


Shortest Undirected Paths in de Bruijn Graphs

Wiktor Zuba 

CWI, Amsterdam, The Netherlands
University of Warsaw, Poland

Oded Lachish 

Birkbeck, University of London, UK

Solon P. Pissis 

CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Computing shortest directed paths in de Bruijn graphs is well studied and well understood. This is not the case for computing *undirected* paths, which is much more challenging algorithmically. In this paper, we present a *general framework* for computing shortest undirected paths in arbitrary de Bruijn graphs, that is, arbitrary subgraphs of the *complete* de Bruijn graph. We then present an application of our techniques for making any arbitrary order- k de Bruijn graph $G(V, E)$ weakly connected by adding a set of edges of minimum total cost. This improves the running time of the recent $(2 - 2/d)$ -approximation algorithm by Bernardini et al. [CPM 2024] from $\mathcal{O}(k|V|^2)$ to $\mathcal{O}(k|V| \log d)$ time, where d is the number of weakly connected components of graph G .

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases string algorithm, graph algorithm, de Bruijn graph, Eulerian graph

Digital Object Identifier 10.4230/LIPIcs.CPM.2025.12

Funding A research visit during which part of the presented ideas were conceived was funded by a Royal Society International Exchanges Award.

Wiktor Zuba: Supported in part by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 101034253.

Solon P. Pissis: Supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

1 Introduction

We start with some basic definitions and notation from [5]. An *alphabet* Σ is a finite set of elements called *letters*. We consider an integer alphabet $\Sigma = [0, \sigma)$. Let $X = X[0] \cdots X[n-1]$ be a *string* of length $n = |X|$ over Σ . By Σ^k we denote the set of all strings of length $k > 0$. For two indices i and $j \geq i$ of X , $X[i..j]$ is the *fragment* of X starting at position i and ending at position j . The fragment $X[i..j]$ is an *occurrence* of the underlying *substring* $P = X[i] \cdots X[j]$; we say that P occurs at *position* i in X . A *prefix* of X is a substring of the form $X[0..j]$ and a *suffix* of X is a substring of the form $X[i..n-1]$. By XY or $X \cdot Y$ we denote the *concatenation* of strings X and Y : $XY = X[0] \cdots X[|X|-1]Y[0] \cdots Y[|Y|-1]$.

Let us fix a collection \mathcal{S} of strings over alphabet Σ . We define the *order- k de Bruijn graph* (dBG, in short) of \mathcal{S} as a directed multigraph, denoted by $G_{\mathcal{S},k}(V, E)$, where V is the set of length- k substrings of the strings in \mathcal{S} and E has an edge (u, v) with multiplicity $m_{u,v}$ if and only if the strings $u[0] \cdot v$ and $u \cdot v[k-1]$ are equal and occurring exactly $m_{u,v}$ times in total in the strings of collection \mathcal{S} . In bioinformatics, \mathcal{S} models a collection of



© Wiktor Zuba, Oded Lachish, and Solon P. Pissis;
licensed under Creative Commons License CC-BY 4.0

36th Annual Symposium on Combinatorial Pattern Matching (CPM 2025).

Editors: Paola Bonizzoni and Veli Mäkinen; Article No. 12; pp. 12:1–12:13

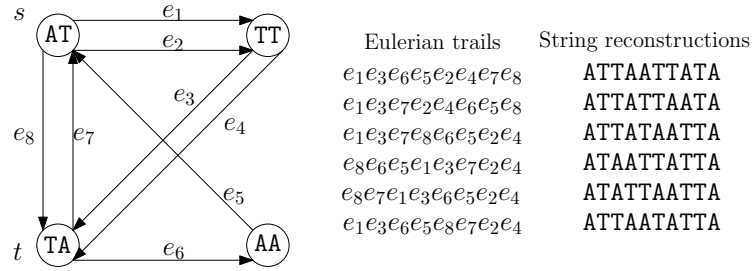


Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

DNA sequences coming from a genome sample through a sequencing experiment, and any Eulerian trail of $G_{S,k}(V, E)$ – a graph walk using each edge of $G_{S,k}(V, E)$ exactly once – represents a *potential* reconstruction of the genome [21, 18]. Inspect Figure 1. This is an idealized model though, since $G_{S,k}$ would never be Eulerian in practice due to sequencing errors [19]; and, furthermore, $G_{S,k}$ would not even be weakly connected. We could make $G_{S,k}$ Eulerian by increasing the multiplicity of some of its *existing* edges [17] or introducing *new ones* [5]. In either case, the natural optimization goal is to minimize the total cost (number) of the added edges. In fact, many algorithms underlying genome assembly tackle similar problems [24, 1, 9, 23].

We also define the *complete de Bruijn graph* of order k over the alphabet Σ , denoted by $G_{\Sigma,k}(V, E)$, as a directed graph, where V is the set of all the strings from Σ^k and E contains an edge (u, v) if and only if string v is obtained from string u by appending letter $v[k-1]$ after its last position and removing letter $u[0]$: $G_{\Sigma,k}(V, E)$ has $|\Sigma|^k$ nodes and $|\Sigma|^{k+1}$ edges.

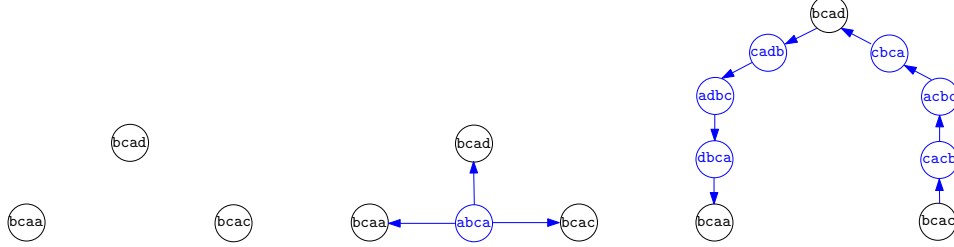


■ **Figure 1** Let \mathcal{S} be a string collection consisting of strings AAT, ATA, ATT, ATT, TAA, TAT, TTA, TTA. Note that, for example, string ATT has multiplicity 2 in \mathcal{S} . The de Bruijn graph $G_{S,2}(V, E)$ is presented on the left; the complete set of node-distinct Eulerian trails from s to t are presented in the middle – we call two trails *node-distinct* if their node sequences are different in accordance with the existing literature [3, 10]. Node-distinct Eulerian trails correspond to distinct strings; the corresponding set of possible string reconstructions is presented on the right.

