

RESEARCH

Open Access



Eulertigs: minimum plain text representation of k -mer sets without repetitions in linear time

Sebastian Schmidt^{1*} and Jarno N. Alanko^{1,2}

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. Our algorithm first constructs the de Bruijn graph in linear time and then uses a Eulerian-cycle-based algorithm to compute the minimum representation, in time linear in the size of the output.

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

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. [1] or Brinda et al. [2] 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) [3], or *simplitigs* [2]. This has the following advantages over encoded representations:

- 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 [4]).
- 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. SSHAHash [5], 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* [2] and *UST* [3]. While neither of these computes a minimum representation, Rahman et al. [3] 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

*Correspondence:

Sebastian Schmidt
sebastian.schmidt@helsinki.fi

¹ Department of Computer Science, University of Helsinki, Helsinki, Finland

² Institute of Biology, National University of Sciences, Kiel, Germany



present a counter-example showing that their lower bound is not tight. Small SPSSs without repetitions are used e.g. in SShash [5] and are also computed by state-of-the-art de Bruijn graph compactors like Cuttlefish 2 [6]. Additionally, the state-of-the-art de Bruijn graph compressor GGAT [7] was extended to compute Euler-tigs, in addition to other variants of SPSSs.

When k -mer repetitions are allowed in an SPSS, there is a known polynomially computable minimum representation, namely *matchtigs* [1]. The *matchtig* algorithm joins unitigs by first iterating all possible joins repeating up to $k - 1$ k -mers, and then using minimum perfect matching to find a set of joins that minimises the size of the representation. This is similar to the algorithm presented in this paper, which leaves out the matching step and only joins unitigs that are adjacent. 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 [1, 2] 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 [2] and [3] 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 [8]. The authors of [8] 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.¹ 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. While defining

¹ 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 [9, Figure 4]. Then, the graph is a collection of complete bipartite graphs [9]. 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.

a bidirected de Bruijn graph is not new [10], we take specific care around k -mers that are their own reverse complement. This technicality is often neglected in the literature, and sidestepped by requiring that the value of k is odd, in which case this special case does not occur. To make sure that our definition is correct for any k , we show that our de Bruijn graph admits exactly the strings that can be spelled from the k -mers that it was constructed from. 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 [4, 6, 11, 12] depend on the value of k and/or are probabilistic due to the use of hashing, minimizers or Bloom filters, or do not use the reverse-complement-aware definition of k -mers [13].

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 [1], removing the possibility of joining walks by repeating k -mers. We give detailed descriptions for all these steps and prove their correctness 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 $CS_k(I) = CS_k(I')$ in $O(|I| \log |\Sigma|)$ time.*

where $CS_k(I) = CS_k(I')$ means that I' is an SPSS of I , and $|I|$ is the cumulative length of I (see Sect. "Preliminaries" 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, this gives an easily computable tight lower bound on the size of a minimum SPSS without repetitions. We also give a counter example where previous heuristics are not necessarily optimal.

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 Sect. "Preliminaries" we give preliminary definitions of well-known concepts. In Sect. "De Bruijn graphs" we define de Bruijn graphs and prove the soundness of the definitions. In Sect. "Linear-time construction of compacted bidirected de Bruijn graphs" we show how to construct de Bruijn graphs by our definitions in linear time. In Sect. "Linear-time minimum SPSS without repetitions" we show how to construct a minimum SPSS without repetitions in linear time if the de Bruijn graph is given. Additionally, we give an example where previous heuristics were not optimal. In Sect. "Experiments" we compare our algorithm and Eulertigs against strings computed with ProphAsm and UST on practical data sets.

Preliminaries

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

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 $\#_M : M \rightarrow \mathbb{Z}^+$ mapping elements to their multiplicities. The cardinality is defined as $|M| := \sum_{s \in M} \#_M(s)$.

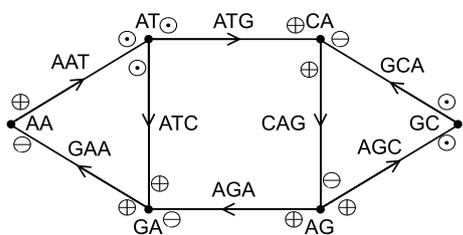
An *alphabet* Σ is an ordered set, and an Σ -word is a string of characters of that set. String concatenation is written as ab for two strings a and b . The set Σ^k is the set of all Σ -words of length k and the set Σ^* is the set of all Σ -words, including the empty word ϵ . Given a positive integer 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 $\text{comp} : \Sigma \rightarrow \Sigma$ mapping characters to characters that is self-inverse (i.e. $\text{comp}(\text{comp}(x)) = x$, also called an *involution*). A *reverse complement function* for alphabet Σ is a function $\text{rc} : \Sigma^* \rightarrow \Sigma^*$ 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.

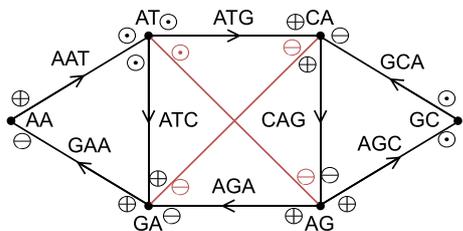
Given an integer k and an alphabet Σ , the k -spectrum of a set of strings $I \subseteq \bigcup_{k' \geq k} \Sigma^{k'}$ is a set of strings $S_k(I) := \{w \in \Sigma^k \mid \exists i \in I : w \text{ is substring of } i \text{ or } \text{rc}(i)\}$. The *canonical k -spectrum* of I is $\text{CS}_k(I) := \{w \in S_k(I) \mid w \text{ is canonical}\}$. 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. [14], 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 $c : V \rightarrow \{1, 0\}$ 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* $(v_1d_1, v'_1d'_1, \eta) \in E$ 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 $(v_1d_1, v'_1d'_1, \eta)^{-1} := (v'_1d'_1, v_1d_1, \eta)$. If not required, we may drop the identifier (i.e. just write $(v_1\ominus, v'_1\odot) \in E$). We count the incidences present in an arc e using multiset notation like $\#_e(vd)$, returning 0 if the arc does not contain the incidence vd , returning 1 if it contains the incidence once and returning 2 if it is a self-loop with that incidence. 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 .

