


# Compression of Multiple $k$ -Mer Sets by Iterative SPSS Decomposition

Kazushi Kitaya ✉

Tokyo, Japan

Tetsuo Shibuya ✉ 🏠 

Human Genome Center, Institute of Medical Science, The University of Tokyo, Japan

---

## Abstract

---

A set of  $k$ -mers is used in many bioinformatics tasks, and much work has been done on methods to efficiently represent or compress a single set of  $k$ -mers. However, methods for compressing multiple  $k$ -mer sets have been less studied in spite of their obvious benefits for researchers and genome-related database maintainers. This paper proposes an algorithm to compress multiple  $k$ -mer sets, which works by iteratively splitting SPSS (spectrum-preserving string sets). In experiments with 3292  $k$ -mer sets constructed from *E. coli* whole-genome sequencing data and 2555  $k$ -mer sets constructed from human RNA-Seq data, the proposed algorithm could reduce the compressed file sizes by 34.7% and 13.2% respectively compared to one of the state-of-the-art colored de Bruijn graph representations. Also, our method used less memory than the colored de Bruijn graph method. This paper also introduces various methods to make the compression algorithm efficient in terms of time and memory, one of which is a parallelizable small-weight SPSS construction algorithm.

**2012 ACM Subject Classification** Applied computing → Molecular sequence analysis

**Keywords and phrases** sequencing data,  $k$ -mer, de Bruijn graph, compression, colored de Bruijn graph

**Digital Object Identifier** 10.4230/LIPIcs.WABI.2021.12

**Supplementary Material** *Software (Source Code)*: <https://github.com/kkty/kmer-sets-compression>; archived at `swh:1:dir:ee89fa1752240bda5cf199275118ba3884b7eac2`

**Funding** This work was supported by JSPS KAKENHI Grant 17H01693, 20K21827, and JST CREST Grant JPMJCR1402JST. The super-computing resource was provided by Human Genome Center, the Institute of Medical Science, the University of Tokyo.

## 1 Introduction

With the advent of next-generation sequencers, the cost of obtaining genomic information has decreased dramatically. It enabled a wide range of research, along with the development of public repositories such as Sequence Read Archive [9]. At the same time, there has been a growing need for efficient methods of processing and storing the increasing amount of data.

For fast processing with less memory, many bioinformatics tasks take as input a set of  $k$ -mers, or the de Bruijn graph [7] represented by a set of  $k$ -mers. For example, the DNA fragment assembly can be done efficiently by finding an Eulerian path in de Bruijn graphs [15]. Also, a set of  $k$ -mers can be thought of as a sketch of the original sequence data, so it can be used to index sequence datasets [17].

Much work has been done on techniques to efficiently represent a single  $k$ -mer set or a single de Bruijn graph. For example, a succinct de Bruijn graph [4] can be used to efficiently represent a de Bruijn graph, keeping essential queries fast, and we can see its usage in MEGAHIT assembler [11] and other works. When we do not need to support fast query



© Kazushi Kitaya and Tetsuo Shibuya;

licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Algorithms in Bioinformatics (WABI 2021).

Editors: Alessandra Carbone and Mohammed El-Kebir; Article No. 12; pp. 12:1–12:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

responses, or just want to efficiently store a set of  $k$ -mers to disk, there are several other approaches that can be simple or more efficient in terms of disk usage. For example, unitigs, a set of strings that can be obtained by finding non-branching paths in de Bruijn graphs, can be used to represent a set of  $k$ -mers efficiently. Finding non-branching paths in de Bruijn graphs (and merging those paths) are formulated as “compacting de Bruijn graphs” and researches have been done on methods to efficiently compact de Bruijn graphs as in [6]. The concept of unitigs was generalized in [16] which introduced spectrum-preserving string sets (SPSS). Also, the paper showed a heuristic algorithm that can be used to get SPSS from a set of  $k$ -mers, which achieves far better compression than unitigs.

In addition to the attempts to efficiently represent a single set of  $k$ -mers or a single de Bruijn graph, researchers have looked for efficient representations of multiple sets of  $k$ -mers. [8] suggested that “colored de Bruijn graphs”, which can represent multiple  $k$ -mer sets obtained from multiple datasets and can efficiently respond to some types of queries, were suited for tasks like genotyping. Further studies have been conducted to find more efficient ways to represent colored de Bruijn graphs, one of which is Rainbowfish [2], which combined the succinct de Bruijn graph [4] with succinct color information representation. Mantis [13], which was primarily designed for indexing datasets, can also be thought of as a colored de Bruijn graph, and its color representation was greatly improved recently by the author of the Rainbowfish paper [1].

As we have seen, there has been research on methods to store a single  $k$ -mer set, some of which support efficient query responses (e.g., succinct de Bruijn graphs), and some of which focus on simplicity (e.g., unitigs) or better compression (e.g., SPSS). Also, there has been research on methods to efficiently store multiple  $k$ -mer sets which support fast query responses (e.g., colored de Bruijn graphs).

