Space-time Trade-offs for the LCP Array of Wheeler DFAs

Nicola Cotumaccio[1,2][0000−0002−1402−5298], Travis Gagie[2][0000−0003−3689−327X], Dominik Köppl[3][0000−0002−8721−4444], and Nicola Prezza[4][0000−0003−3553−4953]

[1] GSSI, Italy nicola.cotumaccio@gssi.it
[2] Dalhousie University, Canada nicola.cotumaccio@dal.ca, travis.gagie@dal.ca
[3] University of Münster, Germany dominik.koeppl@uni-muenster.de
[4] University Ca' Foscari, Venice, Italy nicola.prezza@unive.it

Abstract. Recently, Conte et al. generalized the longest-common prefix (LCP) array from strings to Wheeler DFAs, and they showed that it can be used to efficiently determine matching statistics on a Wheeler DFA [DCC 2023]. However, storing the LCP array requires $O(n \log n)$ bits, n being the number of states, while the compact representation of Wheeler DFAs often requires much less space. In particular, the BOSS representation of a de Bruijn graph only requires a linear number of bits, if the size of alphabet is constant.
In this paper, we propose a sampling technique that allows to access an entry of the LCP array in logarithmic time by only storing a linear number of bits. We use our technique to provide a space-time trade-off to compute matching statistics on a Wheeler DFA. In addition, we show that by augmenting the BOSS representation of a k-th order de Bruijn graph with a linear number of bits we can navigate the underlying variable-order de Bruijn graph in time logarithmic in k, thus improving a previous bound by Boucher et al. which was linear in k [DCC 2015].

Keywords: Wheeler graphs · LCP array · de Bruijn graphs · Matching statistics · Variable-order de Bruijn graphs.

1 Introduction

In 1973, Weiner invented the *suffix tree* of a string [28], a versatile data structure which allows to efficiently handle a variety of problems, including solving pattern matching queries, determining matching statistics, identifying combinatorial properties of the string and computing its Lempel-Ziv decomposition. However, the space consumption of a suffix tree can be too high for some applications (including bioinformatics), so over the past 30 years a number of *compressed* data structures simulating the behavior of a suffix tree have been designed, thus leading to compressed suffix trees [26]. In many applications, one does not need the full functionality of a suffix tree, so it may be sufficient to store only some of these data structures. Among the most popular data structures, we have the suffix array [21], the longest common prefix (LCP) array [21], the Burrows-Wheeler transform (BWT) [6] and the FM-index [13].

In the past 20 years, the ideas behind the suffix array, the BWT and the FM-index have been generalized to trees [12,14], de Bruijn graphs [5], Wheeler graphs [1, 17] and arbitrary graphs and automata [8, 9]. Broadly speaking, Wheeler graphs concisely capture the intuition behind these data structures in a graph setting; thus, they can be regarded as a benchmark for extending suffix tree functionality to graphs. In particular, the LCP array of a string remarkably extends the functionality of the suffix array, and a recent paper [7] shows that the LCP array can also be generalized to Wheeler DFAs, which represents a remarkable step toward fully simulating suffix-tree functionality in a graph setting. However, the solution in [7] is not space efficient: storing the LCP array of a Wheeler DFA requires $O(n \log n)$ bits, n being the number of states. If the size σ of the alphabet is small, this space can be considerably larger than the space required to store the Wheeler DFA itself. As we will see, if $\sigma \log \sigma = o(\log n)$, then the space required to store the Wheeler DFA is $o(n \log n)$, and if $\sigma = O(1)$, then the space required to store the Wheeler DFA is $O(n)$. The latter case is especially relevant in practice, because de Bruijn graphs are the prototypes of Wheeler graphs, and in bioinformatics de Bruijn graphs are defined over the constant-size alphabet $\Sigma = \{A, C, G, T\}$.

In this paper, we show that we can *sample* entries of the LCP array in such a way that, by storing only a linear number of additional bits on top of the Wheeler graph, we can compute each entry of the LCP array in logarithmic time, thus providing a space-time trade-off. More precisely:

Theorem 1. *We can augment the compact representation of a Wheeler DFA $\mathcal{A}$ with $O(n)$ bits ($O(n \log \log \sigma)$ bits, respectively), where n is the number of states and σ is the size of the alphabet, in such a way that we can compute each entry of the LCP array of $\mathcal{A}$ in $O(\log n \log \log \sigma)$ time ($O(\log n)$ time, respectively).*

We present two applications of our result: computing matching statistics on Wheeler DFAs and navigating varriable-order de Bruijn graphs.

Matching Statistics on Wheeler DFAs The problem of computing matching statistics on a Wheeler DFA is defined as follows: given a pattern of length m and a Wheeler DFA with n states, determine the longest suffix of each prefix of m that occurs in the graph (that is, that can be read by following some edges on the graph and concatenating the labels). This problem is a natural generalization of the problem of computing matching statistics on strings. Conte et al. [7] proved the following result:

Theorem 2. *We can augment the compact representation of a Wheeler DFA $\mathcal{A}$ with $O(n \log n)$ bits, where n is the number of states and σ is the size of the alphabet, in such a way that we can compute the matching statistics of a pattern of length m w.r.t to the Wheeler DFA in $O(m \log n)$ time.*

We will show that if we only want to use linear space, then we can use Theorem 1 to obtain the following trade-off.

Theorem 3. *We can augment the compact representation of a Wheeler DFA $\mathcal{A}$ with $O(n \log\log\sigma)$ bits, where n is the number of states and σ is the size of the alphabet, in such a way that we can compute the matching statistics of a pattern of length m w.r.t to the Wheeler DFA in $O(m \log^2 n)$ time.*

Variable-order de Bruijn Graphs Wheeler graphs are a generalization of de Bruijn graphs; in particular, the compact representation of a Wheeler graph is a generalization of the BOSS representation of a de Bruijn graph [5], and our results on the LCP array also apply to a de Bruijn graph. Many assemblers [3, 19, 24, 27] consider all k-mers occurring in a set of reads and build a k-th order de Bruijn graph (on the alphabet $\Sigma = \{A, C, G, T\}$) to perform Eulerian sequence assembly [18,25]. However, the choice of the parameter k impacts the assembly quality, so some assemblers try several choices for k [3,24], which slows down the process because several de Bruijn graphs need to be built. In [4] it was shown that the k-order de Bruijn graph of $\mathcal{S}$ can be used to *implicitly* store the k'-th order de Bruijn graph of $\mathcal{S}$ for *every* $k' \leq k$, thus leading to a *variable-order de Bruijn graph*. The challenge is to navigate this implicit representation (that is, how to follow edges in a forward or backward fashion). In [4], it was shown that the navigation is possible by storing or by simulating an array $\overline{\mathsf{LCP}}_G$ which can be seen as a simplification of the LCP array of the Wheeler graph G. More precisely, we have the following result (see [4]; we assume $\sigma = O(1)$).