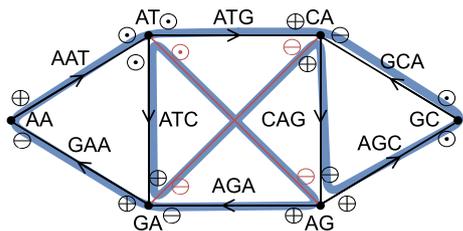
Note that, other than in standard directed graphs, in bidirected graphs arcs can be outgoing or incoming on both ends, and the order of the incidences in the arc does not affect if it is outgoing or incoming to a node. Further, our notation differs from that of standard bidirected graphs in that arcs have a direction. This is required because we will work with arc-centric de Bruijn graphs (see Sect. "De Bruijn graphs"), 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



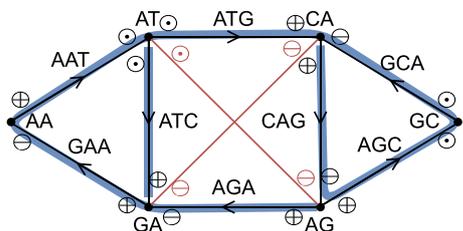
(a) Build the de Bruijn graph of the input strings.



(b) Eulerise the example graph.



(c) Compute a Eulerian cycle.



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

Fig. 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}

Fig. 1a 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 $W := ((v_1d_1, v'_1d'_1, \eta_1), (v_2d_2, v'_2d'_2, \eta_2), \dots, (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ 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 $d'_i = -d_{i+1}$. The length of a walk is $\ell = |W|$. If $v_1 = v_\ell$ and $d_1 = -d'_\ell$, 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 .

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

We define the imbalance of a node $v \in V$ that is not self-complemental as the difference of its outdegree and indegree $\text{imbalance}(v) := \delta^+(v) - \delta^-(v)$. For a self-complemental node $v \in V$ the imbalance is defined as $\text{imbalance}(v) := 1$ if $\delta(v)$ is odd, and $\text{imbalance}(v) := 0$ otherwise. A node $v \in V$ is called unbalanced, if $\text{imbalance}(v) \neq 0$, 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. $(v_1 \oplus, v_2 \ominus, ACCTG) \in E$).

Suffix arrays and suffix trees

Section "Linear-time construction of compacted bidirected de Bruijn graphs" 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 [15] and Mäkinen [16] 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 log-linear time in $|T| \log |\Sigma|$ using e.g. Ukkonen's algorithm [17] or in linear

time in $|T|$ using Farach's algorithm [18]. 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 [19] 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 .

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  $V \leftarrow CS_{k-1}(I)$            /* the nodes are the canonical  $(k-1)$ -mers */
2 foreach  $v \in V$  do
3   if  $rc(v) = v$  then  $c(v) \leftarrow 1$  else  $c(v) \leftarrow 0$ 
4 foreach  $\eta \in CS_k(I)$  do
5    $w \leftarrow pre_{k-1}(\eta)$            /* compute  $v$  */
6    $v \leftarrow canonical\ w$ 
7    $w' \leftarrow suf_{k-1}(\eta)$        /* compute  $v'$  */
8    $v' \leftarrow canonical\ w'$ 
9   if  $c(v) = 1$  then  $d \leftarrow \ominus$            /* 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  $d' \leftarrow \ominus$            /* 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 Fig. 1a for an example. The algorithm inserts an arc for each canonical k -mer, and connects the arcs via nodes according to their $k - 1$ overlaps. Depending on if these overlaps use the reverse complement or if the $k - 1$ -mer of a node is self-complemental, it adds directions to the incidences. A de Bruijn graph computed by this algorithm has the following property.

Lemma 2 *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) $(v'_1 = v_2 \text{ and } d'_1 = -d_2)$ if and only if $\text{suf}_{k-1}(\eta_1) = \text{pre}_{k-1}(\eta_2)$,
- (b) $(v'_1 = v'_2 \text{ and } d'_1 = -d'_2)$ if and only if $\text{suf}_{k-1}(\eta_1) = \text{pre}_{k-1}(\text{rc}(\eta_2))$,
- (c) $(v_1 = v_2 \text{ and } d_1 = -d_2)$ if and only if $\text{suf}_{k-1}(\text{rc}(\eta_1)) = \text{pre}_{k-1}(\eta_2)$, and
- (d) $(v_1 = v'_2 \text{ and } d_1 = -d'_2)$ if and only if $\text{suf}_{k-1}(\text{rc}(\eta_1)) = \text{pre}_{k-1}(\text{rc}(\eta_2))$.

Proof

Observe that the values of w and w' computed in Lines 5 and 7 of Algorithm 1 are equal to $\text{pre}_{k-1}(\eta_1)$ and $\text{suf}_{k-1}(\eta_1)$ for e_1 and equal to $\text{pre}_{k-1}(\eta_2)$ and $\text{suf}_{k-1}(\eta_2)$ for e_2 . Further, observe that the values of v and v' computed in Lines 6 to 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 $\text{pre}_{k-1}(\eta_1), \text{suf}_{k-1}(\eta_1), \text{pre}_{k-1}(\eta_2)$ and $\text{suf}_{k-1}(\eta_2)$. Finally, observe that the sign values d and d' computed in Lines 9 to 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 $d'_1 = -d_2$, then $w'_1 = w_2$ for all possible values of d'_1 , and therefore $\text{suf}_{k-1}(\eta_1) = \text{pre}_{k-1}(\eta_2)$. If $\text{suf}_{k-1}(\eta_1) = \text{pre}_{k-1}(\eta_2)$, 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, $d'_1 = -d_2$ for all possible values of d'_1 .
- (b) If $v'_1 = v'_2$ and $d'_1 = -d'_2$, then $w'_1 = \text{rc}(w'_2)$ for all possible values of d'_1 , and therefore $\text{suf}_{k-1}(\eta_1) = \text{rc}(\text{suf}_{k-1}(\eta_2)) = \text{pre}_{k-1}(\text{rc}(\eta_2))$. If $\text{suf}_{k-1}(\eta_1) = \text{pre}_{k-1}(\text{rc}(\eta_2))$, then $w'_1 = \text{rc}(w'_2)$, and therefore $v'_1 = v'_2$ because v'_1 and v'_2 are the canonicals of w'_1 and w'_2 . Additionally, $d'_1 = -d'_2$ for all possible values of d'_1 .
- (c) If $v_1 = v_2$ and $d_1 = -d_2$, then $\text{rc}(w_1) = w_2$ for all possible values of d_1 , and therefore $\text{suf}_{k-1}(\text{rc}(\eta_1)) = \text{rc}(\text{pre}_{k-1}(\eta_1)) = \text{pre}_{k-1}(\eta_2)$. If $\text{suf}_{k-1}(\text{rc}(\eta_1)) = \text{pre}_{k-1}(\eta_2)$, then $w_1 = \text{rc}(w_2)$, and therefore $v_1 = v_2$ because v_1 and v_2 are the canonicals of w_1 and w_2 . Additionally, $d_1 = -d_2$ for all possible values of d_1 .
- (d) This case is equivalent to the first case when swapping e_1 and e_2 , because $\text{suf}_{k-1}(\eta_1) = \text{pre}_{k-1}(\eta_2) \iff \text{suf}_{k-1}(\text{rc}(\eta_2)) = \text{pre}_{k-1}(\text{rc}(\eta_1))$.

