RESEARCH

Matchtigs: minimum plain text representation of kmer sets

Sebastian Schmidt[1*], Shahbaz Khan[1], Jarno Alanko[1,2] and Alexandru I. Tomescu[1*]

*Correspondence:
sebastian.schmidt@helsinki.fi;
alexandru.tomescu@helsinki.fi
[1]Department of Computer Science, University of Helsinki, Helsinki, Finland
Full list of author information is available at the end of the article

Abstract

Kmer-based methods are widely used in bioinformatics, which raises the question of what is the smallest practically usable representation (i.e. plain text) of a set of kmers. We propose a polynomial algorithm computing a *minimum* such representation (which was previously posed as a potentially NP-hard open problem), as well as an efficient near-minimum greedy heuristic. When compressing genomes of large model organisms, read sets thereof or bacterial pangenomes, with only a minor runtime increase, we decrease the size of the representation by up to 60% over unitigs and 27% over previous work. Additionally, the number of strings is decreased by up to 97% over unitigs and 91% over previous work. Finally we show that a small representation has advantages in downstream applications, as it speeds up queries on the popular kmer indexing tool Bifrost by 1.66× over unitigs and 1.29× over previous work.

Availability:
https://github.com/algbio/matchtigs

Keywords:
kmer sets; plain text compression; graph algorithm; sequence analysis; genomic sequences; minimum-cost flow; Chinese postman problem

1 Background

Motivation. The field of kmer-based methods has seen a surge of publications in the last years. Examples include alignment-free sequence comparison [1, 2, 3], variant calling and genotyping [4, 5, 6, 7, 8], transcript abundance estimation [9], metagenomic classification [10, 11, 12, 13], abundance profile inference [14], indexing of variation graphs [15, 16], estimating the similarity between metagenomic datasets [17], species identification [18, 19] and sequence alignment to de Bruijn graphs [20, 21, 22, 23]. All these methods are based mainly on kmer sets, i.e. on the existence or non-existence of kmers. They ignore further information like for example predecessor and successor relations between kmers which are represented by the topology of a de Bruijn graph [24, 25, 26].

On the other hand, many classical methods such as genome assemblers [26, 27, 28, 29, 30, 31, 32, 33, 34, 35] and related algorithms [36, 37, 38], are based on de Bruijn graphs and their topology. To increase the efficiency of these methods, the graphs are usually compacted by contracting all paths where all inner nodes have in- and outdegree one. These paths are commonly known as *unitigs*, and their first usage can be traced back to [39]. Since unitigs contain no branches in their inner nodes, they do not alter the topology of the graph, and in turn enable the exact same set of analyses. There are highly engineered solutions available to compute a compacted de Bruijn graph by computing unitigs from any set of strings in memory [23] or with external memory [33]. Incidentally, the set of unitigs computed from a set of strings is also a way to store a set of kmers without repetition, and thus in reasonably small space. However, the necessity to preserve the topology of the graph makes unitigs an inferior choice to represent kmer sets, as the sum of their length is still far from optimal, and they consist of many separate strings. The possibility to ignore the topology for kmer-based methods opens more leeway in their representation that can be exploited to reduce the resource consumption of existing and future bioinformatics tools.

The need for such representations becomes apparent when observing the amount of data available to bioinformaticians. For example, the number of complete bacterial genomes available in RefSeq [40] more than doubled between May 2020 and July 2021 from around 9000 [41] to around 21000[1]. And with the ready availability of modern sequencing technologies, the amount of genomic data will increase further in the next years. In turn, analysing this data requires an ever growing amount of computational resources. But this could be relieved through a smaller representation that reduces the RAM usage and speeds up the analysis tools, and thereby allows to run larger pipelines using less computational resources. To fulfil this goal, a *plain text* representation would be the most useful: if the representation has to be decompressed before usage, then this likely erases the savings in RAM and/or adds additional runtime overhead. Formally, a plain text representation is a set of strings that contains each kmer from the input strings (forward, reverse-complemented, or both) and no other kmer. We denote such a set as a *spectrum preserving string set* (SPSS), borrowing the naming from [42] and redefining it slightly[2]. Such a

[1] Our own observation.

[2] While in [42], an SPSS must not contain a kmer twice and must not contain its reverse complement, in our definition (Definition 1 in Section 5.2) these repetitions are explicitly allowed.

plain text representation has the great advantage that some tools (like e.g. Bifrost's query [23]) can use it without modification, and we expect that even those tools that require modifications would not be hard to modify.

Related work. There are two recent attempts at giving a small plain text representation of a kmer set. Both *simplitigs* computed by ProphAsm [41], and *UST-tigs*[3] computed by the UST algorithm [42] greedily compute a node-disjoint path cover of the de Bruijn graph of the kmer set. They are a simultaneous discovery of the same concept, and differ only in some details as well as in the implementation. Both greatly reduce the number of strings (*string count*, SC) as well as the total amount of characters in the strings (*cumulative length*, CL) required to store a kmer set. In [41] the authors show that both SC and CL are greatly reduced for very tangled de Bruijn graphs, like graphs for single large genomes with small kmer length and pangenome graphs with many genomes. The authors of [42] show a significant reduction in SC and CL on various data sets as well as further merits in downstream applications. Specifically, the authors of [41] show an improvement in run time of kmer queries using general-purpose text indexing tools.

In [42] the authors also prove a lower bound on the cumulative length of an SPSS (under their more restricted definition of an SPSS[2]), and show that the length of their SPSS is very close to the lower bound. Further, the authors of both [41] and [42] consider whether computing such a minimum SPSS without repeating kmers might be NP-hard. All these suggest that no further progress might be possible under their restricted SPSS definition.

In the wider field of finding small representations of kmer sets that are not necessarily in plain text, there exists for example ESSCompress [43], which uses an extended DNA alphabet to encode similar kmers in smaller space. Another non-plain text representation is REINDEER [44], which uses substrings of unitigs with kmers of similar abundance to not just store the existence of kmers, but also their abundance in each single genome of a pangenome. Lastly, in [45] the authors use an algorithm similar to ProphAsm and UST to compress multiple kmer sets by separating the unique kmer content of each set from the kmer content shared with other sets.

Our contribution. In this paper we propose the first algorithm to find an SPSS of *minimum* size (CL). Moreover, in contrast to the potential NP-hardness of the restricted definition of an SPSS by [41, 42] (forbidding repeated kmers), we show that the SPSS problem allowing repeated kmers is polynomially solvable, based on a many-to-many min-cost path query and a min-cost perfect matching approach. We further propose a faster and more memory-efficient heuristic to compute a small SPSS that skips the optimal matching step, but still produces close to optimal results in CL, and even better results in terms of SC.

Our experiments over references and read datasets of large model organisms and bacterial pangenomes show that the CL decreases by up to 27% and the SC by up

[3] The paper does not give any specific name for the resulting strings of the UST algorithm, so we named them ourselves.

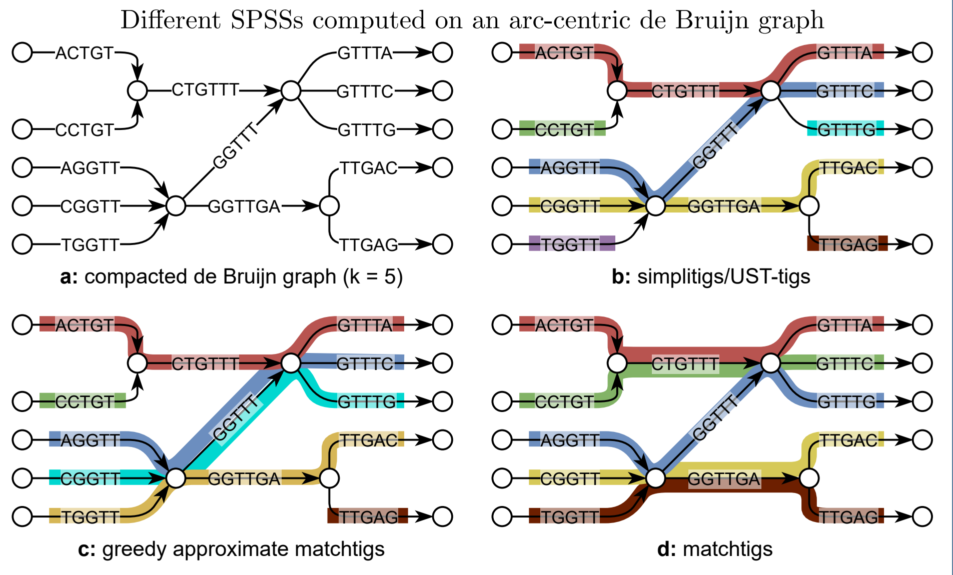
Different SPSSs computed on an arc-centric de Bruijn graph
ACTGT
CCTGT
CTGTTT
GTTTA
GTTTC
GTTTG
GGTTT
AGGTT
CGGTT
TGGTT
GGTTGA
TTGAC
TTGAG
a: compacted de Bruijn graph (k = 5)
b: simplitigs/UST-tigs
c: greedy approximate matchtigs
d: matchtigs

Figure 1 $k = 5$. For simplicity, the reverse complements of all nodes and arcs are omitted. **a:** the original de Bruijn graph in compacted form has 13 unitigs with 70 total characters. **b:** example of simplitigs or UST-tigs with 7 strings and 43 total characters. **c:** example of greedily approximated matchtigs with 6 strings and 40 total characters. **d:** example of matchtigs with 5 strings and 39 total characters.

to 91% over UST[4]. Compared to unitigs, the CL decreases by up to 60% and SC by up to 97%. These improvements come often at just minor costs, as computing our small representation (which includes a run of BCALM2) takes less than twice as long than computing unitigs with BCALM2, and takes less than 35% longer in most cases. Even if the memory requirements for large read datasets increase, they stay within the limits of a modern server.

