Making de Bruijn Graphs Eulerian

Bernardini, Giulia; Chen, Huiping; Loukides, Grigorios; Pissis, Solon P.; Stougie, Leen; Sweering, Michelle

DOI:
10.4230/LIPIcs.CPM.2022.12

Making de Bruijn Graphs Eulerian

Giulia Bernardini
University of Trieste, Italy
CWI, Amsterdam, The Netherlands

Huiping Chen
King's College London, UK

Grigorios Loukides
King's College London, UK

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Leen Stougie
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Michelle Sweering
CWI, Amsterdam, The Netherlands

Abstract

A directed multigraph is called *Eulerian* if it has a circuit which uses each edge exactly once. Euler's theorem tells us that a weakly *connected* directed multigraph is Eulerian if and only if every node is *balanced*. Given a collection S of strings over an alphabet Σ, the *de Bruijn graph* (dBG) of order k of S is a directed multigraph $G_{S,k}(V, E)$, where V is the set of length-$(k-1)$ substrings of the strings in S, and $G_{S,k}$ contains an edge (u, v) with multiplicity $m_{u,v}$, if and only if the string $u[0] \cdot v$ is equal to the string $u \cdot v[k-2]$ and this string occurs exactly $m_{u,v}$ times in total in strings in S. Let $G_{\Sigma,k}(V_{\Sigma,k}, E_{\Sigma,k})$ be the complete dBG of Σ^k. The Eulerian Extension (EE) problem on $G_{S,k}$ asks to extend $G_{S,k}$ with a set $\mathcal{B}$ of nodes from $V_{\Sigma,k}$ and a smallest multiset $\mathcal{A}$ of edges from $E_{\Sigma,k}$ to make it Eulerian. Note that extending dBGs is algorithmically much more challenging than extending general directed multigraphs because some edges in dBGs are by definition *forbidden*. Extending dBGs lies at the heart of sequence assembly [Medvedev et al., WABI 2007], one of the most important tasks in bioinformatics. The novelty of our work with respect to existing works is that we allow not only to duplicate existing edges of $G_{S,k}$ but to also add novel edges and nodes, in an effort to (i) connect multiple components and (ii) reduce the total EE cost. It is easy to show that EE on $G_{S,k}$ is NP-hard via a reduction from shortest common superstring. We further show that EE remains NP-hard, even when we are not allowed to add new nodes, via a highly non-trivial reduction from 3-SAT. We thus investigate the following two problems underlying EE in dBGs:

1. When $G_{S,k}$ is not weakly connected, we are asked to connect its $d > 1$ components using a minimum-weight spanning tree, whose edges are paths on the underlying $G_{\Sigma,k}$ and weights are the corresponding path lengths. This way of connecting guarantees that no new unbalanced node is added. We show that this problem can be solved in $\mathcal{O}(|V|k \log d + |E|)$ time, which is nearly optimal, since the size of $G_{S,k}$ is $\Theta(|V|k + |E|)$.
2. When $G_{S,k}$ is not balanced, we are asked to extend $G_{S,k}$ to $H_{S,k}(V \cup \mathcal{B}, E \cup \mathcal{A})$ such that every node of $H_{S,k}$ is balanced and the total number $|\mathcal{A}|$ of added edges is minimized. We show that this problem can be solved in the optimal $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$ time.

Let us stress that, although our main contributions are theoretical, the algorithms we design for the above two problems are practical. We combine the two algorithms in one method that makes any dBG Eulerian; and show experimentally that the cost of the obtained feasible solutions on real-world dBGs is substantially smaller than the corresponding cost obtained by existing greedy approaches.

2012 ACM Subject Classification Theory of computation → Pattern matching

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).
Editors: Hideo Bannai and Jan Holub; Article No. 12; pp. 12:1–12:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.12

Supplementary Material *Software (Source Code)*: `https://bitbucket.org/eulerian-ext/cpm2022/` archived at `swh:1:dir:d7c2ca6a257600d6d7176b876370c456496af5a2`

Funding The work in this paper is supported in part by: the Netherlands Organisation for Scientific Research (NWO) through project OCENW.GROOT.2019.015 "Optimization for and with Machine Learning (OPTIMAL)" and Gravitation-grant NETWORKS-024.002.003; a CSC scholarship; the Leverhulme Trust RPG-2019-399 project; and 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.

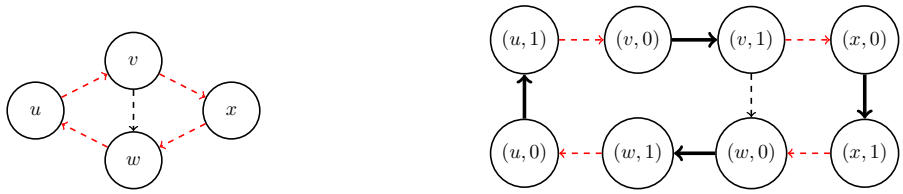
1 Introduction

We start with some basic definitions and notation on strings from [6]. Let $x = x[0] \cdots x[n-1]$ be a *string* of length $n = |x|$ over an integer alphabet $\Sigma = [0, \sigma)$ of σ *letters*. By Σ^k we denote the set of all strings of length $k > 0$. For any two positions i and $j \geq i$ of x, $x[i\,..\,j]$ is the *fragment* of x starting at position i and ending at position j; it is represented in $\mathcal{O}(1)$ space by i and 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 fragment of the form $x[0\,..\,j]$ and a *suffix* of x is a fragment of the form $x[i\,..\,n-1]$. By xy or $x \cdot y$ we denote the *concatenation* of strings x and y: $xy = x[0] \cdots x[|x|-1]y[0] \cdots y[|y|-1]$. Given strings x and y, a *suffix/prefix overlap* of x and y is a suffix of x that is a prefix of y.

The order-k de Bruijn graph (dBG) of a collection S of strings is a directed multigraph $G_{S,k}(V, E)$ such that V is the set of length-$(k-1)$ substrings of the strings in S and $G_{S,k}$ contains an edge (u, v) with multiplicity $m_{u,v}$, if and only if the string $u[0] \cdot v$ is equal to the string $u \cdot v[k-2]$ and this string occurs exactly $m_{u,v}$ times in total in strings in S. When S is generated by a sequencing experiment from a genome, any Eulerian circuit of $G_{S,k}(V, E)$ corresponds to a single genome reconstruction [24, 20]. It goes without saying that genome assembly is one of the most important bioinformatics tasks [25, 29, 11, 22, 21, 26, 27, 18].

However, $G_{S,k}$ is almost surely not Eulerian in practice due to sequencing errors [21]. One could thus try to make it Eulerian by duplicating some of its *existing* edges [19]. In this case, one would naturally like to minimize the total cost of this extension. Even worse, $G_{S,k}$ would likely not be weakly connected, and thus edge duplication is not sufficient to make $G_{S,k}$ Eulerian. In this paper, we introduce the problem of making any arbitrary $G_{S,k}$ Eulerian by allowing not only to duplicate existing edges but to also add novel edges and nodes. The motivation for this is twofold. First, such a process would connect multiple components, which are often unconnected for the values of k used in practice. Second, as this is a generalization of the edge duplication problem [19], it would only reduce the total extension cost, even if the input graph is already weakly connected.

Let us now more formally lay the foundations of our work by first considering a general directed multigraph $G(V, E)$. A directed multigraph is called *Eulerian* if it has a circuit which uses each edge exactly once. Euler's theorem tells us that a weakly *connected* directed multigraph is Eulerian if and only if every node is *balanced*: for any node $v \in V$ the in- and out-degree of v are equal. The *Eulerian Extension* (EE) problem on $G(V, E)$ asks for an Eulerian extension minimizing the total cost of the multiset $\mathcal{A}$ of added edges according to some cost function. A smallest multiset $\mathcal{A}$ over $V \times V$ such that $H(V, E \cup \mathcal{A})$ is Eulerian can be computed in the optimal $\mathcal{O}(|V| + |E|)$ time [5, 9]. We prove that the EE problem becomes significantly more challenging when a subset F of $V \times V$ is *forbidden* (i.e., not feasible):



(a) $\mathcal{H}(V', E')$. **(b)** $G(V' \times \{0,1\}, \{((v,0),(v,1)) | v \in V'\})$.

Figure 1 (a) An instance of the directed Hamiltonian circuit with a solution in red and (b) the instance of Problem 1 to which it reduces. An Eulerian circuit in graph (b), with required bold edges and non-forbidden dashed edges, corresponds to finding a directed Hamiltonian circuit in graph (a). The corresponding solution is in red in graph (b).