Theorem 4.
1. *We can augment the BOSS representation of a k-th order de Bruijn graph with $O(n \log k)$ bits, where n is the number of nodes, so that the underlying variable-order de Bruijn graph can be navigated in $O(\log k)$ time per visited node.*
2. *We can augment the BOSS representation of a k-th order de Bruijn graph with $O(n)$ bits, where n is the number of nodes, so that the underlying variable-order de Bruijn graph can be navigated in $O(k \log n)$ time per visited node.*

Essentially, the first solution in Theorem 4 explicitly stores $\overline{\mathsf{LCP}}_G$, while the second solution in Theorem 4 computes the entries of $\overline{\mathsf{LCP}}_G$ by exploiting the BOSS representation. In general, a big k (close to the size of the reads) allows to retrieve the expressive power on an overlap graph [11], so in Theorem 4 we cannot assume that k is small. On the one hand, the *space* required for the first solution can be too large, because a de Bruijn graph can be stored by using only $O(n)$ bits. On the other hand, the *time* bound in the second solution increases substantially. We can now improve the second solution by providing a data structure that achieves the best of both worlds. As we did in Theorem 1, we can conveniently sample some entries of $\overline{\mathsf{LCP}}_G$. We will prove the following result.

Theorem 5. *We can augment the BOSS representation of a k-th order de Bruijn graph with $O(n)$ bits, where n is the number of nodes, so that the underlying variable-order de Bruijn graph can be navigated in $O(\log k \log n)$ time per visited node.*

2 Definitions

Sets and Relations Let V be a set. A total order on V is a binary relation $\leq$ which is reflexive, antisymmetric and transitive. We say that U is a $\leq$-interval (or simply an interval) if for all $v_1, v_2, v_3 \in V$, if $v_1, v_3 \in U$ and $v_1 < v_2 < v_3$, then $v_2 \in U$. If $u, v \in V$, with $u \leq v$, we denote by $[u, v]$ the smallest interval containing u and v, that is $[u, v] = \{z \in V \mid u \leq z \leq v\}$. In particular, if V is the set of integers, then we assume that $\leq$ is the standard total order, hence $[u, v] = \{u, u+1, \dots, v-1, v\}$.

Strings Let Σ be a finite alphabet, with $\sigma = |\Sigma|$. Let Σ^* be the set of all finite strings on Σ and let Σ^ω be the set of all (countably) infinite strings on Σ. If $\alpha \in \Sigma^*$, then α^R is the reverse string of α. If $\alpha, \beta \in \Sigma^* \cup \Sigma^\omega$, we denote by $\mathsf{lcp}(\alpha, \beta)$ the length of longest common prefix between α and β. In particular, if $\alpha \in \Sigma^*$, then $\mathsf{lcp}(\alpha, \beta) \leq |\alpha|$ and if $\alpha, \beta \in \Sigma^\omega$ with $\alpha = \beta$, then $\mathsf{lcp}(\alpha, \beta) = \infty$. Let $\preceq$ be a fixed total order on Σ. We extend the total order $\preceq$ from Σ to $\Sigma^* \cup \Sigma^\omega$ lexicographically.

DFAs Throughout the paper, let $\mathcal{A} = (Q, E, s_0, F)$ be a deterministic finite automaton (DFA), where Q is the set of states, $E \subseteq Q \times Q \times \Sigma$ is the set of labeled edges, $s_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states. The alphabet Σ is *effective*, that is, every $c \in \Sigma$ labels some edge. Since $\mathcal{A}$ is deterministic, for every $u \in Q$ and for every $a \in \Sigma$ there exists at most one edge labeled a leaving u. Following [1], we assume that (i) s_0 has no incoming edges, (ii) every state is reachable from the initial state and (iii) all edges entering the same state have the same label (*input-consistency*). For every $u \in Q \setminus \{s_0\}$, let $\lambda(u) \in \Sigma$ be the label of all edges entering u. We define $\lambda(s_0) = \#$, where $\# \notin \Sigma$ is a special character such that $\# \prec a$ for every $a \in \Sigma$ (the character # plays the same role as the termination character \$ in suffix arrays, suffix trees and Burrows-Wheeler transforms). As a consequence, an edge (u', u, a) can be simply written as (u', u), because it must be $a = \lambda(u)$.

Compact Data Structures Let A be an array of length n containing elements from a finite totally-ordered set. A *range minimum query* on A is defined as follows: given $1 \leq i \leq j \leq n$, return one of the indices k with $1 \leq k \leq n$ such that (i) $i \leq k \leq j$ and $A[k] = \min\{A[i], A[i+1], \dots, A[j-1], A[j]\}$. We write $k = RMQ_A(i, j)$. Then, there exists a data structure of $2n + o(n)$ such that in $O(1)$ time we can compute $RMQ_A(i, j)$ for every $1 \leq i \leq n$, *without the need to access* A [15,16]. This result is essentially optimal, because every data structure solving range minimum queries on A requires at least $2n - \Theta(\log n)$ bits [16,20].

Let A be a bitvector of length n. Let $rank(A, i) = |\{j \in \{1, 2, \dots, i-1, i\} \mid A[j] = 1\}|$ be the number of 1's among the first i bits of A. Then, there exists a data structure of $n + o(n)$ bits such that in $O(1)$ time we can compute $rank(A, i)$ for $1 \leq i \leq n$ [23].

3 Wheeler DFAs

Let us recall the definition of Wheeler DFA [7].

