Eulertigs: Minimum Plain Text Representation of k-mer Sets Without Repetitions in Linear Time

Sebastian Schmidt[1]
University of Helsinki, Finland

Jarno N. Alanko
University of Helsinki, Finland

Abstract

A fundamental operation in computational genomics is to reduce the input sequences to their constituent k-mers. For maximum performance of downstream applications it is important to store the k-mers in small space, while keeping the representation easy and efficient to use (i.e. without k-mer repetitions and in plain text). Recently, heuristics were presented to compute a near-minimum such representation. We present an algorithm to compute a minimum representation in optimal (linear) time and use it to evaluate the existing heuristics. For that, we present a formalisation of arc-centric bidirected de Bruijn graphs and carefully prove that it accurately models the k-mer spectrum of the input. Our algorithm first constructs the de Bruijn graph in linear time in the length of the input strings (for a fixed-size alphabet). Then it uses a Eulerian-cycle-based algorithm to compute the minimum representation, in time linear in the size of the output.

2012 ACM Subject Classification Applied computing → Computational biology; Theory of computation → Data compression; Theory of computation → Graph algorithms analysis; Theory of computation → Data structures design and analysis

Keywords and phrases Spectrum preserving string sets, Eulerian cycle, Suffix tree, Bidirected arc-centric de Bruijn graph, `k`-mer based methods

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.2

Related Version *Full Version*: `https://www.biorxiv.org/content/10.1101/2022.05.17.492399`

Supplementary Material *Software (Source Code)*: `https://github.com/algbio/matchtigs`, archived at `swh:1:dir:e33705e470606ba7ba1d670041010c056a781fb8`
Software (Experiment Pipeline): `https://doi.org/10.5281/zenodo.6538261`

Funding *Sebastian Schmidt*: Funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO).
Jarno N. Alanko: Funded by NIH NIAID grant No. R01HG011392 and Academy of Finland grant 339070.

1 Introduction

Motivation

A k-mer is a DNA string of length k that is considered equal to itself and its reverse complement. A common pattern in bioinformatics is to reduce a set of input strings to their constituent k-mers. Such representations are at the core of many bioinformatics pipelines – see e.g. Schmidt et al. [23] or Brinda et al. [6] for an overview of applications. The wide-spread use of k-mer sets has prompted the question of what is the smallest *plain text representation* for a set of k-mers. Here, a plain text representation means a set of strings that have the same set of k-mers as the input strings, i.e. the *spectrum* is preserved. Such representations are also called *spectrum preserving string sets* (SPSS) [22], or simplitigs [6]. This has the following advantages over encoded representations:

[1] corresponding author

- When storing k-mer sets to disk, plain text may remove the need of decompression before usage, as some tools that usually take unitigs as input can take any other plain text representation without modification (e.g. Bifrost [13]).
- Within an application, an encoded representation would require decoding whenever a k-mer is accessed, which may slow down the application a lot compared to when each k-mer is in RAM in plain text.

Further, in applications, it might be useful if the representation contains each k-mer exactly once. This is because some applications, like e.g. SSHash [21], are able to take any set of k-mers as input, but cannot easily deal with duplicate k-mers in the input.

Related work

There are two heuristic approaches to the construction of a small SPSS without repetitions, namely *prophasm* [6] and *UST* [22]. While neither of these computes a minimum representation, Rahman et al. [22] also present a lower bound to the minimum size of any representation without repetition, and they show that they are within 3% of this lower bound in practice. They also present a counter-example showing that their lower bound is not tight. Small SPSSs without repetitions are used e.g. in SSHash [21] and are also computed by state-of-the-art de Bruijn graph compactors like Cuttlefish 2 [15].

When k-mer repetitions are allowed in an SPSS, there is a known polynomially computable minimum representation, namely *matchtigs* [23]. While matchtigs are expensive to compute, the authors also present a more efficient greedy heuristic that is able to compute a near-minimum representation on a modern server with no significant penalty in runtime (when compared to computing just unitigs), but a significant increase in RAM usage.

In [6, 23] the authors also showed that decreasing the size of an SPSS results in significantly better performance in downstream applications, i.e. when further compressing the representation with general purpose compressors, or when performing k-mer-based queries.

The authors of both [6] and [22] consider whether computing a minimum representation without repetitions may be NP-hard, as it is equivalent to computing a minimum path cover in a de Bruijn graph, which is NP-hard in general graphs by reduction from Hamiltonian cycle. However, computing a Hamiltonian cycle in a de Bruijn graph is actually polynomial [14]. The authors of [14] argue that de Bruijn graphs are a subclass of *adjoint* graphs, in which solving the Hamiltonian cycle problem is equivalent to solving the Eulerian cycle problem in the *original* of the adjoint graph, which can be computed in linear time[2]. However, the argument is only made for normal directed (and not bidirected) graphs, and thus is not applicable to our setup, where a k-mer is also considered equal to its reverse complement.

Our contributions

Our first technical contribution is to carefully define the notion of a bidirected de Bruijn graph such that the spectrum of the input is accurately modelled in the allowed walks of the graph. Our definition also takes into account k-mers that are their own reverse complement. This technicality is often neglected in the literature, and sidestepped by requiring that the

[2] The original of an adjoint graph can be computed by splitting each node v into two nodes v' and v'' such that v' keeps the incoming arcs, and v'' the outgoing arcs as in [5, Figure 4]. Then, the graph is a collection of complete bipartite graphs [5]. These graphs can be contracted into single nodes, and then we add an arc between the contracted representations of each v' and v''. This can be computed in linear time and is the original graph, since all nodes have become arcs again, and the arcs have the correct predecessors and successors.

value of k is odd, in which case this special case does not occur. We give a suffix-tree-based deterministic linear-time algorithm to construct such a graph, filling a theory gap in the literature, as existing approaches [8, 15, 13, 1] depend on the value of k and/or are probabilistic due to the of use hashing, minimizers or Bloom filters, or do not use the reverse-complement-aware definition of k-mers [7].

Given the bidirected de Bruijn graph, we present an algorithm that computes a minimum plain text representation of k-mer sets without repetitions, which runs in output sensitive linear time. Steps 1 to 3 run in linear time in the number of nodes and arcs in the graph. In short, it works as follows:

1. Add breaking arcs into this graph to make it Eulerian.
2. Compute a Eulerian cycle in the resulting graph.
3. Break that cycle at the breaking arcs.
4. Output the strings spelled by the resulting walks.

The algorithm is essentially an adaption of the matchtigs algorithm [23], removing the possibility of joining walks by repeating k-mers. We give detailed descriptions for all these steps and prove their correctess in our bidirected de Bruijn graph model. Together with our linear-time de Bruijn graph construction algorithm, we obtain the main result of our paper:

▶ **Theorem 1.** *Let k be a positive integer and let I be a set of strings of length at least k over some alphabet Σ. Then we can compute a set of strings I' of length at least k with minimum cumulative length and $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$ in $O(||I|| \log |\Sigma|)$ time.*

where $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$ means that I' is an SPSS of I, and $||I||$ is the cumulative length of I (see Section 2 for accurate definitions). This gives a positive answer to the open question if a minimum SPSS without repetitions can be computed in polynomial time. Additionally, we give an easily computable tight lower bound on the size of a minimum SPSS without repetitions.

For our experiments, we have implemented steps 1 to 4 in Rust, taking the de Bruijn graph as given. The implementation is available on github: `https://github.com/algbio/matchtigs`. Our experimental evaluation shows that our algorithm does not result in significant practical improvements, but for the first time allows to benchmark the quality the heuristics prophasm and UST against an optimal solution. It turns out that both produce close-to-optimal results, but with a different distribution of computational resources.

Our work also shows that using arc-centric de Bruijn graphs can aid the intuition for certain problems, as in this case, the node-centric variant hides the relationship between Eulerian cycles and minimum SPSS without repetition.

Organisation of the paper

In Section 2 we give preliminary definitions of well-known concepts. In Section 3 we define de Bruijn graphs and prove the soundness of the definitions. In Section 4 we show how to construct de Bruijn graphs by our definitions in linear time. In Section 5 we show how to construct a minimum SPSS without repetitions in linear time if the de Bruijn graph is given. In Section 6 we compare our algorithm and Eulertigs against strings computed with prophasm and UST on practical data sets.

2 Preliminaries

In this section we give the prerequisite knowledge required for this paper.

2.1 Bidirected graphs

In this section we define our notion of the bidirected graphs and the incidence model.

A multiset is defined as a set M, and an implicit function $\#_M : M \to \mathbb{Z}^+$ mapping elements to their multiplicities. The cardinality is defined as $|M| := \sum_{s \in M} \#_M(s)$.