▶ **Problem 1.** *Given a directed multigraph $G(V, E)$ and a set $F \subset V \times V$, with $F \cap E = \emptyset$, find a multiset $\mathcal{A}$ of edges over $(V \times V) \setminus F$ such that $H(V, E \cup \mathcal{A})$ is Eulerian and $|\mathcal{A}|$ is minimized; or report* FAIL *if not possible.*

It should be clear that Problem 1 is equivalent to the EE problem when $F = \emptyset$. Note that Problem 1 is directly applicable on arbitrary dBGs, where the set F of forbidden edges is directly implied by the dBG definition: $(u, v) \in F$ if and only if $u[0] \cdot v \neq u \cdot v[k-2]$. We observe that adding nodes may shorten the length of the Eulerian circuit with respect to Problem 1. This observation leads naturally to the following generalization of Problem 1:

▶ **Problem 2.** *Given a directed multigraph $G(V, E)$, a set $\mathcal{V} \supseteq V$, and a set $F \subset \mathcal{V} \times \mathcal{V}$, with $F \cap E = \emptyset$, find a multiset $\mathcal{A}$ of edges over $(\mathcal{V} \times \mathcal{V}) \setminus F$ and a set of nodes $\mathcal{B} \subseteq \mathcal{V}$ such that $H(V \cup \mathcal{B}, E \cup \mathcal{A})$ is Eulerian and $|\mathcal{A}|$ is minimized; or report* FAIL *if not possible.*

We will now prove that both Problems 1 and 2 are NP-hard by reducing from the directed Hamiltonian circuit [13] problem: Given a directed graph, decide whether there exists a directed circuit that visits every node of the graph exactly once.

▶ **Theorem 1.** *Both Problems 1 and 2 are NP-hard.*

Proof. We will first prove that Problem 1 is NP-hard. Consider an instance $\mathcal{H}(V', E')$ of the directed Hamiltonian circuit problem (inspect Figure 1a). We replace each node $v \in V'$ by two nodes $(v, 0), (v, 1)$ and an edge $((v, 0), (v, 1))$ with all the incoming edges incident to the tail $(v, 0)$ and all the outgoing edges incident to the head $(v, 1)$ (inspect Figure 1b).

Note that any sequence of adjacent edges on this modified graph alternates between new edges (corresponding to nodes in $\mathcal{H}$, the bold edges in Figure 1b) and old edges (corresponding to the edges in $\mathcal{H}$ connecting those nodes, dashed in Figure 1b). Finding a Hamiltonian circuit in $\mathcal{H}$ is equivalent to finding a circuit passing through all new edges in the modified graph exactly once. It follows that solving the directed Hamiltonian circuit problem on $\mathcal{H}$ is equivalent to deciding whether the following instance of Problem 1 has a solution of size $|V'|$ (smaller solutions are not possible, larger solutions imply that $\mathcal{H}$ is not Hamiltonian):

$$G(V, E) = G\left(V' \times \{0,1\}, \{((v,0),(v,1)) \mid v \in V'\}\right)$$
$$F = (V \times V) \setminus \left(E \cup \{((u,1),(v,0)) | (u,v) \in E'\}\right).$$

Since the directed Hamiltonian circuit problem is NP-complete [13], it is NP-hard to decide whether a solution to Problem 1 has size at most $|V|/2$. Thus solving Problem 1 is NP-hard.

Note that Problem 2 is equivalent to the EE problem when $F = \emptyset$ and $\mathcal{V} = V$, and that Problem 2 is at least as hard as Problem 1: every instance of Problem 1 can be reduced to some instance of Problem 2 with $\mathcal{V} = V$. Therefore Problem 2 is NP-hard as well. ◀

Related Work. Both Problems 1 and 2 *on general graphs* are closely-related to the *Directed Rural Postman* problem (DRP) [23]: Given a directed (multi-)graph $\mathcal{G}(V', E')$ and a (multi-) set $\mathcal{R} \subseteq E'$ of *required* edges, we are asked to compute a minimum-cost circuit in $\mathcal{G}$ including all edges in $\mathcal{R}$. It is easy to see that any instance of Problem 2 reduces to an instance of DRP with $V' = \mathcal{V}$, $E' = (\mathcal{V} \times \mathcal{V}) \setminus F$ and $\mathcal{R} = E$; a similar reduction works for Problem 1.

Problems 1 and 2 *on arbitrary dBGs* are different versions of the classic Shortest Common Superstring (SCS) problem [12]. In particular, Problem 2 is closely-related to the Multi-SCS problem [5]: Given a set S of strings and a multiplicity $f(s_i)$ of each $s_i \in S$, Multi-SCS asks for a shortest string containing at least $f(s_i)$ occurrences of each $s_i \in S$. When all strings in S are of length k, Multi-SCS is essentially Problem 2 on dBGs of order k.[1] Crochemore et al. showed that Multi-SCS can be solved in linear time when all input strings in S are of length 2. Cazaux and Rivals [4] presented a $\frac{1}{2}$-approximation algorithm for Multi-SCS that maximizes the compression offered by the output string; and a 4-approximation algorithm for Multi-SCS that minimizes the length of the output string. In the Multi-SCCS problem [4], given a set S of strings and a multiplicity $f(s_i)$ of each $s_i \in S$, we are asked to construct a multiset C of cyclic strings of minimum total length such that every string in S occurs $f(s_i)$ times in the strings of C. Cazaux and Rivals [4] showed a linear-time implementation of a greedy algorithm that solves Multi-SCCS exactly (see also [3] for the SCCS problem).

Contributions. Let us now summarize our main contributions on arbitrary dBGs:

1. The reduction leading to Theorem 1 *does not* apply to the case in which the input to Problems 1 or 2 is an arbitrary dBG $G_{S,k}$, as not all edges implied by the reduction may be feasible in $G_{S,k}$. In Section 3 we prove that both problems are NP-hard even on arbitrary dBGs. It is easy to show that Problem 2 is NP-hard via a reduction from SCS. For Problem 1, we make a highly non-trivial reduction from 3-SAT; this is the most involved part of the paper. Since our ultimate goal is to make dBGs Eulerian, we next investigate the following two problems underlying EE in dBGs: *connect* and *balance*.
2. In Section 4, we show an *exact* greedy algorithm to make any $G_{S,k}$, consisting of $d > 1$ weakly connected components, weakly connected, by extending $G_{S,k}$ with a minimum-weight spanning tree, whose edges are paths on the underlying $G_{\Sigma,k}$ and weights are the corresponding path lengths. While there are many optimization criteria for connecting $G_{S,k}$, this way guarantees that no new unbalanced node is added. Our algorithm runs in $\mathcal{O}(|V|k \log d + |E|)$ time, which is *nearly optimal*, since the size of $G_{S,k}$ is $\Theta(|V|k + |E|)$. To achieve this time complexity, we simulate Kruskal's classic algorithm for computing minimum spanning trees [17] using an efficient method to compute shortest paths on the implicit $G_{\Sigma,k}$. This method employs an augmented and modified version of the Aho-Corasick machine [1], which we dynamically update every time we unite two components.
3. Balancing any $G_{S,k}$ with the smallest number of newly added edges can be reduced to Multi-SCCS. By employing the linear-time algorithm of Cazaux and Rivals [4] for Multi-SCCS, we obtain an $\mathcal{O}(k|E|)$-time algorithm for balancing. In Section 5, we show an *exact* greedy algorithm for this problem that runs in the *optimal* $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$ time, where $|\mathcal{A}|$ is the total number of added edges. To achieve this time complexity, similar to Section 4, we simulate Cazaux and Rivals algorithm using another augmented and modified version of the Aho-Corasick machine.

[1] We say "essentially" because Multi-SCS asks for a shortest linear string, whereas Problem 2 asks for an Eulerian circuit, which on a dBG corresponds to a shortest cyclic string.

4. Although our main contributions here are theoretical, the algorithms we design are *practical*. In Section 6, we combine the algorithms of Sections 4 and 5 in one method that makes any $G_{S,k}$ Eulerian; and show experimentally that the cost of the feasible solutions obtained by this method on real-world dBGs constructed over sequencing data is substantially smaller than the cost of solutions obtained by existing string-based greedy approaches. This justifies the need for an approach specifically designed to extend dBGs.