□

Algorithm 2: SPELL

Input: A de Bruijn graph $G = (V, E, c)$ of order k and a walk

$$W = (e_1 := (v_1 d_1, v'_1 d'_1, \eta_1), \dots, e_\ell := (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell)).$$

Output: The string s spelled by W , i.e. $\text{spell}(W)$.

```

1 if  $W$  is empty then
2    $s \leftarrow \epsilon$ 
3 else                                     /* compute the sequence of kmers from  $W$  */
4   foreach  $i \in (1, \dots, \ell)$  do         /* iterate the sequence in order */
5     if  $e_i \in E$  then  $\kappa_i \leftarrow \eta_i$ 
6     else  $\kappa_i \leftarrow \text{rc}(\eta_i)$       /*  $e_i^{-1} \in E$  */
7    $s \leftarrow k - 1$  prefix of  $\kappa_1$ 
8   foreach  $i \in (1, \dots, \ell)$  do         /* iterate the sequence in order */
9     append the last character from  $\kappa_i$  to  $s$ 

```

For a walk $W := (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \dots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ in a de Bruijn graph, its *sequence of k -mers* is $K := (\kappa_1, \dots, \kappa_\ell)$, where for each i we define κ_i as η_i if $e_i \in E$, and as $\text{rc}(\eta_i)$ if $e_i^{-1} \in E$. The string *spell* (W) is the string *spelled* by W , which is defined as its *collapsed* sequence of k -mers, i.e. its sequence of k -mers gets concatenated while overlapping consecutive k -mers by $k - 1$. This is computed by Algorithm 2. It spells out the first k -mer (or its reverse complement) completely, and then adds the last characters of all subsequent k -mers (or their reverse complements). We prove the following lemmas to show that our definition of a de Bruijn graph together with the $\text{spell}(\cdot)$ function is sound for our purposes, i.e. walks in the de Bruijn graph can spell exactly the strings containing subsets of the k -mers used to create the de Bruijn graph. Due to this property, we can use our de Bruijn graph and spell to in the Euler algorithm, such that finding a minimum SPSS is equivalent to finding a minimum walk cover of the de Bruijn graph.

Lemma 3 *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 $W := (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \dots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ be a walk in G , and $K := (\kappa_1, \dots, \kappa_\ell)$ its sequence of k -mers. Then for each consecutive pair of k -mers κ_i, κ_{i+1} it holds that $\text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1})$.*

Proof

Let $i \in \{1, \dots, \ell - 1\}$. 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.

- If $e_i, e_{i+1} \in E$, then by Lemma 2 (a), it holds that $\text{suf}_{k-1}(\eta_i)$ equals $\text{pre}_{k-1}(\eta_{i+1})$. By definition, $\kappa_i = \eta_i$ and $\kappa_{i+1} = \eta_{i+1}$, so $\text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1})$.
- 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 $\text{suf}_{k-1}(\eta_i)$ equals $\text{pre}_{k-1}(\text{rc}(\eta_{i+1}))$. By definition, $\kappa_i = \eta_i$ and $\kappa_{i+1} = \text{rc}(\eta_{i+1})$, so $\text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1})$.
- 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 $\text{suf}_{k-1}(\text{rc}(\eta_i))$ equals

$\text{pre}_{k-1}(\eta_{i+1})$. By definition, $\kappa_i = \text{rc}(\eta_i)$ and $\kappa_{i+1} = \eta_{i+1}$, so $\text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1})$.

- 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 $\text{suf}_{k-1}(\text{rc}(\eta_i))$ equals $\text{pre}_{k-1}(\text{rc}(\eta_{i+1}))$. By definition, $\kappa_i = \text{rc}(\eta_i)$ and $\kappa_{i+1} = \text{rc}(\eta_{i+1})$, so $\text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1})$.

□

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

Lemma 4 *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 $\text{spell}(W)$. Then $K_W = K'_W$.*

Proof

Let $(\kappa_1, \dots, \kappa_\ell) := K_W$. We use induction over the length of W . For an empty W , K is empty, $\text{spell}(W)$ is empty, and therefore K' is empty as well. For $|W| = 1$, Algorithm 2 outputs $\text{spell}(W) = \kappa_1$ 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 $s = \text{spell}(X)$. 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. □

Lemma 5 *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 $\text{CS}_k(w) \subseteq \text{CS}_k(I)$. Then there exists a walk W in G with $\text{spell}(W) = w$.*

Proof

Let $K_w = (\kappa_1, \dots, \kappa_\ell)$ be the sequence of k -mers of w . We construct $W = (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \dots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ 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 $d'_i = -d'_{i+1}$ for all i . Using Lemma 2, we get:

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

To complete the proof we need to show that $\text{spell}(W) = w$. 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 $\text{spell}(W)$ is equivalent to K_W , and therefore $\text{spell}(W) = w$. \square

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 *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 $\text{CS}_k(I) = \text{CS}_k(I')$, if and only if there is a walk cover \mathcal{W} in G that spells the strings in I' .*

Proof

If $\text{CS}_k(I') \subseteq \text{CS}_k(I)$, then for each string $w' \in I'$ it holds that $\text{CS}_k(w') \subseteq \text{CS}_k(I)$. Therefore, by Lemma 5, there exists a walk w in G with $\text{spell}(w) = w'$. Then, the set of all such walks \mathcal{W} spells I' . Further, because $\text{CS}_k(I) \subseteq \text{CS}_k(I')$, the identifier η of each arc $e \in E$ is in $\text{CS}_k(I')$, and therefore in the sequence of k -mers $K_{w'}$ 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, $w' := \text{spell}(w)$ 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 $\text{CS}_k(I) \subseteq \text{CS}_k(I')$. And since \mathcal{W} is a walk cover of G , we get $\text{CS}_k(I) = \text{CS}_k(I')$. \square

Corollary 7 *By setting $I = I'$ in Theorem 6 we see that our de Bruijn graph contains the strings it was constructed from. Further, by Theorem 6 it holds that walks in the de Bruijn graph spell exactly the strings that can be formed from the k -mers that were used to create the graph. Hence, our definition of a de Bruijn graph is sound for all k .*

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 uncompact de Bruijn graphs in our theoretical sections, however all results equally apply to compacted de Bruijn graphs.