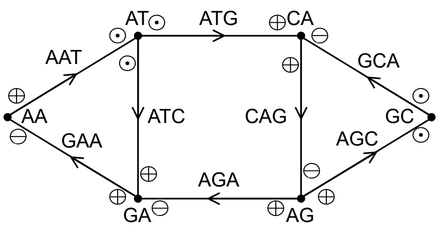
An *alphabet* Σ is an ordered set, and an Σ-*word* is a string of characters of that set. String concatenation is written as ab for two strings a and b. The set Σ^k is the set of all Σ-words of length k and the set Σ^* is the set of all Σ-words, including the empty word ϵ. Given a positive integer k, the *k-suffix* $\text{suf}_k(w)$ (*k-prefix* $\text{pre}_k(w)$) of a word w is the substring of its last (first) k characters. A *k-mer* is a word of length k. A *complement function* over Σ is a function $\text{comp} : \Sigma \to \Sigma$ mapping characters to characters that is self-inverse (i.e. $\text{comp}(\text{comp}(x)) = x$). A *reverse complement* function for alphabet Σ is a function $\text{rc} : \Sigma^* \to \Sigma^*$ defined as $\text{rc}((w_1, \dots, w_\ell)) := (\text{comp}(w_\ell), \dots, \text{comp}(w_1))$, for some arbitrary complement function comp. On sets, rc is defined to compute the reverse complement of each element in the set. Note that rc is self-inverse. A *canonical* k-mer is a k-mer that is lexicographically smaller than or equal to its reverse complement.

Given an integer k and an alphabet Σ, the *k-spectrum* of a set of strings $I \subseteq \bigcup_{k' \geq k} \Sigma^{k'}$ is a set of strings $\text{S}_k(I) := \{w \in \Sigma^k \mid \exists i \in I : w \text{ is substring of } i \text{ or } \text{rc}(i)\}$. The *canonical k-spectrum* of I is $\text{CS}_k(I) := \{w \in \text{S}_k(I) \mid w \text{ is canonical}\}$. For simplicity, the spectrum and canonical spectrum are defined for a single string w as if it were a set $\{w\}$. A *spectrum preserving string set* of a set of strings I is a set of strings I' such that $\text{CS}_k(I) = \text{CS}_k(I')$. The cumulative length of I is $||I|| := \sum_{w \in I} |w|$.

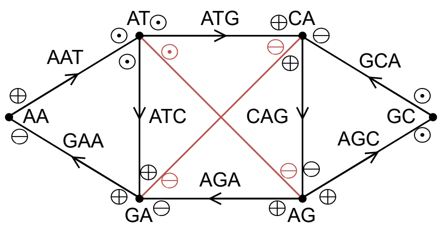
Our definition of a bidirected graph is mostly standard like in e.g. [17], however we allow self-complemental nodes that occur in bidirected de Bruijn graphs. A *bidirected graph* is a tuple $G = (V, E, c)$ with a set of normal and *self-complemental* nodes $v \in V$, a set of arcs $e \in E$, and a function $c : V \to \{1, 0\}$ marking self-complemental nodes with 1, and normal nodes with 0. An *incidence* is a pair vd, where $d \in \{\oplus, \ominus, \odot\}$ is called its *sign* (e.g. $v\oplus$). The negation of a sign is defined as $\neg\oplus := \ominus$, $\neg\ominus := \oplus$ and $\neg\odot := \odot$. For self-complemental nodes $v \in V$, only incidences $v\odot$ are allowed, and for normal nodes only incidences $v\oplus$ and $v\ominus$ are allowed. An *arc* $(v_1 d_1, v_1' d_1', \eta) \in E$ is a tuple of incidences and a unique identifier η, where η can be of any type. The *reversal* of an arc is denoted by $(v_1 d_1, v_1' d_1', \eta)^{-1} := (v_1' d_1', v_1 d_1, \eta)$. If not required, we may drop the identifier (i.e. just write $(v_1\ominus, v_1'\odot) \in E$). If a node $v \in V$ is present with a $\oplus$ ($\ominus$) sign in an arc, then the arc is *outgoing* (*incoming*) from (to) v.

Note that, other than in standard directed graphs, in bidirected graphs arcs can be outgoing or incoming on both ends, and the order of the incidences in the arc does not affect if it is outgoing or incoming to a node. Further, our notation differs from that of standard bidirected graphs in that arcs have a direction. This is required because we will work with arc-centric de Bruijn graphs (see Section 3), which have labels on the arcs and not the nodes. Using the sign of the incidence pairs, it is possible to decide if a node is traversed forwards or backwards, but not if the arc is traversed forwards or backwards. But to decide which label (forwards or reverse complement) to use when computing the string spelled by an arc, the direction is relevant. See Figure 1 (a) for an example of a bigraph, which has labels that make it a de Bruijn graph as well.

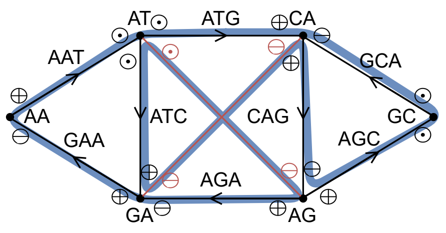
A *walk* in a bigraph is a sequence of arcs $W := ((v_1 d_1, v_1' d_1', \eta_1), (v_2 d_2, v_2' d_2', \eta_2), \dots, (v_\ell d_\ell, v_\ell' d_\ell', \eta_\ell))$ where for every i it holds that $(v_i d_i, v_i' d_i', \eta_i) \in E$ or $(v_i' d_i', v_i d_i, \eta_i) \in E$ (we can arbitrarily walk over arcs forwards and reverse), and for every $i < \ell$ it holds that $v_i' = v_{i+1}$ and $d_i' = \neg d_{i+1}$. The *length* of a walk is $\ell = |W|$. If $v_1 = v_\ell'$ and $d_1 = \neg d_\ell'$, then W is a *cycle*. A bigraph is *connected*, if for each pair of nodes $v_1, v_2 \in V$ there is a walk from v_1 to v_2.



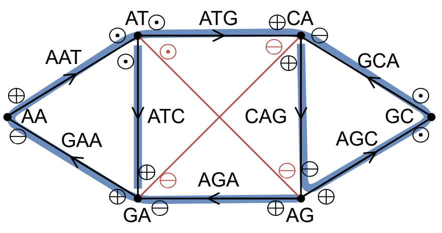
(a) Build the de Bruijn graph of the input strings.



(b) Eulerise the example graph.



(c) Compute a Eulerian cycle.



(d) Break the cycle at the arcs inserted in step (b).

Figure 1 Overview of our algorithm executed on the input strings $\{GAATG, ATCTGCT\}$ with $k = 3$. After step (d), the resulting spelled SPSS is $\{ATC, AGAATGCTG\}$.

For a node $v \in V$, the *multiset of incidences* is defined as $\mathrm{I}(v) := \{vd \mid d \in \{\oplus, \ominus, \odot\}\}$, with multiplicities $\#_{\mathrm{I}(v)}(vd) := \sum_{e \in E} \#_e(vd)$ (treating the arcs as multisets such that self-loops count as two separate incidences). For a node $v \in V$ that is not self-complemental, the *outdegree* is defined as $\delta^+(v) := \#_{\mathrm{I}(v)}(v\oplus)$, and the *indegree* is defined as $\delta^-(v) := \#_{\mathrm{I}(v)}(v\ominus)$. For a self-complemental node $v \in V$, the *degree* is defined as $\delta(v) := \#_{\mathrm{I}(v)}(v\odot)$.

We define the *imbalance* of a node $v \in V$ that is not self-complemental as the difference of its outdegree and indegree $\mathrm{imbalance}(v) := \delta^+(v) - \delta^-(v)$. For a self-complemental node $v \in V$ the imbalance is defined as $\mathrm{imbalance}(v) := 1$ if $\delta(v)$ is odd, and $\mathrm{imbalance}(v) := 0$ otherwise. A node $v \in V$ is called *unbalanced*, if $\mathrm{imbalance}(v) \neq 0$, and *balanced* otherwise.

A *labelled graph* is a bidirected graph $G = (V, E, c)$ where the identifiers of arcs are strings over some alphabet Σ (e.g. $(v_1\oplus, v_2\ominus, ACCTG) \in E$).

2.2 Suffix arrays and suffix trees

Section 4 requires knowledge of suffix arrays and suffix trees. We assume the reader is familiar with these data structures, and briefly give the relevant definitions and properties below. We point the reader to Gusfield [12] and Mäkinen [18] for an in-depth treatment of the topics.

A suffix array SA_T for a string T is an array of length $|T|$ such that $SA_T[i]$ is the starting position of the lexicographically i-th suffix of T. The suffix array interval of a string x is the maximal interval $[i..j]$ such that all the suffixes pointed by $SA_T[i], \dots, SA_T[j]$ have x as a prefix, or the empty interval if x is not a substring of T.

A suffix tree of a string T is a compacted version of the trie of all suffixes of T, such that non-branching paths are merged into single arcs, with arcs pointing away from the root. The compactification concatenates the labels of the arcs on the compacted path. The nodes that were compacted away and are now in the middle of an arc are called implicit nodes, and the rest of the nodes are explicit. A *locus* (plural *loci*) is a node that is either explicit or implicit. A locus v is represented by a pair (u, d), where u is the explicit suffix tree node at the end of the arc containing v (u is equal to v if v is explicit), and d is the depth of locus v in the trie of loci. The suffix array interval of a node is the interval of leaves in the subtree of the node. The suffix array interval of an implicit locus (u, d) is the same as the suffix array interval of u.