Finally we show that besides the smaller size of a minimum SPSS, it also has advantages in downstream applications. As an example of a kmer-based method, we query our compressed representation with the tool Bifrost [23]. This is a state-of-the-art tool supporting kmer-based queries in genomic sequences, using a representation of a kmer set as a set of unitigs. By simply replacing unitigs with greedy matchtigs (and without modifying Bifrost in any way), we get a speedup of 1.66× over unitigs, and 1.29× over UST-tigs and simplitigs.

2 Results

2.1 Matchtigs as a minimum plain text representation of kmer sets

We introduce the *matchtig algorithm* that computes a character-minimum SPSS for a set of genomic sequences. While former heuristics (ProphAsm, UST) did not allow to repeat kmers, our algorithm explicitly searches for all opportunities to reduce the character count in the SPSS by repeating kmers. Consider for example the arc-centric de Bruijn graph in Figure 1a. When representing its kmers without

[4]ProphAsm supports only $k \leq 32$ such that a comparison is often impossible. But where it is possible, it performs only slightly better than UST in terms of CL and SC.

repetition as in Figure 1b, we need 43 characters and 7 strings. But if we allow to repeat kmers as in Figure 1d, we require only 39 characters and 5 strings. It turns out that structures similar to this example occur often enough in real genome graphs to yield significant improvements in both character and string count of an SPSS.

Similar to previous heuristics, our algorithm works on the compacted bidirected de Bruijn graph of the input sequences. However, we require an arc-centric de Bruijn graph, but this can be easily constructed from the node-centric variant (see Section 5.3). In this graph we find a min-cost circular biwalk that visits each biarc at least once, and that can jump between arbitrary nodes at a cost of $k-1$. This formulation is very similar to the classic Chinese postman problem [46], formulated as follows: find a min-cost circular walk in a directed graph that visits each arc at least once. This similarity allows us to adapt a classic algorithm from Edmonds and Johnson that solves the Chinese postman problem [47]. They first reduce the problem to finding a min-cost Eulerisation via a min-cost flow formulation, and then further reduce that to min-cost perfect matching using a many-to-many min-cost path query between unbalanced nodes. It is also similar to a previous theoretical work [48], where the authors solve the Chinese postman problem in a bidirected de Bruijn graph by finding a min-cost Eulerisation via a min-cost flow formulation. As opposed to Edmonds and Johnson and us, in [48] the authors propose to solve the min-cost flow problem directly with a min-cost flow solver. We believe this to be infeasible for our problem, since the arbitrary jumps between nodes require the graph in the flow formulation to have arcs quadratic in the number of nodes.

Our resulting algorithm is polynomial but while it runs fast for large bacterial pangenomes, it proved practically infeasible to build the matching instance for very large genomes ($\geq$ 500Mbp). This is because each of the min-cost paths found translates into roughly one edge in the matching graph, and the number of min-cost paths raises quadratically if the graph gets denser. Thus, our algorithm ran out of memory when constructing it for larger genomes. Hence, for practical purposes, we introduce a greedy heuristic to compute approximate matchtigs. This heuristic does not build the complete instance of the matching problem, but just greedily chooses the shortest path from each unbalanced node to Eulerise the graph. This reduces the amount of paths per node to at most one, and as a result, the heuristic uses significantly less memory, runs much faster, and achieves near optimal speedups when run with multiple threads (see Additional file 5). While it can in theory produce suboptimal results as in Figure 1c, in practice, the size of the greedily computed strings is very close to that of matchtigs, and the number of strings is always smaller.

Moreover, the minimality of matchtigs allows us to exactly compare, for the first time, how close heuristic algorithms for SPSS are to the optimum (on smaller genomes and on bacterial pangenomes, due to the resource-intensiveness of optimal matchtigs).

Our implementations are available[5] as both a library and a command line tool, both written in Rust.

Table 1 Quality and performance of compressing model organisms

genome	algorithm	CL ratio	SC ratio	time [s]		memory [GiB]	
C. elegans (reads)	unitigs	1.00	1.00	2716		5.94	
	UST-tigs	0.58	0.38	3738	(1.38)	15.2	(2.55)
	gMatchtigs	0.45 (0.77)	0.11 (0.30)	4246	(1.56)	73.3	(12.3)
B. mori (reads)	unitigs	1.00	1.00	8467		9.32	
	UST-tigs	0.55	0.36	11738	(1.39)	52.4	(5.62)
	gMatchtigs	0.40 (0.73)	0.06 (0.18)	16331	(1.93)	227	(24.3)
H. sapiens (reads)	unitigs	1.00	1.00	127516		12.9	
	UST-tigs	0.72	0.49	128345	(1.01)	16.4	(1.27)
	gMatchtigs	0.63 (0.87)	0.27 (0.55)	131534	(1.03)	77.9	(6.02)
C. elegans	unitigs	1.00	1.00	68.3		1.21	
	UST-tigs	0.95	0.35	71.4	(1.05)	1.21	(1.00)
	gMatchtigs	0.93 (0.98)	0.07 (0.19)	74.9	(1.10)	1.21	(1.00)
	matchtigs	0.93 (0.98)	0.08 (0.22)	77.7	(1.14)	1.21	(1.00)
B. mori	unitigs	1.00	1.00	270		3.31	
	UST-tigs	0.81	0.35	303	(1.12)	3.31	(1.00)
	gMatchtigs	0.75 (0.93)	0.06 (0.18)	355	(1.31)	3.31	(1.00)
H. sapiens	unitigs	1.00	1.00	2314		10.0	
	UST-tigs	0.85	0.34	2520	(1.09)	10.0	(1.00)
	gMatchtigs	0.80 (0.94)	0.03 (0.09)	3136	(1.35)	13.8	(1.38)

We chose $k = 51$ and a min abundance of 10 for Homo sapiens reads and 1 for all others. The CL and SC ratios are between compressed strings and unitigs, and in parentheses are the ratios between our algorithm and the best competitor (UST). 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, gMatchtigs and matchtigs require an additional tool to be run on the unitigs. The number in parentheses behind time and memory indicates the slowdown/increase over computing just unitigs with BCALM2. All algorithms were run with 28 threads, except for UST which supports only one thread (the preceding run of BCALM2 was still executed with 28 threads). Matchtigs run out of memory for all but C. elegans, so only the corresponding runs are shown. We could not compare against ProphAsm, as it only handles $k \leq 32$. 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.

2.2 Compression of model organisms

We evaluate the performance of our proposed algorithms on three model organisms: *C. elegans*, *B. mori* and *H. sapiens*. We benchmark the algorithms on both sets of short reads (average length 300 for C. elegans and B. mori, and 296 for H. sapiens) and reference genomes of these organisms. On human reads, we filter the data during processing so that we keep only kmers that occur at least 10 times (min abundance = 10).

We use the metrics cumulative length (CL) and string count (SC) as in [41]. The CL is the total number of characters in all strings in the SPSS, and the SC is the number of strings. We evaluate our algorithms against the same large genomes as in [41], using both the reference genome and a full set of short reads of the respective species (see Table 1 for the results). Since UST as well as matchtigs and greedy matchtigs require unitigs as input, and specifically UST needs some extra information in a format only output by BCALM2 [33], we run BCALM2 to compute unitigs from the input strings. Due to the size of the genomes we chose $k = 51$, which is not supported by ProphAsm. Further, for all data sets but the C. elegans reference genome the matchtigs algorithm ran out of memory, so we only compute greedy matchtigs for those.

On read data sets where we keep all kmers, our greedy heuristic achieves an improvement of up to 27% CL and 82% SC over the best competitor (UST-tigs). The human read data set has smaller improvements, however it was processed with a min abundance of 10, yielding longer unitigs with less potential for compression. On reference genomes the improvement in CL is smaller with up to 7%, however the improvement in SC is much larger with up to 91%.

On reference genomes our greedy heuristic achieves an improvement of up to 7% CL and 91% SC over the best competitor (UST-tigs). For C. elegans, where computing matchtigs is feasible as well, we observe that they yield no significant improvement in CL, but are even slightly worse in SC than the greedy heuristic. See Additional file 1 for more quality measurements with different kmer size and min. abundance.

We assume that the improvements correlate inversely with the average length of maximal unitigs of the data set. Our approach achieves a smaller representation by joining unitigs with overlapping ends, avoiding the repetition of those characters. This has a natural limit of saving at most $k-1$ characters per pair of unitigs joint together, so at most $k-1$ characters per unitig. In turn, the maximum fraction of characters saved is bound by $k-1$ divided by the average length of unitigs. In Additional file 1 we have varied the kmer size and min. abundance for our data sets to vary the average length of unitigs. This gives us visual evidence for a correlation between average unitig length and decrease in CL.