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 .

Algorithm

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

$$T = \$w_1\$w_2\$ \dots \$w_m\$ \text{rc}(w_1)\$ \text{rc}(w_2)\$ \dots \$ \text{rc}(w_m)\$,$$

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 Sect. "[Implementation of the subroutines](#)".

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 complementary, 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 complementary 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.

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 Sect. "[Suffix arrays and suffix trees](#)". 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 $|T|$ [16]. 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 [20], using the data structures of Belazzougui et al. [21] 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.

Pseudocode

The pseudocode for computing a compacted de Bruijn graph in linear time is given by Algorithm 4 which uses Algorithm 3 as a subroutine. The data structure D used by the algorithms is that described in Sect. "Linear-time construction of compacted bidirected de Bruijn graphs". Note that if we compute the arc labels as plain strings as in Algorithm 1, we need up to $O(k \log |\Sigma|)$ 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 up to $O(k|I| \log |\Sigma|)$ bits to store all arc labels without referring to the input strings. This contradicts the algorithm being linear in $|I| \log |\Sigma|$. 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 a label that represents ℓ k -mers is $O(\ell \log |\Sigma|)$, and in total the de Bruijn graph represents $O(|I|)$ k -mers.

Algorithm 3: 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 η of the traversed path.

```

1  $[a_f, b_f] \leftarrow \text{contractLeft}(D, [a_f, b_f])$ 
2  $[a_r, b_r] \leftarrow \text{contractRight}(D, [a_r, b_r])$ 
3  $\eta \leftarrow \epsilon$ 
   // extend over  $(k-1)$ -mers that have indegree and outdegree of 1
4 while  $|\text{enumerateRight}(D, [a_f, b_f]) \cup \text{rc}(\text{enumerateLeft}(D, [a_r, b_r]))| =$ 
    $|\text{enumerateLeft}(D, [a_f, b_f]) \cup \text{rc}(\text{enumerateRight}(D, [a_r, b_r]))| = 1$  do
5    $\{\sigma\} \leftarrow \text{enumerateRight}(D, [a_f, b_f]) \cup \text{rc}(\text{enumerateLeft}(D, [a_r, b_r]))$ 
6    $\eta \leftarrow \eta\sigma$ 
7    $[a_f, b_f] \leftarrow \text{extendRight}(D, [a_f, b_f], \sigma)$ 
8    $[a_r, b_r] \leftarrow \text{extendLeft}(D, [a_r, b_r], \text{rc}(\sigma))$ 
9   foreach  $h \in [a_f, b_f] \cup [a_r, b_r]$  do  $S_E[h] \leftarrow \text{true}$ 
10   $[a_f, b_f] \leftarrow \text{contractLeft}(D, [a_f, b_f])$ 
11   $[a_r, b_r] \leftarrow \text{contractRight}(D, [a_r, b_r])$ 
12 if  $S_V[a_f] = \perp$  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 4: LINEARCOMPACTEDDBG

Input: An integer k and a set of strings $I = (w_1, \dots, w_\ell)$ 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
    $\$$ 
2  $T \leftarrow \$w_1\$w_2\$ \dots \$w_\ell\$rc(w_1)\$rc(w_2)\$ \dots \$rc(w_\ell)\$$ 
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  $T$  4
6 foreach  $w_i \in I$  do
7    $[a_f, b_f] \leftarrow (D, \text{pre}_k(w_i))$  prek(wi)
8    $[a_r, b_r] \leftarrow (D, \text{pre}_k(\text{rc}(w_i)))$  prek(rc(wi))
9   foreach  $j \in (k + 1, \dots, |w_i|)$  do
10    if  $S_E[a_f] = \textit{false}$  then k
11      foreach  $h \in [a_f, b_f] \cup [a_r, b_r]$  do  $S_E[h] \leftarrow \textit{true}$ 
12       $\eta \leftarrow$  pointer to  $\text{pre}_k(w_i)$ 
13       $(v_1, d_1, S_E, S_V, V, \eta_1) \leftarrow \text{FINDUNITIGEND}(D, [a_r, b_r], [a_f, b_f], S_E, S_V, V)$ 
14       $(v_2, d_2, S_E, S_V, V, \eta_2) \leftarrow \text{FINDUNITIGEND}(D, [a_f, b_f], [a_r, b_r], S_E, S_V, V)$ 
15       $\eta_1 \leftarrow \text{rc}(\eta_1)$ 
16      if  $\text{rc}(\eta_1\eta\eta_2) < \eta_1\eta\eta_2$  then
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         $r \leftarrow \textit{true}$ 
22      else
23         $r \leftarrow \textit{false}$ 
24      insert  $e = (v_1d_1, v_2d_2, (\eta_1, \eta, \eta_2, r))$  into  $E$ 
25    $[a_f, b_f] \leftarrow (D, [a_f, b_f], w_i[j])$ 
26    $[a_r, b_r] \leftarrow (D, [a_r, b_r], \text{rc}(w_i)[j])$ 
27    $[a_f, b_f] \leftarrow (D, [a_f, b_f])$ 
28    $[a_r, b_r] \leftarrow (D, [a_r, b_r])$ 

```

The comparison on Line 16 of Algorithm 4 can be done in linear time in $|\eta_1| + |\eta_2|$ by finding the suffix array intervals of $\eta_1\eta_2$ and $\text{rc}(\eta_1\eta_2)$ with `extendLeft` and `extendRight` from η and $\text{rc}(\eta)$ 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-compactified de Bruijn graph of $\|I\|$. Therefore, the algorithm can be implemented to run in $O(\|I\| \log |\Sigma|)$ 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 4.

