Shortest Undirected Paths in de Bruijn Graphs

Wiktor Zuba
CWI, Amsterdam, The Netherlands
University of Warsaw, Warsaw, Poland

Oded Lachish
Birkbeck, University of London, 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 algorithmically much more challenging. Here we present a *general framework* for computing shortest undirected paths in arbitrary de Bruijn graphs. 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 minimal total cost. This improves on 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 G.

2012 ACM Subject Classification Theory of computation → Pattern matching

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

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

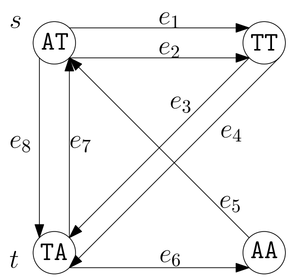
Solon P. Pissis: Supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

1 Introduction

We start with some basic definitions and notation from [3]. 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: $XY = X[0] \cdots X[|X|-1]Y[0] \cdots Y[|Y|-1]$. For strings X and Y, a *suffix/prefix overlap* of X and Y is a suffix of X that is a prefix of Y.

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 DNA sequences coming from a genome sample through a sequencing experiment, and any Eulerian trail of $G_{\mathcal{S},k}(V, E)$ – a graph path using each edge of $G_{\mathcal{S},k}(V, E)$ exactly once – represents a *potential* reconstruction of the genome [15, 12]. Inspect Figure 1. This is an idealised model though as $G_{\mathcal{S},k}$ would never be Eulerian in practice due to sequencing errors [13]; and, furthermore, $G_{\mathcal{S},k}$ would not even be weakly connected. We could make $G_{\mathcal{S},k}$ Eulerian by increasing the multiplicity of some of its *existing* edges [11] or introducing *new ones* [3]. In either case, the natural optimization goal is to minimize the total cost of the added edges. Indeed many algorithms underlying genome assembly tackle similar problems [18, 1, 4, 17].

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 has 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 $|\Sigma|^k$ vertices and $|\Sigma|^{k+1}$ edges.



Eulerian trails	String reconstructions
$e_1e_3e_6e_5e_2e_4e_7e_8$	ATTAATTATA
$e_1e_3e_7e_2e_4e_6e_5e_8$	ATTATTAATA
$e_1e_3e_7e_8e_6e_5e_2e_4$	ATTATAATTA
$e_8e_6e_5e_1e_3e_7e_2e_4$	ATAATTATTA
$e_8e_7e_1e_3e_6e_5e_2e_4$	ATATTAATTA
$e_1e_3e_6e_5e_8e_7e_2e_4$	ATTAATATTA

Figure 1 Let $\mathcal{S}$ be a string collection consisting of AAT, ATA, ATT, ATT, TAA, TAT, TTA, TTA. The de Bruijn graph $G_{\mathcal{S},2}(V, E)$ is presented on the left; the complete set of Eulerian trails from s to t is presented in the middle; the corresponding set of string reconstructions is presented on the right.

Our Motivation. Bernardini et al. [2] studied the problem of *making any arbitrary $G_{\mathcal{S},k}$ weakly connected* by introducing a set of new edges of minimal total cost (as well as the underlying set of new vertices when those do not exist in $G_{\mathcal{S},k}$). Solving this problem is important because we can then use the linear-time algorithm of Bernardini et al. from [3] to balance the weakly connected graph by adding a set of edges of minimal total cost thus making $G_{\mathcal{S},k}$ Eulerian.[1] Recall that making $G_{\mathcal{S},k}$ *directly* Eulerian by adding a set of new edges of minimal total cost is NP-hard from the well-known *shortest common superstring* problem [6]; hence, the *connect-and-balance* strategy, which, generally serves as a good-performing heuristic [3]. Two remarks are in order: first, since the general task is to connect $G_{\mathcal{S},k}$, we can safely assume that $m_{u,v}$ is either 0 or 1 (i.e., multiplicities play no role here); and second, since the task, in particular, is to connect $G_{\mathcal{S},k}$ with the smallest number of edge additions, we should rather seek shortest *undirected* paths in $G_{\mathcal{S},k}$ (i.e., shortest sequences of edges from $G_{\Sigma,k}$, in which the edge directions are neglected). Inspect Figure 2.