The suffix tree can be constructed in linear time in $|T|$ using e.g. Ukkonen's algorithm [24]. The tree comes with a function `child` that takes an explicit node and a character, and returns the child at the end of the arc from that node whose label starts with the given character (if such node exists). This can be implemented in $O(\log |\Sigma|)$ time by binary searching over child pointers sorted by labels. The `child` function can also be easily implemented for implicit loci. Ukkonen's algorithm also produces *suffix links* for the explicit nodes, which map from the suffix tree node of a string cx to the suffix tree node of string x. It is possible to emulate suffix links on the implicit loci using constant-time weighted level-ancestor queries [4] by mapping $(u, d) \mapsto (f_{d-1}(SL(u)), d-1)$, where $SL(u)$ is the destination of a suffix link from u, and $f_{d-1}(SL(u))$ is the furthest suffix tree ancestor from $SL(u)$ at depth at least $d-1$ in the trie of loci. The inverse pointers of suffix links are called *Weiner links*, and they can also be simulated on the implicit loci by mapping $(u, d) \mapsto (WL(u, c), d+1)$, where $WL(u, c)$ is the destination of a Weiner link from u with character c.

3 De Bruijn graphs

Algorithm 1 DeBruijnGraph.

```
Input: An integer $k$ and a set of strings $I$ where each string has length at least $k$.
Output: A de Bruijn graph $G = (V, E, c)$ of order $k$.
1  $V \leftarrow \mathrm{CS}_{k-1}(I)$                    /* the nodes are the canonical $(k-1)$-mers */
2  foreach $v \in V$ do
3      if $\mathrm{rc}(v) = v$ then $c(v) \leftarrow 1$ else $c(v) \leftarrow 0$
4  foreach $\eta \in \mathrm{CS}_k(I)$ do
5      $w \leftarrow \mathrm{pre}_{k-1}(\eta)$                              /* compute $v$ */
6      $v \leftarrow$ canonical $w$
7      $w' \leftarrow \mathrm{suf}_{k-1}(\eta)$                             /* compute $v'$ */
8      $v' \leftarrow$ canonical $w'$
9      if $c(v) = 1$ then $d \leftarrow \odot$               /* compute the direction of $v$ */
10     else if $v = w$ then $d \leftarrow \oplus$
11     else $d \leftarrow \ominus$
12     if $c(v') = 1$ then $d' \leftarrow \odot$             /* compute the direction of $v'$ */
13     else if $v' \neq w'$ then $d' \leftarrow \oplus$       /* note that $\neq$ differs from $=$ above */
14     else $d' \leftarrow \ominus$
15     $e \leftarrow (vd, v'd', \eta)$                      /* insert the arc into the graph */
16     $E \leftarrow E \cup \{e\}$
```

The *de Bruijn graph* of order k of a set of input strings I is defined as a labelled graph constructed by Algorithm 1. See Figure 1 (a) for an example. A de Bruijn graph computed by this algorithm has the following property (see Appendix B for some of the proofs of this section).

▶ **Lemma 2** (Sound labels). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. For all pairs of arcs $e_1 := (v_1 d_1, v_1' d_1', \eta_1), e_2 := (v_2 d_2, v_2' d_2', \eta_2) \in E$ it holds that:*

(a) *($v_1' = v_2$ and $d_1' = \neg d_2$) if and only if* $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$*,*

(b) *($v_1' = v_2'$ and $d_1' = \neg d_2'$) if and only if* $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$*,*

(c) *($v_1 = v_2$ and $d_1 = \neg d_2$) if and only if* $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$*, and*

(d) *($v_1 = v_2'$ and $d_1 = \neg d_2'$) if and only if* $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$*.*

Algorithm 2 SPELL.

```
Input: A de Bruijn graph $G = (V, E, c)$ of order $k$ and a walk
       $W = (e_1 := (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell := (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$.
Output: The string $s$ spelled by $W$, i.e. spell($W$).
1 if $W$ is empty then
2     $s \leftarrow \epsilon$
3 else                                    /* compute the sequence of kmers from $W$ */
4     foreach $i \in (1, \ldots, \ell)$ do            /* iterate the sequence in order */
5         if $e_i \in E$ then $\kappa_i \leftarrow \eta_i$
6         else $\kappa_i \leftarrow \mathrm{rc}(\eta_i)$                          /* $e_i^{-1} \in E$ */
7     $s \leftarrow k - 1$ prefix of $\kappa_1$
8     foreach $i \in (1, \ldots, \ell)$ do            /* iterate the sequence in order */
9         append the last character from $\kappa_i$ to $s$
```

For a walk $W := (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ in a de Bruijn graph, its *sequence of k-mers* is $K := (\kappa_1, \ldots, \kappa_\ell)$, where for each i we define κ_i as η_i if $e_i \in E$, and as $\mathrm{rc}(\eta_i)$ if $e_1^{-1} \in E$. The string spell(W) is the string *spelled* by W, which is defined as its *collapsed* sequence of kmers, i.e. its sequence of k-mers gets concatenated while overlapping consecutive k-mers by $k - 1$. This is computed by Algorithm 2. We prove the following lemmas to show that our definition of the spell($\cdot$) function is sound for our purposes, i.e. correctly spells the string belonging to a walk in a de Bruijn graph.

▶ **Lemma 3** (Sound sequence of k-mers). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let $W := (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ be a walk in G, and $K := (\kappa_1, \ldots, \kappa_\ell)$ its sequence of k-mers. Then for each consecutive pair of kmers κ_i, κ_{i+1} it holds that $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$.*

We define the *sequence of k-mers* $K = (\kappa_1, \ldots, \kappa_\ell)$ of a string $w = (a_1, \ldots, a_{\ell+k-1})$ by $\kappa_i := (a_i, \ldots, a_{i+k-1})$ for each i.

▶ **Lemma 4** (Sound spell). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let W be a walk in G, K_W its sequence of k-mers and K'_W the sequence of k-mers of* spell(W). *Then $K_W = K'_W$.*

▶ **Lemma 5** (Complete representation). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let w be a string with $\mathrm{CS}_k(w) \subseteq \mathrm{CS}_k(I)$. Then there exists a walk W in G with* spell(W) $= w$.

Proof. Let $K_w = (\kappa_1, \ldots, \kappa_\ell)$ be the sequence of k-mers of w. We construct $W = (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ as follows: for each i, let η_i be the canonical of κ_i and $f_i \in E$ be the arc whose identifier is η_i. We set $e_i = f_i$ if κ_i is canonical, and $e_i = f_i^{-1}$ otherwise.

For W to fulfil the definition of a walk we need that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$ for all i. Using Lemma 2, we get:

- If $e_i, e_{i+1} \in E$, then $\mathrm{suf}_{k-1}(\eta_i) = \mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1}) = \mathrm{pre}_{k-1}(\eta_{i+1})$. Therefore, by Lemma 2 a, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.

- If $e_i, e_{i+1}^{-1} \in E$, then $\mathrm{suf}_{k-1}(\eta_i) = \mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1}) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_{i+1}))$. Therefore, by Lemma 2 b, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.
- If $e_i^{-1}, e_{i+1} \in E$, then $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_i)) = \mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1}) = \mathrm{pre}_{k-1}(\eta_{i+1})$. Therefore, by Lemma 2 c, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.
- If $e_i^{-1}, e_{i+1}^{-1} \in E$, then $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_i)) = \mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1}) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_{i+1}))$. Therefore, by Lemma 2 d, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.

To complete the proof we need to show that $\mathrm{spell}(W) = w$. By definition, the sequence of k-mers K_W of W is equivalent to K_w. And since W is a walk, by Lemma 4 we get that the sequence of k-mers of $\mathrm{spell}(W)$ is equivalent to K_W, and therefore $\mathrm{spell}(W) = w$. ◀

A *walk cover* $\mathcal{W}$ of a bigraph G is a set of walks such that for each arc $e \in E$ it holds that e is part of some walk $W \in \mathcal{W}$, or e^{-1} is part of some walk $W \in \mathcal{W}$.

▶ **Theorem 6** (Dualism between SPSS and walk cover). *Let k be a positive integer and let I and I' be sets of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Then it holds that $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$, if and only if there is a walk cover $\mathcal{W}$ in G that spells the strings in I'.*

Proof. If $\mathrm{CS}_k(I') \subseteq \mathrm{CS}_k(I)$, then for each string $w' \in I'$ it holds that $\mathrm{CS}_k(w') \subseteq \mathrm{CS}_k(I)$. Therefore, by Lemma 5, there exists a walk w in G with $\mathrm{spell}(w) = w'$. Then, the set of all such walks $\mathcal{W}$ spells I'. Further, because $\mathrm{CS}_k(I) \subseteq \mathrm{CS}_k(I')$, the identifier η of each arc $e \in E$ is in $\mathrm{CS}_k(I')$, and therefore in the sequence of kmers $K_{w'}$ of some string $w' \in I'$ (possibly as a reverse complement). By Lemma 4 it holds that $K_{w'} = K_w$, where K_w is the sequence of k-mers of walk w. By the definition of the sequence of k-mers of a walk, this implies that w visits e (possible in reverse direction). Since this holds for each $e \in E$, it holds that $\mathcal{W}$ is a walk cover of G.