Our Motivation. Bernardini et al. [4] studied the problem of *making any arbitrary* $G_{S,k}$ *weakly connected* by introducing a set of new edges of minimum total cost (as well as the underlying set of new nodes when those do not exist in $G_{S,k}$); by arbitrary $G_{S,k}$, we mean an arbitrary subgraph of the complete DBG $G_{\Sigma,k}$. Note that making any arbitrary graph weakly connected by introducing a set of new edges of minimum total cost is trivial: if the graph consists of d components, the solution is to greedily add $d - 1$ edges to connect the components. The algorithmic challenge with DBGs follows by their definition: not every pair of nodes can be connected via a single edge. Solving this problem is important because we can then use the linear-time algorithm of Bernardini et al. from [5] to balance the weakly connected graph by adding a set of edges of minimum total cost thus making $G_{S,k}$ Eulerian.¹ Recall that making $G_{S,k}$ *directly* Eulerian by adding a set of new edges of minimum total cost is NP-hard from the well-known *shortest common superstring* problem [12]; hence, the *connect-and-balance* strategy is reasonable and generally serves as a good-performing heuristic [5]. Two remarks are in order: first, since the general task is to connect $G_{S,k}$, we can safely assume that $m_{u,v}$ is either 0 or 1 (i.e., multiplicities play no role here); and second,

¹ By Euler's famous theorem, we know that a weakly connected directed graph is Eulerian if and only if every graph node has equal in-degree and out-degree (except perhaps for the source and target).

since the task, in particular, is to connect $G_{S,k}$ with the smallest number of edge additions, we should rather seek shortest *undirected* paths in $G_{S,k}$ (i.e., shortest sequences of edges from $G_{\Sigma,k}$, in which the edge directions are neglected). Inspect Figure 2.



■ **Figure 2** A de Bruijn graph of order $k = 4$ with 3 weakly connected components (left); an optimal solution, using shortest *undirected* paths, with cost 3 (middle); a feasible solution, using shortest *directed* paths, with cost 8 (right). We color blue the edges and nodes that we add from $G_{\Sigma,k}$.

Bernardini et al. [4] showed that making $G_{S,k}$ weakly connected by adding a set of edges of minimum total cost is NP-hard. They also showed that no polynomial-time approximation scheme (PTAS) exists for making $G_{S,k}$ weakly connected by adding a set of edges of minimum total cost (unless the *unique games conjecture* [13] fails). Finally, they also showed that there exists an $\mathcal{O}(k|V|^2)$ -time $(2 - 2/d)$ -approximation algorithm for the same problem, where d is the number of connected components of $G_{S,k}$. In this paper, we introduce a *general framework* for finding shortest undirected paths in DBGs. In particular, using our framework, we improve the running time of the approximation algorithm of Bernardini et al. from $\mathcal{O}(k|V|^2)$ to $\mathcal{O}(k|V| \log d)$, while maintaining *the same* approximation ratio.

Our Framework. Let us fix d families C_1, C_2, \dots, C_d of nodes (forming connected components in most applications) from $G_{\Sigma,k}$, and let us denote $V = C_1 \sqcup C_2 \sqcup \dots \sqcup C_d$.² Throughout this paper, we treat nodes of DBGs and their length- k string representations as equivalent: indeed, nodes of $G_{\Sigma,k}$ are in a natural bijection with Σ^k . In particular, C_p , for any $p \in [1, d]$, and V are also treated as sets of strings.

We generally aim at obtaining efficient algorithms for finding the minimal distance between the nodes from two different families C_p and C_q ; more formally, for any $p, q \in [1, d]$,

$$\text{dist}(p, q) = \min\{\text{dist}(u, v) : u \in C_p, v \in C_q\},$$

where $\text{dist}(u, v)$ is the *length of a shortest undirected path* from node u to node v in $G_{\Sigma,k}$.

Note that if C_p and C_q form two weakly connected components of $G_{\Sigma,k}$, then $\text{dist}(p, q)$ is precisely the minimum number of edges that must be added to *connect* the two components into a single one – assuming that we also add the nodes implied by those edges. In the same manner, by adding $d - 1$ such undirected paths, we can (greedily) connect C_1, C_2, \dots, C_d to a single component, thus making any arbitrary DBG $G_{S,k}$ weakly connected.

The problem of finding the shortest *directed* path between any two nodes of $G_{\Sigma,k}$ can be solved in the optimal $\mathcal{O}(k)$ time using the preprocessing of the classic KMP algorithm [14]. The same problem for *undirected* paths can also be solved in the optimal $\mathcal{O}(k)$ time [16]. By iteratively applying the latter result, we can compute $\text{dist}(p, q)$ in $\mathcal{O}(k|C_p| \cdot |C_q|)$ time and for all p, q pairs in $\mathcal{O}(k|V|^2)$ total time, which is very slow when $|V|$ is large, even if the number

² We assume that these families are pairwise disjoint. However, our solutions work without this assumption.

d of families is relatively small. Using more refined techniques, based on the generalized suffix tree [25] of the strings from V , we develop algorithms for finding the distances much more efficiently when the families are large or when there are many of them.