Our improvements come at often negligible costs in terms of time and memory. Even for read sets, the run time at most doubles compared to BCALM2 in the worst case. However, the memory consumption rises significantly for read sets. This is due to the high number of unitigs in those graphs and the distance array of Dijkstra's algorithm, whose size is linear in the number of nodes and the number of threads. See Additional file 2 for more performance measurements with different kmer size and min. abundance.

[5] https://github.com/algbio/matchtigs

Table 2 Quality and performance of compressing pangenomes

pangenome	tigs	CL ratio	SC ratio	time [s]		memory [MiB]	
1102x N. gonorrhoeae	unitigs	1.00	1.00	33.5		5235	
	UST-tigs	0.63	0.35	35.1	(1.05)	5235	(1.00)
	simplitigs	0.62	0.33	894	(26.7)	181	(0.03)
	gMatchtigs	0.57 (0.93)	0.18 (0.54)	35.6	(1.06)	5235	(1.00)
	matchtigs	0.57 (0.92)	0.18 (0.56)	36.6	(1.09)	5235	(1.00)
616x S. pneumoniae	unitigs	1.00	1.00	25.4		3165	
	UST-tigs	0.61	0.35	31.5	(1.24)	3165	(1.00)
	simplitigs	0.60	0.33	494	(19.4)	476	(0.15)
	gMatchtigs	0.53 (0.89)	0.13 (0.41)	31.2	(1.23)	3165	(1.00)
	matchtigs	0.52 (0.88)	0.14 (0.44)	46.9	(1.84)	3165	(1.00)
3682x E. coli	unitigs	1.00	1.00	742		7013	
	UST-tigs	0.60	0.35	819	(1.10)	7013	(1.00)
	simplitigs	0.59	0.32	10677	(14.4)	8075	(1.15)
	gMatchtigs	0.51 (0.87)	0.11 (0.33)	853	(1.15)	7765	(1.11)
	matchtigs	0.50 (0.85)	0.12 (0.37)	1342	(1.81)	8301	(1.18)

We chose $k = 31$ and a min abundance of 1. The CL and SC ratios are between compressed strings and unitigs, and in parentheses are the ratios between our algorithm and the best competitor (simplitigs). 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, gMatchtigs and matchtigs require an additional tool to be run on the unitigs and simplitigs are computed directly from the source data. The number in parentheses behind time and memory indicates the slowdown/increase over computing just unitigs with BCALM2. All algorithms were run with 28 threads, except for UST and ProphAsm which support only one thread (for UST, the preceding run of BCALM2 was still executed with 28 threads). 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.

2.3 Compression of pangenomes

In addition to model organisms with large genomes, we evaluate our algorithms on bacterial pangenomes of *N. gonorrhoeae*, *S. pneumoniae* and *E. coli*. We use the same metrics as for model organisms and since the genomes are bacterial, we choose $k = 31$. We show the results in Table 2. See Additional file 3 for more quality measurements with different kmer size and min. abundance, and Additional file 4 for more quality measurements with different kmer size and min. abundance.

Our algorithms improve CL up to 15% (using matchtigs) over the best competitor and SC up to 67% (using greedy matchtigs). Matchtigs always achieve a slightly lower CL and slightly higher SC than greedy matchtigs, but the CL of greedy matchtigs is always at most 2% worse than that of matchtigs. We again assume that the improvements are correlated inversely to the average size of unitigs, as suggested by the experiments in Additional file 3. These improvements come at negligible costs, using at most 23% more time and 11% more memory than BCALM2 when computing greedy matchtigs. For matchtigs, the time increases by less than a factor of two and memory by at most 18% compared to BCALM2.

2.4 Kmer-based short read queries

Matchtigs have further applications beyond merely reducing the size required to store a set of kmers. Due to their smaller size and lower string count, they can make downstream applications more efficient. For example, the kmer-based query tool Bifrost [23] achieves speedups of 1.66 when using matchtigs instead of unitigs, and 1.29 when using matchtigs instead of simplitigs (see Table 3 for the detailed results). Note that we measure the speedup of the search phase only, since the index phase is independent of the query and therefore could be moved into a separate preprocessing

Table 3 Performance characteristics of querying different tigs with Bifrost.

genome	tigs	indexing [s]	searching [s]	total [s]	search speedup		mem [GiB]
3682x E. coli	unitigs	21.00	336.79	360.37	1.00		2.39
	UST-tigs	16.11	267.02	285.52	1.26		2.13
	simplitigs	15.55	261.65	278.91	1.29		2.10
	gMatchtigs	16.19	208.10	226.23	1.62	(1.26)	2.12
	matchtigs	16.33	202.71	220.73	1.66	(1.29)	2.09

The Bifrost query command is run with 16 threads and default settings. 3682 E.coli genomes are queried with all raw reads used to assemble 30 of the genomes. All columns are averages of 10 experiments. The search speedup is with respect to unitigs, and the search speedup in parentheses is with respect to simplitigs. Indexing is the time required to build the minimizer index within Bifrost, and searching is the time required to check if a read is in the pangenome using the minimizer index. The total time is the wall-clock time of bifrost.

step. Note also that these speedups are consistent with those reported when using simplitigs [41] for kmer queries with BWA-MEM [49] (without modification of BWA-MEM).

We also achieve our speedups *without modifying our query tool Bifrost*[6], but just by passing Bifrost a file with our tigs instead of unitigs. This works because for the query, Bifrost does not require any topology information, but just a set of kmers. It builds a minimizer index over the input tigs to efficiently find a given kmer. This process is sped up by the smaller input size, and the lower number of strings. Then, when querying, it iterates over the kmers of the query. It finds the tig containing the kmer using the minimizer index and then extends that match until the query and the tig differ, or the query or tig ends. In that case, it again queries the minimizer index, until all query kmers are checked. This process can be sped up by longer tigs, since then the end of a containing string is reached more rarely. Further, a smaller representation decreases the memory consumption, however does not decrease further when changing from simpltigs or UST-tigs to matchtigs or greedy matchtigs. We assume that this is due to the longer strings requiring more memory while being loaded block-wise in ASCII format before they get stored in a two-bits-per-character compressed format.

3 Discussion

Kmer-based methods have found wide-spread use in many areas of bioinformatics over the past years. However, they usually rely on unitigs to represent the kmer sets, since they can be computed efficiently with standard tools [33, 23]. Unitigs have the additional property that the de Bruijn graph topology can easily be reconstructed from them, since they do not contain branching nodes other than on their first and last kmer. However, this property is not usually required by kmer-based methods, which has opened the question if a smaller set of strings other than unitigs can be used to represent the kmer sets. If such a representation was in plain text, it should be usable in most kmer-based tools, by simply feeding it to the tool instead of unitigs.

Previous work has relaxed the unitig requirement of the representation of the kmer sets to arbitrary strings without kmer repetitions. This resulted in a smaller representation, leading to improvements in downstream applications. Additionally, previous work considered whether that finding an optimal representation without

[6] We did need to add code to measure index and search time separately.

repeated kmers is NP-hard. We have shown that finding that kind of optimum is not necessary, since we get a polynomial algorithm that achieves better compression and improvements in downstream applications if we allow the representation to repeat kmers.

4 Conclusions

Our *optimum* algorithm compresses the representation significantly more than previous work. For practical purposes we also propose a greedy heuristic that achieves near-optimum results, while being suitable for practical purposes in runtime and memory. Specifically, our algorithms achieve a decrease of 27% in size and 91% in string count over UST. Additionally we have shown that our greedy representation speeds up downstream applications, giving an example with a factor of 1.29 compared to previous compressed representations.

Our implementation is available as a stand-alone command-line tool and as a library. We hope that our efficient algorithms result in a wide-spread adoption of near-minimum plain-text representations of kmer sets in kmer-based methods, resulting in more efficient bioinformatics tools.

5 Methods

We first give some preliminary definitions in Section 5.1 and define our problem in Section 5.2. Note that to stay closer to our implementation, our definitions of bidirected de Bruijn graphs differ from those in e.g. [48]. However, the concepts are fundamentally the same. Then in Sections 5.3 to 5.7 we describe how to compute matchtigs. The whole algorithm is summarised by an example in Figure 2. For simplicity, we describe the algorithm using an uncompacted de Bruijn graph. However, in practice it is much more efficient to use a compacted de Bruijn graph, but our algorithm can be adapted easily: simply replace the costs of 1 for each original arc with the number of uncompacted arcs it represents. In Section 5.8 we describe the greedy heuristic.

5.1 Preliminaries

We are given an alphabet Γ and all strings in this work have only characters in Γ. Further, we are given a bijection $\text{comp} : \Gamma \to \Gamma$. The *reverse complement* of a string s is $s^{-1} := \text{rev}(\text{comp}*(S))$ where rev denotes the reversal of a string and $\text{comp}*$ the character-wise application of comp. For an integer k, a string of length k is called a *kmer*. From here on, we only consider strings of lengths at least k, i.e. strings that have at least one kmer as substring. We denote the prefix of length $k-1$ of a kmer s by $\text{pre}(s)$ and its suffix of length $k-1$ by $\text{suf}(s)$. The *spectrum* of a set of strings S is defined as the set of all kmers and their reverse complements that occur in at least one string $s \in S$, formally $\text{spec}_k(S) := \{r \in \Gamma^k \mid \exists s \in S : r \text{ or } r^{-1} \text{ is substring of } s\}$.