Assume that there is a walk cover $\mathcal{W}$ in G that spells the strings in I', and let $w \in \mathcal{W}$ be a walk, K_w its sequence of k-mers, $w' := \mathrm{spell}(w)$ and $K_{w'}$ the sequence of k-mers of w'. Then, by Lemma 4, $K_w = K_{w'}$, which, by the definition of the sequence of k-mers of a walk implies that $\mathrm{CS}_k(I) \subseteq \mathrm{CS}_k(I')$. And since $\mathcal{W}$ is a walk cover of G, we get $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$. ◀

▶ **Corollary 7.** *By setting $I = I'$ in Theorem 6 we can confirm that our definition of a de Bruijn graph is sound in that there is a set of walks that spells the strings used for its construction.*

A *compacted* de Bruijn graph is constructed from a de Bruijn graph by contracting all nodes $v \in V$ that are either self-complemental and have exactly two arcs that have exactly one incidence to v each, or that are not self-complemental and have exactly one incoming and one outgoing arc. For simplicity, we use uncompacted de Bruijn graphs in our theoretical sections, however all results equally apply to compacted de Bruijn graphs.

4 Linear-time construction of compacted bidirected de Bruijn graphs

In this section, we fill a gap in the literature by describing on a high level an algorithm to construct the bidirectional de Bruijn graph of a set of input strings in time linear in the total length of the input strings, independent of the value of k.

4.1 Algorithm

Let $I = \{w_1, \ldots w_m\}$ be the set of input strings. Consider the following concatenation:

$$T = \$w_1\$w_2\$ \ldots \$w_m\$\,\mathrm{rc}(w_1)\$\,\mathrm{rc}(w_2)\$ \ldots \$\,\mathrm{rc}(w_m)\$ \;,$$

where \$ is a special character outside of the alphabet Σ of the input strings. We require an index on T that can answer the following queries: `extendRight`, `extendLeft`, `contractRight` and `contractLeft` in constant time. The extension operations take as input a character $c \in \Sigma$ and the interval of a string x in the suffix array of T, and return the suffix array intervals of xc in the case of `extendRight` and cx in the case of `extendLeft`. The contraction operations are the inverse operations of these, mapping the suffix array intervals of xc to x in the case of `contractRight` and cx to x in the case of `contractLeft`. For efficiency, we also require operations `enumerateRight` and `enumerateLeft`, which take a string x and give all characters such that `extendRight` and `extendLeft` respectively return a non-empty interval, in time that is linear in the number of such characters. Implementations for all the six subroutines are given in Section 4.2.

Using these operations, we can simulate the regular non-bidirected de Bruijn graph of T. Each k-mer of the input strings for a fixed k corresponds to a disjoint interval in the suffix array of T. The nodes are represented by their suffix array intervals. The outgoing arcs from a $(k-1)$-mer x are those characters c where `extendRight`(x, c) returns a non-empty interval. We can enumerate all the characters c with this property in constant time using `enumerateRight`(x). The incoming arcs can be enumerated symmetrically with the `enumerateLeft`(x). Finally, we can find the destination or origin of an arc labelled with x by running a `contractLeft` or `contractRight` operation respectively on x.

To construct the bidirected de Bruijn graph, we merge together nodes that are the reverse complement of each other. To find which nodes are complemental, we scan the input strings I while maintaining the suffix array interval of the current k-mer using `extendRight` and `contractLeft` operations, while at the same time maintaining the suffix array interval of the reverse complement using `extendLeft` and `contractRight` operations. Whenever we merge two nodes, we combine the incoming and outgoing arcs, assigning the incidences of the arcs according to the incidence rules in our definition. We are able to tell in constant time which k-mer of a pair of complemental k-mers is canonical by comparing the suffix array intervals of the k-mers: the k-mer whose suffix array interval has a smaller starting point is the canonical k-mer. If the starting points are the same, the k-mer is self-complemental.

Using the `enumerateRight` and `enumerateLeft` functions, we can check if a node would be contracted in a compacted de Bruijn graph. By extending k-mers over such nodes, we can in linear time also output only the arcs and nodes of a compacted de Bruijn graph. For storing the labels, we use one pointer into the input strings to store a single k-mer, as well as a flag that is set whenever the label is not canonical. If a label has multiple k-mers, then we store the remaining k-mers as explicit strings, however without their overlap with the "pointer-k-mer". This way, we can store each label in $O(\ell)$ space, where ℓ is the number of k-mers in the label. We additionally store the first and last character of each label, as an easy way to make the SPELL function run in output sensitive linear time.

4.2 Implementation of the subroutines

All required the subroutines `extendRight`, `extendLeft`, `contractRight`, `contractLeft`, `enumerateRight` and `enumerateLeft` can be implemented with the suffix tree of T by simulating the trie of the suffix tree loci as described in Section 2.2. The suffix array intervals of explicit nodes can be stored with the nodes, so that we can operate on loci (u, d) and retrieve the suffix array intervals on demand. The operation `extendRight` follows an arc from a locus to a child, and the operation `contractRight` is implemented by going to the parent of the current locus. The operation `contractLeft` follows a suffix link from the current locus, and `extendLeft` follows a Weiner link. The operations `enumerateRight` and `enumerateLeft` are implemented by storing the children and the Weiner links from explicit suffix tree nodes as neighbor lists. The total number of these links is linear in $|T|$ [18]. With this implementation, the slowest operations are `extendRight` and `extendLeft`, taking $O(\log |\Sigma|)$ time to binary search the neighbor lists. We therefore obtain the following result:

▶ **Theorem 8.** *The compacted arc-centric bidirected de Bruijn graph of order k of a set of input strings I from the alphabet Σ can be constructed in time $O(||I|| \log |\Sigma|)$.*

We note that the same operations can also be implemented on top of the bidirectional BWT index of Belazzougui and Cunial [2], using the data structures of Belazzougui et al. [3] for the enumeration operations. This gives an index that supports all the required subroutines in *constant time*. The drawback of the bidirectional BWT index is that only randomized construction algorithms are known, but the expected time is still linear in $|T|$. We leave as an open problem the construction of the compacted arc-centric bidirected de Bruijn graph in deterministic linear time independent of the alphabet size.

5 Linear-time minimum SPSS without repetitions

Let I be a set of strings. To compute an SPSS without repetitions we first build a compacted de Bruijn graph G from I. Because of Theorem 6, finding an SPSS is equivalent to finding a walk cover in G. Further, with Lemma 4, we get that an SPSS without repetitions is equivalent to a walk cover that visits each arc exactly once (either once forwards, or once reverse, but not both forwards and reverse). We call such a walk cover a *unique walk cover*.

For minimality, observe that the cumulative length of an SPSS S relates to its equivalent set of walks $\mathcal{W}$ as follows:

$$||S|| = \sum_{W \in \mathcal{W}} (k - 1 + |W|) \tag{1}$$

This is because in Algorithm 2, in Line 7, $k - 1$ characters are appended to the result, and then in the loop in Line 8, one additional character per arc in W is appended. We cannot alter the sum $\sum_{W \in \mathcal{W}} |W|$, since we need to cover all arcs in G. However we can alter the number of strings, and decreasing or increasing this number by one will decrease or increase the cumulative length of S by $k - 1$. Therefore, finding a minimum SPSS of I without repetitions equals finding a unique walk cover of G that has a minimum number of walks.

Note that computing a minimum SPSS in a bigraph that is not connected is equivalent to separately computing an SPSS in each maximal connected subgraph. Therefore we restrict to connected bigraphs from here on.

5.1 A lower bound for an SPSS without repetitions

Using the imbalance of the nodes of a bigraph, we can derive a lower bound for the number of walks in a walk cover.

▶ **Lemma 9.** *Let $v \in V$ be an unbalanced node in a bigraph $G = (V, E, c)$. Then in a unique walk cover $\mathcal{W}$ of G, either at least $|\text{imbalance}(v)|$ walks start in v, or at least $|\text{imbalance}(v)|$ walks end in v.*

Proof. If v is self-complemental, then its imbalance is 1, so by definition v has an odd number of incident arcs. Each walk that does not start or end in v needs to enter and leave v via two distinct arcs whenever it visits v. But since the number of incident arcs is odd, there is at least one arc that cannot be covered this way, implying that a walk needs to start or end in this arc.

If v is not self-complemental and has a positive imbalance, then it has imbalance(v) more outgoing arcs then incoming arcs. Since walks need to leave v with the opposite sign than they entered v, at least imbalance(v) arcs cannot be covered by walks that do not start or end in v. If v has negative imbalance, the situation is symmetric. ◀

▶ **Definition 10** (Imbalance of a bigraph). *The imbalance* imbalance(G) *of a bigraph* $G = (V, E, c)$ *is the sum of the absolute imbalance of all nodes* $\sum_{v \in V} |\,\text{imbalance}(v)|$.