Definition 1. *Let $\mathcal{A} = (Q, E, s_0, F)$ be a DFA. A* Wheeler order *on $\mathcal{A}$ is a total order $\leq$ on Q such that $s_0 \leq u$ for every $u \in Q$ and:*

1. *(Axiom 1) If $u, v \in Q$ and $u < v$, then $\lambda(u) \preceq \lambda(v)$.*
2. *(Axiom 2) If $(u', u), (v', v) \in E$, $\lambda(u) = \lambda(v)$ and $u < v$, then $u' < v'$.*

A DFA $\mathcal{A}$ is Wheeler *if it admits a Wheeler order.*

Every DFA admits at most one Wheeler order [1], so the total order $\leq$ in Definition 1 is *the* Wheeler order on $\mathcal{A}$. In the following, we fix a Wheeler DFA $\mathcal{A} = (Q, E, s_0, F)$, with $n = |Q|$ and $e = |E|$, and we write $Q = \{u_1, \dots, u_n\}$, with $u_1 < u_2 < \cdots < u_n$ in the Wheeler order. In particular, $u_1 = s_0$. Following [7], we assume that s_0 has a self-loop labeled #, which is consistent with Axiom 1, because $\# \prec a$ for every $a \in \Sigma$). This implies that every state has at least one incoming edge, so for every state u_i there exists at least one infinite string $\alpha \in \Sigma^\omega$ that can be read starting from u_i and following edges in a backward fashion. We denote by I_{u_i} the nonempty set of all such strings. Formally:

Definition 2. *Let $1 \leq i \leq n$. Define:*

$$I_{u_i} = \{\alpha \in \Sigma^\omega \mid \textit{there exist integers } f_1, f_2, \dots \textit{ in } [1,n] \textit{ such that (i) } f_1 = i, \textit{(ii) } (u_{f_{k+1}}, u_{f_k}) \in E \textit{ for every } k \geq 1 \textit{ and (iii) } \alpha = \lambda(u_{f_1})\lambda(u_{f_2})\dots\}.$$

For every $1 \leq i \leq n$, let $p_{\min}(i)$ be the smallest $1 \leq i' \leq n$ such that $(u_{i'}, u_i) \in E$ and let $p_{\max}(i)$ be the largest $1 \leq i'' \leq n$ such that $(u_{i''}, u_i) \in E$. Both $p_{\min}(i)$ and $p_{\max}(i)$ are well-defined because every state has at least one incoming edge. For every $1 \leq i \leq n$, define $p^1_{\min}(i) = p_{\min}(i)$ and recursively, for $j \geq 2$, let $p^j_{\min}(i) = p_{\min}(p^{j-1}_{\min}(i))$. Then, $\lambda(u_i)\lambda(p_{\min}(i))\lambda(p^2_{\min}(i))\lambda(p^3_{\min}(i))\dots$ is the lexicographically *smallest* string in I_{u_i}, which we denote by $\min_i$ [7]. Analogously, one can define the lexicographically *largest* string in I_{u_i} by using $p_{\max}$. Moreover, in [7] it was shown that:

$$\min\nolimits_1 \preceq \max\nolimits_1 \preceq \min\nolimits_2 \preceq \max\nolimits_2 \preceq \cdots \preceq \max\nolimits_{n-1} \preceq \min\nolimits_n \preceq \max\nolimits_n.$$

Intuitively, the previous equation shows that the permutation of the set of all states of $\mathcal{A}$ induced by the Wheeler order can be seen as a generalization of the permutation of positions induced by the prefix array of a string α (or equivalently, the suffix array of the reverse string of α^R). Indeed, a string α can also be seen as a DFA $\mathcal{A}' = (Q', E', s'_0, F')$, where $Q' = \{q'_0, q'_1 \dots, q'_{|\alpha|}\}$, $s'_0 = q'_0$, $F' = \{q'_{|\alpha|}\}$ (the set F plays no role here), $\lambda(q'_i)$ is the i-th character of α for $1 \leq i \leq n$ and $E' = \{(q'_{i-1}, q'_i) \mid 1 \leq i \leq n\}$ (every state is reached by exactly one string so the minimum and the maximum string reaching each state are equal).

Let $1 \leq r \leq s \leq n$ and let $c \in \Sigma$. Let $E_{r,s,c}$ be the set of all states that can be reached from a state in $[r, s]$ by following edges labeled c; formally, $E_{r,s,c} = \{1 \leq$

$j \leq n \mid \lambda(u_j) = c$ and $(u_i, u_j) \in E$ for some $i \in [r, s]$ $\}$. Then, $E_{r,s,c}$ is again an interval, that is, there exist $1 \leq r' \leq s' \leq n$ such that $E_{r,s,c} = [r', s']$ [17]. This property enables a compression mechanism that generalizes the Burrows-Wheeler transform [6] and the FM-index [13] to Wheeler DFAs. The Wheeler DFA $\mathcal{A}$ can be stored by using only $2e + 4n + e \log \sigma + \sigma \log e$ bits (up to lower order terms), including n bits to mark the set F of final states and n bits to mark all $1 \leq i \leq n$ such that $\lambda(u_i) \neq \lambda(u_{i-1})$, which allows us to retrieve each $\lambda(u_i)$ in $O(1)$ time by using a rank query [17] (recall that n is the number of states and e is the number of edges). Since $\mathcal{A}$ is a DFA, we have $e \leq n\sigma$, so the required space is $O(n\sigma \log \sigma)$. If the alphabet is small — that is, if $\sigma \log \sigma = o(\log n)$ — then the number of required bits is $o(n \log n)$; if $\sigma = O(1)$, then the number of required bits is $O(n)$. This compact representation supports efficient navigation of the graph and it allows to solve pattern matching queries. More precisely, by resorting to state-of-the art select queries [23] in $O(\log \log \sigma)$ time (i) for $1 \leq i \leq n$, we can compute $p_{\min}(i)$ and $p_{\max}(i)$ and (ii) given $1 \leq r \leq s \leq n$ and $c \in \Sigma$, we can compute $[r', s'] = E_{r,s,c}$ [17]. In particular, query (ii) is the so-called *forward-search*, which generalizes the analogous mechanism of the FM-index, thus allowing to solve pattern matching queries on the graph.

The Wheeler order generalizes the notion of suffix array from strings to DFA. It is also possible to generalize LCP-arrays from strings to graph [7].