2 Preliminaries

We fix an integer $k > 1$ and an integer alphabet Σ. Given a collection S of strings over Σ, we denote by $G_{S,k}(V, E)$ the de Bruijn graph (dBG) of order k of S (defined in Section 1). The cardinality of E (i.e., the sum of edge multiplicities) is $|E| = ||S|| - (k-1)|S|$, where $||S||$ is the total length of the strings in S. Let $d^-(u)$ and $d^+(u)$ be, respectively, the in- and out-degree of node u of $G_{S,k}$. An undirected graph is said to be *connected* if for every pair u and v of nodes in the graph there exists a path from u to v. A directed graph is called *weakly connected* if by replacing all of its directed edges with undirected edges we obtain a connected (undirected) graph. A *spanning tree* of a weakly connected graph is a weakly connected subgraph which covers all the nodes of the graph with the minimum possible number of edges. A weakly connected graph $G_{S,k}$ is called *Eulerian* if every node u in $G_{S,k}$ is balanced, i.e., $d^+(u) = d^-(u)$. The dBG of order k of Σ^k is called the *complete de Bruijn graph* of order k over Σ; we denote it by $G_{\Sigma,k}(V_{\Sigma,k}, E_{\Sigma,k})$, where $V_{\Sigma,k} = \Sigma^{k-1}$ and $E_{\Sigma,k} = \{(s[0\,..\,k-2], s[1\,..\,k-1]) \mid s \in \Sigma^k\}$.

Throughout, we assume that we are given the graph $G_{S,k}$ of an arbitrary string collection S, which we denote by $G(V, E)$.

3 Eulerian Extension of de Bruijn Graphs is NP-hard

In this section, we investigate the hardness of Problems 1 and 2 on arbitrary dBGs.

Eulerian Extension of de Bruijn Graphs (Extend-DBG)
Input: A de Bruijn graph $G(V, E)$ of order k over alphabet Σ.
Output: An Eulerian graph $H(V \cup \mathcal{B}, E \cup \mathcal{A})$ with $\mathcal{B} \subseteq V_{\Sigma,k}$, $\mathcal{A}$ over $E_{\Sigma,k}$ and minimized $|\mathcal{A}|$.

Extend-DBG can be solved in linear time when $k = 2$ [5]. When $k > 2$, Extend-DBG can be shown to be NP-hard via a simple reduction from the Length-k Shortest Common Superstring problem (k-SCS), a special case of the SCS problem in which all input strings are of length k. k-SCS is NP-hard, for any $k > 2$ [12]. Any instance of k-SCS on some alphabet Σ can be reduced to an instance of Extend-DBG on a dBG of order k over $\Sigma \cup \{\#\}$, with $\# \notin \Sigma$. The nodes of such dBG are the length-$(k-1)$ prefixes and suffixes of each input string of k-SCS plus a special node $\#^{k-1}$. All the edges naturally correspond to the input strings of k-SCS, except for a special edge encoding $\#^k$. An Eulerian circuit of a minimum-size Eulerian extension of such graph then corresponds to a shortest common cyclic superstring $\tilde{s}$, which can be trivially transformed into a solution s to k-SCS (a shortest common linear superstring) by removing substring $\#^k$, so that the first letter of s is the first letter of $\tilde{s}$ after the last $\#$, and the last letter of s is the last letter of $\tilde{s}$ before the first $\#$.

Since a common superstring always exists (any concatenation of the strings is a cyclic superstring), the reduction implicitly assumes that it is always possible to connect a dBG to make it Eulerian. While this is true for Extend-DBG, as a path of length at most $k-1$

exists between any two nodes if new nodes can be added to the graph, the assumption is wrong if we are *only* allowed to connect pairs of nodes of the input graph. If we tried to solve k-SCS via Extend-DBG with this restriction, we would only consider suffix/prefix overlaps of length $(k-1)$ (corresponding to two consecutive edges of the dBG given by the reduction) or $(k-2)$ (corresponding to edges added between two existing nodes when solving Extend-DBG), which is clearly wrong. It is therefore interesting to see if Extend-DBG remains NP-hard even with this restriction.

We start by formally defining this restricted version of the EE problem on dBGs.

Restricted Eulerian Extension of de Bruijn Graphs (R-Extend-DBG)
Input: A de Bruijn graph $G(V,E)$ of order k over alphabet Σ.
Output: An Eulerian graph $H(V, E \cup \mathcal{A})$ with $\mathcal{A}$ over $(V \times V) \cap E_{\Sigma,k}$ and minimized $\lvert\mathcal{A}\rvert$; or report FAIL if not possible.

R-Extend-DBG can also be solved in linear time when $k = 2$ [5]. However, proving that R-Extend-DBG is NP-hard for $k > 2$ turns out to be significantly more challenging. We prove this via a reduction from 3-SAT, a well-known NP-hard problem [15]. Let $\{x_1, \ldots, x_\ell\}$ be a set of *variables*. A *literal* is a variable x_i or a negated variable $\neg x_i$. A *clause* is a disjunction of literals. A *formula* $F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ is in conjunctive normal form (CNF), if it is a conjunction of n clauses. The k-SAT problem is deciding whether a formula F in CNF form with every clause in F consisting of at most k literals is satisfiable.

▶ **Theorem 2.** *R-Extend-DBG is NP-hard if $G(V,E)$ is of order $k = 3$.*

Proof. Consider a 3-SAT instance with a set $\{x_1, \ldots x_\ell\}$ of ℓ variables and a formula $F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ of n clauses, where each clause contains three literals. We construct a dBG with $k = 3$ for which solving R-Extend-DBG problem tells us whether F is satisfiable or not. The dBG is constructed over the alphabet:

$$\Sigma = \{x_i, \neg x_i, y_i, z_i\}_{i\in[1,\ell]} \cup \{a_j, b_j\}_{j\in[1,n]} \cup \{x_{ij}, \neg x_{ij}\}_{i\in[1,\ell]}^{j\in[1,n]} \cup \{c_1, c_2\},$$

where the letters within each set are pairwise distinct and all sets are pairwise disjoint. The dBG will consist of the union of some gadget subgraphs, as described next. For each variable x_i, we define a variable-gadget $G_v(x_i)(V_v^i, E_v^i)$, conceptually corresponding to $x_i \vee \neg x_i$, with five nodes and four edges:

$$V_v^i = \{z_i x_i, x_i y_i, y_i z_i, z_i \neg x_i, \neg x_i y_i\}, \quad E_v^i = \{x_i y_i z_i, y_i z_i x_i, \neg x_i y_i z_i, y_i z_i \neg x_i\}.$$

For every clause $C_j = l_1^j \vee l_2^j \vee l_3^j$, with $l_1^j, l_2^j, l_3^j \in \{x_i, \neg x_i\}_{i\in[1,\ell]}$, we define a corresponding clause-gadget $G_c(C_j)(V_c^j, E_c^j)$, with seven nodes and six edges:

$$V_c^j = \{a_j b_j, b_j l_{1j}^j, l_{1j}^j a_j, b_j l_{2j}^j, l_{2j}^j a_j, b_j l_{3j}^j, l_{3j}^j a_j\},$$

$$E_c^j = \{a_j b_j l_{1j}^j, a_j b_j l_{2j}^j, a_j b_j l_{3j}^j, l_{1j}^j a_j b_j, l_{2j}^j a_j b_j, l_{3j}^j a_j b_j\}.$$

In this definition, l_{tj}^j for each $t \in \{1,2,3\}$ are just placeholders, such that $l_{tj}^j = x_{ij}$ if $l_t^j = x_i$ and $l_{tj}^j = \neg x_{ij}$ if $l_t^j = \neg x_i$: for example, in Figure 2, $l_{11}^1 = x_{11}$, $l_{21}^1 = x_{21}$, and $l_{31}^1 = \neg x_{31}$ because $C_1 = (x_1 \vee x_2 \vee \neg x_3)$.