However, methods to represent multiple  $k$ -mer sets with a primary focus on compression capability are not well studied, in spite of their obvious benefits for researchers who want to work with multiple datasets and for maintainers of genome-related databases who want to store a number of datasets.

This paper tackles this problem by introducing a new algorithm to compress multiple  $k$ -mer sets, which works by iteratively splitting SPSS.

With the proposed algorithm, it was possible to compress 3292  $k$ -mer sets constructed from *E. coli* whole-genome sequencing data and 2555  $k$ -mer sets constructed from human RNA-Seq data to 2.42% and 6.04%, respectively. They were better by 60.0% and 25.0% than compressing them individually, which led to 6.03% and 8.06%. They were also better by 34.7% and 13.2% than one of the recent colored de Bruijn graph representations, Mantis [13, 1], which led to 3.70% and 6.97%. Also, our method used 26.1GB and 24.8GB of memory, better by 22.1% and 9.5% than Mantis, which used 33.5GB and 27.4GB.

This paper also introduces various methods to make the compression algorithm efficient in terms of speed and memory usage, one of which is a small-weight SPSS construction algorithm that performs well with multiple threads. On 16-core machines, the proposed algorithm was faster by 83.2% than the existing non-parallelizable algorithm, UST [16].

## **2** Preliminaries

This chapter covers the concepts of  $k$ -mers, canonical  $k$ -mers, de Bruijn graphs, compacted de Bruijn graphs, and spectrum-preserving string sets (SPSS), which will be utilized by the proposed algorithms.

## 2.1 $k$ -mer and canonical $k$ -mer

A string  $s$  is called a  $k$ -mer when  $|s| = k$ . Here, the alphabet  $\Sigma$  is equal to  $\{A, C, G, T\}$ .

Let  $s$  be a  $k$ -mer. For each  $c \in \Sigma$ ,  $next(s, c)$  is the  $k$ -mer that can be constructed by concatenating the suffix of length  $k - 1$  in  $s$  and  $c$ . Similarly, for each  $c \in \Sigma$ ,  $prev(s, c)$  is the  $k$ -mer that can be constructed by concatenating  $c$  and the prefix of length  $k - 1$  in  $s$ .

Let  $s$  be a  $k$ -mer. The reverse complement of  $s$  is the  $k$ -mer that can be constructed by reversing  $s$  and replacing characters A, C, G, and T with T, G, C, and A, respectively.

Let  $s$  be a  $k$ -mer and  $s'$  be the reverse complement of  $s$ . The canonical form of  $s$  (and the canonical form of  $s'$ ) is  $\min(s, s')$ . Here, we use the dictionary order of strings.

Let  $s$  be a  $k$ -mer.  $s$  is a canonical  $k$ -mer if the canonical form of  $s$  is equal to  $s$ .

In addition to  $k$ -mers, we defined canonical  $k$ -mers and other related terms to handle data from double-stranded structures like DNA. Instruments like next-generation sequencers produce a set of strings (reads). We often pre-process the data by listing all the  $k$ -mers that appears as a substring in the reads and making a set of canonical  $k$ -mers from them.

## 2.2 de Bruijn graph of canonical $k$ -mers

Let  $S = \{s_1, s_2, \dots, s_n\}$  be a set of canonical  $k$ -mers. The de Bruijn graph  $G$  represented by  $S$  is an undirected graph with the following properties.

- It has  $n$  vertices. Let them be  $\{v_1, v_2, \dots, v_n\}$ .
- Each vertex has two sides: “left” side and “right” side.
- Each vertex is labeled by a  $k$ -mer. Let us suppose that for each  $i$ ,  $label(v_i) = s_i$ .
- There is an edge from the right side of  $v_i$  to the left side of  $v_j$  if  $c \in \Sigma$  exists such that  $next(label(v_i), c)$  is equal to  $label(v_j)$ .
- There is an edge from the right side of  $v_i$  to the right side of  $v_j$  if  $c \in \Sigma$  exists such that  $next(label(v_i), c)$  is equal to the reverse complement of  $label(v_j)$ .
- There is an edge from the left side of  $v_i$  to the right side of  $v_j$  if  $c \in \Sigma$  exists such that  $prev(label(v_i), c)$  is equal to  $label(v_j)$ .
- There is an edge from the left side of  $v_i$  to the left side of  $v_j$  if  $c \in \Sigma$  exists such that  $prev(label(v_i), c)$  is equal to the reverse complement of  $label(v_j)$ .

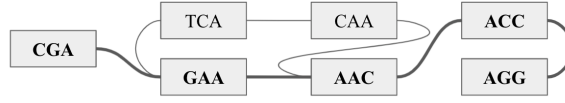
A sequence of vertices  $p = \{v_1, v_2, \dots, v_m\}$  in a de Bruijn graph  $G$  is a path in  $G$  if the following conditions are satisfied.