Definition 3. *The* LCP-array *of the Wheeler DFA $\mathcal{A}$ is the array* $\mathsf{LCP}_{\mathcal{A}} = \mathsf{LCP}_{\mathcal{A}}[2, 2n]$ *which contains the following* $2n - 1$ *values in the following order:* $\mathsf{lcp}(\min_1, \max_1)$, $\mathsf{lcp}(\max_1, \min_2)$, $\mathsf{lcp}(\min_2, \max_2)$, $\ldots$, $\mathsf{lcp}(\max_{n-1}, \min_n)$, $\mathsf{lcp}(\min_n, \max_n)$. *In other words,* $\mathsf{LCP}[2i] = \mathsf{lcp}(\min_i, \max_i)$ *for* $1 \leq i \leq n$ *and* $\mathsf{LCP}_{\mathcal{A}}[2i - 1] = \mathsf{lcp}(\max_{i-1}, \min_i)$ *for* $2 \leq i \leq n$.

It can be proved that for every $2 \leq i \leq n$, if $\mathsf{LCP}_{\mathcal{A}}[i]$ is finite, then $\mathsf{LCP}_{\mathcal{A}}[i] < 3n$ [7]. As a consequence, $\mathsf{LCP}_{\mathcal{A}}$ can be stored by using $O(n \log n)$ bits.

4 A Space-time Trade-off for the LCP Array

By storing an LCP array on top of the compact representation of a Wheeler graph, we have additional information that we can use to efficiently solve problems such as computing the matching statistics; however, we need to store $O(n \log n)$ bits. As we have seen, $O(n \log n)$ dominates the number of bits required to store $\mathcal{A}$ itself, if the alphabet is small. In this section, we show that we can store a data structure of only $O(n \log \log \sigma)$ bits which allows to compute every entry $\mathsf{LCP}_{\mathcal{A}}[i]$ in $O(\log n)$ time, thus proving Theorem 1. This will be possible by sampling some entries of $\mathsf{LCP}_{\mathcal{A}}$. The sampling mechanism is obtained by conveniently defining an auxiliary graph from the entries of the LCP array. We will immediately describe our technique, our sampling mechanism being general-purpose.

Sampling Let $G = (V, H)$ be a finite (unlabeled) directed graph such that every node has at most one incoming edge. For every $v \in V$ and for every $i \geq 0$,

Algorithm 1 Building $V(h)$

```
V(h) ← ∅
U ← ∅
while there exists v ∈ V such that (a) v(i) is defined for 0 ≤ i ≤ h − 1, (b) v(i) ≠ v(j) for
0 ≤ j < i ≤ h − 1, (c) v(i) ∉ U for 0 ≤ i ≤ h − 1 do
    Pick such a v, add v(h − 1) to V(h) and add v(i) to U for every 0 ≤ i ≤ h − 1
end while
```

Algorithm 2 Input: $h \in [2, 2n]$. Output: $\mathsf{LCP}_{\mathcal{A}}[h]$.

```
procedure MAIN_FUNCTION(h)
    Initialize a global bit array D[2, 2n] to zero    ▷ D[2, 2n] marks the entries already considered
    return LCP(h)
end procedure

procedure LCP(h)
    D[h] ← 1
    if C[h] = 1 then                                  ▷ The desired value has been sampled
        return LCP*_A[rank(C, h)]
    else if h is odd then
        i ← ⌈h/2⌉
        if λ(u_{i−1}) ≠ λ(u_i) then
            return 0
        else
            k ← p_max(i − 1)
            k′ ← p_min(i)
            j ← RMQ_{LCP_A}(2k + 1, 2k′ − 1)
            if D[j] = 1 then      ▷ We have already considered this entry before, so there is a cycle
                return ∞
            else
                return 1 + LCP(j)
            end if
        end if
    else
        i ← h/2
        k ← p_min(i)
        k′ ← p_max(i)
        j ← RMQ_{LCP_A}(2k, 2k′)
        if D[j] = 1 then          ▷ We have already considered this entry before, so there is a cycle
            return ∞
        else
            return 1 + LCP(j)
        end if
    end if
end procedure
```

there exists at most one node $v' \in V$ such that there exists a directed path from v' to v having i edges; if v' exists, we denote it by $v(i)$. Fix a parameter $h \geq 1$. Let us prove that there exists $V(h) \subseteq V$ such that (i) $|V(h)| \leq \frac{|V|}{h}$ and (ii) for every $v \in V$ there exists $0 \leq i \leq 2h-2$ such that $v(i)$ is defined and either $v(i) \in V(h)$ or $v(i)$ has no incoming edges or $v(i) = v(j)$ for some $0 \leq j < i$. We build $V(h)$ incrementally following Algorithm 1. Let us prove that, at the end of the algorithm, properties (i) and (ii) are true. For every $v \in V(h)$, define $S_v = \{v, v(1), v(2) \ldots, v(h-1)\}$, which is possible because by construction if $v \in V(h)$, then $v(i)$ is defined for every $0 \leq i \leq h-1$. It must be $v(i) \neq v(j)$ for $0 \leq i < j \leq h-1$, so $|S_v| = h$. If $v, v' \in V(h)$ and $v \neq v'$, then by construction S_v and $S_{v'}$ are disjoint. As a consequence, $|V| \geq \sum_{v \in V(h)} |S_v| = \sum_{v \in V(h)} h = h|V_h|$

and so $|V_h| \leq \frac{|V|}{h}$, which proves property (i). Let us prove property (ii). Pick $v \in V$; we must prove that there exists $0 \leq i \leq 2h-2$ such that $v(i)$ is defined and either $v(i) \in V(h)$ or $v(i)$ has no incoming edges or $v(i) = v(j)$ for some $0 \leq j < i$. We distinguish three cases:

1. there exists i with $1 \leq i \leq h-1$ such that $v(i-1)$ is defined but $v(i)$ is not defined. Then, $v(i-1)$ has no incoming edges.
2. there exist i, j with $0 \leq j < i \leq h-1$ such that $v(j)$ and $v(i)$ are defined and $v(i) = v(j)$. In this case, the conclusion is immediate.
3. $v(i)$ is defined for every $0 \leq i \leq h$ and $v(i) \neq v(j)$ for $0 \leq j < i \leq h-1$. Since Algorithm 1 has terminated, then there exists $0 \leq j \leq h-1$ such that $v(j) \in U$. The construction of U implies that there exists $v' \in V$ and $0 \leq j \leq h-1$ such that $v(j) = v'(j')$ and $v'(h-1) \in V(h)$. As a consequence $v(h-1+j-j') = v(j)(h-1-j') = (v'(j'))(h-1-j') = v'(h-1) \in V(h)$. Since $j \leq h-1$ and $j' \geq 0$, we conclude $h-1+j-j' \leq 2h-2$ and we are done.