Finally, the main component-gadget $G_m(V_m, E_m)$ is daisy-shaped: it has a central node and 2ℓ *petals*, one for each variable x_i and negated variable $\neg x_i$, each consisting of a simple cycle of length $n+3$ beginning and ending at the central node. In the following definition we use l_j for each $j \in [1,n]$ again as placeholders, to be replaced with x_{ij} in the petal of x_i, and with $\neg x_{ij}$ in the petal of $\neg x_i$, for all $i \in [1,\ell]$:

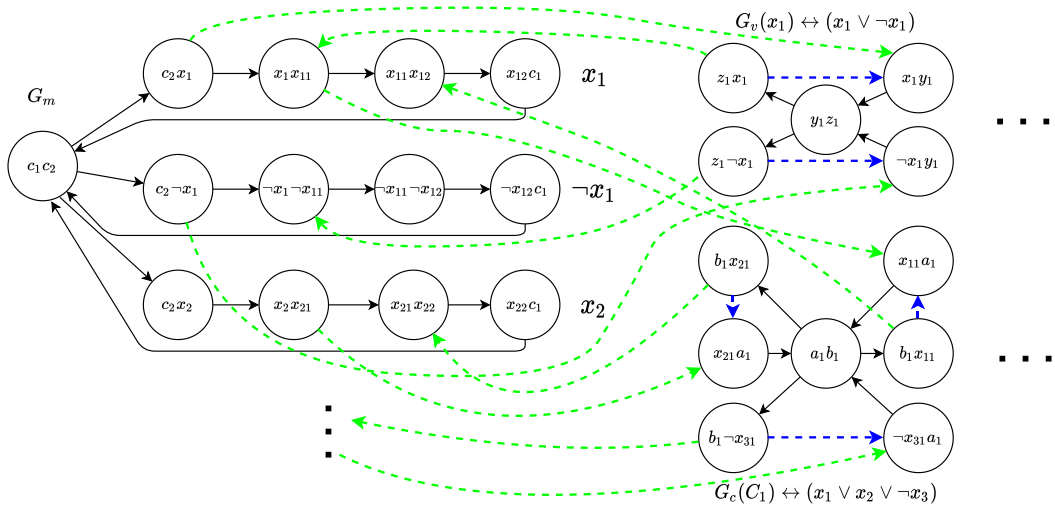


Figure 2 An instance of R-Extend-DBG that is equivalent to the 3-SAT problem $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$. The first half of the G_m component is shown on the left. The edges of the gadgets are shown in black. Other feasible edges within components are shown in blue, while feasible edges between components are shown in green.

$$V_m = \{c_1c_2\} \cup \left\{c_2l, ll_1, l_1l_2, \ldots, l_{n-1}l_n, l_nc_1 \mid l \in \{x_i, \neg x_i\}_{i\in[1,\ell]}\right\}$$

$$E_m = \left\{c_1c_2l, c_2ll_1, ll_1l_2, l_1l_2l_3, \ldots, l_{n-1}l_nc_1, l_nc_1c_2 \mid l \in \{x_i, \neg x_i\}_{i\in[1,\ell]}\right\}.$$

Note that each of these gadgets is connected and they are all mutually disjoint. An example is shown in Figure 2. To balance the nodes, we need to add at least 2 edges for each variable-gadget and at least 3 edges for each clause-gadget. For example, adding the blue edges in Figure 2 would balance the graph. However, to additionally make the graph connected, we would need to add other additional edges.

We want to minimize the number of added edges to make the graph balanced and connected. That is equivalent to minimizing the number of nodes visited with multiplicity. We will prove that we need at least $3\ell(n+3) + 6\ell + 9n$ edges and that this is sufficient if and only if the formula F is satisfiable. Consider the cut separating all gadgets G_v from the other components. A gadget $G_v(x_i)$ can only be reached from nodes c_2x_i and $c_2\neg x_i$ of G_m (see the green edges in Figure 2). Therefore, there must be at least ℓ additional edges leaving nodes of the form c_2l of G_m, one for each $G_v(x_i)$, with $l \in \{x_i, \neg x_i\}$. Such nodes are then visited at least 3ℓ times in total, as all the 2ℓ of them must be visited at least once by following the solid edges of G_m, and ℓ of them (one for each $G_v(x_i)$) must be visited at least once more. Since the graph must be balanced, the number of edges traversing the cut reaching the G_v gadgets equals the number of edges leaving them. Thus, the nodes of G_m of the form ll_j, which are the only ones reachable from the G_v gadgets, are also visited at least 3ℓ times. Moreover, in order for G_m to remain balanced, the nodes of the form $l_{j-1}l_j$ and l_jl_{j+1} (with $l_0 = l, l_{n+1} = c_1$) on the petals on which nodes of the form c_2l or ll_j are visited twice must be visited at least twice too. It follows that there are at least $3\ell(n+3)$ visits of the nodes in G_m (each of the $n+3$ nodes on each of the 2ℓ petals are visited at least once; and the nodes of at least ℓ petals must be visited twice), while the number of visits in the G_v components is at least 6ℓ and the number of visits in the G_c components is at least $9n$ (in order for them to be balanced), yielding the desired *lower bound*.

For this bound to be tight, there cannot be any extra visits to nodes in G_v and G_c gadgets. Hence, for the graph to be balanced, we need the number of visits to $c_2l, ll_1, l_1l_2, \ldots, l_nc_1$ to be equal for each fixed literal l. It follows that, in order to remain within $3\ell(n+3)$ visits to nodes of G_m, for exactly one of the literals x_i and $\neg x_i$ these nodes are visited twice, while for the other literal these nodes are visited only once. Note that we can only connect to $G_c(C_j)$ if for one of its literals l node $l_{j-1}l_j$ of G_m is visited twice. Therefore we can only connect the graph with the lower bound number of nodes if F is satisfiable.

We will now show that if F is satisfiable then this number is enough. We balance each G_v gadget with 2 edges and each G_c with 3 edges (blue edges in Figure 2). We also add the length-$(n+3)$ cycles in G_m corresponding to the ℓ true literals. We will show that we can connect each of the G_v and each of the G_c gadgets to G_m. That we can do so without increasing the number of edges results from the following claim.

▷ Claim. Let X and Y be two distinct balanced connected components with non-required (i.e., appearing with a higher multiplicity than in the original graph) edges azb and czd in X and Y, respectively. Then there exists a connected balanced graph on the nodes of X and Y with the same number of edges.

Proof. Since azb and czd are non-required, we can remove them. Note that both X and Y are still connected: since az and zb (resp. cz and zd) are the only unbalanced nodes in X (resp. Y), they must lie in the same component. Now add azd and czb. These edges are feasible because all endpoints are already present in the graph. This rebalances the graph and connects the two components. ◁

Since either x_i or $\neg x_i$ is true, one of them is the mid symbol of a non-required edge in G_m, thus we can link all G_v gadgets to G_m using the interchange of edges described in the proof of the claim (in Figure 2, we trade the blue edge in $G_v(x_i)$ and one copy of (c_2x_i, x_ix_{i1}) for the green edges, or do it for $\neg x_1$). Moreover, since we assumed that F is satisfiable, at least one literal l of each C_j is true. The edges in $G_c(C_j)$ and G_m with l in the middle are non-required, thus we can link all G_c gadgets to G_m, making the graph connected. ◀

4 Connecting de Bruijn Graphs with Paths in Near-Optimal Time

We present an exact $\mathcal{O}(|V|k\log d + |E|)$-time algorithm for connecting any dBG $G(V,E)$ of order k by arranging its $d > 1$ weakly connected components in a tree. The tree nodes are the components themselves and the tree edges are paths of minimum total length between such components. Since our ultimate goal is to both connect and balance G, by connecting G in this way, we make sure that the new nodes we add are already *balanced*.[2] To formally define the connecting problem we consider, we first need the following definition.

▶ **Definition 3** (Condensed Graph). *Given a dBG $G(V,E)$ of order k over an alphabet Σ with a set $\mathcal{C}$ of weakly connected components, its* condensed graph *$\widehat{G}(\widehat{V},\widehat{E})$ is a weighted directed multigraph whose nodes $\widehat{V}$ are in a bijection with $\mathcal{C}$. The edges have integer weights in $[1,k-1]$: there is an edge $(i,j) \in \widehat{E}$ for each pair of nodes $u_i \in C_i$, $u_j \in C_j$, with $C_i, C_j \in \mathcal{C}$, and its weight is the length of a shortest path from u_i to u_j in the complete dBG $G_{\Sigma,k}$.*

We now formally define the problem we consider in this section.

[2] Note that the graph resulting from this algorithm would, in general, not be balanced.

Algorithm 1 Connecting a de Bruijn Graph with Paths.

1: Find the d connected components of G, construct, and preprocess the AC machine of the nodes of G
2: **for** $i \in [1, d-1]$ **do**
3: Select a backward edge (s, u) encoding a longest suffix/prefix overlap
4: $(s_\alpha, s_\beta) \leftarrow \texttt{components}(s, u)$
5: Add to $\mathcal{P}$ the path from s_α to s_β, which connects components α and β
6: Update the labels of the states and the backward edges
7: Prune the backward edges connecting two single-color states of the same color

Connecting de Bruijn Graphs with Paths (Connect-DBG)
Input: A de Bruijn graph $G(V, E)$ of order k over alphabet $\Sigma = [0, \sigma), \sigma \leq (k-1)|V|$.
Output: A minimum-weight spanning tree $\mathcal{T}$ of the condensed graph $\widehat{G}$ of G.