▶ **Theorem 11** (Lower bound). *Let G be a bigraph. A walk cover $\mathcal{W}$ of G has a minimum string count of* imbalance$(G)/2$.

Proof. Let $v \in V$ be an unbalanced node. Then, by Lemma 9 at least $|\,\text{imbalance}(v)|$ walks start in v or at least $|\,\text{imbalance}(v)|$ walks end in v. Since each walk has exactly one start node and one end node, $\mathcal{W}$ has a minimum string count of imbalance$(G)/2$. ◀

5.2 Eulerising a bigraph

Algorithm 3 EULERISE.

```
Input: Bigraph $G = (V, E, c)$.
Output: Eulerised bigraph $G' = (V, E', c)$.
1  $G' \leftarrow G$                                  /* $G$ and $G'$ share $V$ and $c$ */
2  $L \leftarrow$ empty list           /* collect missing incidences to balance $G'$ */
3  foreach $v \in V$ do
4      $i \leftarrow$ imbalance$(v)$
5      if $c(v) = 1$ then
6          if $i \neq 0$ then append $v\odot$ to $L$
7      else
8          if $i > 0$ then append $i$ copies of $v\ominus$ to $L$
9          if $i < 0$ then append $i$ copies of $v\oplus$ to $L$
10 while $|L| > 0$ do                      /* insert missing incidences as arcs */
11     $vd \leftarrow$ remove the first incidence from $L$
12     $v'd' \leftarrow$ remove the first incidence from $L$
13     insert 1 arc $(vd, v'd', |L|)$ into $E'$          /* use distinct identifiers */
```

A directed graph is called *Eulerian*, if all nodes have indegree equal to outdegree, i.e. are balanced [10]. If the graph is strongly connected[3], then this is equivalent to the graph admitting a *Eulerian cycle*, i.e. a cycle that visits each arc exactly once. The same notion can be used with bidirected graphs, using our definition of imbalance.

▶ **Definition 12** (Eulerian bigraph). *A bigraph is* Eulerian, *if all nodes have imbalance zero.*

A connected bigraph can be transformed into a Eulerian bigraph by adding arcs using Algorithm 3. See Figure 1 (b) for an example.

[3] Strongly connected means that there is a directed path from each node v_1 to each node v_2.

▶ **Lemma 13.** *The imbalance of a bigraph is even.*

Proof. Adding or removing an arc changes the imbalance of two nodes by 1, or of one node by two. In both cases, the imbalance of the graph can only change by -2, 0, or 2. Since the imbalance of a graph without arcs is 0, this implies that there can be no graph with odd imbalance. ◀

▶ **Lemma 14.** *Given a connected bigraph $G = (V, E, c)$, Algorithm 3 outputs a Eulerian bigraph $G' = (V, E', c)$.*

Proof. Algorithm 3 is well-defined, since by Lemma 13, it holds that L has even length in each iteration of the loop in Line 10, so the removal operation in Line 12 always has something to remove.

The output of Algorithm 3 is a valid bigraph, since for self-complemental nodes $v \in V$, only incidences $v\odot$ are added to G', and for not self-complemental nodes $v \in V$, only incidences $v\oplus$ and $v\ominus$ are added to G'.

Further, the output is a Eulerian bigraph, because for all $v \in V$, it holds that imbalance(v) is 0, by the following argument:

- If $c(v) = 1$ and v has imbalance zero in G, then its imbalance stays the same in G'. If it has imbalance 1, then one incident arc is inserted, making its degree even and its imbalance therefore zero.
- If $c(v) = 0$ and v has positive imbalance i in G, then i incoming arcs are added to v (counting incoming self-loops twice), and no outgoing arcs are added. Therefore, it has imbalance zero in G'. By symmetry, if v has negative imbalance in G, it has imbalance zero in G'. ◀

▶ **Lemma 15.** *Given a bigraph $G = (V, E, c)$, Algorithm 3 terminates after $O(|V| + |E|)$ steps.*

Proof. For the list data structure we choose a doubly linked list, and for the graph an adjacency list (and array with an entry for each node containing a doubly linked list for the arcs).

The loop in Line 3 runs $|V|$ times and each iteration runs in $O(|\operatorname{imbalance}(v)|)$ for a node v, because a doubly linked list supports appending in constant time. The sum of absolute imbalances of all nodes cannot exceed $2|E|$, because each arc adds at most 1 to the absolute imbalance of at most two nodes, or adds at most 2 to the absolute imbalance of at most one node. Therefore, the length of list L after completing the loop is at most $2|E|$, and the loop runs in $O(|V| + |E|)$ time.

The loop in Line 10 runs at most $|L| \leq 2|E|$ times and performs only constant-time operations, since L is a doubly linked list and we can insert arcs into an adjacency list in constant time. Therefore, this loop also runs in $O(|V| + |E|)$ time. ◀

With Lemmas 14 and 15 we get the following.

▶ **Theorem 16.** *Algorithm 3 is correct and runs in $O(|V| + |E|)$ time.*

5.3 Computing a Eulerian cycle in a bigraph

After Eulerising the bigraph, we can compute a Eulerian cycle using Algorithm 4. We do this similarly to Hierholzer's classic algorithm for Eulerian cycles [10]. First we find an arbitrary cycle. Then, as long as there are unused arcs left, we search along the current cycle for unused arcs, and find additional cycles through such unused arcs. We integrate each of those additional cycles into the main cycle. See Figure 1 (c) for an example of a Eulerian cycle.

Algorithm 4 EulerianCycle.

```
   Input: Connected Eulerian bigraph $G = (V, E, c)$.
   Output: Eulerian cycle $W$.
1  while $|E| > 0$ do
2      if $|W| = 0$ then
3          $(vd, v'd', \eta) \leftarrow$ remove some arc from $E$
4          $W' \leftarrow ((vd, v'd', \eta))$                    /* doubly linked list */
5      else                  /* search a used arc that connects to an unused arc */
6          $(vd, v'd', \eta) \leftarrow$ dereference $first\_unfinished$
7          while $E$ has no arc with incidence $v'\neg d'$ do
8              advance $first\_unfinished$ to the next arc in $W$
9              $(vd, v'd', \eta) \leftarrow$ dereference $first\_unfinished$
       // extend $W'$ without repeating arcs until it closes a cycle
10     while $E$ contains an arc $e = (v_e d_e, v'_e d'_e, \eta_e)$ with incidence $v'\neg d'$ do
11         remove $e$ from $E$
12         if $v_e d_e = v'\neg d'$ then $(vd, v'd', \eta) \leftarrow (v_e d_e, v'_e d'_e, \eta_e)$
13         else $(vd, v'd', \eta) \leftarrow (v'_e d'_e, v_e d_e, \eta_e)$          /* $v'_e d'_e = v'\neg d'$ */
14         append $(vd, v'd', \eta)$ to $W'$
15     if $|W| = 0$ then
16         $W \leftarrow W'$
17         $first\_unfinished \leftarrow$ pointer to the first arc in $W$
18     else
19         insert $W'$ after $first\_unfinished$ in $W$
20         $W' \leftarrow ()$                              /* empty doubly linked list */
```

▶ **Lemma 17.** *Given a connected Eulerian bigraph $G = (V, E, c)$, Algorithm 4 terminates and outputs a Eulerian cycle W.*

Proof. For $W = (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ to be a Eulerian cycle, it must be a cycle that contains each arc exactly once.

The sequence W' constructed by the loop in Line 10 is a walk by construction, and since G is Eulerian it is a cycle after the loop terminates. After finding the initial cycle in the first iteration of the outer loop, each additional cycle is started from a node on the initial cycle, and is a cycle again. Therefore it can be inserted into the original cycle without breaking its cycle property.

Since each arc is deleted when being added to W', there is no duplicate arc in W. And if the algorithm terminates, then $|E| = 0$ (Line 1), so W contains all arcs.

For termination, consider that if W is not complete after the first iteration of the outer loop, then the loop in Line 7 searches for an unused arc using the $first\_unfinished$ pointer. Since the prefix of W up to including $first\_unfinished$ is never modified (Line 19), and $first\_unfinished$ is only advanced when its pointee cannot reach any arc anymore, it holds that no arc in W can reach an arc in E when $first\_unfinished$ gets advanced over the end of W. Since G was initially Eulerian and only Eulerian cycles have been removed from G, this implies that all nodes visited by W are still balanced and therefore have no incident arcs anymore. And since G was originally connected, W has visited all nodes, i.e. $|E| = 0$. Therefore, $first\_unfinished$ cannot be advanced over the end of W, because the outer loop terminates before that.

To complete the proof of termination, consider that in each iteration of the outer loop, at least one arc gets removed from E. In the first iteration, this happens at least in Line 3, and in all following iterations, this happens in Line 11. ◀

▶ **Lemma 18.** *Given a connected Eulerian bigraph $G = (V, E, c)$, Algorithm 4 terminates after $O(|V| + |E|)$ steps.*

Proof. We use a doubly linked list for W and W', and an adjacency list for G. Then all lines can be executed in constant time.