Computing the LCP Array Using a Linear Number of Bits First, let us store a data structure of $O(n)$ bits which in $O(1)$ time determines $RMQ_{\mathsf{LCP}_\mathcal{A}}(i,j)$ for every $2 \leq i \leq j \leq 2n$.

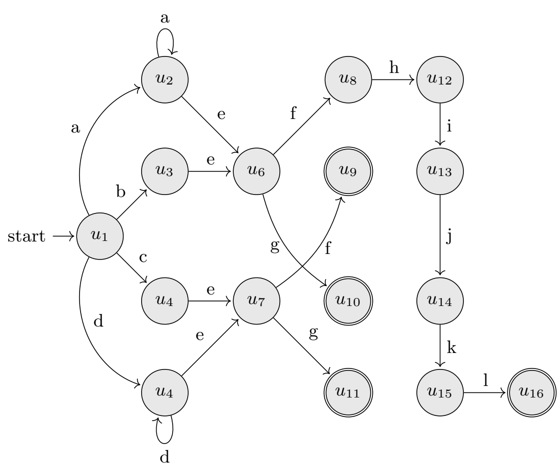
Notice that $\mathsf{LCP}_\mathcal{A}[2i] \geq 1$ for $1 \leq i \leq n$ because the first character of $\min_i$ and the first character of $\max_i$ are equal to $\lambda(u_i)$. Moreover, we have $\mathsf{LCP}_\mathcal{A}[2i-1] \geq 1$ if and only if $\lambda(u_{i-1}) = \lambda(u_i)$, for $2 \leq i \leq n$.

Consider the entry $\mathsf{LCP}_\mathcal{A}[2i-1] = \mathsf{lcp}(\max_{i-1}, \min_i)$, for $2 \leq i \leq n$, and assume that $\mathsf{LCP}_\mathcal{A}[2i-1] \geq 1$. Let $k = p_{\max}(i-1)$ and $k' = p_{\min}(i)$. Since $\mathsf{LCP}_\mathcal{A}[2i-1] \geq 1$, then there exists $a \in \Sigma$ such that $\max_{i-1} = a \max_k$ and $\min_{i-1} = a \min_{k'}$. In particular, $(u_k, u_{i-1}, a) \in E$ and $(u_{k'}, u_i, a) \in E$, so from Axiom 2 we obtain $k < k'$. Moreover, we have $\mathsf{LCP}_\mathcal{A}[2i-1] = \mathsf{lcp}(\max_{i-1}, \min_i) = \mathsf{lcp}(a \max_k, a \min_{k'}) = 1 + \mathsf{lcp}(\max_k, \min_{k'})$. Notice that:

$$
\begin{aligned}
&\mathsf{lcp}(\max_k, \min_{k'}) = \min\{\mathsf{lcp}(\max_k, \min_{k+1}), \mathsf{lcp}(\min_{k+1}, \max_{k+1}), \dots, \\
&= \mathsf{lcp}(\min_{k'-1}, \max_{k'-1}), \mathsf{lcp}(\max_{k'-1}, \min_{k'})\} = \\
&= \min\{\mathsf{LCP}_\mathcal{A}[2k+1], \mathsf{LCP}_\mathcal{A}[2k+2], \dots, \mathsf{LCP}_\mathcal{A}[2k'-2], \mathsf{LCP}_\mathcal{A}[2k'-1]\}.
\end{aligned}
$$

Let $j = RMQ_{\mathsf{LCP}_\mathcal{A}}(2k+1, 2k'-1)$. Then, $\mathsf{LCP}_\mathcal{A}[j] = \min\{\mathsf{LCP}_\mathcal{A}[2k+1], \mathsf{LCP}_\mathcal{A}[2k+2], \dots, \mathsf{LCP}_\mathcal{A}[2k'-2], \mathsf{LCP}_\mathcal{A}[2k'-1]\}$, so $\mathsf{LCP}_\mathcal{A}[2i-1] = 1 + \mathsf{LCP}_\mathcal{A}[j]$ (we assume $t + \infty = \infty$ for every $t \geq 0$), and we have reduced the problem of computing $\mathsf{LCP}_\mathcal{A}[2i-1]$ to the problem of computing $\mathsf{LCP}_\mathcal{A}[j]$. In the following, let $\mathcal{R}(2i-1) = j$. Given $2 \leq i \leq n$, we can compute $j = \mathcal{R}(2i-1)$ in $O(\log\log\sigma)$ time, because we can compute $k = p_{\max}(i-1)$ and $k' = p_{\min}(i)$ in $O(\log\log\sigma)$ time and we can compute j in $O(1)$ time by means of a range minimum query.

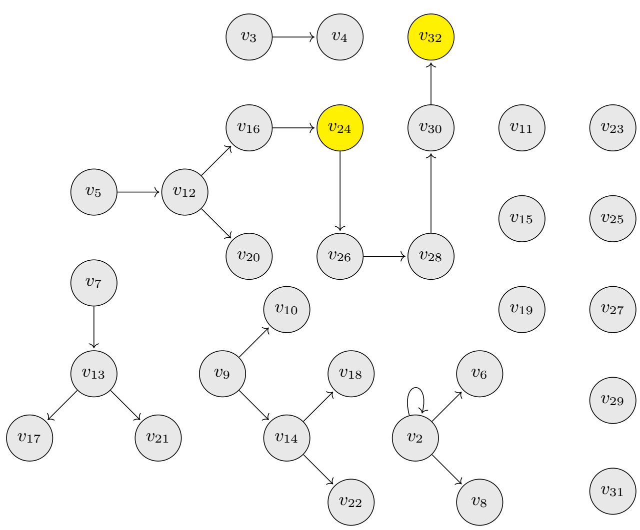
We proceed analogously with the entries $\mathsf{LCP}_\mathcal{A}[2i] = \mathsf{lcp}(\min_i, \max_i)$, for $1 \leq i \leq n$ (it must necessarily be $\mathsf{LCP}_\mathcal{A}[2i] \geq 1$). Let $k = p_{\min}(i)$ and $k' = p_{\max}(i)$; by the definitions of $p_{\min}$ and $p_{\max}$ it must be $k \leq k'$. Hence, $\mathsf{LCP}_\mathcal{A}[2i] = 1 + \mathsf{lcp}(\min_k, \max_{k'})$ and similarly $\mathsf{lcp}(\min_k, \max_{k'}) = \min\{\mathsf{LCP}_\mathcal{A}[2k], \mathsf{LCP}_\mathcal{A}[2k+$