In particular, given the collection C_1, C_2, \dots, C_d , we consider the following problems:

- **One-to-One**(p, q): output $\text{dist}(p, q)$. Here we are given, in addition, p and q , and we are asked to find the length of a shortest undirected path between *any* node of C_p and *any* node of C_q .
- **One-to-All**(p): output $\text{dist}(p, q)$, for every $q \in [1, d]$. Here we are given, in addition, p , and we are asked to find the length of a shortest undirected path between *any* node of C_p and *any* node of C_q , for every $q \in [1, d]$.
- **All-to-All**: output $\text{dist}(p, q)$, for all $p, q \in [1, d]$. Here we are asked to find the length of a shortest undirected path between *any* node of C_p and *any* node of C_q , for all $p, q \in [1, d]$.
- **Top**(r): Here we are given, in addition, r , and we are asked to output, for every $q \in [1, d]$, r distinct $p \in [1, d]$ with the smallest value of $\text{dist}(p, q)$, breaking ties arbitrarily.

Let us remark that the algorithms underlying our framework are *constructive* – whenever we compute $\text{dist}(p, q)$, we also know a pair $u \in C_p, v \in C_q$ of nodes that are at distance $\text{dist}(p, q)$. By applying the technique from [16], we can enhance the output of the above algorithms with optimal paths realising those distances at the additional linear cost in the size of the output – k times the length of a shortest path (every node is explicitly encoded using k letters). If the output is given in a compacted form – the differences between two consecutive nodes on the path (the new letter introduced and whether it is put in the front or back) – then the additional cost reduces to the length of the shortest path.

Our Results. We make the following specific contributions:

- an algorithm solving **One-to-One**(p, q) in $\mathcal{O}(k(|C_p| + |C_q|))$ time and space;
- an algorithm solving **One-to-All**(p) in $\mathcal{O}(k|V|)$ time and space;
- an algorithm solving **All-to-All** in $\mathcal{O}(dk|V|)$ time and $\mathcal{O}(k|V|)$ space;
- an algorithm solving **Top**(r) in $\mathcal{O}(rk|V|)$ time and space.

Application. By plugging our results directly in the approximation algorithm of Bernardini et al. [4], we improve the running time of their algorithm for connecting DBGs from $\mathcal{O}(k|V|^2)$ to $\mathcal{O}(dk|V|)$. Using more refined techniques, we obtain an even further improvement; in particular, we design an $\mathcal{O}(k|V| \log d)$ -time algorithm for the same task, while maintaining the same approximation ratio of $(2 - 2/d)$ [4].

Paper Organization. In Section 2, we present some preliminaries that essentially summarize the work in [16]. In Section 3, we present a *simple* linear-time algorithm for computing shortest paths in undirected DBGs. In Section 4, we present our framework: how distances between sets of nodes can be computed more efficiently in many settings. In Section 5, we apply our framework to the problem of making any arbitrary DBG weakly connected [4].

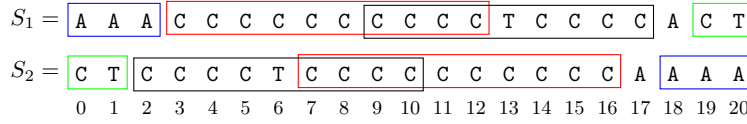
2 Preliminaries

Let $[k]$ denote the set $\{0, 1, \dots, k-1\}$, and let $S[i..j] = S[i..j+1)$ denote the substring $S[i]S[i+1]\dots S[j]$ of S . Let $S_1, S_2 \in \Sigma^k$ represent nodes v_1 and v_2 (respectively) of $G_{\Sigma, k}$.

Let U be a common substring of S_1 and S_2 and assume that it occurs in those strings at positions i and j respectively, with $i \leq j$. Notice that we can transform S_1 into S_2 by first removing the first i letters of S_1 (appending i arbitrary letters at its end at the same

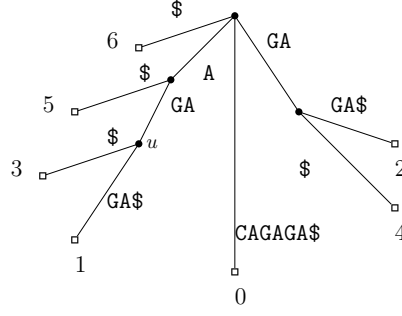
time), then adding the first j letters of S_2 to its front (removing j letters from its end – in particular all the letters added in the previous step), and then symmetrically remove the last $k - j - |U|$ letters and add the length- $(k - j - |U|)$ suffix of S_2 in their place. In total, this requires $i + j + 2 \cdot (k - j - |U|) = 2k - 2|U| - (j - i)$ operations, and each such operation corresponds to an edge of the complete dBG. It turns out that one cannot get a path from v_1 to v_2 of shorter length other than by choosing a *different* common substring or *different* occurrences. This fact is summarized in Lemma 1 by Liu [16]. Inspect Figure 3.

► **Lemma 1** ([16]). *Let v_1, v_2 be two nodes of $G_{\Sigma, k}$ and $S_1, S_2 \in \Sigma^k$ be the string representations of v_1, v_2 , respectively. Then $\text{dist}(v_1, v_2) = 2k - \max_{i,j \in [k]} (2 \cdot |U_{i,j}| + |j - i|)$, where $U_{i,j}$ is the longest common prefix of $S_1[i..k-1]$ and $S_2[j..k-1]$.*



■ **Figure 3** Consider the strings S_1 and S_2 for $k = 21$. The distance from S_1 to S_2 in the directed de Bruijn graph is equal to 19 (CT is the longest suffix of S_1 that is a prefix of S_2), and the distance from S_2 to S_1 is equal to 18 (AAA is the longest suffix of S_2 that is a prefix of S_1). In the undirected case, the distance between the two nodes is $2 \cdot 21 - 2 \cdot 9 - 7 = 17$ witnessed by the common substring C^4TC^4 . Notice that even though C^{10} is a longer common substring it cannot be used to obtain a shortest path between the nodes because it appears in S_1 at position 3 and in S_2 at position 7 (and $2 \cdot 10 + 4 < 2 \cdot 9 + 7$). This example shows that *it is not enough* to look at prefixes, suffixes or the longest common substring when computing the distance in the undirected de Bruijn graph.