A solution $\mathcal{T}$ to Connect-DBG naturally corresponds to a set $\mathcal{P}$ of paths on $G_{\Sigma,k}$ that make G weakly connected: an edge (i, j) of $\mathcal{T}$ corresponds to a shortest path from some node $u_i \in C_i$ to some node $u_j \in C_j$, and in turn, by the definition of dBG, such path is determined by the longest suffix/prefix overlap of u_i and u_j. Our algorithm essentially mimics the Kruskal algorithm [17] on the condensed graph $\widehat{G}$. However let us stress that we do not construct $\widehat{G}$ explicitly, as it would take $\Theta(k|V|^2)$ time, and moreover using the Kruskal algorithm as-is would require $\mathcal{O}(|V|^2 \log |V|)$ time (because $\widehat{G}$ has $\Theta(|V|^2)$ edges). We rather exploit the properties of dBGs and compute $\mathcal{T}$ by searching for longest suffix/prefix overlaps of the nodes of G. Our algorithm greedily selects, at each iteration, a longest suffix/prefix overlap (encoding a shortest path) of any two nodes that belong to different components. To do so, we define an augmented and modified version of the Aho-Corasick (AC) machine [1] of all the nodes of G, which we dynamically update every time we unite two components. The AC machine generalizes the Knuth-Morris-Pratt [16] algorithm for a set of strings. Informally, it is a finite-state machine that resembles a trie with additional *backward edges* (also called *failure transitions*) between the various states. There is exactly one failure transition $f(u) = v$ from each state u. Suffix/prefix overlaps can then be found using the following lemma.

▶ **Lemma 4** (Aho-Corasick lemma [1]). *Let u and v be two strings representing two distinct states of the AC machine, and identify the states with such strings. Then, $f(u) = v$ if and only if v is the longest proper suffix of u that is also a prefix of some string in the machine.*

We first assign each connected component of G a distinct color, and modify the AC machine of the nodes of G so that we maintain the three following invariants, in any iteration i of the algorithm:

I1. Each state has up to $d - i$ colors. Each terminal state is colored by its current connected component; each non-terminal state has the union of colors of the descending subtree.
I2. There are no backward edges connecting two single-color states of the same color.
I3. There are up to $k - 1$ backward edges outgoing from each terminal state s, each labeled by the color of s. There are no backward edges connecting non-terminal states.

Intuitively, we prune each backward edge connecting two single-color states colored α, because in this case all the nodes of G with the corresponding suffix/prefix overlap are in the same component α, and thus this edge cannot be used to unite unconnected components of G.

Algorithm 1 consists of four main phases: (i) preprocessing (Line 1); (ii) greedily selecting backward edges (Line 3); (iii) recoloring (Line 6); and (iv) pruning (Line 7).

(i) Preprocessing. We first identify the connected components of G, build the AC machine of its nodes and color its states according to invariant I1. We maintain the colors of a state u using a list LC_u and a dynamic hashtable HC_u. A key c of HC_u is a color of u, and its value is a pair of pointers: one to the position of c in LC_u, the second to any terminal state colored c below u. We also keep a counter `colors-cnt`(u) of the number of distinct colors of u.

From each terminal state s, we then follow the unique path of backward edges to the root and, for each state u on this path, we add a backward edge (s, u) of the same color as s, according to invariant I3. We maintain the backward edges outgoing from s with a list LB_s of their heads and a dynamic hashtable HB_s. A key u of HB_s is a state in LB_s (the head of an outgoing backward edge); its value is a pointer to the position of u in LB_s.

We keep the backward edges incoming to u with a list LE_u of their tails, and we maintain their colors with a dynamic hashtable HE_u. A key c of HE_u is the color of one such edges; its value is the list $\mathrm{HE}_u[c]$ of the positions in LE_u of the edges colored c. To add an incoming backward edge (s, u) of color α to u, we first append s to LE_u; we then look up the value of α in HE_u. If we find it, we append to $\mathrm{HE}_u[\alpha]$ the position of s in LE_u; otherwise, we create key α and initialize $\mathrm{HE}_u[\alpha]$ with the position of s in LE_u. Finally, we prune all the backward edges connecting two non-terminal states and, for each non-terminal state u colored c with `colors-cnt`$(u) = 1$, we query HE_u and prune from the machine all the backward edges (s, u) represented in the list $\mathrm{HE}_u[c]$ (using HB_s). For each color c, we also maintain a global counter `global-cnt`(c) of the total number of states and backward edges colored c.

(ii) Selecting backward edges. We select the backward edges in an order given by a reverse BFS, starting from the deepest states and proceeding level by level towards the root. At each visited state u of string depth (level) ℓ, we search for incoming backward edges, encoding a suffix/prefix overlap of length ℓ (Lemma 4), in the list LE_u. We select an edge of LE_u at each subsequent iteration, and only when LE_u is empty we move on to the next state. Note that the same backward edge (s, u) can be selected in multiple iterations, as it can be used to unite the component α of s with all the components coloring u, thus it will only be pruned when all such components are united with α.

To unite two components using a suffix/prefix overlap implied by a backward edge (s, u), we select two appropriate nodes of G by `components`(s, u), which takes as input a terminal state s of color α and a non-terminal state u, and outputs $s_\alpha = s$ and a terminal state s_β descending from u of some color $\beta \neq \alpha$; or returns `FAIL` if no such s_β exists (i.e., only when s and u both have the same single color α). We also add the path from s_α to s_β into $\mathcal{P}$.

(iii) Updating the colors. When we unite two components α and β, we change all labels α into β if `global-cnt`$(\alpha) \leq$ `global-cnt`(β); and change β into α otherwise. At each iteration one color is removed from the machine, and thus after iteration i there are $d - i$ distinct colors (I1). We update the colors of the states starting from the terminals and proceeding towards the root. To change color α to β in a non-terminal state u, we look up α in the hashtable HC_u, delete it from the list LC_u by following the first pointer of $\mathrm{HC}_u[\alpha]$ and remove the entry of α from HC_u. We then look up β in HC_u: if it is not there, we insert key β in HC_u with second pointer equal to the second pointer of $\mathrm{HC}_u[\alpha]$ and append β to LC_u. We also update the counter `colors-cnt`(u) of the number of colors of u and the counter `global-cnt`(β) of the total number of states and edges colored β accordingly: if β was already in HC_u, we decrease `colors-cnt`(u) by one (because of deleting α) and leave `global-cnt`(β) unchanged; otherwise we leave `colors-cnt`(u) unchanged and increase `global-cnt`(β) by one.

When we change the color α of a terminal state s into β, we must also change to β the color of the backward edges from s. To do so, for each edge (s, u) we look up α in HE_u. If we do not find it, then the color of all the edges with head u has already been updated. Otherwise, we access the list pointed by $\mathrm{HE}_u[\alpha]$ (which contains s and possibly other terminal states colored α). We insert β in HE_u and copy $\mathrm{HE}_u[\alpha]$ in $\mathrm{HE}_u[\beta]$, if β was not already there; or we append $\mathrm{HE}_u[\alpha]$ to the list $\mathrm{HE}_u[\beta]$, if $\mathrm{HE}_u[\beta]$ already existed. In both cases, we set $\texttt{global-cnt}(\beta)$ to $\texttt{global-cnt}(\beta) + \texttt{len}(\mathrm{HE}_v[\alpha])$ and remove the entry of α from HE_u.

(iv) Pruning. If, after updating the colors in the machine, the only remaining color of a non-terminal state u is β (i.e., $\texttt{colors-cnt}(u) = 1$), we query HE_u with key β. If we find β, we prune all edges (s, u) with s in the list pointed by $\mathrm{HE}_u[\beta]$, and also delete the entry for β from HE_u. To prune an edge (s, u), we look up u in HB_s, delete u from LB_s following the pointer $\mathrm{HB}_s[u]$ and finally delete the entry of u from HB_s. This ensures invariant I2, because the backward edges in $\mathrm{HE}_u[\beta]$ are all and only those of color β with head u.

▶ **Theorem 5.** *Connect-DBG can be solved in $\mathcal{O}(|V|k \log d + |E|)$ time using $\mathcal{O}(k|V| + |E|)$ working space.*

Proof. For the correctness of Algorithm 1 we first show that, at any iteration, the backward edges in our machine represent all suffix/prefix overlaps of nodes in two currently distinct components of G. By Lemma 4, for each state u on the path of backward edges from a terminal state s to the root in the AC machine of V, the partial path ending at u encodes a suffix/prefix overlap between s and any terminal state below u; and each possible suffix/prefix overlap between s and any other node in V corresponds to one such partial path. During preprocessing, we replace each such partial path with a single backward edge (s, u); and by invariants I1-I3, we only keep the backward edges (s, u) encoding an overlap between s and some node of V in a different component (some other nodes in the same component may have the same overlap, but function `components` makes the algorithm ignore them).