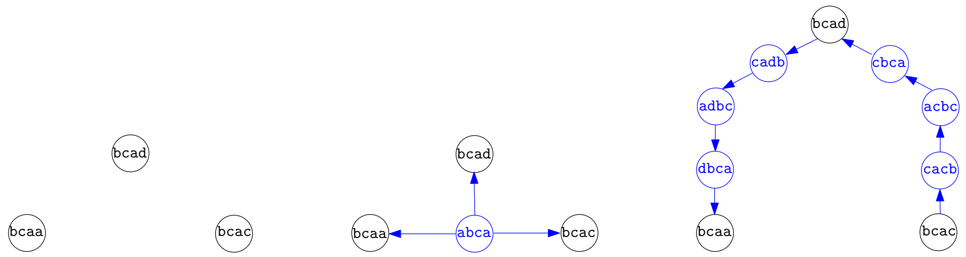


Figure 2 An input dBG of order $k = 4$ with 3 weakly connected components (left); an optimal solution, using shortest undirected paths, with cost 3 (middle); a feasible solution, using shortest directed paths, with cost 8 (right). We color blue the edges and vertices we have added from $G_{\Sigma,k}$.

Bernardini et al. [2] showed that making $G_{\mathcal{S},k}$ weakly connected by adding a set of edges of minimal total cost is NP-hard. They also showed that no polynomial-time approximation scheme (PTAS) exists for making $G_{\mathcal{S},k}$ weakly connected by adding a set of edges of minimal total cost (unless the *unique games conjecture* [7] 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 $G_{\mathcal{S},k}$. In this paper, we introduce *a general framework* for finding shortest undirected paths in dBGs. In particular, by employing our framework, we improve on 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 $C_1, C_2, \ldots, C_d$ of vertices (forming connected components in most applications) from $G_{\Sigma,k}$, and let us denote $V = C_1 \sqcup C_2 \sqcup \cdots \sqcup C_d$.[2] Throughout this whole paper, we treat vertices of dBGs and their length-k string representations as equivalent: indeed, vertices of $G_{\Sigma,k}$ are in a natural bijection with Σ^k. In particular, C_p, for any $p \in [1, d]$, or V are also treated as sets of strings.

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

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

where $\mathsf{dist}(u, v)$ denotes the *length of a shortest undirected path* from vertex u to vertex v.

Note that if C_p and C_q form two weakly connected components of $G_{\Sigma,k}$, then $\mathsf{dist}(p, q)$ is precisely the minimal number of edges that must be added to *connect* the two components into a single one – assuming that we also add the vertices implied by those edges. In the same

[1] By Euler's famous theorem, we know that a weakly connected directed graph is Eulerian if and only if every graph vertex has equal in-degree and out-degree (except perhaps for the source and target).
[2] We assume that these families are pairwise disjoint. However, our solutions work without this assumption.

manner, by adding $d-1$ such undirected paths, we can (greedily) connect $C_1, C_2, \ldots, C_d$ into a single component, thus making any arbitrary dBG $G_{\mathcal{S},k}$ weakly connected.

The problem of finding the shortest *directed* path between any two vertices of $G_{\Sigma,k}$ can be solved in the optimal $\mathcal{O}(k)$ time using the preprocessing of the classic KMP algorithm [8]. The same problem for *undirected* paths can also be solved in the optimal $\mathcal{O}(k)$ time [10]. By iteratively applying the latter result, we can compute $\mathsf{dist}(p,q)$ 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 d of families is relatively small. By using more refined techniques, based on the generalized suffix tree [19] 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 $C_1, C_2, \ldots, C_d$, we consider the following problems:

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

Let us remark that the algorithms underlying our framework are *constructive*: whenever we compute $\mathsf{dist}(p,q)$, we also know the pair $u \in C_p, v \in C_q$ of vertices that are at distance $\mathsf{dist}(p,q)$ – by applying the technique from [10], we can enhance the output of the above algorithms with the optimal paths realising those distances at the additional linear cost in the size of the output – k times the length of the shortest path (every vertex is explicitly encoded using k letters); alternatively we can pay only the length of the shortest path if the output is given in a compacted form: the difference between the next two vertices on the path in the form of the new letter introduced and whether it is put in the front or back.

