(l, r]_c$ corresponding to all the bridges (l, r, c) of G; the bridges of G can be reconstructed in $O(\log\sigma)$ time each using operation $\mathsf{OUTL}(v_i)[j]$ of Lemma 13. Each interval $(l, r]_c \in I$ overlapping $t > 1$ blocks (i.e. $l + 1 \le k\sigma \le (k + t - 2)\sigma < r$ for some $k \in \{0, \ldots, n/\sigma - 1\}$) is broken into t "pieces" $(l_1 = l, r_1]_c, \ldots, (l_t, r_t = r]_c$: a suffix of a block, followed by full blocks, followed by a prefix of a block. Each piece $(l_i, r_i]_c$, overlapping the k-th block for some k, is inserted in interval set I_k. The pieces $(l_1, r_1]_c, \ldots, (l_t, r_t]_c$ are connected using a doubly-linked list. Intervals of I fully contained in a block are not split in any piece and just inserted in I_k, with k being the block they overlap. Since no more than σ intervals can pairwise intersect at any point $i \in [n]$, for every k at most σ "pieces" of at most σ intervals are inserted in I_k: in total, the number of intervals in all the sets I_k is therefore $\sum_{k=0}^{n/\sigma-1} |I_k| \in O(|I| + \sigma \cdot n/\sigma) = O(n)$ (because $|I| \in O(n)$ by Lemma 12). We build an interval tree data structure $\mathcal{T}(I_k)$ ([25, Ch 8.8], [10, Ch 17.3]) on each I_k. Interval stabbing queries are answered locally (on the tree associated with the block containing the stabbing position i). Interval deletion queries require to also follow the linked list associated with the deleted interval, to delete all ℓ/σ "pieces" of the original interval (of length ℓ) of I. Each interval piece is deleted in constant time since we do not need to re-balance the tree. Our claim follows by observing that $|I_k| \in O(\sigma^2)$ for every k (because no more than σ intervals can pairwise intersect at any point), so (i) each $\mathcal{T}(I_k)$ is built in $O(|I_k| \log |I_k|) = O(|I_k| \log\sigma)$ time (overall, all trees are built in $O(n \log\sigma)$ time), and (ii) each pointer (tree edges and linked list pointers) uses just $O(\log\sigma)$ bits: observe that linked list pointers always connect intervals belonging to adjacent trees $\mathcal{T}(I_k)$, $\mathcal{T}(I_{k\pm 1})$, so they point inside a memory region of size $O(|I_k|) \subseteq O(\sigma^2)$ and thus require $O(\log |I_k|) \subseteq O(\log\sigma)$ bits each. ◀

After building data structure $\mathcal{I}$, in Line 4 we identify all integers $1 < i \leq n$ such that $\mathsf{LCP}^*_G[i] = 0$: these correspond to consecutive nodes v_{i-1}, v_i with different incoming labels: $\lambda(v_{i-1}) \neq \lambda(v_i)$.

We keep the following invariant before and after the execution of each iteration of the `while` loop at Line 5: for every $1 < i \leq n$, *exactly one* of the following three conditions holds. (i) the pair $(i, \mathsf{LCP}^*_G[i])$ has already been output at line 7, (ii) $(i, d) \in W$ for some $d \geq 0$, in which case it holds that $\mathsf{LCP}^*_G[i] = d$ or (iii) $(l, r, c) \in \mathcal{I}$, where (l, r, c) is the bridge associated with v_i according to Lemma 12.

The invariant clearly holds before entering the `while` loop, because for every $1 < i \leq n$ we either push $(i, 0)$ in W at Line 4 whenever $\lambda(v_{i-1}) \neq \lambda(v_i)$ (thereby satisfying condition (ii) since $\mathsf{LCP}^*_G[i] = \mathsf{lcp}(\gamma^*_{i-1}, \gamma^*_i) = 0$), or insert (l, r, c) in $\mathcal{I}$ at Line 3, where (l, r, c) is the bridge associated with v_i (thereby satisfying condition (iii)). At this point, condition (i) does not hold for any $1 < i \leq n$. Notice that by Definition 11, no bridge is associated with v_i such that $\lambda(v_{i-1}) \neq \lambda(v_i)$ (and vice versa), so the invariant's conditions are indeed mutually exclusive.

We show that the invariant holds after every operation in the body of the main loop. Assume we pop (i, d) from W at Line 6. Then, this means that (before popping) condition (ii) of our invariant holds, and in particular $\mathsf{LCP}^*_G[i] = d$. At line 7 the algorithm correctly outputs $(i, d = \mathsf{LCP}^*_G[i])$, so condition (i) of the invariant now holds for position i (while condition (ii) does not hold anymore, and condition (iii) did not hold even before: remember that the three conditions are mutually exclusive). The invariant is not modified (so it still holds) for the other positions $i' \neq i$.