The correctness of Algorithm 1 then directly follows from the above and from the correctness of the Kruskal algorithm [17] for computing a minimum-weight spanning tree.

For the complexity analysis, we bound the time for each of the four main phases as follows: (i) preprocessing by $\mathcal{O}(k|V| + |E|)$; (ii) selecting backward edges by $\mathcal{O}(k|V|)$; (iii) recoloring by $\mathcal{O}(|V|k \log d)$; and (iv) pruning by $\mathcal{O}(k|V|)$. The working space is bounded by $\mathcal{O}(k|V| + |E|)$, the size of G.

(i) Preprocessing. Computing the connected components of G and giving each one a color $c \in [1, d]$ requires $\mathcal{O}(|V| + |E|)$ time, with $|E|$ the number of *distinct* edges of G [14]. Building the AC machine of V takes $\mathcal{O}(k|V|)$ time because each string is of length $k - 1$ [1, 8]. To implement HE, HC and HB we use perfect hashing, supporting insertions and deletions of key-value pairs, and to retrieve any entry with a given key. The running time per operation is $\mathcal{O}(1)$ *with high probability* [7, Theorem 1.1]. Colors are assigned to the states of the AC machine in $\mathcal{O}(k|V|)$ time, starting from the terminal states and proceeding up to the root.

For each of the $|V|$ terminal states, we follow a path of backward edges of length up to $k - 1$ (as the string depth of the machine is $k - 1$ and backward edges connect states with strictly decreasing string depth) and add up to $k - 2$ backward edges in $\mathcal{O}(1)$ time per edge (s, u) by using the hashtables HE_u, HC_u and HB_s. This takes $\mathcal{O}(k|V|)$ time in total. Finally, the initial pruning of backward edges requires $\mathcal{O}(k|V|)$ total time, as we visit each non-terminal state u, look up at most one key in HE_u, and possibly delete the edges (s, u) represented by the list stored at HE_u by using the hashtable HB_s.

(ii) Selecting backward edges. Each step of the reverse BFS takes $\mathcal{O}(1)$ time, and we abort it when we have selected $d-1$ backward edges. A state can be visited multiple times only if there are still incoming backward edges that can be selected, and in this case we select one of them at each visit. Since $d \leq |V|$, the whole visit requires $\mathcal{O}(k|V|+d) = \mathcal{O}(k|V|)$ time in total. Moreover, for each selected edge (s,u), we compute $\texttt{components}(s,u)$ in $\mathcal{O}(1)$ time by visiting up to two elements in the color list of u LC_u.[3]

(iii) Updating the colors. Changing color α to β in a non-terminal state u takes $\mathcal{O}(1)$ time by using HC_u. Changing from α to β the color of all the backward edges (s,u) outgoing from a terminal state s requires accessing the list at $\mathrm{HE}_u[\alpha]$ and appending the whole list $\mathrm{HE}_u[\alpha]$ to the (possibly empty) list $\mathrm{HE}_u[\beta]$. This procedure amortizes to $\mathcal{O}(1)$ time for each recolored backward edge.

We next show that the algorithm does $\mathcal{O}(|V|k\log d)$ recolorings of states and edges over all iterations via an auxiliary data structure: a rooted binary tree with the d colors (components) as leaves. Each leaf c is weighted with $\texttt{global-cnt}(c)$, which is the total number of occurrences of color c the machine. Internal nodes in the tree represent the component unions done by the algorithm, each weighted with the number of states and backward edges that are recolored in the corresponding step, i.e., the lightest weight of its two children. We remark that this tree is not part of the algorithm, but rather it is just a conceptual aid to count the number of color updates in the worst case. The total number of recolorings done by Algorithm 1 is given by the sum of all the weights on the internal nodes. Let $f(w,d)$ be the maximum such sum on a tree with d leaves with a total weight of w. We will prove the following claim by induction on d.

▷ Claim. $f(w,d) \leq w\log_2(d)$.

Proof.

Induction basis: If $d=1$, then the tree consists of the root and one leaf, so $f(w,d) = 0 = w\log_2(d)$.

Induction hypothesis: For all $d' < d$, we have $f(w,d') \leq w\log_2(d')$.

Induction step: Consider a situation with d colors. The root of the tree corresponds to the final recoloring, when the last two components are merged. The two subtrees starting from its children have weight w_1 and w_2 and d_1 and d_2 leaves, respectively. Without loss of generality $w_1 \leq w_2$. We now bound $f(w,d)$:

$$\begin{aligned} f(w,d) &\leq f(w_1,d_1) + f(w_2,d_2) + \min(w_1,w_2) \leq w_1\log_2(d_1) + w_2\log_2(d_2) + w_1 \\ &\leq w_1\log_2(\min(d_1,d_2)) + w_2\log_2(\max(d_1,d_2)) + w_1 \\ &= w_1\log_2(2\min(d_1,d_2)) + w_2\log_2(\max(d_1,d_2)) \\ &\leq (w_1+w_2)\log_2(d_1+d_2) = w\log_2(d). \end{aligned}$$

◁

We conclude that $f(w,d) \leq w\log_2(d)$ for all $d \in \mathbb{N}$. Observe that $w = \mathcal{O}(k|V|)$, because the color of each of the $|V|$ terminal states propagates to at most $k-2$ non-terminal states (the depth of the machine is $k-1$), and there are up to $k-2$ backward edges from each terminal state; and therefore $w\log_2(d) = \mathcal{O}(|V|k\log d)$.

(iv) Pruning. Pruning a backward edge (s,u) requires $\mathcal{O}(1)$ time using the hashtables HE_u and HB_s. Since there are up to $k|V|$ backward edges, deletions take $\mathcal{O}(k|V|)$ time overall. ◀

[3] To compute $\texttt{components}(s,u)$ with s colored α in $\mathcal{O}(1)$ time, we maintain a pointer in the header of the color list of each state. We either select the color β of the header of LC_u or, if it is equal to α, we advance the pointer, which guarantees finding $\beta \neq \alpha$, as the lists do not contain duplicates. In both cases we follow the pointer at $\mathrm{HC}_u[\beta]$ to find s_β in $\mathcal{O}(1)$ time.

5 Balancing de Bruijn Graphs in Optimal Time

We present an exact $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$-time algorithm for balancing any dBG $G(V, E)$ of order k so that the number $|\mathcal{A}|$ of newly added edges is minimized. As a consequence of the Euler's theorem, when the input graph G is weakly connected, our algorithm makes it Eulerian with the smallest possible cost. Let us first formally define the problem.

Balancing de Bruijn Graphs (Balance-DBG)
Input: A de Bruijn graph $G(V, E)$ of order k over alphabet $\Sigma = [0, \sigma), \sigma \leq (k-1)|V|$.
Output: A balanced graph $H = (V \cup \mathcal{B}, E \cup \mathcal{A})$ with $\mathcal{B} \subseteq V_{\Sigma,k}$, $\mathcal{A}$ over $E_{\Sigma,k}$ and minimized $|\mathcal{A}|$.

It is easy to see that Balance-DBG can be reduced to the Multi-SCCS problem (defined in Section 1). In particular, Balance-DBG reduces to an instance of Multi-SCCS with $S = E$. A greedy algorithm, which keeps merging the suffix and prefix with the longest overlap until we are left with only cyclic strings, is known to solve Multi-SCCS exactly [4]. Cazaux and Rivals showed a linear-time implementation of this algorithm [4, Theorem 10], which implies an $\mathcal{O}(k|E|)$-time algorithm for Balance-DBG. In a dBG, this algorithm corresponds to finding a minimum-weight matching between the heads and the tails of the edges, where the weight is given by the length of a shortest directed path from the head to the tail. The greedy algorithm constructs a matching by repeatedly adding a feasible edge of minimum weight. Although such greedy algorithm is not exact on general weighted bipartite graphs [10], it turns out to be optimal in the *special case* of dBGs following from the optimality of the greedy algorithm for Multi-SCCS [4]. In balanced nodes, all heads and tails can be matched for a cost of zero. The greedy algorithm will match those first, so it thus suffices to only match up the excess heads and tails at unbalanced nodes. In what follows, we describe a different implementation of the greedy strategy which gives optimal time complexity for the special instances arising from Balance-DBG. Similar to Section 4, we employ an augmented and modified version of the AC machine.