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 . By 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:

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

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.

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 $\text{imbalance}(v)$ more outgoing arcs than incoming arcs. Since walks need to leave v with the opposite sign than they entered v , at least $\text{imbalance}(v)$ arcs cannot be covered by walks that do not start or end in v . If v has negative imbalance, the situation is symmetric. \square

Definition 10 The imbalance $\text{imbalance}(G)$ 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 *Let G be a bigraph. A walk cover \mathcal{W} of G has a minimum string count of $\text{imbalance}(G)/2$.*

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 $\text{imbalance}(G)/2$. \square

Eulerising a bigraph

A directed graph is called *Eulerian*, if all nodes have indegree equal to outdegree, i.e. are balanced [22]. If the graph is strongly connected,² 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.

² Strongly connected means that there is a directed path from each node v_1 to each node v_2 .

Algorithm 5: 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 */

```

Definition 12 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 5. See Fig. 1b for an example. The algorithm lists all nodes that are out of balance, and inserts arbitrary arcs to balance them.

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. \square

Lemma 14 *Given a connected bigraph $G = (V, E, c)$, Algorithm 5 outputs a Eulerian bigraph $G' = (V, E', c)$.*

Proof

Algorithm 5 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 5 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' .

\square

Lemma 15 *Given a bigraph $G = (V, E, c)$, Algorithm 5 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 $O(|\text{imbalance}(v)|)$ 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. \square

With Lemma 14 and 15 we get the following.

Theorem 16 *Algorithm 5 is correct and runs in $O(|V| + |E|)$ time.*

Computing a Eulerian cycle in a bigraph**Algorithm 6: 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  $v'-d'$  do
8       advance  $first\_unfinished$  to the next arc in  $W$ 
9        $(vd, v'd', \eta) \leftarrow$  dereference  $first\_unfinished$ 
10    // extend  $W'$  without repeating arcs until it closes a cycle
11    while  $E$  contains an arc  $e = (v_e d_e, v'_e d'_e, \eta_e)$  with incidence  $v'-d'$  do
12      remove  $e$  from  $E$ 
13      if  $v_e d_e = v'-d'$  then  $(vd, v'd', \eta) \leftarrow (v_e d_e, v'_e d'_e, \eta_e)$ 
14      else  $(vd, v'd', \eta) \leftarrow (v'_e d'_e, v_e d_e, \eta_e)$  /*  $v'_e d'_e = v'-d'$  */
15      append  $(vd, v'd', \eta)$  to  $W'$ 
16    if  $|W| = 0$  then
17       $W \leftarrow W'$ 
18       $first\_unfinished \leftarrow$  pointer to the first arc in  $W$ 
19    else
20      insert  $W'$  after  $first\_unfinished$  in  $W$ 
21       $W' \leftarrow ()$  /* empty doubly linked list */

```

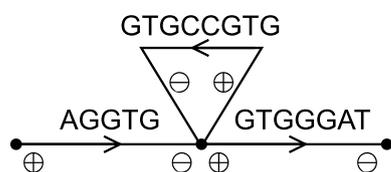


Fig. 2 An example de Bruijn graph with $k = 4$ in which UST and ProphAsm may compute a suboptimal solution. The optimal solution here is a single string AGGTGCCGTGGAT

After Eulerising the bigraph, we can compute a Eulerian cycle using Algorithm 6. We do this similarly to Hierholzer's classic algorithm for Eulerian cycles [22]. 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 Fig. 1c for an example of a Eulerian cycle.

Lemma 17 *Given a connected Eulerian bigraph $G = (V, E, c)$, Algorithm 6 terminates and outputs a Eulerian cycle W .*

Proof

For $W = (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \dots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ 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

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

Genome	Algorithm	CL ratio	SC ratio	Time [s]	Memory [GiB]
<i>C. elegans</i> (reads)	Unitigs	1.789	2.831	1963	5.96
	UST	1.035	1.080	2815	(1.43) 15.2 (2.54)
	Eulertigs	1	1	2741	(1.40) 24.9 (4.18)
<i>B. mori</i> (reads)	Unitigs	1.912	3.136	6844	9.35
	UST	1.050	1.118	10053	(1.47) 52.4 (5.60)
	Eulertigs	1	1	9412	(1.38) 78.4 (8.38)
<i>H. sapiens</i> (reads)	Unitigs	1.418	2.143	55007	13.0
	UST	1.016	1.044	55772	(1.01) 16.4 (1.26)
	Eulertigs	1	1	55856	(1.02) 26.5 (2.05)
<i>C. elegans</i>	Unitigs	1.060	3.154	53.3	1.22
	UST	1.002	1.089	57.1	(1.07) 1.22 (1.00)
	Eulertigs	1	1	62.5	(1.17) 1.22 (1.00)
<i>B. mori</i>	Unitigs	1.262	3.310	244	3.32
	UST	1.018	1.156	281	(1.15) 3.32 (1.00)
	Eulertigs	1	1	295	(1.21) 3.32 (1.00)
<i>H. sapiens</i>	Unitigs	1.195	3.532	1788	10.0
	UST	1.015	1.192	2020	(1.13) 10.0 (1.00)
	Eulertigs	1	1	2127	(1.19) 10.0 (1.00)

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*

Table 2 Experiments on (references of) pangenomes with $k = 31$ and a min abundance of 1

Pangenome	Tigs	CL ratio	SC ratio	Time [s]		Memory [MiB]	
1102x <i>N. gonorrhoeae</i>	Unitigs	1.615	3.052	29.1		4351	
	UST	1.022	1.072	31.4	(1.08)	4351	(1.00)
	ProphAsm	1.00004	1.00013	734	(25.2)	208	(0.05)
	Eulertigs	1	1	30.2	(1.04)	4351	(1.00)
616x <i>S. pneumoniae</i>	Unitigs	1.679	3.055	26.1		3146	
	UST	1.026	1.080	30.8	(1.18)	3146	(1.00)
	ProphAsm	1.00004	1.00012	412	(15.8)	434	(0.14)
	Eulertigs	1	1	29.3	(1.12)	3146	(1.00)
3682x <i>E. coli</i>	Unitigs	1.705	3.092	334		7117	
	UST	1.031	1.092	418	(1.25)	7117	(1.00)
	ProphAsm	1.00008	1.00023	7066	(21.1)	7221	(1.01)
	Eulertigs	1	1	398	(1.19)	7117	(1.00)
~309kx Salmonella	Unitigs	1.830	3.151	82417		13007	
	UST	1.049	1.126	82836	(1.01)	13007	(1.00)
	Eulertigs	1	1	82732	(1.00)	13007	(1.00)
2505x Human	Unitigs	1.479	3.201	77582		411472	
	ProphAsm	1.00004	1.00017	82797*	(1.07)	411472*	(1.00)
	Eulertigs	1	1	79198	(1.02)	411472	(1.00)