Our Results. We make the following specific contributions:

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

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

Paper Organization. In Section 2, we present some preliminaries essentially summarizing the work in [10]. 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 vertices 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 [2].

2 Preliminaries

Let $[k]$ denote the set $\{0, 1, \ldots, k-1\}$, and let $S[i\,..\,j]$ denote the substring $S[i]S[i+1]\cdots S[j]$ of S. Let $S_1, S_2 \in \Sigma^k$ represent vertices 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 $i \le j$. 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 $k - j - |U|$ new ones in their place. In total this requires $i + j + 2 \cdot (k - j - |U|) = 2k - 2|U| - (j - i)$ operations. It turns out one cannot get a path from v_1 to v_2 of shorter length other then by choosing a *different* common substring or *different* occurrences. This fact is summarized in Lemma 1 by Liu [10]. Inspect Figure 3.

▶ **Lemma 1** ([10]). *Let v_1, v_2 be two vertices of $G_{\Sigma,k}$ and $S_1, S_2 \in \Sigma^k$ be the string representations of v_1, v_2, respectively. Then* $\mathsf{dist}(v_1, v_2) = 2k - \max_{i,j\in[k]} (2 \cdot |U_{i,j}| + |j - i|)$, *where $U_{i,j}$ is the longest common prefix of $S_1[i\,..\,k-1]$ and $S_2[j\,..\,k-1]$.*

$S_1 =$	A	A	A	C	C	C	C	C	C	C	C	C	C	T	C	C	C	C	A	C	T
$S_2 =$	C	T	C	C	C	C	T	C	C	C	C	C	C	C	C	C	C	A	A	A	A
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Figure 3 Consider strings S_1 and S_2 for $k = 21$. The distance from S_1 to S_2 in the directed dBG 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 case of the undirected dBG the distance between the two vertices is $2 \cdot 21 - 2 \cdot 9 - 7 = 17$ witnessed by the common substring $\mathtt{C}^4\mathtt{TC}^4$. Notice that even though $\mathtt{C}^{10}$ 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 shows that *it is not enough* to look at prefixes, suffixes or the longest common substring when computing the distance in the undirected dBG.

Suffix Tree. The classic indexing solution for standard strings is the suffix tree [19]. 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 $\mathcal{F}$, which takes $\mathcal{O}(|\mathcal{F}|)$ space [14]. Each edge in the compacted trie has a label represented as a fragment of a string in $\mathcal{F}$. The *suffix tree* $\mathsf{ST}(S)$ is the compacted trie of the suffixes of string S. Assuming S ends with a unique terminating letter \$, every suffix $S[i\,..\,|S|]$ is represented by a leaf decorated by index i; see Fig. 4 for an example. The suffix tree occupies $\mathcal{O}(|S|)$ space and it can be constructed in $\mathcal{O}(|S|)$ time [19, 5]. It supports pattern matching queries for any pattern of length m in $\mathcal{O}(m + |\mathrm{Occ}|)$ time, where Occ is the set of output occurrences. The suffix tree can also be generalized to a collection $\mathcal{S} = \{S_1, \ldots, S_N\}$ of N strings as the compacted trie of the suffixes of string $S_1\$_1 \cdots S_N\$_N$, where $\$_i \notin \Sigma$, for $i \in [1, N]$, is a unique terminating symbol.