The loop in Line 10 removes one arc from E each iteration, so it runs at most $|E|$ times in total (over all iterations of the outer loop). The loop in Line 7 advances $first\_unfinished$ each iteration. Since the algorithm is correct by Lemma 17, $|W| \leq |E|$ and $first\_unfinished$ never runs over the end of $first\_unfinished$, so the loop runs at most $|E|$ times in total (over all iterations of the outer loop).

The condition for the loop in Line 10 is true at least once in each iteration of the outer loop, since the preceding branch sets up $(vd, v'd', \eta)$ such that it has a successor (in the first iteration because of Eulerianess). So in each iteration of the outer loop, at least one arc gets removed, so the outer loop runs at most $|E|$ times in total.

As a result, all loops individually run at most $|E|$ times, therefore Algorithm 4 terminates after $O(|V| + |E|)$ steps. ◀

With Lemmas 17 and 18 we get the following.

▶ **Theorem 19.** *Algorithm 4 is correct and runs in $O(|V| + |E|)$ time.*

5.4 Computing a minimum SPSS without repetitions

We convert the Eulerian cycle into a walk cover of the original bigraph by breaking it at all arcs inserted by Algorithm 3, and removing those arcs (see Figure 1 (d) for an example). This results in a walk cover with either one walk, if Algorithm 3 inserted zero or one arcs, or imbalance$(G)/2$ arcs, if Algorithm 3 inserted more arcs. By Theorem 11, this is a minimum number of walks, and therefore the SPSS spelled by these walks is minimum as well. Constructing the de Bruijn graph takes $O(||I|| \log \Sigma)$ time, and it has $O(||I||)$ k-mers, so it holds that $|V| \in O(||I||)$ and $|E| \in O(||I||)$. Further, spelling the walk cover takes time linear to the cumulative length of the spelled strings. Since we compute a minimum representation, it holds that the output is not larger than the total length of the input strings. Therefore we get:

▶ **Theorem 1.** *Let k be a positive integer and let I be a set of strings of length at least k over some alphabet Σ. Then we can compute a set of strings I' of length at least k with minimum cumulative length and $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$ in $O(||I|| \log |\Sigma|)$ time.*

6 Experiments

We ran our experiments on a server running Linux with two 64-core AMD EPYC 7H12 processors with 2 logical cores per physical core, 1.96TiB RAM and an SSD. Our data sets are the same as in [23], and we also adapted their metrics *cumulative length* (CL), which is the total count of characters in all strings, and *string count* (SC), which is the number of strings. Our implementation does not use the formalisation of bidirected graphs introduced in this work, but instead uses the formalisation from [23]. For constructing de Bruijn graphs, we do not implement our purely theoretical linear time algorithm, since practical de Bruijn

Table 1 Experiments on references and read sets of single genomes with $k = 51$ and a min abundance of 10 for human and 1 for the others. The CL and SC ratios are compared to the CL-optimal Eulertigs. For time and memory, we report the total time and maximum memory required to compute the tigs from the respective data set. BCALM2 directly computes unitigs, while UST- and Eulertigs require a run of BCALM2 first before they can be computed themselves. Prophasm can only be run for $k \leq 32$, which does not make sense for large genomes. The number in parentheses behind time and memory indicates the slowdown/increase over computing just unitigs with BCALM2. BCALM2 was run with 28 threads, while all other tools support only one thread. The lengths of the genomes are 100Mbp for C. elegans, 482Mbp for B. mori and 3.21Gbp for H. sapiens and the read data sets have a coverage of 64x for C. elegans, 58x for B. mori and 300x for H. sapiens.

genome	algorithm	CL ratio	SC ratio	time [s]		memory [GiB]	
C. elegans (reads)	unitigs	1.789	2.831	1888		5.97	
	UST	1.035	1.080	2738	(1.45)	15.2	(2.54)
	Eulertigs	1	1	3735	(1.98)	25.0	(4.19)
B. mori (reads)	unitigs	1.912	3.136	7737		9.36	
	UST	1.050	1.118	10937	(1.41)	52.4	(5.60)
	Eulertigs	1	1	13793	(1.78)	79.4	(8.48)
H. sapiens (reads)	unitigs	1.418	2.143	56966		13.0	
	UST	1.016	1.044	57736	(1.01)	16.4	(1.26)
	Eulertigs	1	1	58861	(1.03)	29.2	(2.25)
C. elegans	unitigs	1.060	3.154	54.7		1.22	
	UST	1.002	1.089	58.0	(1.06)	1.22	(1.00)
	Eulertigs	1	1	65.9	(1.21)	1.22	(1.00)
B. mori	unitigs	1.262	3.310	224		3.32	
	UST	1.018	1.156	258	(1.16)	3.32	(1.00)
	Eulertigs	1	1	315	(1.41)	3.32	(1.00)
H. sapiens	unitigs	1.195	3.532	3166		10.0	
	UST	1.015	1.192	3369	(1.06)	10.0	(1.00)
	Eulertigs	1	1	3717	(1.17)	10.0	(1.00)

graph construction is a well-researched field [8, 13, 15, 9, 20, 19], and we want to focus more on computing the compressed representation from unitigs. UST only supports unitigs constructed by BCALM2 [8], since it needs certain additional data. BCALM2 is not a linear time algorithm, but works efficient in practice. Therefore, we use BCALM2 to construct a node-centric de Bruijn graph, and then convert it to an arc-centric variant using a hash table.

Our experimental pipeline is constructed with [16] and using the bioconda software repository [11]. We ran all multithreaded tools with up to 28 threads and never used more than 128 cores of our machine at once to prevent hyperthreading from affecting our timing. The code to reproduce our experiments is available at `https://doi.org/10.5281/zenodo.6538261`.

The performance figures are all very similar, with two exceptions. Prophasm does not support parallel computation at the moment, therefore its runtime is much higher. Compared to that, all other algorithms use parallel computation to compute unitigs, but computing

Table 2 Experiments on (references of) pangenomes with $k = 31$ and a min abundance of 1. The CL and SC ratios are compared to the CL-optimal Eulertigs. For time and memory, we report the total time and maximum memory required to compute the tigs from the respective data set. BCALM2 directly computes unitigs, while UST- and Eulertigs require a run of BCALM2 first before they can be computed themselves. Prophasm is run directly on the source data. The number in parentheses behind time and memory indicates the slowdown/increase over computing just unitigs with BCALM2. BCALM2 was run with 28 threads, while all other tools support only one thread. The N. gonorrhoeae pangenome contains 8.36 million unique kmers, the S. pneumoniae pangenome contains 19.3 million unique kmers and the E. coli pangenome contains 341 million unique kmers.

pangenome	tigs	CL ratio	SC ratio	time [s]		memory [MiB]	
1102x N. gonorrhoeae	unitigs	1.615	3.052	24.2		4328	
	UST	1.022	1.074	26.1	(1.08)	4328	(1.00)
	prophasm	1.00004	1.00013	774	(31.9)	208	(0.05)
	Eulertigs	1	1	26.9	(1.11)	4328	(1.00)
616x S. pneumoniae	unitigs	1.679	3.055	21.4		3135	
	UST	1.027	1.081	25.9	(1.21)	3135	(1.00)
	prophasm	1.00004	1.00012	436	(20.3)	434	(0.14)
	Eulertigs	1	1	28.5	(1.33)	3135	(1.00)
3682x E. coli	unitigs	1.705	3.092	334		7146	
	UST	1.031	1.092	416	(1.24)	7146	(1.00)
	prophasm	1.00008	1.00023	7456	(22.3)	7221	(1.01)
	Eulertigs	1	1	471	(1.41)	7146	(1.00)

the final tigs from unitigs seems to be negligible compared to computing the de Bruijn topology. Moreover, running UST or Eulertigs on read data sets of larger genomes consumes significantly more memory than computing just unitigs. This is likely because BCALM2 uses external memory to compute unitigs, while the other tools simply load the whole set of unitigs into memory.

It is notable that the Eulertigs algorithm is always slower than UST. This may be because of the Eulertig algorithm being more complex, but also because our loading and storing routines might not be as efficient. While UST uses node-centric de Bruijn graphs and can therefore directly make use of the topology output by BCALM2 (which is a fasta file with arcs stored as custom annotations), we need to convert the graph into arc-centric format. This is supported by e.g. the B. mori short read data set, on which the computation of Eulertigs uses only 11% of the runtime for the algorithm itself, while 89% are from loading the graph (including the conversion to arc-centric) and storing the result.

In terms of CL, we see that the SPSS computed with UST mostly remains within the expected 3% of the lower bound, but they are up to 5% above the lower bound on more compressible data sets. The SPSS computed by prophasm is very close to the optimum in all cases, and we assume that this difference in quality is because prophasm extends paths both forwards and backwards, while the UST heuristic merely extends them forwards.

Looking at SC, we see that Eulertigs are always the lowest, which is due to the string count directly being connected to the cumulative length by Equation (1). This also explains the correlation between CL and SC, which can be observed in all cases.