(a)

State	i	$\mathsf{LCP}_{\mathcal{A}}[i]$	k	k'	$\mathcal{R}(i)$
1	1				
	2	∞	1	1	2
2	3	0	-	-	-
	4	1	1	2	3
3	5	0	-	-	-
	6	∞	1	1	2
4	7	0	-	-	-
	8	∞	1	1	2
5	9	0	-	-	-
	10	1	1	5	9
6	11	0	-	-	-
	12	1	2	3	5
7	13	1	3	4	7
	14	1	4	5	9
8	15	0	-	-	-
	16	2	6	6	12
9	17	2	6	7	13
	18	2	7	7	14
10	19	0	-	-	-
	20	2	6	6	12
11	21	2	6	7	13
	22	2	7	7	14
12	23	0	-	-	-
	24	3	8	8	16
13	25	0	-	-	-
	26	4	12	12	24
14	27	0	-	-	-
	28	5	13	13	26
15	29	0	-	-	-
	30	6	14	14	28
16	31	0	-	-	-
	32	7	15	15	30

(b)



(c)

i	$C[i]$	$\mathsf{LCP}^*_{\mathcal{A}}$
1		3
2	0	7
3	0	
4	0	
5	0	
6	0	
7	0	
8	0	
9	0	
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	
16	0	
17	0	
18	0	
19	0	
20	0	
21	0	
22	0	
23	0	
24	1	
25	0	
26	0	
27	0	
28	0	
29	0	
30	0	
31	0	
32	1	

(d)

Fig. 1: (a) A Wheeler DFA. States are numbered according to the Wheeler order. (b) The array $\mathsf{LCP}_{\mathcal{A}}$, and the values needed to compute $G = (V, H)$. We assume that a range minimum query returns the *largest* position of a minimum value. (c) The graph $G = (V, H)$, with $V(\lceil \log n \rceil) = V(4) = \{v_{24}, v_{32}\}$ (yellow states). (d) The data structures that we store.

$1], \ldots, \mathsf{LCP}_{\mathcal{A}}[2k' - 1], \mathsf{LCP}_{\mathcal{A}}[2k']\}$. Let $j = RMQ_{\mathsf{LCP}_{\mathcal{A}}}(2k, 2k')$. In the following, let $\mathcal{R}(2i) = j$. Given $1 \leq i \leq n$, we can compute $j = \mathcal{R}(2i)$ in $O(\log \log \sigma)$ time. See Figure 1 for an example.

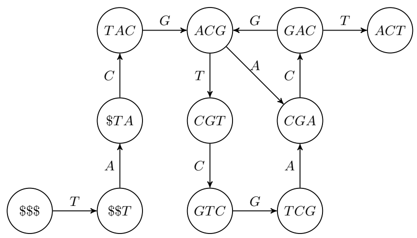
Now, consider the (unlabeled) directed graph $G = (V, H)$ defined as follows. Let V be a set of $2n-1$ nodes $v_2, v_3, \ldots, v_{2n}$. Moreover, $v_i \in V$ has no incoming edge in G if $\mathcal{R}(i)$ is not defined, which happens if $\mathsf{LCP}_{\mathcal{A}}[i] = 0$ (and so i is odd and $\lambda(u_{i-1}) \neq \lambda(u_i)$); $v_i \in V$ has exactly one incoming edge if $\mathcal{R}(i)$ is defined, namely, $(v_{\mathcal{R}(i)}, v_i)$. Note that v_{2i} has an incoming edge for every $1 \leq i \leq n$. Let $h \geq 1$ be a parameter. We know that there exists $V(h) \subseteq V$ such that (i) $|V(h)| \leq \frac{|V|}{h}$ and (ii) for every $v_i \in V$ there exists $0 \leq k \leq 2h - 2$ such that $v_i(k)$ is defined and either $v_i(h) \in V(h)$ or $v_i(h)$ has no incoming edges or $v_i(h) = v_i(l)$ for some $0 \leq l < h$. Notice that if $v_i(h) = v_i(l)$ for some $0 \leq l < h$, then $\mathsf{LCP}_{\mathcal{A}}[i] = \infty$ (because there is a cycle and so $v_i(h')$ is defined for every $h' \geq 0$). Let $n' = |V(h)|$, and let $\mathsf{LCP}^*_{\mathcal{A}}[1, n']$ an array storing the value $\mathsf{LCP}_{\mathcal{A}}[i]$ for each $v_i \in V(h)$, sorted by increasing i. Since $n' \leq \frac{|V|}{h} = \frac{2n-1}{h}$, storing $\mathsf{LCP}^*_{\mathcal{A}}[1, n']$ takes $n'O(\log n) = O(\frac{n \log n}{h})$ bits. We store a bitvector $C[2, 2n]$ such that $C[i] = 1$ if and only if $v_i \in V(h)$ for every $2 \leq i \leq 2n$; we augment C with $o(n)$ bits so that it supports rank queries in $O(1)$ time. For every $2 \leq i \leq 2n$, in $O(1)$ time we can check whether $\mathsf{LCP}_{\mathcal{A}}[i]$ has been stored in $\mathsf{LCP}^*_{\mathcal{A}}$ by checking whether $C[i] = 1$, and if $C[i] = 1$ it must be $\mathsf{LCP}_{\mathcal{A}}[i] = \mathsf{LCP}^*_{\mathcal{A}}[rank(C, i)]$.

From our discussion, it follows that Algorithm 2 correctly computes $\mathsf{LCP}_{\mathcal{A}}[i]$ for every $2 \leq i \leq n$. Property (ii) ensures that the function *lcp* is called at most h times. Every call requires $O(\log \log \sigma)$ time, so the running time of our algorithm is $O(h \log \log \sigma)$ (the initialization of $D[2, 2n]$ in Algorithm 2 can be simulated in $O(1)$ time [22]). We conclude that we store $O(n + \frac{n \log n}{h})$ bits, and in $O(h \log \log \sigma)$ time we can compute $\mathsf{LCP}_{\mathcal{A}}[i]$ for every $2 \leq i \leq n$.