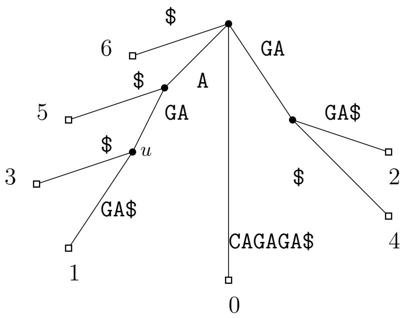


Figure 4 Suffix tree $\mathsf{ST}(S)$ of string $S = \texttt{CAGAGA\$}$. The node spelling string `AG` is *implicit* and thus dissolved in this compacted trie; the node u spelling string `AGA` is *explicit* and thus stored. In particular, the root node, the branching nodes, and the leaf nodes form the set of explicit nodes.

3 Simple Pairwise Distance Computation using Suffix Tree

Let $I_1(U) = \{i : S_1[i \mathinner{.\,.} i+|U|) = U\}$ and let $I_2(U) = \{j : S_2[j \mathinner{.\,.} j+|U|) = U\}$. The distance $\mathsf{dist}(v_1, v_2)$ as described in Lemma 1 can also be expressed as follows: [3]

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

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

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

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

▶ **Observation 2.** *$L_v^x = \min_{w \in \mathit{Children}(v)} L_w^x$ for a branching node v. For a leaf v the set from which L_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 $\mathsf{dist}(u_1, u_2)$ – after precomputing those values, it suffices to find the optimal value 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.

▶ **Lemma 3.** *Let v_1, v_2 be two vertices of $G_{\Sigma,k}$ and $S_1, S_2 \in \Sigma^k$ be the string representations of v_1, v_2, respectively. Given $\mathsf{ST}(\{S_1, S_2\})$, we can compute $\mathsf{dist}(v_1, v_2)$ in $\mathcal{O}(k)$ time.*

Let us remark that a different, yet much more complicated, $\mathcal{O}(k)$-time algorithm for computing $\mathsf{dist}(v_1, v_2)$ using suffix trees has already been described in [10].

Recall that $\mathsf{dist}(p, q) = \min_{v_1 \in C_p, v_2 \in C_q} \mathsf{dist}(v_1, v_2)$. A naïve application of Lemma 3 for computing $\mathsf{dist}(p, q)$ by explicitly computing the pairwise distance between all the pairs of vertices runs in $\mathcal{O}(k|C_p| \cdot |C_q|)$ time. In the next section, we present our framework: how distances between sets of vertices 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 generalised 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 $C_1, C_2, \ldots, C_d$, p and q, and we are asked to find the length of the shortest undirected path between any vertex of C_p and any vertex 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 get that $\mathsf{dist}(p, q)$ can be expressed by Equation (1) if we naturally extend the set $I_x(U) = \{i : \exists_{S_x \in C_x} S_x[i \mathinner{..} i + |U|) = U\}$, for $x \in \{p, q\}$. In particular, for Equation (2), the required change is the use of the generalised suffix tree of the strings representing $C_p \cup C_q$ and the use of L^x_v, R^x_v, for $x \in \{p, q\}$, based on the suffixes of all the strings representing C_x. Inspect Figure 5.

Since the size of $\mathsf{ST}(C_p \cup C_q)$ 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 $\mathsf{dist}(v_1, v_2)$, 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 $C_1, C_2, \ldots, C_d$ and p, and we are asked to find the length of the shortest undirected path between any vertex of C_p and any vertex of C_q, for every $q \in [1, d]$. By 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, and which is later used for solving Top(r). Our high-level goal is to reduce the number of vertices over which we must optimize Equation (2).

Let $A(v)$ denote the set of all ancestors of v in $\mathsf{ST}(V)$ (including the node itself). Further let $\mathsf{Min}^x_v = \max_{w \in A(v)}[2 \cdot d(w) - L^x_w]$, and $\mathsf{Max}^x_v = \max_{w \in A(v)}[2 \cdot d(w) + R^x_w]$, for $x \in \{p, q\}$ and $q \in [1, d]$. Recall that $V = C_1 \sqcup C_2 \sqcup \cdots \sqcup C_d$. Recall that as noted in Observation 2 for the leaf nodes of $\mathsf{ST}(V)$, denoted by $\mathsf{Leaves}(\mathsf{ST}(V))$, the sets $I_x(U)$ are either equal to $\{i^x_v\}$ (when $S[i^x_v \mathinner{..} k-1] = U$ for $S \in C_x$ is represented by node v) or empty. [4] In particular

[4] Note that for a leaf v, $d(v) = k - i^x_v$; 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 $\mathsf{dist}(p, p)$ anyway.