In Line 8, we retrieve and remove every bridge (l, r, c) such that $l < i \leq r$. Let $v_{i'}$ be the node associated with bridge (l, r, c) (Lemma 12). The fact that we remove (l, r, c) from $\mathcal{I}$ temporarily invalidates the invariant for i' (none of (i-iii) hold), but in the `for` loop at Line 9 we immediately re-establish the invariant by pushing in W pair $(i', d+1)$ and observing that indeed $\mathsf{LCP}^*_G[i'] = d+1$ (i.e. condition (ii) of the invariant holds for position i'). To see that $\mathsf{LCP}^*_G[i'] = d+1$ holds true first observe that, since (i) we use a queue W for pairs $(i, \mathsf{LCP}^*_G[i])$, (ii) initially (Line 4), we only push in W pairs of the form $(i, 0)$, and (iii) whenever we pop a pair (i, d) we push pairs of the form $(j, d+1)$ for some $i, j \in [n]$, LCP values are popped in line 6 in nondecreasing order. In particular, for every $d \geq 0$, no pair of the form $(i, d+1)$ is popped from the queue until all pairs of the form (j, d) are popped. From this observation and since i is the first integer stabbing bridge (l, r, c) in Line 8 (since we remove bridges from $\mathcal{I}$ immediately after they are stabbed), it must be $\mathsf{lcp}(\gamma^*_l, \gamma^*_r) = \min_{l<j\leq r} \mathsf{LCP}^*_G[j] = \mathsf{LCP}^*_G[i] = d$. Then, since $\gamma^*_{i'-1} = c\gamma^*_l$ and $\gamma^*_{i'} = c\gamma^*_r$ hold, we have that $\mathsf{LCP}^*_G[i'] = \mathsf{lcp}(\gamma^*_{i'-1}, \gamma^*_{i'}) = d+1$.

We proved that our invariant always holds true; in particular, it holds when the algorithm terminates. The fact that conditions (i-iii) are mutually exclusive, immediately implies that no LCP value is output more than once, i.e. the first components of the output pairs $(i, \mathsf{LCP}^*_G[i])$ are all distinct.

At the end of the algorithm's execution, $W = \emptyset$ holds. To prove that the algorithm computes every LCP value, suppose for a contradiction, that there exists $i \in [n]$ such that $(i, \mathsf{LCP}^*_G[i])$ is never output in Line 7. Without loss of generality, choose i yielding the smallest such $\mathsf{LCP}^*_G[i]$. Note that $\mathsf{LCP}^*_G[i] > 0$, since all pairs $(j, \mathsf{LCP}^*_G[j] = 0)$ are inserted in W at Line 4, thus they are output at Line 7. Then, condition (i) of our invariant does not hold for i. Also condition (ii) cannot hold, otherwise it would be $(i, \mathsf{LCP}^*_G[i]) \in W \neq \emptyset$. We conclude that condition (iii) must hold for i: $(l, r, c) \in \mathcal{I}$, where (l, r, c) is the bridge associated with v_i by Lemma 12. In turn, this implies that no pair $(j, \mathsf{LCP}^*_G[j])$ for $l < j \leq r$ has been output in Line 7 (otherwise, such a j stabbing (l, r, c) would have caused the removal of (l, r, c) from $\mathcal{I}$). By Lemma 12, $v_l \xrightarrow{c} v_{i-1}$ and $v_r \xrightarrow{c} v_i$ hold. Since we assume that $\inf_u \neq \inf_v$ for all $u \neq v \in V$, we can apply Lemma 7 and obtain that it must hold $\mathsf{LCP}^*_G[j] = \mathsf{LCP}^*_G[i] - 1$ for some $l < j \le r$, which contradicts to minimality of $\mathsf{LCP}^*_G[i]$. We conclude that the algorithm computes every LCP value.

Next, we analyze the algorithm's running time and working space. By Lemma 14, $\mathcal{I}$ is built in $O(n \log \sigma)$ time using $O(n \log \sigma)$ bits of working space. The `while` loop at Line 5 iterates at most $O(n)$ times because (i) at most n elements are pushed into the queue W at the beginning (Line 4), and (ii) an element $(i, d) \in \mathbb{N}^2$ can be pushed into the queue at Line 11 only after a bridge is stabbed and removed; thus at most $|\mathcal{I}| \in O(n)$ elements can be pushed into the queue. As a result, Line 8 (query $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$) is executed $O(n)$ times. Recall (Lemma 14) that such a query runs in $O(\log \sigma + |R| + \ell/\sigma)$ time, where $|R|$ is the number of characters labeling stabbed intervals (equivalently, the number of stabbed intervals since no two intervals labeled with the same character can intersect) and ℓ is the total cumulative length of the stabbed intervals. Since overall the calls to $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$ will ultimately remove all bridges from $\mathcal{I}$, we conclude that the sum of all cardinalities $|R|$ equals $|\mathcal{I}| \in O(n)$, and the sum of all cumulative lengths ℓ equals the sum of all the bridges' lengths: $\sum_{(l,r,c)\in\mathcal{I}}(r - l + 1) \in O(n\sigma)$ (because no two intervals labeled with the same character can intersect). Appying Lemma 14 we conclude that, overall, all calls to $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$ cost $O(n \log \sigma + n + n\sigma/\sigma) = O(n \log \sigma)$ time. Line 10 takes $O(\log \sigma)$ time by Lemma 13. We represent the queue W in $O(n)$ bits of space, using the same strategy of Beller *et al.* (see [6] for all details): W is represented internally with two queues W_t and W_{t+1}, containing pairs of the form (i, t) and $(i, t + 1)$, respectively (in fact, only the first element i of the pair needs to be stored). Pairs are popped from the former queue and pushed into the second. As long as $|W_{t+1}| \le n/\log n$, W_{t+1} is represented as a simple vector of integers. As soon as $|W_{t+1}| > n/\log n$, we switch to a packed bitvector representation of n bits ($n/\log n$ words) marking with a bit set all i such that $(i, t + 1) \in W_{t+1}$. When W_t becomes empty, we delete it, create a new queue W_{t+2}, and start popping from W_{t+1} and pushing into W_{t+2}. If W_{t+1} is still represented as a vector of integers, popping is trivial. Otherwise (packed bitvector), all integers in W_{t+1} can be popped in overall $O(n/\log n + |W_{t+1}|) \subseteq O(|W_{t+1}|)$ time using bitwise operations.