Suffix Tree. The classic indexing solution for standard strings is the suffix tree [25]. Given a set \mathcal{F} of strings, the *compacted trie* of these strings is the trie obtained by compressing each path of nodes of degree one in the trie of the strings in \mathcal{F} . The dissolved nodes are called *implicit* while the preserved nodes are called *explicit*. For any node u (implicit or explicit) of the compacted trie, $\text{str}(u)$ denotes the string spelled from the root of the trie to u ; the *string depth* of u is then defined as $d(u) = |\text{str}(u)|$. Each edge in the compacted trie has a label represented as a fragment of a string in \mathcal{F} . Thus, the compacted trie uses $\mathcal{O}(|\mathcal{F}|)$ space [20]. The *suffix tree* $\text{ST}(S)$ is the compacted trie of the suffixes of string S . Assuming S ends with a unique terminating symbol $\$,$ every suffix $S[i..|S|]$ of S is represented by a leaf annotated with index i (the starting position of the suffix in S). The root node, the branching nodes, and the leaf nodes form the set of explicit nodes of $\text{ST}(S)$; inspect Figure 4. The suffix tree occupies $\mathcal{O}(|S|)$ space and it can be constructed in $\mathcal{O}(|S|)$ time [25, 11]. The edges of $\text{ST}(S)$ can be accessed in $\mathcal{O}(1)$ time if stored using perfect hashing [2]. The suffix tree then supports pattern matching queries for any pattern P of length m in $\mathcal{O}(m + |\text{Occ}|)$ time, where Occ is the set of output occurrences of P in S . The suffix tree can also be generalized to a collection $\mathcal{S} = \{S_1, \dots, S_N\}$ of N strings as the compacted trie $\text{ST}(\mathcal{S})$ of the suffixes of string $S_1\$_1 \cdots S_N\$_N$, where $\$_i \notin \Sigma$, for $i \in [1, N]$, are distinct terminating symbols.



■ **Figure 4** The suffix tree $\text{ST}(S)$ of string $S = \text{CAGAGA}\$$. The node representing string AG is *implicit* and thus it is not stored explicitly in this compacted trie. The node u representing string AGA is *explicit* (it is a *branching* node) and thus it is stored; we have $\text{str}(u) = \text{AGA}$ and $d(u) = 3 = |\text{AGA}|$.

3 Simple Pairwise Distance Computation using Suffix Tree

Let $I_1(U) = \{i : S_1[i..i + |U|] = U\}$ and let $I_2(U) = \{j : S_2[j..j + |U|] = U\}$. The sets $I_1(U)$ and $I_2(U)$ represent the occurrences of string U in strings S_1 and S_2 , respectively. The distance $\text{dist}(v_1, v_2)$ as described in Lemma 1 can also be expressed as follows:³

$$\text{dist}(v_1, v_2) = 2k - \max_U (2|U| + \max[\max(I_1(U)) - \min(I_2(U)), \max(I_2(U)) - \min(I_1(U))]). \quad (1)$$

Equation (1) changes the order of the maxima: the outer one is over any substring U , and the inner one is over the occurrences (starting positions) of U . We will next view Equation (1) through the lens of the generalized suffix tree of S_1 and S_2 .

Let $\text{ST}(\{S_1, S_2\})$ be the generalized suffix tree of S_1 and S_2 . Since $|S_1| = |S_2| = k$, $\text{ST}(\{S_1, S_2\})$ has $\mathcal{O}(k)$ explicit nodes (and edges). For an explicit node v of $\text{ST}(\{S_1, S_2\})$, let $d(v)$ be its string depth (the length of the substring represented), $L_v^x = \min\{i : S_x[i..k-1] \text{ is a leaf descendant of } v\}$, and $R_v^x = \max\{i : S_x[i..k-1] \text{ is a leaf descendant of } v\}$ for $x \in \{1, 2\}$.

Each common substring U is represented by an explicit (or implicit) node of $\text{ST}(\{S_1, S_2\})$. Notice, that if both U and $U' = Ua$ for some letter $a \in \Sigma$ appear at the very same positions (both in S_1 and S_2), then the value of the formula for U' is larger than the one for U by exactly 2, hence it is enough to focus on the explicit nodes of $\text{ST}(\{S_1, S_2\})$ when we want to compute the optimal value. Hence Equation (1) can also be expressed as follows:

$$\text{dist}(v_1, v_2) = 2k - \max_{v \in \text{ST}(\{S_1, S_2\})} (2 \cdot d(v) + \max[R_v^1 - L_v^2, R_v^2 - L_v^1]). \quad (2)$$

► **Observation 2.** $L_v^x = \min_{w \in \text{Children}(v)} L_w^x$ (respectively $R_v^x = \max_{w \in \text{Children}(v)} R_w^x$) for a branching node v , where $\text{Children}(v)$ is the set of direct descendants of v in the suffix tree. For a leaf v , the set from which L_v^x (resp. R_v^x) is taken is either a singleton or an empty set.

A direct consequence of Observation 2 is that the values L_v^1, R_v^1, L_v^2 , and R_v^2 can be computed bottom up. We thus compute these 4 values using a bottom-up traversal (and the depth $d(v)$ via a top-down traversal) for each node in $\mathcal{O}(k)$ total time. This gives a *simple* $\mathcal{O}(k)$ -time algorithm for computing $\text{dist}(u_1, u_2)$ – after constructing the suffix tree and computing those values, it suffices to find the optimal value of Equation (2) over all the explicit nodes.

³ If $I_1(U) = \emptyset$ or $I_2(U) = \emptyset$, then the distance for this U as witness is equal to ∞ , and hence the distance remains the same whether such U are considered or not, thus the abstract formula can iterate over Σ^* .

► **Lemma 3.** *Let v_1, v_2 be two nodes of $G_{\Sigma, k}$ and $S_1, S_2 \in \Sigma^k$ be the string representations of v_1, v_2 , respectively. We can compute $\text{dist}(v_1, v_2)$ in $\mathcal{O}(k)$ time using $\text{ST}(\{S_1, S_2\})$.*

We note that a different, yet much more complicated, $\mathcal{O}(k)$ -time algorithm for computing $\text{dist}(v_1, v_2)$ using suffix trees has already been given by Liu in [16].