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, the *E. coli* pangenome contains 341 million unique kmers, the Salmonella pangenome contains 657 million unique kmers and the human pangenome contains 2.8 billion unique kmers. Due to its size, Prophasm could not be run on the Salmonella pangenome. Also due to size, BCALM2 did not run on the human pangenome, hence we used Cuttlefish 2. To still be able to compare against competitors, we ran Prophasm on the unitigs produced by Cuttlefish 2 (UST requires extra information specific to BCALM2)

* Indicates that resource usage includes running Cuttlefish 2 for Prophasm

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. \square

Lemma 18 *Given a connected Eulerian bigraph $G = (V, E, c)$, Algorithm 6 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 Eulerianity). 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.

Table 3 Experiments on references and read sets of single genomes with $k = 52$ and a min abundance of 10 for human and 1 for the others

Genome	Algorithm	CL ratio	SC ratio	Time [s]	Memory [GiB]
<i>C. elegans</i> (reads)	Unitigs	1.788	2.824	2278	5.94
	UST	1.034	1.079	3164	(1.39) 15.0 (2.53)
	Eulertigs	1	1	3101	(1.36) 24.8 (4.17)
<i>B. mori</i> (reads)	Unitigs	1.911	3.133	7157	9.35
	UST	1.050	1.117	10530	(1.47) 52.3 (5.59)
	Eulertigs	1	1	10006	(1.40) 78.3 (8.38)
<i>H. sapiens</i> (reads)	Unitigs	1.414	2.135	56418	12.0
	UST	1.016	1.043	57174	(1.01) 16.1 (1.35)
	Eulertigs	1	1	57252	(1.01) 25.9 (2.17)
<i>C. elegans</i>	Unitigs	1.059	3.145	72.9	1.22
	UST	1.002	1.088	76.2	(1.05) 1.22 (1.00)
	Eulertigs	1	1	82.0	(1.13) 1.22 (1.00)
<i>B. mori</i>	Unitigs	1.259	3.296	259	3.33
	UST	1.017	1.153	295	(1.14) 3.33 (1.00)
	Eulertigs	1	1	311	(1.20) 3.33 (1.00)
<i>H. sapiens</i>	Unitigs	1.190	3.521	1509	10.0
	UST	1.014	1.190	1708	(1.13) 10.0 (1.00)
	Eulertigs	1	1	1845	(1.22) 10.0 (1.00)

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*

Table 4 Experiments on (references of) pangenomes with $k = 32$ and a min abundance of 1

Pangenome	Tigs	CL ratio	SC ratio	Time [s]	Memory [MiB]
1102x <i>N. gonorrhoeae</i>	Unitigs	1.623	3.053	37.1	6725
	UST	1.023	1.074	39.3	(1.06) 6725 (1.00)
	ProphAsm	1.00005	1.00015	764	(20.6) 210 (0.03)
	Eulertigs	1	1	38.3	(1.03) 6725 (1.00)
616x <i>S. pneumoniae</i>	Unitigs	1.685	3.050	37.8	4036
	UST	1.026	1.079	42.2	(1.12) 4036 (1.00)
	ProphAsm	1.00005	1.00014	446	(11.8) 439 (0.11)
	Eulertigs	1	1	41.3	(1.09) 4036 (1.00)
3682x <i>E. coli</i>	Unitigs	1.710	3.089	457	7193
	UST	1.031	1.092	542	(1.18) 7193 (1.00)
	ProphAsm	1.00006	1.00018	7148	(15.6) 7318 (1.02)
	Eulertigs	1	1	521	(1.14) 7193 (1.00)
~309kx Salmonella	Unitigs	1.831	3.141	169935	13860
	UST	1.048	1.124	170358	(1.00) 13860 (1.00)
	Eulertigs	1	1	170248	(1.00) 13860 (1.00)

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, the *E. coli* pangenome contains 341 million unique kmers, the Salmonella pangenome contains 657 million unique kmers and the human pangenome contains 2.8 billion unique kmers. Due to its size, Prophasm could not be run on the Salmonella pangenome. Also due to size, BCALM2 did not run on the human pangenome, hence we used Cuttlefish 2. To still be able to compare against competitors, we ran Prophasm on the unitigs produced by Cuttlefish 2 (UST requires extra information specific to BCALM2). Cuttlefish 2 supports only odd k , hence the human pangenome is excluded from this experiment

Table 5 Experiments on references and read sets of single genomes with $k = 102$ and a min abundance of 10 for human and 1 for the others

Genome	Algorithm	CL ratio	SC ratio	Time [s]	Memory [GiB]
<i>C. elegans</i> (reads)	Unitigs	1.742	2.588	5585	5.91
	UST	1.023	1.049	6292	(1.13) 11.8 (2.00)
	Eulertigs	1	1	6565	(1.18) 21.6 (3.66)
<i>B. mori</i> (reads)	Unitigs	1.891	3.003	34979	10.8
	UST	1.042	1.093	38272	(1.09) 47.1 (4.36)
	Eulertigs	1	1	38939	(1.11) 77.3 (7.17)
<i>H. sapiens</i> (reads)	Unitigs	1.334	1.927	191808	9.15
	UST	1.008	1.021	192219	(1.00) 10.8 (1.18)
	Eulertigs	1	1	192464	(1.00) 13.8 (1.50)
<i>C. elegans</i>	Unitigs	1.042	3.061	176	2.14
	UST	1.001	1.063	179	(1.01) 2.14 (1.00)
	Eulertigs	1	1	186	(1.05) 2.14 (1.00)
<i>B. mori</i>	Unitigs	1.133	2.805	756	3.15
	UST	1.005	1.071	771	(1.02) 3.15 (1.00)
	Eulertigs	1	1	801	(1.06) 3.15 (1.00)
<i>H. sapiens</i>	Unitigs	1.060	3.189	5204	17.4
	UST	1.003	1.101	5277	(1.01) 17.4 (1.00)
	Eulertigs	1	1	5474	(1.05) 17.4 (1.00)