Let $Z^+ \subset V$ be the nodes with higher out-degree d^+ than in-degree d^-, and $Z^- \subset V$ the nodes with $d^- > d^+$. We construct the AC machine of $Z^+ \cup Z^-$ and preprocess it as follows. We label by $-$ each terminal state $s \in Z^-$ and initialize a counter $m_s = d_s^- - d_s^+$; we also label by $+$ each state encoding a *prefix* of $s \in Z^+$ and initialize a counter $m_s = d_s^+ - d_s^-$ for s. In addition, for every non-terminal state u, we compute a set $D(u)$ of all its descendant terminal states $s \in Z^+$. From each terminal state s labelled $-$, we follow the unique path of backward edges to the root: for each non-terminal state u labelled $+$ on this path, we add a backward edge (s, u). We finally prune all backward edges that do not link a $-$ state with a $+$ state: we maintain the backward edges of the machine similar to Section 4. Our algorithm first sets $\mathcal{A} = \emptyset$ and $\mathcal{B} = \emptyset$ and then iteratively adds edges to $\mathcal{A}$ and nodes to $\mathcal{B}$ as follows.

We traverse the machine in reverse BFS order starting at the terminal states (this traversal was proposed by Ukkonen in [28]). When we encounter the head of a backward edge at a state u, we find the terminal state $s^- \in Z^-$ at the tail of the edge and any terminal state $s^+ \in D(u) \subseteq Z^+$. Let s be the shortest string with prefix s^+ and suffix s^- (as an application of Lemma 4). We add $\min\{m_{s^+}, m_{s^-}\}$ copies of $s[i \mathinner{..} i+k-1]$, for all $i \in [0, |s|-k]$, to $\mathcal{A}$; we add $s[i \mathinner{..} i+k-2]$ to $\mathcal{B}$ if it is not in $V \cup \mathcal{B}$; and we decrease both m_{s^+} and m_{s^-} by $\min\{m_{s^+}, m_{s^-}\}$. When $m_{s^-} = 0$, we delete all backward edges starting from the terminal state s^- and update the edge lists of their heads accordingly. When $m_{s^+} = 0$, we delete s^+ from the D sets of its ancestors. If any $D(u)$ becomes empty, we delete all incoming edges at state u. The algorithm terminates when there are no more backward edges in the machine.

▶ **Theorem 6.** *Balance-DBG can be solved in the optimal $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$ time using $\mathcal{O}(k|V| + |E|)$ working space.*

Proof. The correctness of the algorithm follows from the observation that in order to solve Balance-DBG via Multi-SCCS it suffices to consider the nodes in $Z^+ \cup Z^-$, and from the fact that a greedy strategy solves Multi-SCCS exactly [4, Theorem 10] (see the discussion above). We thus conclude that the presented algorithm is correct.

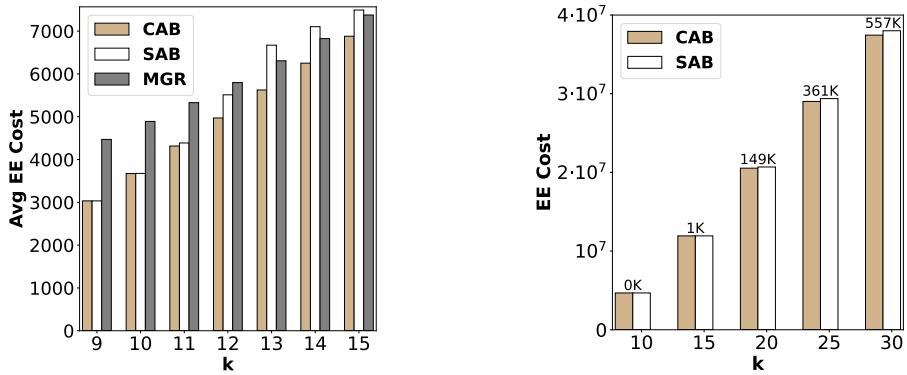
Constructing Z^+ and Z^- and computing the initial values of m counters takes $\mathcal{O}(|E|)$ time via a traversal of G. Constructing, traversing and updating the AC machine takes $\mathcal{O}(k|V|)$ time because $|Z^+ \cup Z^-| \leq |V|$ and each node in V is a string of length $k-1$. Note that any edge added in $\mathcal{A}$ and any node added in $\mathcal{B}$ can be represented in $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ space using two nodes in V. Thus, the total time required to output graph $H = (V \cup \mathcal{B}, E \cup \mathcal{A})$ is $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$. The working space is bounded by $\mathcal{O}(k|V| + |E|)$, the size of G. ◀

6 Experiments

Methods and Setup. We designed a method for Extend-DBG based on our theoretical findings. The method first connects the input dBG based on our exact algorithm underlying Theorem 5 and then balances it by our exact algorithm underlying Theorem 6. We remark that both these algorithms are exact but their combination is generally not, which is consistent with Extend-DBG being NP-hard. To further help balancing, our method connects the graph using only unbalanced nodes. Our method is called CAB (for *connect* and *balance*).

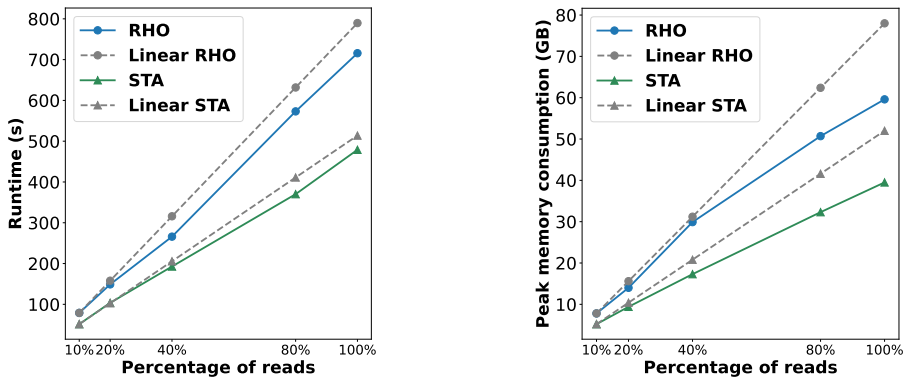
We compared CAB to the $\frac{1}{2}$-approximation algorithm for Multi-SCS that maximizes the compression offered by the output string [4]. We refer to this algorithm as MGR (for Multi-SCS *Greedy*). To specifically examine the impact of our connect framework on extension cost, we also designed a "hybrid" method, referred to as SAB (for *SCS* and *balance*). SAB first connects the graph based on the greedy algorithm [2] for SCS and then balances it by the algorithm of Theorem 6, as CAB does. The intuition is that any (shortest) common superstring s of set V corresponds to a connected extended dBG. To connect G, we consider all the potential additional edges implied by s and greedily add to G a smallest subset of them that makes G connected. The pseudocode of SAB is provided in Algorithm 2 of Appendix A.

We implemented the above methods in `C++` and ran them on a single core of an AMD Opteron 6386 SE 2.8GHz CPU with 252GB RAM running GNU/Linux. Our source code is available at `https://bitbucket.org/eulerian-ext/cpm2022/`. We used two whole-genome shotgun benchmark datasets that are available from `http://gage.cbcb.umd.edu/data/index.html`: (i) *Rhodobacter sphaeroides* (RHO); and (ii) *Staphylococcus aureus* (STA). The number of reads in RHO and STA is $2,050,868$ (Library 1) and $1,294,104$ (Library 1), respectively. In both datasets, the average read length is 101bp and the insert length is 180bp. Tables 1a and 1b in Appendix A show the characteristics of the two datasets. Although MGR works in polynomial time [4], no efficient (e.g., linear-time or near-linear-time) implementation of MGR is known. This is in contrast to SAB, which uses a linear-time implementation of the greedy algorithm for SCS [2] to connect the graph. Since our implementation of MGR works in quadratic time in the input size, we used randomly selected *samples* for each dataset and every k in the comparison against MGR. The samples were constructed by selecting 650 reads from each dataset uniformly at random and had roughly 40K to 55K nodes and 60K edges.



(a) RHO samples. **(b)** RHO.

Figure 3 (a) Average Eulerian Extension (EE) cost vs. k over five random samples of RHO. (b) EE cost vs. k on the whole RHO dataset. The difference between the EE costs of SAB and CAB is shown on the top of each pair of bars (K stands for thousands).