Recall that $\text{dist}(p, q) = \min_{v_1 \in C_p, v_2 \in C_q} \text{dist}(v_1, v_2)$. A naïve application of Lemma 3 for computing $\text{dist}(p, q)$ by explicitly computing the pairwise distance between all the pairs of nodes runs in $\mathcal{O}(k|C_p| \cdot |C_q|)$ time. In the next section, we present our framework: how distances between sets of nodes can be computed more efficiently in many settings.

4 Our Framework for Shortest Undirected Paths in de Bruijn Graphs

In this section, we provide simple and efficient solutions to the considered distance (shortest-path) problems. Our solutions rely only on the generalized suffix tree of the strings in question (V or $C_p \cup C_q$) and standard operations (graph traversals and dynamic programming) and hence are not only of theoretical interest, but should also admit efficient implementations.

4.1 One-to-One

Recall that in **One-to-One**, we are given C_1, C_2, \dots, C_d , p and q , and we are asked to find the length of a shortest undirected path between any node of C_p and any node of C_q .

Note that in Equation (1), it does not really matter from which string in C_p (resp. C_q) the occurrence of U at position i (resp. j) originates: by changing the order of the minima in the formula, we have that $\text{dist}(p, q)$ can be expressed by Equation (1) if we naturally extend the set $I_x(U) = \{i : \exists S \in C_x S[i \dots i + |U|] = U\}$, for $x \in \{p, q\}$. In particular, with the use of L_v^x, R_v^x , for $x \in \{p, q\}$, based on the suffixes of all the strings representing C_x , Equation (2) generalizes to the following one:

$$\text{dist}(p, q) = 2k - \max_{v \in \text{ST}(C_p \cup C_q)} (2 \cdot d(v) + \max[R_v^p - L_v^q, R_v^q - L_v^p]). \quad (3)$$

Inspect Figure 5.

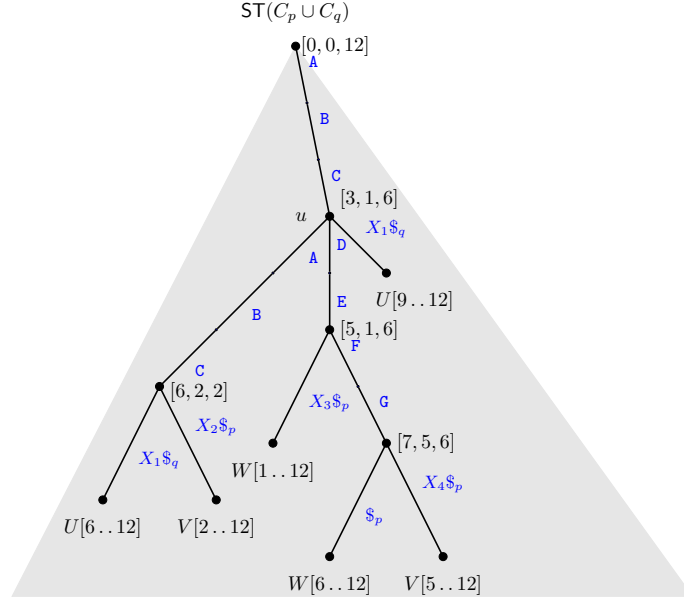
Since the size of $\text{ST}(C_p \cup C_q)$ is in $\mathcal{O}(k \cdot (|C_p| + |C_q|))$, and it can be constructed in linear time, by repeating the same operations as for the computation of $\text{dist}(v_1, v_2)$, we obtain:

► **Theorem 4.** *We can solve **One-to-One** in $\mathcal{O}(k \cdot (|C_p| + |C_q|))$ time and space.*

4.2 One-to-All

Recall that in **One-to-All**(p), we are given C_1, C_2, \dots, C_d and p , and we are asked to find the length of a shortest undirected path between any node of C_p and any node of C_q , for every $q \in [1, d]$. Using Theorem 4, we directly obtain a solution to **One-to-All**(p) and to **All-to-All** running in $\mathcal{O}(k(|V| + d|C_p|))$ and $\mathcal{O}(dk|V|)$ time, respectively, using $\mathcal{O}(k \max_q |C_q|)$ space. We now proceed to a more refined processing of the suffix tree that allows us to obtain a more efficient algorithm for **One-to-All**, which is later used to solve **Top**(r). Our high-level goal is to reduce the number of nodes over which we must optimize Equation (3).

Let $A(v)$ denote the set of all ancestors of v in $\text{ST}(V)$ (including the node itself). Further let $\text{Min}_v^x = \max_{w \in A(v)} [2 \cdot d(w) - L_w^x]$, and $\text{Max}_v^x = \max_{w \in A(v)} [2 \cdot d(w) + R_w^x]$, for $x \in [1, d]$. Recall that $V = C_1 \sqcup C_2 \sqcup \dots \sqcup C_d$. Recall also that as noted in Observation 2 for the leaf nodes of $\text{ST}(V)$, denoted by $\text{Leaves}(\text{ST}(V))$, the sets $I_x(U)$ are either equal to a singleton



■ **Figure 5** The information $[d(v), L_v^p, R_v^p]$ computed for the explicit non-leaf nodes v of $\text{ST}(C_p \cup C_q)$ restricted to the part of the suffix tree where the only edge going out of the root is labeled by letter A. Let $U \in C_q$, $V, W \in C_p$, and $k = 13$, for $U = \text{CBDCCCABCABCE}$, $V = \text{CDABCABCDEFGB}$, and $W = \text{BABCDEABCDEFGB}$. The labels on the edges leading to leaves are compacted for simplicity (and represented only with the suffixes X_1, \dots, X_4). By additionally computing $L_u^q = 6$ and $R_u^q = 9$, for the explicit node u representing ABC , we get the distance $2 \cdot 13 - 2 \cdot 3 - \max(9 - 1, 6 - 6) = 12$ witnessed by the common substring ABC . By comparing the distances witnessed by all the common substrings (nodes of the suffix tree), we obtain the minimal distance 10 witnessed by ABCABC .