7 Conclusions

We have presented a linear and hence optimal algorithm for computing a minimum SPSS without repetitions for a fixed alphabet size. This closes the open question about its complexity raised in [6, 22]. Using our optimal algorithm, we were able to accurately evaluate the existing heuristics and show that they are very close to the optimum in practice. Further, we have published our algorithm as a command-line tool on github, allowing it to easily be used in other projects.

Further, we have presented how bidirected de Bruijn graphs can be formalised without excluding any corner cases. We have also shown how such a graph can be constructed in linear time for a fixed-size alphabet. The construction of the compacted arc-centric bidirected de Bruijn graph in linear time independent of the alphabet size stays an open problem.

References

1 Anton Bankevich, Andrey V Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A Pevzner. Multiplex de bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature biotechnology*, pages 1–7, 2022.

2 Djamal Belazzougui and Fabio Cunial. Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

3 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *European Symposium on Algorithms*, pages 133–144. Springer, 2013.

4 Djamal Belazzougui, Dmitry Kosolobov, Simon J Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching*, 2021.

5 Jacek Blazewicz, Alain Hertz, Daniel Kobler, and Dominique de Werra. On some properties of dna graphs. *Discrete Applied Mathematics*, 98(1-2):1–19, 1999.

6 Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biology*, 22(1):1–24, 2021.

7 Bastien Cazaux, Thierry Lecroq, and Eric Rivals. From indexing data structures to de Bruijn graphs. In *Symposium on combinatorial pattern matching*, pages 89–99. Springer, 2014.

8 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.

9 Victoria G Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. Practical dynamic de bruijn graphs. *Bioinformatics*, 34(24):4189–4195, 2018.

10 Herbert Fleischner. *Eulerian graphs and related topics*. Elsevier, 1990.

11 Björn Grüning, Ryan Dale, Andreas Sjödin, Brad A Chapman, Jillian Rowe, Christopher H Tomkins-Tinch, Renan Valieris, and Johannes Köster. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nature Methods*, 15(7):475–476, 2018.

12 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.

13 Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*, 21(1):1–20, 2020.

14 Marta Kasprzak. Classification of de Bruijn-based labeled digraphs. *Discrete Applied Mathematics*, 234:86–92, 2018. Special Issue on the Ninth International Colloquium on Graphs and Optimization (GO IX), 2014. doi:10.1016/j.dam.2016.10.014.

15 Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2. *bioRxiv*, 2021.

16 Johannes Köster and Sven Rahmann. Snakemake – A scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

17 Vamsi Kundeti, Sanguthevar Rajasekaran, and Heiu Dinh. An efficient algorithm for chinese postman walk on bi-directed de bruijn graphs. In Weili Wu and Ovidiu Daescu, editors, *Combinatorial Optimization and Applications*, pages 184–196, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

18 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

19 Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, 2019.

20 Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.

21 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *bioRxiv*, 2022. `doi:10.1101/2022.01.15.476199`.

22 Amatur Rahman and Paul Medevedev. Representation of k-mer sets using spectrum-preserving string sets. *Journal of Computational Biology*, 28(4):381–394, 2021.

23 Sebastian Schmidt, Shahbaz Khan, Jarno Alanko, and Alexandru I Tomescu. Matchtigs: minimum plain text representation of kmer sets. *bioRxiv*, 2021.

24 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

A Authors contributions

JNA and SS discovered the problem, SS solved the problem when the de Bruijn graph is given and wrote most of the manuscript, JNA designed the linear-time de Bruijn graph construction algorithm and wrote Section 4. SS implemented the algorithm and conducted and evaluated the experiments. All authors reviewed and approved the final version of the manuscript.

B Omitted proofs

▶ **Lemma 2** (Sound labels). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. For all pairs of arcs $e_1 := (v_1d_1, v'_1d'_1, \eta_1), e_2 := (v_2d_2, v'_2d'_2, \eta_2) \in E$ it holds that:*

(a) *$(v'_1 = v_2$ and $d'_1 = \neg d_2)$ if and only if $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$,*

(b) *$(v'_1 = v'_2$ and $d'_1 = \neg d'_2)$ if and only if $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$,*

(c) *$(v_1 = v_2$ and $d_1 = \neg d_2)$ if and only if $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$, and*

(d) *$(v_1 = v'_2$ and $d_1 = \neg d'_2)$ if and only if $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$.*

Proof. Observe that the values of w and w' computed in Lines 5 and 7 of Algorithm 1 are equal to $\mathrm{pre}_{k-1}(\eta_1)$ and $\mathrm{suf}_{k-1}(\eta_1)$ for e_1 and equal to $\mathrm{pre}_{k-1}(\eta_2)$ and $\mathrm{suf}_{k-1}(\eta_2)$ for e_2. Further, observe that the values of v and v' computed in Lines 6 and 8 are equal to v_1 and v'_1 for e_1 and equal to v_2 and v'_2 for e_2. This makes v_1, v'_1, v_2 and v'_2 the canonicals of $\mathrm{pre}_{k-1}(\eta_1)$, $\mathrm{suf}_{k-1}(\eta_1)$, $\mathrm{pre}_{k-1}(\eta_2)$ and $\mathrm{suf}_{k-1}(\eta_2)$. Finally, observe that the sign values d and d' computed in Lines 9–14 are equal to d_1 and d'_1 for e_1 and equal to d_2 and d'_2 for e_2.

(a) If $v'_1 = v_2$ and $d'_1 = \neg d_2$, then $w'_1 = w_2$ for all possible values of d'_1, and therefore $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$.
If $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$, then $w'_1 = w_2$, and therefore $v'_1 = v_2$ because v'_1 and v_2 are the canonicals of w'_1 and w_2. Additionally, $d'_1 = \neg d_2$ for all possible values of d'_1.

(b) If $v'_1 = v'_2$ and $d'_1 = \neg d'_2$, then $w'_1 = \mathrm{rc}(w'_2)$ for all possible values of d'_1, and therefore $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{rc}(\mathrm{suf}_{k-1}(\eta_2)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$.
If $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$, then $w'_1 = \mathrm{rc}(w'_2)$, and therefore $v'_1 = v'_2$ because v'_1 and v'_2 are the canonicals of w'_1 and w'_2. Additionally, $d'_1 = \neg d'_2$ for all possible values of d'_1.

(c) If $v_1 = v_2$ and $d_1 = \neg d_2$, then $\mathrm{rc}(w_1) = w_2$ for all possible values of d_1, and therefore $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{rc}(\mathrm{pre}_{k-1}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$.
If $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$, then $w_1 = \mathrm{rc}(w_2)$, and therefore $v_1 = v_2$ because v_1 and v_2 are the canonicals of w_1 and w_2. Additionally, $d_1 = \neg d_2$ for all possible values of d_1.
(d) This case is equivalent to the first case when swapping e_1 and e_2, because $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2) \iff \mathrm{suf}_{k-1}(\mathrm{rc}(\eta_2)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_1))$. ◀

▶ **Lemma 3** (Sound sequence of k-mers). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let $W := (e_1 = (v_1d_1, v'_1d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ be a walk in G, and $K := (\kappa_1, \ldots, \kappa_\ell)$ its sequence of k-mers. Then for each consecutive pair of kmers κ_i, κ_{i+1} it holds that $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$.*

Proof. Let $i \in \{1, \ldots, \ell-1\}$. By the definition of *walk* it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d_{i+1}$. We can apply Lemma 2 case by case.
(a) If $e_i, e_{i+1} \in E$, then by Lemma 2 a, it holds that $\mathrm{suf}_{k-1}(\eta_i)$ equals $\mathrm{pre}_{k-1}(\eta_{i+1})$. By definition, $\kappa_i = \eta_i$ and $\kappa_{i+1} = \eta_{i+1}$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$.
(b) If $e_i, e_{i+1}^{-1} \in E$, then by Lemma 2 b applied to e_i, e_{i+1}^{-1}, it holds that $\mathrm{suf}_{k-1}(\eta_i)$ equals $\mathrm{pre}_{k-1}(\mathrm{rc}(\eta_{i+1}))$. By definition, $\kappa_i = \eta_i$ and $\kappa_{i+1} = \mathrm{rc}(\eta_{i+1})$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$
(c) If $e_i^{-1}, e_{i+1} \in E$, then by Lemma 2 c applied to e_i^{-1}, e_{i+1}, it holds that $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_i))$ equals $\mathrm{pre}_{k-1}(\eta_{i+1})$. By definition, $\kappa_i = \mathrm{rc}(\eta_i)$ and $\kappa_{i+1} = \eta_{i+1}$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$.
(d) If $e_i^{-1}, e_{i+1}^{-1} \in E$, then by Lemma 2 d applied to e_i^{-1}, e_{i+1}^{-1}, it holds that $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_i))$ equals $\mathrm{pre}_{k-1}(\mathrm{rc}(\eta_{i+1}))$. By definition, $\kappa_i = \mathrm{rc}(\eta_i)$ and $\kappa_{i+1} = \mathrm{rc}(\eta_{i+1})$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$. ◀

▶ **Lemma 4** (Sound spell). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let W be a walk in G, K_W its sequence of k-mers and K'_W the sequence of k-mers of* $\mathrm{spell}(W)$*. Then $K_W = K'_W$.*

Proof. Let $(\kappa_1, \ldots, \kappa_\ell) := K_W$. We use induction over the length of W. For an empty W, K is empty, $\mathrm{spell}(W)$ is empty, and therefore K' is empty as well. For $|W| = 1$, Algorithm 2 outputs $\mathrm{spell}(W) = \kappa_1$ and it holds that $K'_W = (\kappa_1) = K_W$.