An *arc-centric de-Bruijn graph* (or short *de-Bruijn graph*) $\text{DBG}_k(S) = (V, E)$ of order k of a set of strings S is defined as a standard directed graph with nodes $V := \{\text{pre}(s) \mid s \in \text{spec}_k(S)\} \cup \{\text{suf}(s) \mid s \in \text{spec}_k(S)\}$ and arcs $E := \{(\text{pre}(s), \text{suf}(s)) \mid s \in \text{spec}_k(S)\}$. On top of this, we use the following notions of bidirectedness. An ordered pair of reverse-complementary nodes $[v, v^{-1}] \in V \times V$ is called a *binode* and

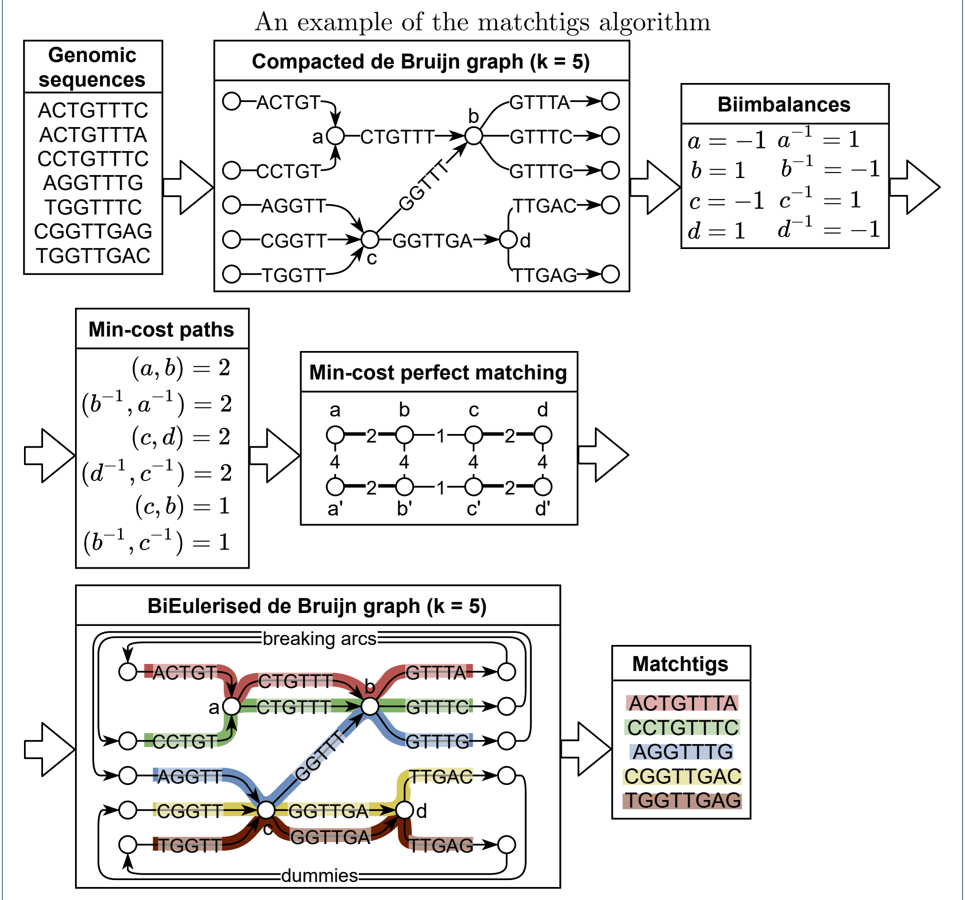

Figure 2 We first build a compacted de Bruijn graph (for simplicity, the reverse complements of the nodes and arcs have been left out), in which we then compute the bi-imbalances (for simplicity, we left out the bi-imbalances of source and sink nodes). Between nodes with nonzero bi-imbalance we compute the min-cost paths and build a min-cost perfect matching instance from the paths (solution edges highlighted in bold). Then we biEulerise the de Bruijn graph by multiplying the biarcs corresponding to solution edges of the matching problem, and inserting arbitrary breaking arcs between remaining unmatched nodes. Finally we compute a biEulerian cycle in the resulting graph, and cut it apart at all breaking arcs to create the matchtigs.

an ordered pair of reverse-complementary arcs $[(a, b), (b^{-1}, a^{-1})] \in E \times E$ is called a *biarc*. Even though these pairs are ordered, reversing the order still represents the same binode/biarc, just in the other direction. If an arc or a node is its own reverse-complement (called *self-complemental*), then it is written as biarc $[(a, b)]$ or binode $[v]$. See Figure 3 for examples of different bigraphs.

Since de Bruijn graphs are defined as standard directed graphs, we use the following standard definitions. The set of incoming (outgoing) arcs of a node is denoted by $E^-(v)$ ($E^+(v)$), and the indegree (outdegree) is $d^-(v) := |E^-(v)|$ ($d^+(v) := |E^+(v)|$). A *walk* in a de Bruijn graph is a sequence of adjacent arcs (followed in the forward direction) and a *unitig* is a walk in which all inner nodes (nodes with at least two incident walk-arcs) have exactly one incoming and one outgoing arc. The length $|w|$ of a walk w is the length of the sequence (counting repeated arcs as often as they are repeated). A *compacted de-Bruijn graph* is a de Bruijn graph in which all maximal unitigs have been replaced by a single arc. A

Examples of de Bruijn graphs

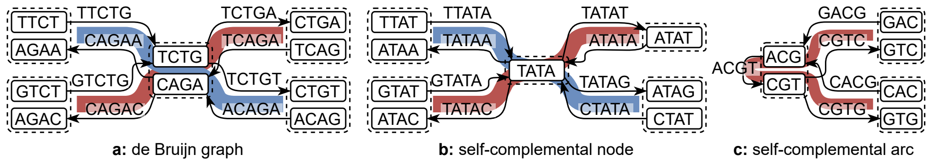

Figure 3 Binodes are surrounded by a dashed box, where self-complemental binodes contain only one graph node. **a:** a de Bruijn graph of the strings TTCTGA and GTCTGT. The colored lines are an arc-covering set of biwalks. **b:** a de Bruijn graph containing two self-complemental nodes. The colored lines are a set of biwalks that visit each biarc adjacent to [TATA] exactly once. **c:** a de Bruijn graph containing a self-complemental arc. The colored line is a biwalk that visits each biarc adjacent to [ACG, CGT] exactly once.

circular walk is a walk that starts and ends in the same node, and a *Eulerian cycle* is a circular walk that contains each arc exactly once. A graph that admits a Eulerian cycle is *Eulerian*.

Assuming the complemental pairing of nodes and arcs defined above, we can define the following bidirected notions of walks and standard de Bruijn graph concepts. *Biwalks* and *circular biwalks* are defined equivalently to walks, except that they are sequences of biarcs. A biwalk w in a de Bruijn graph spells a string $\text{spell}(w)$ of overlapping visited kmers. That is, $\text{spell}(w)$ is constructed by concatenating the string a from w's first biarc $[(a,b),(b^{-1},a^{-1})]$ (or $[(a,b)]$) with the last character of b of the first and all following biarcs. See Figure 3 for examples of bidirected de Bruijn graphs and notable special cases.

5.2 Problem overview

We are given a set of input strings I where each string has length at least k, and we want to compute a minimum spectrum preserving string set, defined as follows.

Definition 1 (SPSS) *A* spectrum preserving string set *(or* SPSS*) of I is a set S of strings of length at least k such that $\text{spec}_k(I) = \text{spec}_k(S)$, i.e. both sets of strings contain the same kmers, either directly or as reverse complement.*

Note that our definition allows kmers and their reverse complements to be repeated in the SPSS, both in the same string and in different strings. This is the only difference to the definition for UST [42] and simplitigs [41], which can be formalised as a set of strings S_{UST} of length at least k such that for each kmer in the spectrum, either itself or its reverse complement (but *not both*) occurs *exactly once* in *exactly one* string in S_{UST}.

Definition 2 (Minimum SPSS) *The* size *$||S||$ of an SPSS S is defined as*

$$||S|| = \sum_{s \in S} |s|,$$

where $|s|$ denotes the length of string s. A minimum *SPSS is an SPSS of minimum size.*

On a high level, our algorithm functions as follows (see also Figure 2).

1 Create a bidirected de Bruijn graph from the input strings (see Section 5.3).
2 Compute the *bi-imbalances* of each node (see Section 5.5).
3 Compute the min-cost bipaths of length at most $k-1$ from all nodes with negative bi-imbalance to all nodes with positive bi-imbalance (see Section 5.7).
4 Solve a min-cost matching instance with costs for unmatched nodes to choose a set of shortest bipaths with minimum cost by reduction to min-cost perfect matching (see Section 5.6).
5 *BiEulerise* the graph with the set of bipaths as well as arbitrary arcs between unmatched nodes.
6 Compute a *biEulerian cycle* in the resulting graph (see Section 5.5).
7 Break the cycle into a set of biwalks and translate them into a set of strings, which is the output minimum SPSS (see Section 5.4).

Note that in our implementation, a substantial difference is that we do not build the de Bruijn graph ourselves, but we use a compacted de Bruijn graph that we build from a set of maximal unitigs computed with BCALM2.