By choosing $h = \lceil \frac{\log n}{\log \log \sigma} \rceil$, we conclude that our data structure can be stored using $O(n \log \log \sigma)$ bits and it allows to compute $\mathsf{LCP}_{\mathcal{A}}[i]$ for every $2 \leq i \leq n$ in $O(\log n)$ time. By choosing $h = \lceil \log n \rceil$ we conclude that our data structure can be stored using $O(n)$ bits and it allows to compute $\mathsf{LCP}_{\mathcal{A}}[i]$ for every $2 \leq i \leq n$ in $O(\log n \log \log \sigma)$ time.

5 Applications

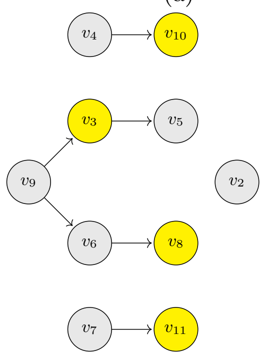
Matching Statistics Let us recall how the bounds in Theorem 2 are obtained. The space bound is $O(n \log n)$ bits because we need to store $\mathsf{LCP}_{\mathcal{A}}$. We also store a data structure to solve range minimum queries on $\mathsf{LCP}_{\mathcal{A}}$, which only takes $O(n)$ bits. The time bound $O(m \log n)$ follows from performing $O(m)$ steps to compute all matching statistics. In each of these $O(m)$ steps, we may need to perform a binary search on $\mathsf{LCP}_{\mathcal{A}}$. In each step of the binary search, we need to solve a range minimum query once and we need to access $\mathsf{LCP}_{\mathcal{A}}$ once, so the binary search takes $O(\log n)$ time per step. By Theorem 1, if we store only $O(n \log \log \sigma)$ bits, we can access $\mathsf{LCP}_{\mathcal{A}}$ in $O(\log n)$ time, so the time for the binary search becomes $O(\log^2 n)$ per step and Theorem 3 follows.



(a)

i	Node	$\overline{\mathsf{LCP}}_G[i]$	k	k'	$\mathcal{R}(i)$
1	$$$				
2	CGA	0	-	-	-
3	$TA	1	9	9	9
4	GAC	0	-	-	-
5	TAC	2	3	3	3
6	GTC	1	4	11	9
7	ACG	0	-	-	-
8	TCG	2	6	6	6
9	$$T	0	-	-	-
10	ACT	1	2	4	4
11	CGT	1	6	7	7

(b)



(c)

i	$C[i]$	$\overline{\mathsf{LCP}}^*_G$
1		1
2	0	2
3	1	1
4	0	1
5	0	
6	0	
7	0	
8	1	
9	0	
10	1	
11	1	

(d)

Fig. 2: The 3-rd order de Bruijn graph for the set $\mathcal{S} = \{CGAC, GACG, GACT, TACG, GTCG, ACGA, ACGT, TCGA, CGTC\}$ from [4]. We proceed like in Figure 1 (now we only consider odd entries of LCP_G, and $h = \lceil \log k \rceil = 2$).

Variable-order de Bruijn Graphs Let $k \geq 0$ be a parameter, and let $\mathcal{S}$ be a set of strings on the alphabet $\Sigma = \{A, C, G, T\}$ (in this application we always assume $\sigma = O(1)$).

The k-th order de Bruijn graph of $\mathcal{S}$ is defined as follows. The set of nodes is the set of all strings of Σ of length k that occur as a substring of some string in $\mathcal{S}$. There is an edge from node α to node β labeled $c \in \Sigma$ if and only if (i) the suffix of α of length $k-1$ is equal to the prefix of β of length $k-1$ and (ii) the last character of β is c. If some node α has no incoming edges, then we add nodes $\$^i\alpha_{k-i}$ for $1 \leq i \leq k$, where α_j is the prefix of α of length j and \$ is a special character, and we add edges as above; see Figure 2 for an example. Wheeler DFAs are a generalization of de Bruijn graphs (we do not need to define an initial state and a set of final states, because here we are not interested in studying the applications of de Bruijn graphs and Wheeler automata to automata theory [2, 10]); the Wheeler order is the one such that node α comes before node β if and only if the string α^R is lexicographically smaller than the string β^R [17].

Notice that, in a k-th order de Bruijn graph G, all strings that can be read from node α by following edges in a backward fashion start with α^R (as usual, we assume that node $\$\$\$$ has a self-loop labeled $\$$). As a consequence, it holds $\mathsf{LCP}_G[2i] \geq k$ for every $1 \leq i \leq n$ and $\mathsf{LCP}_G[2i-1] \leq k-1$ for every $2 \leq i \leq n$ (so any value in an odd entry is smaller than any value in an even entry).

As we saw in the introduction, in [4] it was shown that the k-order de Bruijn graph of $\mathcal{S}$ can be used to *implicitly* store the k'-th order de Bruijn graph of $\mathcal{S}$ for *every* $k' \leq k$, thus leading to a *variable-order de Bruijn graph*. The navigation of a variable-order de Bruijn graph is possible by storing or by simulating the values in the odd entries of the LCP array. Formally, in order to avoid confusion, we define $\overline{\mathsf{LCP}}_G[i] = \mathsf{LCP}_G[2i-1]$ for every $2 \leq i \leq n$; see Figure 2. Note that $\overline{\mathsf{LCP}}_G[i] \leq k-1$ for every $2 \leq i \leq n$, so $\overline{\mathsf{LCP}}_G$ can be stored by using $O(n \log k)$ bits. Notice that Theorem 1 also applies to $\overline{\mathsf{LCP}}_G[i]$ (we do not need to store values in the even entries because a value in an odd entry is smaller than a value in an even entry, so even entries are never selected in the sampling process when answering a range minimum query on LCP_G). However, we can now choose a better parameter $h \geq 1$ in our sampling process. Indeed, each entry of $\overline{\mathsf{LCP}}_G$ can be stored by using $O(\log k)$ bits (not $O(\log n)$ bits), so if we choose $h = \lceil \log k \rceil$, we conclude that we can augment the BOSS representation of a de Bruijn graph with $O(n)$ bits such that for every $2 \leq i \leq n$ we can compute $\overline{\mathsf{LCP}}_G[i]$ in $O(\log k)$ time.