$\{i_v^x\}$ (when $S[i_v^x \dots k - 1] = U$ for $S \in C_x$ is represented by node v) or empty.⁴ In particular i_v^x exists only for a single x (as symbols $\$x$ are distinct). We can thus define Equation (4) as a more refined version of Equation (3):

$$\text{dist}(p, q) = 2k - \max_{v \in \text{Leaves}(\text{ST}(V))} (\max [\text{Max}_v^p - i_v^q, \text{Min}_v^p + i_v^q]). \quad (4)$$

► **Example 5.** Consider the example from Figure 5: $U = \text{CBDCCCABCABCE} \in C_q$, $V = \text{CDABCABCDEFGB} \in C_p$, and $W = \text{BABCDEABCDEFGB} \in C_p$. Consider the leaf nodes representing the strings $V[2 \dots 12]$ and $U[6 \dots 12]$. Both leaf nodes have the following ancestors (other than themselves) and the following $[d, L_v^p, R_v^p, L_v^q, R_v^q]$ values:

- ABCABC : $[6, 2, 2, 6, 6]$;
- ABC : $[3, 1, 6, 6, 9]$;
- empty string ε (root node): $[0, 0, 12, 0, 12]$.

Let v be the leaf node representing $U[6 \dots 12]$. We iterate over all ancestors w of v :

- $\text{Min}_v^p = \max_{w \in A(v)} [2 \cdot d(w) - L_w^p] = \max\{-\infty, 10, 5, 0\} = 10$;
- $\text{Max}_v^p = \max_{w \in A(v)} [2 \cdot d(w) + R_w^p] = \max\{-\infty, 14, 12, 12\} = 14$.

⁴ Note that for a leaf v , $d(v) = k - i_v^x$; that is, the label $\$$ is not taken into account in computation of the length of the common substring represented – this plays a role only when computing $\text{dist}(p, p)$ anyway.

We have $2k - \max[\text{Max}_v^p - i_v^q, \text{Min}_v^p + i_v^q] = 26 - (10 + 6) = 10$, which gives us the minimal distance, between C_q and C_p witnessed by ABCABC.

The following lemma is crucial for the correctness of our approach.

► **Lemma 6.** *Equations (3) and (4) are equivalent.*

Proof. Let v be the node of ST for which Equation (3) attains the optimal value, which w.l.o.g. is equal to $2k - \max(2 \cdot d(v) + R_v^q - L_v^p)$, and let u, w be the leaf descendants of v for which $L_v^p = i_u^p$ and $R_v^q = i_w^q$.

Since v is an ancestor of w , we know that $\text{Min}_w^p \geq 2 \cdot d(v) - i_u^p$, hence $\text{Min}_w^p + i_w^q \geq 2 \cdot d(v) - i_u^p + i_w^q = 2 \cdot d(v) - L_v^p + R_v^q$, thus the value of Equation (4) (witnessed by node w) is at least as large as the value of Equation (3) (witnessed by the node v).

For the converse inequality, w.l.o.g. the value of Equation (4) is equal to $2k - [\text{Min}_w^p + i_w^q]$ for a leaf node w . By the definition of Min_w^p there exists an ancestor v of w such that $\text{Min}_w^p = 2 \cdot d(v) - L_v^p$. Hence $2 \cdot d(v) - L_v^p + R_v^q \geq \text{Min}_w^p + i_w^q$ (as $R_v^q \geq i_w^q$), which shows that the value of Equation (3) cannot be smaller than the value of Equation (4). ◀

► **Observation 7.** $\text{Min}_v^p = \max(\text{Min}_{\text{Parent}(v)}^p, 2d(v) - L_v^p)$.⁵

A direct consequence of Observation 7 is that the values Min_v^p and Max_v^p can be computed top down. By computing these 2 values using a top-down traversal in $\mathcal{O}(k|V|)$ total time and space for all explicit nodes and computing Equation (4) for the leaves, we obtain:

► **Theorem 8.** *We can solve One-to-All in $\mathcal{O}(k|V|)$ time and space.*

By directly computing the values Min_v^p and Max_v^p , for all $p \in [1, d]$, we obtain another algorithm to solve All-to-All in $\mathcal{O}(dk|V|)$ time. For this algorithm, the space used is $\mathcal{O}(dk|V|)$. By simply using Theorem 8 d times, the required space is reduced to $\mathcal{O}(k|V|)$. The computation of all the values in a single run over the generalized suffix tree has other nice properties, however – as shown in the next section it allows to restrict the output while also reducing the computation time and space.

4.3 Top

Recall that in $\text{Top}(r)$, we are given C_1, C_2, \dots, C_d and an integer r , and we are asked to output, for every $q \in [1, d]$, r distinct $p \in [1, d]$ with the smallest value of $\text{dist}(p, q)$, breaking ties arbitrarily.

We start with the following simple yet crucial observation: If for a fixed leaf v of $\text{ST}(V)$, the values of Min_v^p and Max_v^p over p are ordered nonincreasingly, it suffices to know the first r of those for each type (Min and Max), that is, we do not need to compute all the $2d$ values.

Recall that the Min_v^p and Max_v^p values are computed by first computing the values L_w^p and R_w^p bottom up and then computing the values Min_w^p and Max_w^p top down for every node w of $\text{ST}(V)$. If we store the best (smallest for L and largest for Min, Max and R) r values (over $p \in [1, d]$) for each of those 4 types, the values for the parent/children can be computed in time proportional to the number of nodes multiplied by r . Indeed when computing the smallest (up to) r values of L_w^p , it suffices to find the r smallest elements of the values stored in the children; hence that can be computed in $\mathcal{O}(rc)$ time, where c is the number of children of w – this sums up to $\mathcal{O}(rk|V|)$ total time over all explicit nodes of $\text{ST}(V)$. Here

⁵ The equation for Max_v^p is analogous: $\max(\text{Max}_{\text{Parent}(v)}^p, 2d(v) + R_v^p)$.

we use $\mathcal{O}(rc)$ time to exclude the duplicate values $p \in [1, d]$ – a check if this set C_p is already represented can be done using an extra integer array of size d (only one array for the whole computation) with $\mathcal{O}(1)$ -time updates.