5.3 Building a compacted bidirected arc-centric de Bruijn graph from a set of strings

When building the graph we first compute unitigs from the input strings using BCALM2. Then we initialise an empty graph and do the following for each unitig:

1 We insert the unitig's first $k-1$-mer and its reverse complement as binode by inserting the two nodes separately and marking them as a bidirected pair, if it does not already exist. The existence is tracked with a hashmap, storing the two nodes corresponding to a kmer and its reverse complement if it exists.
2 We do the same for the last $k-1$-mer of the unitig.
3 We add a biarc between the two binodes by inserting one forward arc between the forward nodes of the binodes, and one reverse arc between the reverse complement nodes of the binodes. The forward arc is labelled with the unitig, and the reverse arc is labelled with its reverse complement.

To save memory, we store the unitigs in a single large array, where each character is encoded as two-bit number. The keys of the hashmap and the labels of the arcs are pointers into the array, together with a flag for the reverse complement. Nodes do not need a label, as their label can be inferred from any of its incident arcs' label. Recall that in the description of our algorithm, we use an uncompacted graph only for simplicity.

5.4 Reduction to the bidirected partial-coverage Chinese postman problem

We first compute the arc-centric de-Bruijn graph $\text{DBG}_k(I)$ of the given input string set I as described in Section 5.3. In $\text{DBG}_k(I)$, an SPSS S is represented by a *biarc-covering* set of biwalks W (the reverse direction of a biarc does not need to be covered separately). That is a set of biwalks such that each biarc is element of at least one biwalk (see Figure 3a). According to the definition of spell, the size of S is related to W as follows:

$$||S|| = \sum_{w \in W} |\operatorname{spell}(w)| = |W|(k-1) + \sum_{w \in W} |w|. \tag{1}$$

Each walk costs $k-1$ characters because it contains the node a from its first biarc $[(a,b),(b^{-1},a^{-1})]$ (or $[(a,b)]$), and it additionally costs one character per arc.

To minimise $||S||$, we transform the graph by adding all arcs in $(V \times V) \setminus E$ and marking them as *breaking* arcs. We additionally add a cost function that assigns all *original* (non-breaking) biarcs a cost of 1 and all breaking biarcs a cost of $k-1$. In the transformed graph we find a circular biwalk w^* of minimum cost that covers at least all original biarcs. This is similar to the directed Chinese postman problem (DCPP). In the DCPP, the task is to find a circular min-cost arc-covering walk in a directed graph. It is a classical problem, known to be solvable in $O(n^3)$ time [50] with a flow-based algorithm using e.g. [51] to compute min-cost flows. The partial coverage variant of the DCPP (requiring to cover only a subset of the arcs) is also known as the rural postman problem [52]. Further, the bidirected variant of the DCPP was discussed before in [48], and the authors also solved it using min-cost flow in $O(n^2 \log^2(n))$ time.

We break the resulting biwalk at all breaking biarcs if it contains any. In that case, the strings spelled by the resulting biarcs form an SPSS since they cover all original biarcs in the graph, and due to the choice of the cost function, the SPSS is minimum. If the resulting biwalk contains no breaking biarcs, we search for a longest sequence of biarcs that are repeated in forward or reverse. Then we break the circular biwalk into a linear one by removing those kmers. Since we have chosen a longest sequence of kmers, the resulting SPSS is minimum again. If the resulting biwalk repeats no biarc either, then we break it at an arbitrary node without removing any biarc. Since there is no repetition, the resulting SPSS is minimum again.

5.5 Solving the bidirected partial-coverage Chinese postman problem with min-cost integer flows

Edmonds and Johnson [47] introduced a polynomial-time flow-based approach that is adaptable to solve our variant of the DCPP. They show that finding a minimum circular arc-covering walk in a directed graph is equivalent to finding a minimum *Eulerisation* of the graph, and then any Eulerian cycle in the Eulerised graph. A Eulerisation is a multiset of arc-copies from the graph that makes the graph Eulerian if added[7], and a minimum Eulerisation is one whose sum of arc costs is minimum among all such multisets. To find such a minimum cost set of arcs, they formulate a min-cost flow problem as an integer linear program as follows:

$$\min \sum_{e \in E} c_e x_e \text{ s.t.} \tag{2}$$

$$x_e \text{ are non-negative integers, and} \tag{3}$$

$$\forall v \in V : \sum_{e \in E^-(v)} x_e - \sum_{e \in E^+(v)} x_e = d^+(v) - d^-(v). \tag{4}$$

The variables x_e are interpreted as the amount of flow through arc e, and the variables c_e denote the cost for assigning flow to an arc. The costs are equivalent to the arc costs in the weighted graph specified by the DCPP instance. Objective (2)

[7]Either by connecting nodes with missing outgoing arcs directly to nodes with missing incoming arcs, or by connecting them via a path of multiple arcs.

minimises the costs of inserted arcs as required. To ensure that the resulting flow can be directly translated into added arcs, Condition (3) ensures that the resulting flow is non-negative and integral. Lastly, Equation (4) is the *balance constraint*, ensuring that the resulting flow is a valid Eulerisation of the graph. This constraint makes nodes with missing outgoing arcs into *sources*, and nodes with missing incoming arcs into *sinks*, with demands matching the number of missing arcs. Note that in contrast to classic flow problems, this formulation contains no capacity constraint. For a solution of this linear program, the corresponding Eulerisation contains x_e copies of each arc e.

To adapt this formulation to our variant of the DCPP, we need to make three modifications, namely: allow for partial coverage, adjust the balance constraint for biwalks and ensure that the resulting flow is *bidirected*, i.e. that the flow of each arc equals the flow of its reverse complement.

Partial coverage. In the partial coverage Chinese postman problem, we are additionally given a set $F \subseteq E$ of arcs to be covered. In contrast to the standard DCPP, a solution walk only needs to cover all the arcs in F. In our case, the set F is the set of original arcs of the graph before Eulerisation. To solve the partial coverage Chinese postman problem we define outgoing covered arcs $F^+(v) := F \cap E^+(v)$, and incoming covered arcs $F^-(v) := F \cap E^-(v)$ for a node v, as well as the covered outdegree $d_F^+(v) := |F^+(v)|$ and the covered indegree $d_F^-(v) := |F^-(v)|$. Then we reformulate the balance constraint as:

$$\forall v \in V : \sum_{e \in E^-(v)} x_e - \sum_{e \in E^+(v)} x_e = d_F^+(v) - d_F^-(v).$$

The resulting set of arcs is a minimum Eulerisation of the graph (V, F), and a Eulerian walk in this graph is equivalent to a minimum circular F-covering walk in the original graph.

Bidirected balance. In contrast to Edmonds and Johnson, we are interested in a minimum circular *bi*walk that covers each original *bi*arc. Analogue to the formulation for directed graphs, we define a *biEulerian cycle* to be a bidirected cycle that visits each biarc exactly once. Further, we define a *biEulerian graph* to be a graph that admits a biEulerian cycle, and a *biEulerisation* to be a multiset of biarc-copies from the graph that makes a graph biEulerian if added, and a minimum biEulerisation is one whose sum of arc costs is minimum among all such multisets.

We can compute a biEulerisation in the same way as we compute a Eulerisation, the only change is in the balance constraint. Observe that for a Eulerian graph, the *imbalance* $i_v := d^-(v) - d^+(v)$ is zero for each node [53], because each node is entered exactly as often as it is exited. For binodes, the definition of the *bi-imbalance* bi_v of a binode $[v, v^{-1}]$ or $[v]$ follows the same idea. However, in contrast to directed graphs, there are two (mutually exclusive[8]) special cases.

[8] Since the labels of an arc are of length k and those of a node are of length $k-1$, only one can have even length, so only one can be self-complemental for DNA alphabets.

Binodes $\phi = [v, v^{-1}] \in V \times V$ with $v \neq v^{-1}$ may have incident self-complemental arcs $[(v, v^{-1})]$ and/or $[(v^{-1}, v)]$ (see Figure 3c for an example). If e.g. only $[(v, v^{-1})]$ exists, then to traverse it, a biwalk needs to enter v twice. First, it needs to reach $[v, v^{-1}]$ via some biarc, and after traversing $[(v, v^{-1})]$, it needs to leave $[v^{-1}, v]$ via a different biarc, whose reverse complement enters $[v, v^{-1}]$. If only $[(v^{-1}, v)]$ exists, then the situation is symmetric. Therefore, for balance of $[v, v^{-1}]$, the self-complemental biarc $[(v, v^{-1})]$ requires two biarcs entering $[v, v^{-1}]$ and the self-complemental biarc $[(v^{-1}, v)]$ requires two biarcs leaving $[v, v^{-1}]$. If both self-complemental arcs exist (e.g. both $[(ATA, TAT)]$ and $[(TAT, ATA)]$ for a binode $[ATA, TAT]$), then a biwalk can traverse them consecutively from e.g. $[v, v^{-1}]$ by traversing first $[(v, v^{-1})]$ and then $[(v^{-1}, v)]$, ending up in $[v, v^{-1}]$ again, such that the self-complemental arcs have a neutral contribution to the bi-imbalance. Resulting, the bi-imbalance of ϕ is

$$bi_v = d^+(v) - d^-(v) + (\mathbb{1}_{(v,v^{-1}) \in E^+(v)} - \mathbb{1}_{(v^{-1},v) \in E^-(v)}),$$

where $\mathbb{1}_P$ is 1 if the predicate P is true and 0 otherwise.