We finally obtain:

▶ **Lemma 15.** *Given a deterministic Wheeler pseudoforest $G = (V, E)$ such that $\inf_u \neq \inf_v$ for all $u \neq v \in V$ represented with the data structure of Lemma 13, the reduced LCP array LCP^*_G of G can be computed in $O(n \log \sigma)$ time and $O(n \log \sigma)$ bits of working space on top of the input. The algorithm does not allocate memory for the output array LCP^*_G: the entries of this array are streamed to output in the form of pairs $(i, \mathsf{LCP}^*_G[i])$ sorted by their second component, from smallest to largest.*

Putting everything together (pre-processing, Lemma 15, and post-processing), we obtain the main result of our paper, Theorem 1.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

2 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. *Regular Languages meet Prefix Sorting*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.

3 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2023. doi:10.1007/978-3-031-43980-3_1.

4 Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable κ-spectra via subset rank queries on the spectral burrows-wheeler transform. In Jonathan W. Berry, David B. Shmoys, Lenore Cowen, and Uwe Naumann, editors, *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023, Seattle, WA, USA, May 31 - June 2, 2023*, pages 225–236. SIAM, 2023. doi:10.1137/1.9781611977714.20.

5 Ruben Becker, Manuel Cáceres, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Francisco Olivares, and Nicola Prezza. Sorting Finite Automata via Partition Refinement. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.15.

6 T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows–Wheeler transform. *J. Discrete Algorithms*, 18:22–31, 2013.

7 Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de bruijn graphs. In *2015 Data Compression Conference*, pages 383–392, 2015. doi:10.1109/DCC.2015.70.

8 M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DEC Systems Research Center, 1994.

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

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (4th ed.)*. The MIT Press, 2022.

11 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $o(|e|^2 + |v|^{5/2})$ time. In *2022 Data Compression Conference (DCC)*, pages 272–281, 2022. doi:10.1109/DCC52660.2022.00035.

12 Nicola Cotumaccio. Prefix Sorting DFAs: A Recursive Algorithm. In Satoru Iwata and Naonori Kakimura, editors, *34th International Symposium on Algorithms and Computation (ISAAC 2023)*, volume 283 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ISAAC.2023.22.

13 Nicola Cotumaccio. A Myhill-Nerode Theorem for Generalized Automata, with Applications to Pattern Matching and Compression. In Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshtanov, editors, *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, volume 289 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2024.26.

14 Nicola Cotumaccio. Enhanced graph pattern matching, 2024. arXiv:2402.16205.

15 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part i. *J. ACM*, 70(4), August 2023. doi:10.1145/3607471.

16 Nicola Cotumaccio, Travis Gagie, Dominik Köppl, and Nicola Prezza. Space-time trade-offs for the LCP array of wheeler dfas. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2023. doi:10.1007/978-3-031-43980-3_12.

17 Nicola Cotumaccio and Nicola Prezza. *On Indexing and Compressing Finite Automata*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.
18 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.
19 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. doi:10.1016/j.tcs.2017.06.016.
20 Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval*, pages 1–6, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
21 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*, SODA '03, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics.
22 Sung-Hwan Kim, Francisco Olivares, and Nicola Prezza. Faster Prefix-Sorting Algorithms for Deterministic Finite Automata. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2023.16.
23 U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.
24 Yakov Nekrich. A Dynamic Stabbing-Max Data Structure with Sub-Logarithmic Query Time. In *Proceedings of the 22nd International Symposium on Algorithms and Computations (ISAAC)*, pages 170–179, 2011. doi:10.1007/978-3-642-25591-5_19.
25 Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
26 Nicola Prezza and Giovanna Rosone. Space-efficient construction of compressed suffix trees. *Theoretical Computer Science*, 852:138–156, 2021.
27 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43–es, November 2007. doi:10.1145/1290672.1290680.
28 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012. Special Issue: Combinatorial Pattern Matching (CPM 2010). doi:10.1016/j.ic.2011.03.007.