The first solution in Theorem 4 consists in storing a wavelet tree on $\overline{\mathsf{LCP}}_G$, which requires $O(n \log k)$ bits and allows to navigate the graph in $O(\log k)$ time per visited node. The second solution in Theorem 4 does not store $\overline{\mathsf{LCP}}_G$ at all; whenever needed, an entry of $\overline{\mathsf{LCP}}_G$ is computed in $O(k)$ time by exploiting the BOSS representation of the de Bruijn graph. The second solution only stores a data structures of $O(n)$ bits to solve range minimum queries. The details can be found in [4]. Essentially, the time bound $O(k \log n)$ comes from performing binary searches on $\overline{\mathsf{LCP}}_G$ while explicitly computing an entry of $\overline{\mathsf{LCP}}_G$ at each step in $O(k)$ time. However, we have seen that, while staying within the $O(n)$ space bound, we can augment the BOSS representation so that we can compute the entries of $\overline{\mathsf{LCP}}_G$ in $O(\log k)$ time, so the time bound $O(k \log n)$ becomes $O(\log k \log n)$, which implies Theorem 5.

References

1. Alanko, J., D'Agostino, G., Policriti, A., Prezza, N.: Regular languages meet prefix sorting. In: Proc. of the 31st Symposium on Discrete Algorithms, (SODA'20). pp. 911–930. SIAM (2020). `https://doi.org/10.1137/1.9781611975994.55`, `https://doi.org/10.1137/1.9781611975994.55`
2. Alanko, J., D'Agostino, G., Policriti, A., Prezza, N.: Wheeler languages. Information and Computation **281**, 104820 (2021)
3. Bankevich, A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., Pyshkin, A.V., Sirotkin, A.V., Vyahhi, N., Tesler, G., Alekseyev, M.A., Pevzner, P.A.: SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. Journal of Computational Biology **19**(5), 455–477 (2012). `https://doi.org/10.1089/cmb.2012.0021`, `https://doi.org/10.1089/cmb.2012.0021`, pMID: 22506599

4. Boucher, C., Bowe, A., Gagie, T., Puglisi, S.J., Sadakane, K.: Variable-order de Bruijn graphs. In: 2015 Data Compression Conference. pp. 383–392 (2015). https://doi.org/10.1109/DCC.2015.70
5. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: Algorithms in Bioinformatics. pp. 225–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
6. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep. (1994)
7. Conte, A., Cotumaccio, N., Gagie, T., Manzini, G., Prezza, N., Sciortino, M.: Computing matching statistics on Wheeler DFAs. In: 2023 Data Compression Conference (DCC). pp. 150–159 (2023). https://doi.org/10.1109/DCC55655.2023.00023
8. Cotumaccio, N.: Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In: 2022 Data Compression Conference (DCC). pp. 272–281 (2022). https://doi.org/10.1109/DCC52660.2022.00035
9. Cotumaccio, N., Prezza, N.: On indexing and compressing finite automata. In: Proc. of the 32nd Symposium on Discrete Algorithms, (SODA'21). pp. 2585–2599. SIAM (2021). https://doi.org/10.1137/1.9781611976465.153, https://doi.org/10.1137/1.9781611976465.153
10. Cotumaccio, N., D'Agostino, G., Policriti, A., Prezza, N.: Co-lexicographically ordering automata and regular languages – part i (2023)
11. Díaz-Domínguez, D., Gagie, T., Navarro, G.: Simulating the DNA Overlap Graph in Succinct Space. In: Pisanti, N., Pissis, S.P. (eds.) 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 128, pp. 26:1–26:20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.CPM.2019.26, http://drops.dagstuhl.de/opus/volltexte/2019/10497
12. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05). pp. 184–193 (2005). https://doi.org/10.1109/SFCS.2005.69
13. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS'00). pp. 390–398 (2000). https://doi.org/10.1109/SFCS.2000.892127
14. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM **57**(1) (nov 2009). https://doi.org/10.1145/1613676.1613680, https://doi.org/10.1145/1613676.1613680
15. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010: Theoretical Informatics. pp. 158–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
16. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing **40**(2), 465–492 (2011). https://doi.org/10.1137/090779759, https://doi.org/10.1137/090779759
17. Gagie, T., Manzini, G., Sirén, J.: Wheeler graphs: A framework for BWT-based data structures. Theoretical Computer Science **698**, 67–78 (2017). https://doi.org/https://doi.org/10.1016/j.tcs.2017.06.016, https://www.sciencedirect.com/science/article/pii/S0304397517305285, algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo)

18. Idury, R.M., Waterman, M.S.: A new algorithm for DNA sequence assembly. Journal of computational biology : a journal of computational molecular cell biology **2 2**, 291–306 (1995)
19. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., Wang, J.: De novo assembly of human genomes with massively parallel short read sequencing. Genome research **20**, 265–72 (12 2009). https://doi.org/10.1101/gr.097261.109
20. Liu, M., Yu, H.: Lower bound for succinct range minimum query. In: Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing. p. 1402–1415. STOC 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3357713.3384260, https://doi.org/10.1145/3357713.3384260
21. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058, https://doi.org/10.1137/0222058
22. Navarro, G.: Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. ACM Comput. Surv. **46**(4) (mar 2014). https://doi.org/10.1145/2535933, https://doi.org/10.1145/2535933
23. Navarro, G.: Compact Data Structures - A Practical Approach. Cambridge University Press (2016), http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-al
24. Peng, Y., Leung, H.C.M., Yiu, S.M., Chin, F.Y.L.: IDBA – a practical iterative de Bruijn graph de novo assembler. In: Berger, B. (ed.) Research in Computational Molecular Biology. pp. 426–440. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
25. Pevzner, P.A., Tang, H., Waterman, M.S.: An Eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences **98**(17), 9748–9753 (2001). https://doi.org/10.1073/pnas.171285098, https://www.pnas.org/doi/abs/10.1073/pnas.171285098
26. Sadakane, K.: Compressed suffix trees with full functionality. Theor. Comp. Sys. **41**(4), 589–607 (2007). https://doi.org/10.1007/s00224-006-1198-x, https://doi.org/10.1007/s00224-006-1198-x
27. Simpson, J., Wong, K., Jackman, S., Schein, J., Jones, S., Birol, I.: ABySS: A parallel assembler for short read sequence data. Genome research **19**, 1117–23 (02 2009). https://doi.org/10.1101/gr.089532.108
28. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th IEEE Annual Symposium on Switching and Automata Theory. pp. 1–11 (1973). https://doi.org/10.1109/SWAT.1973.13