After the computation of L_w^p (resp. R_w^p) for all the nodes, the r largest values Min_w^p (resp. Max_w^p) can be obtained using Observation 7 from the values stored in the parent node, and the r values $2 \cdot d(w) - L_w^p$ (resp. $2 \cdot d(w) + R_w^p$) – which are already sorted nonincreasingly due to the sorted order on L_w^p (resp. R_w^p).

Finally, we once again simply iterate over the leaves of $\text{ST}(V)$, and gather the r smallest values of $\text{dist}(p, q)$ over all the leaves representing a suffix of a string from C_q (using a bucket queue). We obtain the following result.

► **Theorem 9.** *Top(r) can be computed in $\mathcal{O}(rk|V|)$ time and space.*

5 Application: Connecting de Bruijn Graphs Efficiently

While our framework may have other applications revolving around DBGs (e.g., [6, 7]), in this section, we showcase the application of making an arbitrary DBG weakly connected.

Let us fix an arbitrary DBG of order k consisting of d weakly connected components and also denote it by $G(V, E)$. Bernardini et al. [4] proved the following result.

► **Theorem 10** ([4]). *For any order- k DBG $G(V, E)$ consisting of d weakly connected components, there exists an $\mathcal{O}(k|V|^2)$ -time $(2 - 2/d)$ -approximation algorithm for making G weakly connected by adding a set of edges of minimum total cost.*

We improve Theorem 10 by slashing a factor of $|V|/\log d$ from the running time. Let $G'(V', E')$ be the graph obtained from the complete DBG $G_{\Sigma, k}$ by collapsing each component C_p , $p \in [1, d]$, of $G(V, E)$ into one super-node. The solution underlying Theorem 10 consists of the following three steps:

- (i) Construct the metric closure⁶ of G' .
- (ii) Compute a minimum spanning tree of the metric closure.
- (iii) Convert the minimum spanning tree into a set of nodes and a set of edges to be added to G to make it weakly connected.

The correctness follows directly from the fact that a minimum spanning tree for the metric closure of G' is a $(2 - 2/d)$ -approximation for the minimum Steiner tree [15],⁷ where d is the number of terminals and thus the number of weakly connected components of G . Step (i) requires $\mathcal{O}(k|V|^2)$ time by applying Lemma 3. Step (ii) can be done in $\mathcal{O}(d^2)$ time by applying, e.g., Prim's algorithm [22]. Finally, Step (iii) can be done by applying again Lemma 3 to compute a shortest undirected path. This sums up to $\mathcal{O}(k|V|^2)$ total time.

To complement Theorem 10, Bernardini et al. also showed that making $G(V, E)$ weakly connected by adding a set of edges of minimum total cost is NP-hard and admits no PTAS.

Theorem 10 can be improved using our framework: Theorem 4 directly outputs the weights of the edges of the metric closure G' in $\mathcal{O}(dk|V|)$ time; this improves the running time from $\mathcal{O}(k|V|^2)$ to $\mathcal{O}(dk|V|)$ time. In particular, with the use of the **Top** queries, we can obtain an even more efficient solution by removing the construction of the metric closure and instead computing its spanning tree directly from our input.

⁶ Recall that the *metric closure* of a graph G' is the complete graph in which each edge is weighted by the shortest path distance between the nodes in G' .

⁷ Recall that the *minimum Steiner tree* problem asks, given a graph G' with nonnegative edge weights and a subset of *terminal* nodes, to compute a tree of minimum weight that contains all terminals.

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be an arbitrary *undirected weighted* graph. Further, let E_{Top} be a subset of \mathcal{E} defined by choosing, for each node $v \in \mathcal{V}$, an edge incident with v with the smallest weight (one of such edges in case of ties), and then removing, from each cycle obtained this way, the heaviest edge (breaking ties arbitrarily). We next state the following well-known lemma (cf. [8]), and also prove it here for completeness.

► **Lemma 11.** *E_{Top} is a subset of a minimum spanning tree of $\mathcal{G}(\mathcal{V}, \mathcal{E})$.*

Proof. We show that starting from any arbitrary spanning tree of \mathcal{G} , we can modify it using a greedy approach so that the obtained spanning tree contains all edges of E_{Top} , and has a total weight at most as large as the weight of the initial spanning tree.

We iterate over the edges of E_{Top} ; we add the edge to the spanning tree and then remove one edge from the newly created cycle: the one with the largest weight and, among possibly many such edges, we prefer an edge that does not belong to E_{Top} .

Clearly, such an operation cannot increase the weight of the spanning tree – the worst-case scenario is that the newly added edge is immediately removed. If by applying this procedure, we never remove any edge from E_{Top} , then the claimed solution exists.

Towards the contradiction assume that after adding an edge $e_1 \in E_{\text{Top}}$, we create a cycle in which *all* the heaviest edges belong to E_{Top} . Take one such edge – it was chosen for a node v . The other edge of the cycle incident with v cannot be lighter (by the definition of E_{Top}), hence by assumption it must have the same weight and hence must also belong to E_{Top} (by the assumption of this paragraph of the proof); we move on to the node for which this edge was chosen, and continue in the same way. Since the cycle is finite, at some point we have to come back to v – but this means that the edges from E_{Top} formed a cycle – a contradiction with the definition of E_{Top} . Thus, E_{Top} is a subset of a minimum spanning tree of $\mathcal{G}(\mathcal{V}, \mathcal{E})$. ◀

Recall that $G'(V', E')$ is the graph obtained from the complete dBG $G_{\Sigma, k}$ by collapsing each component C_p , $p \in [1, d]$, of $G(V, E)$ into one super-node. We show the following lemma (implementing the algorithm of [8]).

► **Lemma 12.** *We can find a minimum spanning tree of $G'(V', E')$ in $\mathcal{O}(k|V| \log d)$ time using $\mathcal{O}(k|V|)$ space.*

Proof. Let us remark that we do not explicitly construct G' or $G_{\Sigma, k}$.