For self-complemental binodes $\psi = [v] \in V$, there is no concept of incoming or outgoing biarcs, since any biarc can be used to either enter or leave $[v]$ (see Figure 3b for an example). Therefore, for balance, biarcs need to be paired arbitrarily, resulting in the bi-imbalance

$$bi_v \equiv d^+(v) \mod 2.$$

Finally, we include partial coverage to above bi-imbalance formulations by limiting the incoming and outgoing arcs to F. Further, to distinguish between self-complemental binodes and others, we denote the set of self-complemental binodes as $S \subseteq V$ and the set of binodes that are not self-complemental as $T := V \setminus S$. Then we get the following modified coverage constraint:

$$\begin{aligned} \forall v \in T : \quad & \sum_{e \in E^-(v)} x_e - \sum_{e \in E^+(v)} x_e \\ & = d_F^+(v) - d_F^-(v) + \left(\mathbb{1}_{(v,v^{-1}) \in F^+(v)} - \mathbb{1}_{(v^{-1},v) \in F^-(v)}\right), \text{ and} \quad (5) \\ \forall v \in S : \quad & \sum_{e \in E^+(v)} x_e + d_F^+(v) \equiv 0 \mod 2. \quad (6) \end{aligned}$$

Valid bidirected flow. To adapt Edmonds' and Johnson's formulation to biwalks, we additionally need to ensure that the resulting flow yields a set of biarcs, i.e. that each arc has the same flow as its reverse complement:

$$\forall e \in E : x_e = x_{e^{-1}} \quad (7)$$

Adapted flow formulation. With the modifications above, we can adapt the formulation of Edmonds and Johnson to solve the bidirected partial-coverage Chinese postman problem. We define F to be the arcs in the original graph, and set

$E := V \times V$. We further set $c_e = 1$ for $e \in F$ and $c_e = k - 1$ otherwise. Lastly we define S and T as above. Then we get the following modified formulation:

$$\min \quad \sum_{e \in E} c_e x_e \text{ s.t.} \tag{2}$$

$$x_e \text{ are non-negative integers, and} \tag{3}$$

$$\forall v \in T: \quad \sum_{e \in E^-(v)} x_e - \sum_{e \in E^+(v)} x_e$$
$$= d_F^+(v) - d_F^-(v) + \left(\mathbb{1}_{(v,v^{-1}) \in F^+(v)} - \mathbb{1}_{(v^{-1},v) \in F^-(v)}\right), \text{ and} \tag{5}$$

$$\forall v \in S: \quad \sum_{e \in E^+(v)} x_e + d_F^+(v) \equiv 0 \mod 2, \text{ and} \tag{6}$$

$$\forall e \in E: \quad x_e = x_{e^{-1}} \tag{7}$$

In this min-cost integer flow formulation[9] of the bidirected partial-coverage Chinese postman problem, analogue to the formulation of Edmonds and Johnson, sources and sinks are nodes with missing outgoing or incoming arcs with demands matching the number of missing arcs in F. Our formulation would not be solvable for practical de-Bruijn graphs because inserting a quadratic amount of arcs into the graph is infeasible. However, most of the breaking arcs are not needed, since in a minimum solution they can only carry flow if they directly connect a source to a sink: otherwise, the longer source-sink path they are part of could be replaced with a single breaking arc that directly connects a source to a sink to create a solution with lower costs. But even reducing the number of breaking arcs like this might not be enough if the graph contains too many sources and sinks. We therefore reduce the linear program to a min-cost matching instance, similar to Edmonds and Johnson.

5.6 Solving the min-cost integer flow formulation with min-cost matching

To solve the bidirected partial-coverage Chinese postman problem with min-cost matching, we observe that flow traverses the graph from a source to a sink only via min-cost paths, since all arcs have infinite capacity. Due to the existence of the breaking arcs with low cost $(k-1)$, we can further restrict it to use only paths of length at most $k-2$ without affecting minimality. However, since we are interested in a low number of strings in our minimum SPSS, we also allow paths of length $k-1$. We can precompute these min-cost paths efficiently in parallel (see Section 5.7 below). Then it remains to decide which combination of min-cost paths and breaking arcs yield a minimum solution.

To simplify this problem, observe that the pairing of sources and sinks that are connected via breaking arcs does not matter. Any pairing uses the same amount of breaking arcs, and therefore has the same costs. It only matters that these nodes are not connected by a lower-cost path that does not use any breaking arcs. As a

[9]It is conceptually similar to that proposed in [48], however different because the basic definitions differ, and we further allow for special arcs that do not need to be covered.

result, we can ignore breaking arcs when searching a minimum solution, and instead introduce costs for unmatched nodes. Using this, we can formulate a min-cost matching problem with penalty costs for unmatched nodes, which can be reduced to a min-cost perfect matching problem.

For the construction of our undirected matching graph M we define the set of sources $A \subseteq T$ as all nodes with negative bi-imbalance, and the set of sinks $B \subseteq T$ as all nodes with positive bi-imbalance. Then we add $|bi_v|$ (absolute value of the bi-imbalance of v) copies of each node from A, B and S to M. Further, for each min-cost path from a node $a \in A \cup S$ to a node $b \in B \cup S$ we add an edge (undirected arc) from each copy of a to each copy of b in M with costs equal to the costs of the path. We ignore self loops at nodes in S since they do not alter the imbalance, and nodes in A and B cannot have self loops. Then, to fulfil Condition (7) and to reduce the size of the matching problem, we merge all nodes and arcs with their reverse complement (the unmerged graph is built here to simplify our explanations, in our implementation we directly build the merged graph). Lastly, we assign each unmatched node the costs $(k-1)/2$, as each pair of unmatched nodes produces costs $k-1$ for using a breaking arc.

We reduce M to an instance of the min-cost perfect matching problem using the reduction described in [54]. For that we duplicate the graph, and add an edge with costs $k-1$ between each node and its copy.[10]

After this reduction, we use Blossom V [55] to compute a solution. This gives us a multiset of arcs that we complete with the breaking arcs required to balance the unmatched nodes to create a biEulerisation of the input graph. Following the approach from Edmonds and Johnson, we find a biEulerian cycle in the resulting graph which is a solution to the bidirected partial-coverage Chinese postman problem as required.

5.7 Efficient computation of many-to-many min-cost paths

Apart from solving the matching problem, finding the min-cost paths between sources and sinks is the most computationally heavy part of our algorithm.

We solve it using Dijkstra's shortest path algorithm [56] in a one-to-many variant and execute it in parallel for all sources. To be efficient, we create a queue with blocks of source nodes, and the threads process one block at a time. A good block size balances between threads competing for access to the queue, and threads receiving a very imbalanced workload. Since our min-cost paths are short (at most $k-1$ arcs), in most executions of Dijkstra's algorithm only a tiny fraction of the nodes in the graph are visited. But Dijkstra's algorithm wastefully loops over all nodes to reset their costs before each execution. To save time, we introduce an epoch value for each node together with a global expected epoch value, and instead of resetting all nodes before each execution, we increment the expected epoch value. Then, when reading the costs of a node, we reset it if the node has an outdated epoch value. Additionally, whenever the costs of a node are read or written, we set its epoch value to the expected epoch value. As another optimisation, we abort the execution

[10]By duplicating the graph, virtually each edge's costs are doubled, since each edge exists twice afterwards. However, the edges between nodes and their duplicate exist only once, so their costs require doubling.

early when Dijkstra reaches costs greater than $k-1$, since we are only interested in paths up to costs $k-1$.

Finally, in our implementation, we do not compute the actual sequences of arcs of the paths. Instead of copying the path arcs when biEulerising the graph, we insert special dummy arcs with a length equal to the length of the path. When breaking the final biEulerian cycle, if there are no breaking arcs but dummy arcs, then we break at a longest dummy arc to produce a minimum solution. If there are neither breaking nor dummy arcs, we proceed as described above. Then, when reporting the final set of strings, we define spell$(\cdot)$ to append the last ℓ characters of b when encountering a dummy biarc $[(a, b), (b^{-}1, a^{-}1)]$ (or $[(a, b)]$) of length ℓ.

5.8 Efficient computation of the greedy heuristic

The greedy heuristic biEulerises the graph by greedily adding min-cost paths between unbalanced nodes, as opposed to choosing an optimal set via min-cost matching like our main algorithm. It then continues like the main algorithm, finding a biEulerian cycle, breaking it into walks and spelling out the strings.

To be efficient, the min-cost paths are again computed in parallel, and we apply all optimisations from Section 5.7. The parallelism however poses a problem for the greedy computation: if a binode with one missing incoming biarc is reached by two min-cost paths in parallel, then if both threads add their respective biarcs, we would overshoot the bi-imbalance of that binode. To prevent that, we introduce a lock for each node, and before inserting a biarc into the graph we lock all (up to) four incident nodes. By locking the nodes in order of their ids we ensure that no deadlock can occur. Since the number of threads is many orders of magnitude lower than the number of nodes, we assume that threads almost never need to wait for each other. In addition to the parallelisation, we abort Dijkstra's algorithm early when we have enough paths to fix the imbalance for the binode. This sometimes requires to execute Dijkstra's algorithm again if a potential sink node was used by a different thread in parallel. But again, since the number of threads is many orders of magnitude lower than the number of nodes, we assume that this case almost never occurs.