For $|W| \geq 2$ we consider that $K_X = K'_X$ holds for a prefix X of W with $|X| = |W| - 1$. When $i = |W|$ at the beginning of the loop in Line 8, then $s = \mathrm{spell}(X)$. By Lemma 3 it holds that the last $k-1$ characters of s are equal to the first $k-1$ characters of κ_ℓ. Therefore, by appending the last character from κ_ℓ to s, κ_ℓ is appended to K'_X forming K'_W. Therefore, last k-mer of K'_W equals the last k-mer of K_W, and the first $\ell - 1$ k-mers of K'_W equal those of K_W by induction. ◀

C Pseudocode for linear-time construction of compacted de Bruijn graphs

The pseudocode for computing a compacted de Bruijn graph in linear time is given by Algorithm 6 which uses Algorithm 5 as a subroutine. The data structure D used by the algorithms is that described in Section 4. Note that if we compute the arc labels as plain strings as in Algorithm 1, we need up to $O(k)$ bits to store a single-k-mer arc. And since arcs are not substrings of input strings (but potentially combinations of input strings), we would need a string set of up to $O(k||I||)$ characters to store all arc labels without referring to the input strings. This contradicts the algorithm being linear in $||I||$. However, we can store the labels as tuples (p, η, q, r), where $p\eta q$ is the label where p and q are explicit strings while η is a pointer to a k-mer in the input. If r is true, then the label must be reverse complemented to match that defined by Algorithm 1. With this fix, the size of each label is linear in the number of k-mers it represents, and in total the de Bruijn graph represents $O(||I||)$ k-mers.

The comparison on Line 16 of Algorithm 6 can be done in linear time in $|\eta_1| + |\eta_2|$ by finding the suffix array intervals of $\eta_1\eta\eta_2$ and $\mathrm{rc}(\eta_1\eta\eta_2)$ with `extendLeft` and `extendRight` from η and $\mathrm{rc}(\eta)$ respectively, and comparing the starts of the intervals. This way, the total time taken by all those comparisons is proportional to the sum of $|\eta_1| + |\eta_2|$ over all unitigs, which is linear in $||I||$ because each character of η_1 and η_2 can be mapped to a distinct edge in the non-compacted de Bruijn graph of $||I||$. Therefore, the algorithm can be implemented to run in $O(||I||)$ time.

Our pseudocode does not compute the first and last character of each arc-label, but this can be easily computed in constant time using w_i, η_1 and η_2 in Algorithm 6.

Algorithm 5 FindUnitigEnd.

```
Input: A data structure $D$, a pair of suffix-intervals $[a_f, b_f], [a_r, b_r]$, an array $S_E$
       mapping from suffix-space to boolean, an array $S_V$ mapping from suffix
       space to nodes, a set of nodes $V$. Each node in $V$ contains a parameter $c$.
Output: A node $v$ at the end of the unitig and a sign $d$, as well as the updated
        $S_E, S_V, V$ and the label $\eta$ of the traversed path.
1  $[a_f, b_f] \leftarrow$ contractLeft($D, [a_f, b_f]$)
2  $[a_r, b_r] \leftarrow$ contractRight($D, [a_r, b_r]$)
3  $\eta \leftarrow \epsilon$
   // extend over $(k-1)$-mers that have indegree and outdegree of 1
4  while |enumerateRight($D, [a_f, b_f]$) $\cup$ rc(enumerateLeft($D, [a_r, b_r]$))| =
    |enumerateLeft($D, [a_f, b_f]$) $\cup$ rc(enumerateRight($D, [a_r, b_r]$))| = 1 do
5      $\{\sigma\} \leftarrow$ enumerateRight($D, [a_f, b_f]$) $\cup$ rc(enumerateLeft($D, [a_r, b_r]$))
6      $\eta \leftarrow \eta\sigma$
7      $[a_f, b_f] \leftarrow$ extendRight($D, [a_f, b_f], \sigma$)
8      $[a_r, b_r] \leftarrow$ extendLeft($D, [a_r, b_r], \mathrm{rc}(\sigma)$)
9      foreach $h \in [a_f, b_f] \cup [a_r, b_r]$ do $S_E[h] \leftarrow$ true
10     $[a_f, b_f] \leftarrow$ contractLeft($D, [a_f, b_f]$)
11     $[a_r, b_r] \leftarrow$ contractRight($D, [a_r, b_r]$)
12 if $S_V[a_f] = \bot$ then
13     insert node $v$ into $V$
14     foreach $h \in [a_f, b_f] \cup [a_r, b_r]$ do $S_V[h] \leftarrow v$
15 else $v \leftarrow S_V[a_f]$
16 if $a_f = a_r$ then $c(v) \leftarrow 1$ ; $d \leftarrow \odot$          /* $v$ self-complemental */
17 else if $a_f < a_r$ then $c(v) \leftarrow 0$ ; $d \leftarrow \ominus$          /* $v$ canonical */
18 else $c(v) \leftarrow 0$ ; $d \leftarrow \oplus$                   /* $v$ not canonical */
19 return $(v, d, S_E, S_V, V, \eta)$
```

Algorithm 6 LinearCompactedDbg.

```
Input: An integer $k$ and a set of strings $I = (w_1, \ldots, w_\ell)$ where each string has
       length at least $k$.
Output: A de Bruijn graph $G = (V, E, c)$ of order $k$.
1  $V \leftarrow \emptyset$; $E \leftarrow \emptyset$                    // $c$ is stored as parameter of each node
   // $ is a special character outside of the alphabet
2  $T \leftarrow \$w_1\$w_2\$ \ldots \$w_\ell\$\,\mathrm{rc}(w_1)\$\,\mathrm{rc}(w_2)\$ \ldots \$\,\mathrm{rc}(w_\ell)\$$
3  $S_V \leftarrow$ array of length $|T|$ filled with $\perp$ mapping from suffix space to nodes in $V$
4  $S_E \leftarrow$ array of length $|T|$ filled with $false$ marking used $k$-mers
5  build data structure $D$ over $T$                              // See text in Section 4
6  foreach $w_i \in I$ do
7      $[a_f, b_f] \leftarrow$ find$(D, \mathrm{pre}_k(w_i))$           // Suffix array interval of $\mathrm{pre}_k(w_i)$
8      $[a_r, b_r] \leftarrow$ find$(D, \mathrm{pre}_k(\mathrm{rc}(w_i)))$    // Suffix array interval of $\mathrm{pre}_k(\mathrm{rc}(w_i))$
9      foreach $j \in (k+1, \ldots, |w_i|)$ do
10         if $S_E[a_f] = false$ then                        // create arc from unused $k$-mer
11             foreach $h \in [a_f, b_f] \cup [a_r, b_r]$ do $S_E[h] \leftarrow true$
12             $\eta \leftarrow$ pointer to $\mathrm{pre}_k(w_i)$
               // find unitig start by finding the end on the rev. comp.
13             $(v_1, d_1, S_E, S_V, V, \eta_1) \leftarrow$ FindUnitigEnd$(D, [a_r, b_r], [a_f, b_f], S_E, S_V, V)$
               // find unitig end
14             $(v_2, d_2, S_E, S_V, V, \eta_2) \leftarrow$ FindUnitigEnd$(D, [a_f, b_f], [a_r, b_r], S_E, S_V, V)$
               // Reverse because finding the start was done in reverse
15             $\eta_1 \leftarrow \mathrm{rc}(\eta_1)$
16             if $\mathrm{rc}(\eta_1\eta\eta_2) < \eta_1\eta\eta_2$ then       // arc labels are always canonical
17                 swap $v_1$ and $v_2$
18                 swap $d_1$ and $d_2$
19                 $d_1 \leftarrow \neg d_1$
20                 $d_2 \leftarrow \neg d_2$
21                 $r \leftarrow true$
22             else
23                 $r \leftarrow false$
24             insert $e = (v_1 d_1, v_2 d_2, (\eta_1, \eta, \eta_2, r))$ into $E$
25         $[a_f, b_f] \leftarrow$ extendRight$(D, [a_f, b_f], w_i[j])$
26         $[a_r, b_r] \leftarrow$ extendLeft$(D, [a_r, b_r], \mathrm{rc}(w_i)[j])$
27         $[a_f, b_f] \leftarrow$ contractLeft$(D, [a_f, b_f])$
28         $[a_r, b_r] \leftarrow$ contractRight$(D, [a_r, b_r])$
```