We start by producing a set of edges E_{Top} for G' using a $\text{Top}(r)$ query for $r = 2$. Such a query returns for each component C_q of G , that is, equivalently for each super-node q of G' , two super-nodes closest to it. In particular, what is implied by this is that even if one of those two is q itself (which happens in most cases), the other one must be different. By Theorem 9, in $\mathcal{O}(k|V|)$ total time, we find, for each super-node of G' , one incident edge with the smallest weight possible – by taking this set of edges and removing from each cycle a single edge we obtain a valid set of edges E_{Top} .

By Lemma 11, we can safely report this set of edges as part of the output. We can also contract the super-nodes of G' connected with the edges from E_{Top} into other single super-nodes obtaining graph G'' . Now every spanning tree of G' that contains E_{Top} is equivalent to the union of a spanning tree of G'' and the set E_{Top} , hence the problem reduces to finding the minimum spanning tree of G'' .

Notice that G'' is represented by the very same input to our original problem on a dBG, just with some of the sets C_p merged together, and so we can use the same generalized suffix tree just with different labels p, q . We can thus repeat the same approach until we reach a graph with a single super-node. Note that each such iteration takes $\mathcal{O}(k|V|)$ time

(Theorem 9), and that there can be no more than $\log_2 d$ such iterations because each time *every* super-node gets connected to another one, the number of super-nodes (components) drop by at least the factor 2 – the statement follows. ◀

By applying Lemma 12 to the solution from [4] we obtain the following improved result.

► **Theorem 13.** *For any order- k d BG $G(V, E)$ consisting of d weakly connected components, there exists an $\mathcal{O}(k|V|\log d)$ -time $(2 - 2/d)$ -approximation algorithm for making G weakly connected by adding a set of edges of minimum total cost.*

Since the algorithm underlying Theorem 13 is near-optimal, the main open question is whether we can improve the approximation ratio.

References

- 1 Nidia Obscura Acosta, Veli Mäkinen, and Alexandru I. Tomescu. A safe and complete algorithm for metagenomic assembly. *Algorithms Mol. Biol.*, 13(1):3:1–3:12, 2018. doi:10.1186/S13015-018-0122-7.
- 2 Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. Iceberg hashing: Optimizing many hash-table criteria at once. *J. ACM*, 70(6):40:1–40:51, 2023. doi:10.1145/3625817.
- 3 Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe text indexing. *ACM J. Exp. Algorithmics*, 26:1.10:1–1.10:26, 2021. doi:10.1145/3461698.
- 4 Giulia Bernardini, Huiping Chen, Inge Li Gørtz, Christoffer Krogh, Grigorios Loukides, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Connecting de Bruijn Graphs. In Shunsuke Inenaga and Simon J. Puglisi, editors, *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*, volume 296 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2024.6.
- 5 Giulia Bernardini, Huiping Chen, Grigorios Loukides, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Making de Bruijn graphs Eulerian. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CPM.2022.12.
- 6 Giulia Bernardini, Chang Liu, Grigorios Loukides, Alberto Marchetti-Spaccamela, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Missing value replacement in strings and applications. *Data Min. Knowl. Discov.*, 39(2):12, 2025. doi:10.1007/S10618-024-01074-3.
- 7 Giulia Bernardini, Alberto Marchetti-Spaccamela, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Constructing strings avoiding forbidden substrings. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.9.
- 8 Otakar Borůvka. O jistém problému minimálním. *Práce Moravské přírodovědecké společnosti*, sv. III(spis 3):37–58, 1926. URL: <https://dml.cz/handle/10338.dmlcz/500114>.
- 9 Massimo Cairo, Shahbaz Khan, Romeo Rizzi, Sebastian Schmidt, Alexandru I. Tomescu, and Elia C. Zironcelli. The hydrostructure: a universal framework for safe and complete algorithms for genome assembly, 2021. arXiv:2011.12635.
- 10 Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giulia Punzi. Beyond the BEST theorem: Fast assessment of eulerian trails. In Evripidis Bampis and Aris Pagourtzis, editors, *Fundamentals of Computation Theory - 23rd International Symposium, FCT 2021, Athens, Greece, September 12-15, 2021, Proceedings*, volume 12867 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2021. doi:10.1007/978-3-030-86593-1_11.

- 11 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.
- 12 John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980. doi:10.1016/0022-0000(80)90004-5.
- 13 Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the 17th Annual IEEE Conference on Computational Complexity, Montréal, Québec, Canada, May 21-24, 2002*, page 25. IEEE Computer Society, 2002. doi:10.1109/CCC.2002.1004334.
- 14 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 15 Lawrence T. Kou, George Markowsky, and Leonard Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, 1981. doi:10.1007/BF00288961.
- 16 Zhen Liu. Optimal routing in the de Bruijn networks. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, pages 537–544. IEEE Computer Society, 1990. doi:10.1109/ICDCS.1990.89261.
- 17 Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *7th WABI*, volume 4645 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 2007. doi:10.1007/978-3-540-74126-8_27.
- 18 Paul Medvedev and Mihai Pop. What do Eulerian and Hamiltonian cycles have to do with genome assembly? *PLOS Computational Biology*, 17(5):1–5, May 2021. doi:10.1371/journal.pcbi.1008928.
- 19 Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010. doi:10.1016/j.ygeno.2010.03.001.
- 20 Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968. doi:10.1145/321479.321481.
- 21 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci*, 98(17):9748–9753, 2001. doi:10.1073/pnas.171285098.
- 22 Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957. doi:10.1002/j.1538-7305.1957.tb01515.x.
- 23 Sebastian Schmidt, Shahbaz Khan, Jarno N Alanko, Giulio E Pibiri, and Alexandru I Tomescu. Matchtigs: minimum plain text representation of k-mer sets. *Genome Biology*, 24(1):136, 2023. doi:10.1186/s13059-023-02968-z.
- 24 Alexandru I. Tomescu and Paul Medvedev. Safe and complete contig assembly through omnitigs. *J. Comput. Biol.*, 24(6):590–602, 2017. doi:10.1089/CMB.2016.0141.
- 25 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.