5.9 Minimising string count

In the paper we studied SPSSes of minimum total length (minimum CL). In this section we note that an SPSS with a minimum number of strings (minimum SC), and with no constraint on the total length, is also computable in polynomial time.

The high-level idea, ignoring reverse complements, is as follows. Given the arc-centric de Bruijn graph G, construct the directed acyclic graph G^* of strongly connected components (SCCs) of G. In G^*, every SCC is a node, and we have as many arcs between two SCCs as there are pairs of nodes in the two SCCs with an arc between them. Clearly, all arcs in a single SCC are coverable by a single walk. Moreover, for two SCCs connected by an arc, their two covering walks can be connected via this arc into a single walk covering all arcs of both SCCs. Thus, the minimum number of walks needed to cover all arcs of G^* (i.e., minimum SC SPSS) equals the minimum number of paths needed to cover all arcs of the acyclic graph G^*. This is a classic problem solvable in polynomial time with network flows (see e.g. [57] among many).

However, such an SPSS of minimum SC very likely has a large CL, because covering an SCC with a single walk might repeat quadratically many arcs, and connecting the covering walks of two adjacent SCCs might additionally require to repeat many arcs to reach the arc between them.

5.10 Experimental evaluation

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. We downloaded the genomes of the model organisms from RefSeq [40]: Caenorhabditis elegans with accession GCF_000002985.6, Bombyx mori with accession GCF_000151625.1 and Homo sapiens with accession GCF_000001405.39. These are the same genomes as in [41], except that we get HG38 from RefSeq for citability. The short reads were downloaded from the sequence read archive [58]: Caenorhabditis elegans with accession SRR14447868.1, Bombyx mori with accession DRR064025.1 and Homo sapiens with accessions SRR2052337.1 to SRR2052425.1.

For the pangenomes we downloaded the 616 Streptococcus pneumoniae genomes from the sequence read archive, using the accession codes provided in Table 1 in [59]. We downloaded the 1102 Neisseria gonorrhoeae genomes from [60]. Up to here the pangenomes are retrieved in the same way as in [41]. We additionally used `grep` to select 3682 Escherichia coli genomes from `ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/bacteria/assembly_summary.txt`. For the short read query we used 30 read data sets from the sequence read archive with the accessions listed in Additional file 6.

We used snakemake [61] and the bioconda software repository [62] to craft our experiment pipeline. Both the build and the query were checked for correctness by checking the kmer sets built against unitigs and the query results against the same query run on unitigs. Whenever we measured runtime of queries and builds for Additional file 5 (Performance with different amounts of threads), we only let a single experiment run, even if the experiment used only one core. When running the other builds we ran multiple processes at the same time, but never using more threads than the processor has physical cores (thus avoiding any effects introduced by sharing logical cores). Due to the size of the experiments, when running a tool we copied its input to the SSD, and copied the results back to our main storage. The copying was not part of the measurements. We made sure that the server is undisturbed, except that we monitored the experiment status and progress with `htop` and `less`. We limited each run to 256GiB of RAM per process, which prevented us from running matchtigs on larger inputs. Further, ProphAsm supports only $k \leq 32$, so it was not run for k larger than that. See Section *Availability of data and materials* for availability of our implementation and experiment code.

Acknowledgements
We are very grateful to Paul Medvedev and Amatur Rahman for helpful initial discussions on this problem. We further wish to thank the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources. Finally we wish to thank the Rust community (`https://users.rust-lang.org`) for explanations about language-specific details of parallel implementations.

Funding
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, SAFEBIO), and by the Academy of Finland (grants No. 322595, 328877).

Availability of data and materials
The implementation of the matchtigs and greedy matchtigs algorithms is available on github at https://github.com/algbio/matchtigs. The name of the project is *matchtigs*. It is platform independent, and can be compiled locally or installed 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 at https://doi.org/10.5281/zenodo.5766024, and the implementation together with all code to reproduce the experiments is available at https://doi.org/10.5281/zenodo.5751234. See Section 5.10 for the availability of the non-original data used for our experiments.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
SK and AIT formulated the problem. AIT and SK designed an optimal algorithm and SK implemented and evaluated its prototype. Thereafter, SS improved the algorithm's design to its current form and provided the final implementation as well as optimisations required to make it practically relevant. SS developed the greedy heuristic and implemented it. JA, SS and AIT designed the experiments and interpreted the results. SS performed all experiments, and developed all further code published in the context of this work. SS wrote the manuscript under the supervision of the other authors. All authors reviewed and approved the final version of the manuscript.

Authors' information
Orcids: Sebastian Schmidt https://orcid.org/0000-0003-4878-2809, Shahbaz Khan https://orcid.org/0000-0001-9352-0088, Jarno Alanko https://orcid.org/0000-0002-8003-9225 and Alexandru I. Tomescu https://orcid.org/0000-0002-5747-8350

Author details
[1]Department of Computer Science, University of Helsinki, Helsinki, Finland. [2]Faculty of Computer Science, Dalhousie University, Halifax, Canada.

References

1. Zielezinski, A., Vinga, S., Almeida, J., Karlowski, W.M.: Alignment-free sequence comparison: benefits, applications, and tools. Genome Biology **18**(1), 1–17 (2017)
2. Zielezinski, A., Girgis, H.Z., Bernard, G., Leimeister, C.-A., Tang, K., Dencker, T., Lau, A.K., Röhling, S., Choi, J.J., Waterman, M.S., *et al.*: Benchmarking of alignment-free sequence comparison methods. Genome Biology **20**(1), 1–18 (2019)
3. Luhmann, N., Holley, G., Achtman, M.: Blastfrost: fast querying of 100,000 s of bacterial genomes in bifrost graphs. Genome Biology **22**(1), 1–15 (2021)
4. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de Bruijn graphs. Nature Genetics **44**(2), 226–232 (2012)
5. Nordström, K.J., Albani, M.C., James, G.V., Gutjahr, C., Hartwig, B., Turck, F., Paszkowski, U., Coupland, G., Schneeberger, K.: Mutation identification by direct comparison of whole-genome sequencing data from mutant and wild-type individuals using k-mers. Nature Biotechnology **31**(4), 325–330 (2013)
6. Bradley, P., Gordon, N.C., Walker, T.M., Dunn, L., Heys, S., Huang, B., Earle, S., Pankhurst, L.J., Anson, L., De Cesare, M., *et al.*: Rapid antibiotic-resistance predictions from genome sequence data for staphylococcus aureus and mycobacterium tuberculosis. Nature Communications **6**(1), 1–15 (2015)
7. Shajii, A., Yorukoglu, D., William Yu, Y., Berger, B.: Fast genotyping of known snps through approximate k-mer matching. Bioinformatics **32**(17), 538–544 (2016)
8. Sun, C., Medvedev, P.: Toward fast and accurate snp genotyping from whole genome sequencing data for bedside diagnostics. Bioinformatics **35**(3), 415–420 (2019)
9. Bray, N.L., Pimentel, H., Melsted, P., Pachter, L.: Near-optimal probabilistic rna-seq quantification. Nature Biotechnology **34**(5), 525–527 (2016)
10. Ames, S.K., Hysom, D.A., Gardner, S.N., Lloyd, G.S., Gokhale, M.B., Allen, J.E.: Scalable metagenomic taxonomy classification using a reference genome database. Bioinformatics **29**(18), 2253–2260 (2013)
11. Wood, D.E., Salzberg, S.L.: Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome Biology **15**(3), 1–12 (2014)
12. Břinda, K., Salikhov, K., Pignotti, S., Kucherov, G.: Prophyle: a phylogeny-based metagenomic classifier using the burrows-wheeler transform. Poster at HiTSeq 2017 (2017)
13. Corvelo, A., Clarke, W.E., Robine, N., Zody, M.C.: taxmaps: comprehensive and highly accurate taxonomic classification of short-read data in reasonable time. Genome Research **28**(5), 751–758 (2018)
14. Simon, H.Y., Siddle, K.J., Park, D.J., Sabeti, P.C.: Benchmarking metagenomics tools for taxonomic classification. Cell **178**(4), 779–794 (2019)
15. Sirén, J.: Indexing variation graphs. In: 2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 13–27 (2017). SIAM
16. Garrison, E., Sirén, J., Novak, A.M., Hickey, G., Eizenga, J.M., Dawson, E.T., Jones, W., Garg, S., Markello, C., Lin, M.F., *et al.*: Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nature Biotechnology **36**(9), 875–879 (2018)
17. Benoit, G.: Simka: fast kmer-based method for estimating the similarity between numerous metagenomic datasets. In: RCAM (2015)
18. David, S., Mentasti, M., Tewolde, R., Aslett, M., Harris, S.R., Afshar, B., Underwood, A., Fry, N.K., Parkhill, J., Harrison, T.G.: Evaluation of an optimal epidemiological typing scheme for legionella pneumophila with whole-genome sequence data using validation guidelines. Journal of clinical microbiology **54**(8), 2135–2148 (2016)