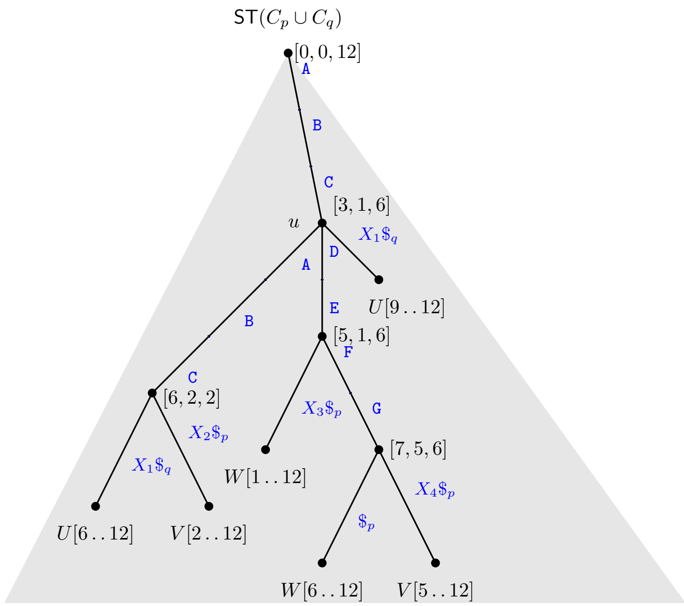


Figure 5 The information $[d, L^p_v, R^p_v]$ computed for the explicit non-leaf nodes v of $\mathsf{ST}(C_p \cup C_q)$ restricted to the part of the suffix tree where the first edge going out of the root is A. Let $U \in C_q$, $V, W \in C_p$, and $k = 13$, for $U =$ `CBDCCCABCABCE`, $V =$ `CDABCABCDEFGB`, and $W =$ `BABCDEABCDEFG`. The labels on the edges leading to leaves are compacted for simplicity (and represented only with suffixes $X_1, \ldots, X_4$). By additionally computing $L^q_u = 6$ and $R^q_u = 9$, 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 then comparing the distances witnessed by all the common substrings (nodes of the ST), we obtain the minimal distance 10 witnessed by `ABCABC`.

i^x_v exists only for a single x. We can thus define Equation (3) as a more refined version of Equation (2):

$$\mathsf{dist}(p, q) = 2k - \max_{v \in \mathsf{Leaves}(\mathsf{ST}(V))} \left(\max\left[\mathsf{Max}^p_v - i^q_v, \mathsf{Min}^p_v + i^q_v\right]\right). \tag{3}$$

▶ **Example 5.** Consider the example from Figure 5: $U =$ `CBDCCCABCABCE` $\in C_q$, $V =$ `CDABCABCDEFGB` $\in C_p$, and $W =$ `BABCDEABCDEFG` $\in C_p$. Consider the leaf nodes representing strings $V[2..12]$ and $U[6..12]$. Both leaf nodes have the following ancestors (other than themselves) and the following $[d, L^p_v, R^p_v, L^q_v, R^q_v]$ 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 $U[6..12]$. We iterate over all ancestors w of v:

- $\mathsf{Min}^p_v = \max_{w \in A(v)}[2 \cdot d(w) - L^p_w] = \max\{-\infty, 10, 5, 0\} = 10$;
- $\mathsf{Max}^p_v = \max_{w \in A(v)}[2 \cdot d(w) + R^p_w] = \max\{-\infty, 14, 12, 12\} = 14$.