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*

Table 6 Experiments on (references of) pangenomes with $k = 64$ and a min abundance of 1

Pangenome	Tigs	CL ratio	SC ratio	Time [s]	Memory [MiB]
1102x <i>N. gonorrhoeae</i>	Unitigs	1.805	3.026	57.3	6116
	UST	1.028	1.069	59.4	(1.04) 6116 (1.00)
	Eulertigs	1	1	59.2	(1.03) 6116 (1.00)
616x <i>S. pneumoniae</i>	Unitigs	1.767	3.008	42.4	5375
	UST	1.026	1.068	46.9	(1.10) 5375 (1.00)
	Eulertigs	1	1	47.0	(1.11) 5375 (1.00)
3682x <i>E. coli</i>	Unitigs	1.803	3.037	637	6897
	UST	1.030	1.076	720	(1.13) 6897 (1.00)
	Eulertigs	1	1	724	(1.14) 6897 (1.00)
~309kx Salmonella	Unitigs	1.873	3.021	202386	15580
	UST	1.042	1.098	202838	(1.00) 15580 (1.00)
	Eulertigs	1	1	202816	(1.00) 15580 (1.00)

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, the *E. coli* pangenome contains 341 million unique kmers, the Salmonella pangenome contains 657 million unique kmers and the human pangenome contains 2.8 billion unique kmers. Due to its size, Prophasm could not be run on the Salmonella pangenome. Also due to size, BCALM2 did not run on the human pangenome, hence we used Cuttlefish 2. To still be able to compare against competitors, we ran Prophasm on the unitigs produced by Cuttlefish 2 (UST requires extra information specific to BCALM2). Cuttlefish 2 supports only odd k , hence the human pangenome is excluded from this experiment. Prophasm supports only $k \leq 32$, hence it is excluded from this experiment

As a result, all loops individually run at most $|E|$ times, therefore Algorithm 6 terminates after $O(|E|)$ steps. Because G is connected, this is equivalent to $O(|V| + |E|)$ steps. \square

With Lemma 17 and 18 we get the following.

Theorem 19 *Algorithm 6 is correct and runs in $O(|V| + |E|)$ time.*

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 5, and removing those arcs (see Fig. 1d for an example). This results in a walk cover with either one walk, if Algorithm 5 inserted zero or one arcs, or imbalance $(G)/2$ arcs, if Algorithm 5 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 $CS_k(I) = CS_k(I')$ in $O(|I| \log |\Sigma|)$ time.

Previous heuristics were not optimal

The heuristics implemented by UST [3] and Prophasm [2] are not optimal, as shown experimentally below. Here, we also give a simple counter-example to argue that the previous heuristics were not optimal. Even though the previous algorithms were described in node-centric de Bruijn graphs, we describe them here in the arc-centric variants to stick with the terminology of this paper.

UST works by starting from an arbitrary arc and extending forwards to unused arcs as long as possible. If there is no unused arc, but the last chosen arc has a successor that is the start of another walk, then the walks are joined. On the other hand, ProphAsm works by choosing an arbitrary arc and extending both forwards and backwards to unused arcs as long as possible. Both algorithms may fail to produce an optimal solution in the example given in Fig. 2. They may both first choose AGGTG and then continue to GTGGGAT, producing a string AGGTGGGAT. When they then process GTGCCGTG, they cannot join it with the previous string, hence they

produce two strings of a cumulative length of 17. The optimal solution in Fig. 2 has one string with a cumulative length of 14.

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 [1], 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 [1]. For constructing de Bruijn graphs, we do not implement our purely theoretical linear time algorithm, since practical de Bruijn graph construction is a well-researched field [4, 6, 11, 23–25], and we want to focus more on computing the compressed representation from unitigs. UST only supports unitigs constructed by BCALM2 [11], since it needs certain additional data. BCALM2 is not a linear time algorithm, but is 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 union-find data structure. For the human pangenome, which hits some built-in limit of BCALM2, we use Cuttlefish 2 [6] instead. This prevents us from running UST, but instead we run ProphAsm on the unitigs computed by Cuttlefish 2.

Our experimental pipeline is constructed with [26] and using the bioconda software repository [27]. 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 licensed under the Creative Commons Attribution 4.0 International license and available on zenodo [28]. We additionally provide our implementation of the Eulertigs algorithm on zenodo [29] as well as github [30], conda [31] and crates.io [32].

The performance figures in Tables 1 and 2 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 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. In terms of CL, we

see that the SPSS computed with UST mostly remains within the expected 3% of the lower bound, but it is 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 Eq. 1. This also explains the correlation between CL ratio and SC ratio, which can be observed in all cases.

We conduct our experiments also with even k in Tables 3 and 4 to prove that our implementation also works with even k . To verify that the strings are correct, the tigs computed for the E. coli pangenome are compared against each other by loading all k -mers into a hash table and checking if different tigs contain the same k -mers. There are no significant differences between the experiments with even k and odd k . Additionally, we conduct our experiments with higher k in Tables 5 and 6 to show that performance stays the same when k is increased. The increase in k seems to increase the runtimes of BCALM2 and Cuttlefish 2, but the runtime of our Eulertigs implementation does not change significantly. As a result, the ratio between the runtimes of BCALM2 and Cuttlefish 2 and the runtime of our Eulertigs implementation becomes smaller for larger k .

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 [2, 3]. 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.com [30] and [conda](https://conda.io) [31] and a library on crates.io [32], allowing it to easily be used in any k -mer-based tool. While the difference in cumulative length between previous heuristics and the optimum is not large, our tool works for any value of k , and is, combined with a de Bruijn graph compactor such as BCALM, much faster than ProphAsm, which achieves nearly indistinguishable cumulative length. Hence, our tool is better suited to be used any k -mer-based application including SSHAsh [5], and specifically in the applications listed in [2] which include compressed storage on disk and k -mer-based queries.

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.

Acknowledgements

We wish to thank Andrea Cracco for providing us with the ~309kx Salmonella pangenome. We also wish to thank the anonymous reviewers for their useful constructive feedback, which improved the presentation of the paper, the implementation and the experimental results. Open access funded by Helsinki University Library.