(a) Runtime of CAB. **(b)** Memory of CAB.

Figure 4 Runtime and peak memory consumption vs. reads of CAB on RHO and STA for $k = 30$. The solid lines are the results for CAB; the dashed lines are the results that would be produced by a linear scaling of the algorithm.

Eulerian Extension (EE) Cost. Figure 3a shows the average EE cost of all methods on five random samples of the RHO dataset, for varying k. Our CAB method outperformed both MGR and SAB in all tested cases. MGR performed poorly for small k values, as the edge multiplicities are larger and the extension cost is heavily determined by balancing, whereas SAB performed poorly for larger k values, as the edge multiplicities are smaller and the EE cost is heavily determined by connecting. Our results are very promising because MGR was also orders of magnitude slower than CAB (as expected).

Figure 3b shows the EE cost on the whole RHO dataset, for varying k. We show the result only for the SAB and CAB methods, since MGR could not terminate in reasonable time. Note that there is no difference between the two methods for $k = 10$, as in this case the input graph is connected and thus both SAB and CAB balance it in the same optimal way. However, for $k > 10$, CAB outperforms SAB consistently, and the difference generally increases with k. This shows that, unlike SAB, our method is able to connect the graph with a small cost, even when the graph has a large number of components.

Analogous results to those of Figure 3 for the STA dataset are in Figure 5 of Appendix A.

Runtime and Peak Memory Consumption. Figures 4a and 4b show that the runtime and peak memory consumption of CAB scale (even better than) *linearly* with the input size, which confirms our complexity analysis (see Theorems 5 and 6). The results for SAB are omitted to avoid cluttering the figures; SAB was several times slower but consumed slightly less memory, mainly due to the space-efficient SCS algorithm [2] it employs.

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
2. Jarno Alanko and Tuukka Norri. Greedy shortest common superstring approximation in compact space. In *24th SPIRE*, volume 10508 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2017. doi:10.1007/978-3-319-67428-5_1.
3. Bastien Cazaux and Eric Rivals. A linear time algorithm for shortest cyclic cover of strings. *J. Discrete Algorithms*, 37:56–67, 2016. doi:10.1016/j.jda.2016.05.001.
4. Bastien Cazaux and Eric Rivals. Superstrings with multiplicities. In *29th CPM*, volume 105 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CPM.2018.21.
5. Maxime Crochemore, Marek Cygan, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Algorithms for three versions of the shortest common superstring problem. In *21st CPM*, pages 299–309, 2010. doi:10.1007/978-3-642-13509-5_27.
6. Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
7. Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *17th ICALP*, pages 6–19, 1990. doi:10.1007/BFb0032018.
8. Shiri Dori and Gad M. Landau. Construction of Aho Corasick automaton in linear time for integer alphabets. *Inf. Process. Lett.*, 98(2):66–72, 2006. doi:10.1016/j.ipl.2005.11.019.
9. Frederic Dorn, Hannes Moser, Rolf Niedermeier, and Mathias Weller. Efficient algorithms for Eulerian extension and Rural Postman. *SIAM J. Discret. Math.*, 27(1):75–94, 2013. doi:10.1137/110834810.
10. Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1:1–1:23, 2014. doi:10.1145/2529989.
11. Sara El-Metwally, Taher Hamza, Magdi Zakaria, and Mohamed Helmy. Next-generation sequence assembly: Four stages of data processing and computational challenges. *PLoS Comput. Biol.*, 9(12), 2013. doi:10.1371/journal.pcbi.1003345.
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. Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.
14. John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973. doi:10.1145/362248.362272.
15. Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computation*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.
16. 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.
17. Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. doi:10.1090/S0002-9939-1956-0078686-7.

18 Paul Medvedev. Modeling biological problems in computer science: a case study in genome assembly. *Briefings Bioinform.*, 20(4):1376–1383, 2019. doi:10.1093/bib/bby003.

19 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.

20 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.

21 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.

22 Niranjan Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14:157–167, 2013.

23 Clifford S. Orloff. A fundamental problem in vehicle routing. *Networks*, 4(1):35–64, 1974. doi:10.1002/net.3230040105.

24 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.

25 Michael C. Schatz, Arthur L. Delcher, and Steven L. Salzberg. Assembly of large genomes using second-generation sequencing. *Genome Res.*, 20(9):1165–1173, 2010. doi:10.1101/gr.101360.109.

26 Jared T. Simpson and Mihai Pop. The theory and practice of genome sequence assembly. *Annu Rev Genomics Hum Genet*, 16:153–172, 2015.

27 Jang-il Sohn and Jin-Wu Nam. The present and future of de novo whole-genome assembly. *Briefings Bioinform.*, 19(1):23–40, 2018. doi:10.1093/bib/bbw096.

28 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990. doi:10.1007/BF01840391.

29 Bilal Wajid and Erchin Serpedin. Review of general algorithmic features for genome assemblers for next generation sequencers. *Genomics, Proteomics & Bioinformatics*, 10(2):58–73, 2012. doi:doi.org/10.1016/j.gpb.2012.05.006.

A Omitted Details from Section 6

Algorithm 2 SAB.

Compute the connected components of $G(V, E)$
$s \leftarrow \mathsf{SCS}(V)$ ▷ A (shortest) common superstring of V using the algorithm of [2]
Let $Q_1 = u_1, \ldots, u_{|V|}$ be the sequence of all nodes in V as they occur in s
Let $Q_2 = (u_1, u_2), \ldots, (u_{|V|-1}, u_{|V|})$ be the sequence of edges as they occur in Q_1
Sort Q_2 in decreasing order w.r.t. the length of the longest suffix/prefix overlap of (u_i, u_j)
$i \leftarrow 0$
while $G'(V, E)$ is not weakly connected **do** ▷ Connects the graph
  $(u, v) \leftarrow Q_2[i]$ ▷ Gets the ith longest suffix/prefix overlap
  if the components where u and v lie are not currently connected **then**
    Let q be the shortest string with u as prefix and v as suffix
    Extend E with all edges $(q[p\,..\,p+k-2], q[p+1\,..\,p+k-1])$ occurring in q
    Extend V with all new nodes $q[p\,..\,p+k-1] \notin V$ occurring in q
  $i \leftarrow i + 1$
Algorithm of Theorem 6 on graph $G'(V, E)$ to find multiset $\mathcal{A}$ ▷ Balances the graph

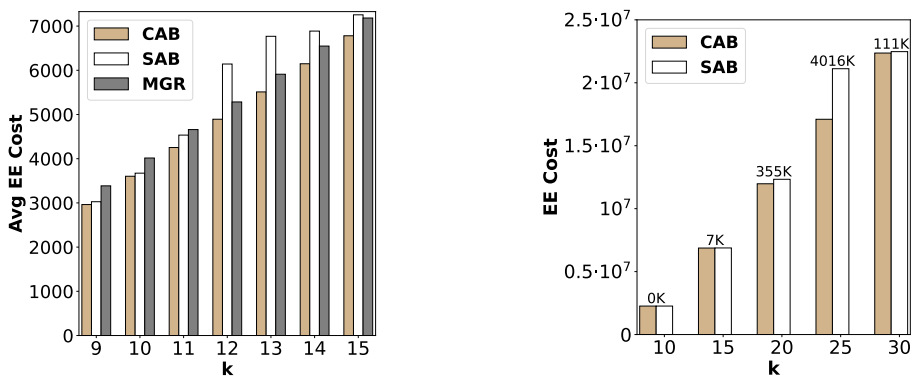
Table 1 Datasets characteristics.

(a) *Rhodobacter sphaeroides* (RHO).

k	# nodes	# edges	# distinct edges	# components
10	1,013,904	185,506,278	3,338,995	1
15	37,858,157	175,579,617	46,337,190	528
20	61,265,275	165,433,984	62,546,892	44,386
25	64,861,977	155,232,772	65,087,335	131,266
30	65,383,451	145,014,018	65,356,249	199,627

(b) *Staphylococcus aureus* (STA).

k	# nodes	# edges	# distinct edges	# components
10	1,047,172	117,107,289	3,974,601	1
15	40,262,854	110,650,401	42,924,890	1,637
20	45,318,307	104,188,673	45,512,480	152,945
25	45,833,210	97,727,029	45,825,958	211,943
30	45,498,694	91,266,009	45,354,736	259,333



(a) STA samples. **(b)** STA.

Figure 5 (a) Average Eulerian Extension (EE) cost vs. k over five random samples of STA. (b) EE cost vs. k on the whole STA dataset. The difference between the EE costs of SAB and CAB is shown on the top of each pair of bars (K stands for thousands).