We have $2k - \max\left[\mathsf{Max}^p_v - i^q_v, \mathsf{Min}^p_v + i^q_v\right] = 26 - (10 + 6) = 10$, 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* (2) *and* (3) *are equivalent.*

Proof. Let v be the node of ST for which Equation (2) 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 $\mathsf{Min}_w^p \geq 2 \cdot d(v) - i_u^p$, hence $\mathsf{Min}_w^p + i_w^q \geq 2 \cdot d(v) - i_u^p + i_w^q = 2 \cdot d(v) - L_v^p + R_v^q$, thus the value of Equation (3) (witnessed by the node w) is at least as large as the value of Equation (2) (witnessed by the node v).

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

◀

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

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 (3) 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 solving 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 $\mathsf{Top}(r)$, we are given $C_1, C_2, \ldots, C_d$ 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 $\mathsf{dist}(p, q)$, breaking ties arbitrarily.

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

Recall that 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 $\mathsf{ST}(V)$. If we store the best 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 out 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 the explicit nodes of $\mathsf{ST}(V)$. Here 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 non-increasingly due to the sorted order on L_w^p (resp. R_w^p).

Finally, we once again simply iterate over the leaves of $\mathsf{ST}(V)$, and gather the r smallest values of $\mathsf{dist}(p,q)$ over all the leaves representing a suffix of a string from C_q (with a use of a bucket queue) obtaining:

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

5 Application: Connecting de Bruijn Graphs

We anticipate that our framework has many applications revolving around dBGs. 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 components and also denote it by $G(V,E)$. Bernardini et al. [2] proved the following result.

▶ **Theorem 10** ([2]). *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 minimal total cost.*

In this section, 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 in [2] consists of the following three steps:

(i) Construct the metric closure of G' – we do not explicitly construct G' or $G_{\Sigma,k}$.
(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 [9],[5] 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 [16]. Finally, Step (iii) can be done by applying again Lemma 3 to compute the shortest undirected paths. This requires $\mathcal{O}(k|V|^2)$ 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 minimal 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. Notably, with the use of the Top queries, we can obtain an even more efficient solution by dropping the construction of the metric closure and instead computing its spanning tree directly from our input.

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 vertex $v \in \mathcal{V}$, an edge incident with v with the smallest weight (one of such edges in case of ties), and then by removing, from each cycle obtained this way, the heaviest edge (breaking ties arbitrarily). We show the following lemma.

▶ **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 the edges from E_{Top}, and has total weight at most as large as the weight of the initial spanning tree.

[5] Recall that the minimum Steiner tree problem asks, given a graph $G'(V',E')$ with non-negative edge weights and a subset of *terminal* nodes, to compute a tree of minimum weight that contains all terminals.

We iterate over the edges from 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.

Hence we next assume that after adding an edge $e_1 \in E_{\mathrm{Top}}$, we create a cycle in which *all* the heaviest edges belong to E_{Top}. Take one such edge – it is associated with a vertex v. The other edge from 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 vertex associated with this edge, and continue the same way. Since the graph 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.

▶ **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 from producing a set of edges E_{Top} for G' with a use of a $\mathsf{Top}(r)$ 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 sum 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 [2] we obtain the following improved result.

▶ **Theorem 13.** *For any order-k dBG $G(V, E)$ 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 minimal 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.
2. 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.
3. 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.
4. Massimo Cairo, Shahbaz Khan, Romeo Rizzi, Sebastian Schmidt, Alexandru I. Tomescu, and Elia C. Zirondelli. The hydrostructure: a universal framework for safe and complete algorithms for genome assembly, 2021.
5. 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.
6. John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980.
7. 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.
8. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
9. Lawrence T. Kou, George Markowsky, and Leonard Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, 1981.
10. 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.
11. 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.
12. Paul Medvedev and Mihai Pop. What do Eulerian and Hamiltonian cycles have to do with genome assembly? *PLOS Computational Biology*, 17(5):1–5, 05 2021.
13. Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.
14. Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
15. 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.
16. Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.
17. 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.
18. Alexandru I. Tomescu and Paul Medvedev. Safe and complete contig assembly through omnitigs. *J. Comput. Biol.*, 24(6):590–602, 2017.
19. 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.