Author 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 Sect. "Linear-time construction of compacted bidirected de Bruijn graphs". SS implemented the algorithm and conducted and evaluated the experiments. All authors read and approved the final manuscript.

Funding

Open Access funding provided by University of Helsinki including Helsinki University Central Hospital. This work was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFE BIO), and by NIH NIAID grant No. R01HG011392 and Academy of Finland grant 339070.

Availability of data and materials

The implementation of the Eulertig algorithm is available on [github](https://github.com) [30]. The name of the project is *matchtigs*. It is platform independent, and can be compiled locally or installed from [bioconda](https://bioconda.org) as described in the README of the project. It is licensed under the 2-clause BSD license. The version used for our experiments is available on [zenodo](https://zenodo.org) [29], and the implementation together with all code to reproduce the experiments is available at [28]. The experiment code is licensed under the Creative Commons Attribution 4.0 International license. See [1] for the availability of the non-original data used for our experiments.

Declarations

Ethics approval and consent to participate

This study only uses publicly available datasets, hence an ethics approval or consent to participate is not required.

Competing interests

The authors declare that they have no competing interests.

Received: 13 February 2023 Accepted: 13 May 2023

Published online: 04 July 2023

References

- Schmidt S, Khan S, Alanko J, Tomescu AI. Matchtigs: minimum plain text representation of kmer sets. *bioRxiv*. 2021. <https://doi.org/10.1101/2021.12.15.472871>.
- Břinda K, Baym M, Kucherov G. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biol*. 2021;22(1):1–24.
- Rahman A, Medvedev P. Representation of k-Mer sets using spectrum-preserving string sets. *J Comput Biol*. 2021;28(4):381–94.
- Holley G, Melsted P. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biol*. 2020;21(1):1–20.
- Pibiri GE. Sparse and skew hashing of k-mers. *bioRxiv*. 2022. <https://doi.org/10.1101/2022.01.15.476199>.

6. Khan J, Kokot M, Deorowicz S, Patro R. Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2. *bioRxiv*. 2021. <https://doi.org/10.1101/2021.12.14.472718>.
7. Cracco A, Tomescu AI. Extremely-fast construction and querying of compacted and colored de Bruijn graphs with GGCAT. *bioRxiv*. 2022. <https://doi.org/10.1101/2022.10.24.513174>.
8. Kasprzyk M. Classification of de Bruijn-based labeled digraphs. *Discrete Appl Math*. 2018;234:86–92. <https://doi.org/10.1016/j.dam.2016.10.014>.
9. Blazewicz J, Hertz A, Kobler D, de Werra D. On some properties of DNA graphs. *Discrete Appl Math*. 1999;98(1–2):1–19.
10. Rahman A, Medvedev P. Assembler artifacts include misassembly because of unsafe units and underassembly because of bidirected graphs. *Genome Res*. 2022;32(9):1746–53.
11. Chikhi R, Limasset A, Medvedev P. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*. 2016;32(12):201–8.
12. Bankevich A, Bzikadze AV, Kolmogorov M, Antipov D, Pevzner PA. Multiplex de Bruijn graphs enable genome assembly from long, high-fidelity reads. *Nat Biotechnol*. 2022. <https://doi.org/10.1038/s41587-022-01220-6>.
13. Cazaux B, Lecroq T, Rivals E. From indexing data structures to de Bruijn graphs. In: Kulikov AS, Kuznetsov SO, Pevzner P, editors. *Symposium on combinatorial pattern matching*. Springer: Berlin; 2014. p. 89–99.
14. Kundeti V, Rajasekaran S, Dinh H. An efficient algorithm for Chinese postman walk on bi-directed de Bruijn graphs. In: Wu W, Daescu O, editors. *Combinatorial optimization and applications*. Berlin, Heidelberg: Springer; 2010. p. 184–96.
15. Gusfield D. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge: Cambridge University Press; 1997. <https://doi.org/10.1017/cbo9780511574931>.
16. Mäkinen V, Belazzougui D, Cunial F, Tomescu AI. *Genome-scale algorithm design*. Cambridge: Cambridge University Press; 2015.
17. Ukkonen E. On-line construction of suffix trees. *Algorithmica*. 1995;14(3):249–60.
18. Farach M. Optimal suffix tree construction with large alphabets. In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE. 1997; p. 137–43.
19. Belazzougui D, Kosolobov D, Puglisi SJ, Raman R. Weighted ancestors in suffix trees revisited. In: *32nd Annual Symposium on Combinatorial Pattern Matching*. 2021.
20. Belazzougui D, Cunial F. Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In: *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. 2019; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
21. Belazzougui D, Cunial F, Kärkkäinen J, Mäkinen V. Versatile succinct representations of the bidirectional burrows-wheeler transform. In: Bodlaender HL, Italiano GF, editors. *European symposium on algorithms*. Springer: Berlin; 2013. p. 133–44.
22. Fleischer H. *Eulerian graphs and related topics*. The Netherlands: Elsevier; 1990.
23. Crawford VG, Kuhnle A, Boucher C, Chikhi R, Gagie T. Practical dynamic de Bruijn graphs. *Bioinformatics*. 2018;34(24):4189–95.
24. Muggli MD, Bowe A, Noyes NR, Morley PS, Belk KE, Raymond R, Gagie T, Puglisi SJ, Boucher C. Succinct colored de Bruijn graphs. *Bioinformatics*. 2017;33(20):3181–7.
25. Muggli MD, Alipanahi B, Boucher C. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics*. 2019;35(14):51–60.
26. Köster J, Rahmann S. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*. 2012;28(19):2520–2.
27. Grüning B, Dale R, Sjödin A, Chapman BA, Rowe J, Tomkins-Tinch CH, Valieris R, Köster J. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nat Methods*. 2018;15(7):475–6.
28. Schmidt S. Euler tigs experiments. Zenodo. 2022. <https://doi.org/10.5281/zenodo.7371148>.
29. Schmidt S. Euler tigs. Zenodo. 2022. <https://doi.org/10.5281/zenodo.7371184>.
30. Schmidt S. Matchtigs. GitHub. <https://github.com/algbio/matchtigs>. Accessed 15 Apr 2023.
31. Schmidt S. Matchtigs. Bioconda. <https://anaconda.org/bioconda/matchtigs>. Accessed 15 Apr 2023.
32. Schmidt S. Matchtigs. Crates.io. <https://crates.io/crates/matchtigs>. Accessed 15 Apr 2023.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.