19. Chattaway, M.A., Schaefer, U., Tewolde, R., Dallman, T.J., Jenkins, C.: Identification of escherichia coli and shigella species from whole-genome sequences. Journal of clinical microbiology **55**(2), 616–623 (2017)
20. Clausen, P.T., Aarestrup, F.M., Lund, O.: Rapid and precise alignment of raw reads against redundant databases with kma. BMC bioinformatics **19**(1), 1–8 (2018)
21. Pandey, P., Almodaresi, F., Bender, M.A., Ferdman, M., Johnson, R., Patro, R.: Mantis: A fast, small, and exact large-scale sequence-search index. Cell systems **7**(2), 201–207 (2018)
22. Marchet, C., Kerbiriou, M., Limasset, A.: Indexing De Bruijn graphs with minimizers. In: Recomb-Seq 2019-9th RECOMB Satellite Workshop on Massively Parallel Sequencing, pp. 1–16 (2019)
23. Holley, G., Melsted, P.: Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. Genome Biology **21**(1), 1–20 (2020)
24. Pevzner, P.A.: l-Tuple DNA sequencing: computer analysis. Journal of Biomolecular structure and dynamics **7**(1), 63–73 (1989)
25. Idury, R.M., Waterman, M.S.: A new algorithm for DNA sequence assembly. Journal of Computational Biology **2**(2), 291–306 (1995)
26. Pevzner, P.A., Tang, H., Waterman, M.S.: An Eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences **98**(17), 9748–9753 (2001). doi:10.1073/pnas.171285098
27. Chaisson, M.J., Pevzner, P.A.: Short read fragment assembly of bacterial genomes. Genome Research **18**(2), 324–330 (2008)
28. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J., Birol, I.: Abyss: a parallel assembler for short read sequence data. Genome Research **19**(6), 1117–1123 (2009)
29. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., *et al.*: De novo assembly of human genomes with massively parallel short read sequencing. Genome Research **20**(2), 265–272 (2010)
30. Bankevich, A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., Pyshkin, A.V., Sirotkin, A.V., Vyahhi, N., Tesler, G., Alekseyev, M.A., Pevzner, P.A.: SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. Journal of Computational Biology **19**(5), 455 (2012). doi:10.1089/cmb.2012.0021
31. Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., He, G., Chen, Y., Pan, Q., Liu, Y., *et al.*: Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. Gigascience **1**(1), 2047–217 (2012)
32. Chikhi, R., Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms for Molecular Biology **8**(1), 22 (2013)
33. Chikhi, R., Limasset, A., Medvedev, P.: Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics **32**(12), 201–208 (2016)
34. Jackman, S.D., Vandervalk, B.P., Mohamadi, H., Chu, J., Yeo, S., Hammond, S.A., Jahesh, G., Khan, H., Coombe, L., Warren, R.L., Birol, I.: ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter. Genome Research **27**(5), 768–777 (2017). doi:10.1101/gr.214346.116
35. Ruan, J., Li, H.: Fast and accurate long-read assembly with wtdbg2. Nature Methods **17**(2), 155–158 (2020)
36. Tomescu, A.I., Medvedev, P.: Safe and Complete Contig Assembly Through Omnitigs. Journal of Computational Biology **24**(6), 590–602 (2017). doi:10.1089/cmb.2016.0141
37. Acosta, N.O., Mäkinen, V., Tomescu, A.I.: A safe and complete algorithm for metagenomic assembly. Algorithms for Molecular Biology **13**(1), 1–12 (2018)
38. Cairo, M., Khan, S., Rizzi, R., Schmidt, S., Tomescu, A.I., Zirondelli, E.C.: The hydrostructure: a universal framework for safe and complete algorithms for genome assembly. arXiv preprint arXiv:2011.12635 (2020)
39. Kececioglu, J.D., Myers, E.W.: Combinatorial algorithms for DNA sequence assembly. Algorithmica **13**(1), 7–51 (1995)
40. O'Leary, N.A., Wright, M.W., Brister, J.R., Ciufo, S., Haddad, D., McVeigh, R., Rajput, B., Robbertse, B., Smith-White, B., Ako-Adjei, D., *et al.*: Reference sequence (refseq) database at ncbi: current status, taxonomic expansion, and functional annotation. Nucleic Acids Research **44**(D1), 733–745 (2016)
41. Břinda, K., Baym, M., Kucherov, G.: Simplitigs as an efficient and scalable representation of de Bruijn graphs. Genome Biology **22**(1), 1–24 (2021)
42. Rahman, A., Medevedev, P.: Representation of k-mer sets using spectrum-preserving string sets. Journal of Computational Biology **28**(4), 381–394 (2021)
43. Rahman, A., Chikhi, R., Medvedev, P.: Disk compression of k-mer sets. Algorithms for Molecular Biology **16**(1), 1–14 (2021)
44. Marchet, C., Iqbal, Z., Gautheret, D., Salson, M., Chikhi, R.: REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. Bioinformatics **36**(Supplement_1), 177–185 (2020). doi:10.1093/bioinformatics/btaa487. https://academic.oup.com/bioinformatics/article-pdf/36/Supplement_1/i177/33860751/btaa487.pdf
45. Kitaya, K., Shibuya, T.: Compression of Multiple k-Mer Sets by Iterative SPSS Decomposition. In: Carbone, A., El-Kebir, M. (eds.) 21st International Workshop on Algorithms in Bioinformatics (WABI 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 201, pp. 12–11217. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). doi:10.4230/LIPIcs.WABI.2021.12. https://drops.dagstuhl.de/opus/volltexte/2021/14365
46. Kwan, M.-k.: Graphic programming using odd or even points. Chinese Mathematics **1**, 273–277 (1962)
47. Edmonds, J., Johnson, E.L.: Matching, euler tours and the chinese postman. Mathematical programming **5**(1), 88–124 (1973)
48. Medvedev, P., Georgiou, K., Myers, G., Brudno, M.: Computability of models for sequence assembly. In: Giancarlo, R., Hannenhalli, S. (eds.) Algorithms in Bioinformatics, 7th International Workshop, WABI 2007, Philadelphia, PA, USA, September 8-9, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4645, pp. 289–301. Springer, Berlin, Heidelberg (2007). doi:10.1007/978-3-540-74126-8_27
49. Li, H.: Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. arXiv preprint arXiv:1303.3997 (2013)

50. Lenstra, J.K., Kan, A.R.: Complexity of vehicle routing and scheduling problems. Networks **11**(2), 221–227 (1981)
51. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM (JACM) **19**(2), 248–264 (1972)
52. Christofides, N., Campos, V., Corberán, A., Mota, E.: In: Gallo, G., Sandi, C. (eds.) An algorithm for the Rural Postman problem on a directed graph, pp. 155–166. Springer, Berlin, Heidelberg (1986). doi:10.1007/BFb0121091. https://doi.org/10.1007/BFb0121091
53. Even, S.: Graph Algorithms. Computer Science Press, Rockville, MD (1979)
54. Schäfer, G.: Weighted matchings in general graphs. Master's thesis, Saarland University (2000)
55. Kolmogorov, V.: Blossom V: a new implementation of a minimum cost perfect matching algorithm. Mathematical Programming Computation **1**(1), 43–67 (2009)
56. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
57. Cáceres, M., Cairo, M., Mumey, B., Rizzi, R., Tomescu, A.I.: Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. arXiv preprint arXiv:2107.05717 (2021). To appear in the Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)
58. Leinonen, R., Sugawara, H., Shumway, M.: The sequence read archive. Nucleic Acids Research **39**(suppl_1), 19–21 (2010)
59. Croucher, N.J., Finkelstein, J.A., Pelton, S.I., Parkhill, J., Bentley, S.D., Lipsitch, M., Hanage, W.P.: Population genomic datasets describing the post-vaccine evolutionary epidemiology of streptococcus pneumoniae. Scientific data **2**(1), 1–9 (2015)
60. Grad, Y.H., Harris, S.R., Kirkcaldy, R.D., Green, A.G., Marks, D.S., Bentley, S.D., Trees, D., Lipsitch, M.: Genomic epidemiology of gonococcal resistance to extended-spectrum cephalosporins, macrolides, and fluoroquinolones in the united states, 2000–2013. The Journal of infectious diseases **214**(10), 1579–1587 (2016)
61. Köster, J., Rahmann, S.: Snakemake—a scalable bioinformatics workflow engine. Bioinformatics **28**(19), 2520–2522 (2012)
62. Grüning, B., Dale, R., Sjödin, A., Chapman, B.A., Rowe, J., Tomkins-Tinch, C.H., Valieris, R., Köster, J.: Bioconda: sustainable and comprehensive software distribution for the life sciences. Nature Methods **15**(7), 475–476 (2018)

Additional Files

Additional file 1 — Quality of compressing model organisms (pdf)
Additional CL and SC data from the experiments from Table 1, with varying k and min. abundance. Also, the average unitig length and total unitig count is plotted.

Additional file 2 — Performance of compressing model organisms (pdf)
Additional run time and memory data from the experiments from Table 1, with varying k and min. abundance.

Additional file 3 — Quality of compressing pangenomes (pdf)
Additional CL and SC data from the experiments from Table 2, with varying k and min. abundance. Also, the average unitig length and total unitig count is plotted.

Additional file 4 — Performance of compressing pangenomes (pdf)
Additional run time and memory data from the experiments from Table 2, with varying k and min. abundance.

Additional file 5 — Performance with different amounts of threads (pdf)
A run time and memory comparison of compressing the E. coli pangenome and the H. sapiens reference genome with different amounts of threads and different compression methods.

Additional file 6 — Accessions of the reads used for the query experiment (txt)
A list of sequence read archive accession numbers for the 30 sets of E.coli short reads used in the experiment in Section 2.4