- For each  $1 \leq i < j \leq m$ ,  $v_i \neq v_j$ .
- For each  $1 \leq i \leq m - 1$ ,  $v_i$  and  $v_{i+1}$  are connected by an edge.
- For each  $2 \leq i \leq m - 1$ , if  $v_{i-1}$  is connected to the left side of  $v_i$ ,  $v_{i+1}$  is connected to the right side of  $v_i$ .
- For each  $2 \leq i \leq m - 1$ , if  $v_{i-1}$  is connected to the right side of  $v_i$ ,  $v_{i+1}$  is connected to the left side of  $v_i$ .

The string represented by  $p$  is the string that will be returned by the following procedure.

1. If  $m = 1$ , return  $label(v_1)$ .
2. If  $v_2$  is connected to the right side of  $v_1$ , initialize  $s$  with  $label(v_1)$ . Otherwise, initialize  $s$  with the reverse complement of  $label(v_1)$ .
3. For  $i = 2, 3, \dots, m$ , perform the following operations.
  - a. If  $v_{i-1}$  is connected to the left side of  $v_i$ , append the last character of  $label(v_i)$  to the end of  $s$ . Otherwise, append the last character of the reverse complement of  $label(v_i)$  to  $s$ .
4. Return  $s$ .

Figure 1 shows an example of de Bruijn graphs along with a path in it.



■ **Figure 1** The de Bruijn graph represented by a set of canonical  $k$ -mers  $\{CGA, TCA, CAA, ACC, GAA, AAC, AGG\}$  and a path in it that represents “CGAACCT”.

### 2.3 Unitigs and compacted de Bruijn graph

A compacted de Bruijn graph is a graph that can be built by “merging” non-branching paths of a de Bruijn graph. In this section, the concept of compacted de Bruijn graphs is introduced along with the concept of unitigs.

Let  $G = (V, E)$  be a de Bruijn graph. The unitigs of  $G$  are a set of strings that can be constructed with the following procedure.

1. Find a min-size path cover  $W$  in  $G$  such that each edge in any of its paths does not share a vertex’s side with other edges in  $E$ .
2. For each path  $p$  in  $W$ , make the string represented by  $p$ .

Note that the path cover is uniquely determined when we do not care about orientations.

Also, note that unitigs can be defined by edge contraction as well. But we define them as above because we are focusing on paths in de Bruijn graphs throughout this paper.

Let  $pre(s_i)$  be the  $k$ -mer that corresponds to the prefix of  $s_i$  whose length is  $k$ , and  $suf(s_i)$  be the  $k$ -mer that corresponds to the suffix of  $s_i$  whose length is  $k$ .

Let  $S = \{s_1, s_2, \dots, s_n\}$  be the unitigs of a de Bruijn graph  $G$ . The compacted de Bruijn graph  $G'$  corresponding to  $G$  is an undirected graph with the following properties. It is similar to the definition of a de Bruijn graph.

- It has  $n$  vertices. Let them be  $\{v_1, v_2, \dots, v_n\}$ .
- Each vertex has two sides: “left” side and “right” side.
- Each vertex is labeled by a string. Let us suppose that for each  $i$ ,  $label(v_i) = s_i$ .
- There is an edge from the right side of  $v_i$  to the left side of  $v_j$  if  $c \in \Sigma$  exists such that  $next(suf(label(v_i)), c)$  is equal to  $pre(label(v_j))$ .
- There is an edge from the right side of  $v_i$  to the right side of  $v_j$  if  $c \in \Sigma$  exists such that  $next(suf(label(v_i)), c)$  is equal to the reverse complement of  $suf(label(v_j))$ .
- There is an edge from the left side of  $v_i$  to the right side of  $v_j$  if  $c \in \Sigma$  exists such that  $prev(pre(label(v_i)), c)$  is equal to  $suf(label(v_j))$ .
- There is an edge from the left side of  $v_i$  to the left side of  $v_j$  if  $c \in \Sigma$  exists such that  $prev(pre(label(v_i)), c)$  is equal to the reverse complement of  $pre(label(v_j))$ .

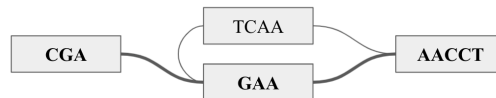
A path in a compacted de Bruijn graph can be defined the same way as for a de Bruijn graph.

Let  $p = \{v_1, v_2, \dots, v_m\}$  be a path in a compacted de Bruijn graph. The string represented by  $p$  is the string that will be returned by the following procedure.

1. If  $m = 1$ , return  $label(v_1)$ .
2. If  $v_2$  is connected to the right side of  $v_1$ , initialize  $s$  with  $label(v_1)$ . Otherwise, initialize  $s$  with the reverse complement of  $label(v_1)$ . Here, the reverse complement of a string is defined the same way as for  $k$ -mers.

3. For  $i = 2, 3, \dots, m$ , perform the following operation.
  - a. If  $v_{i-1}$  is connected to the left side of  $v_i$ , append the suffix of  $label(v_i)$  whose length is  $|label(v_i)| - (k - 1)$  to the end of  $s$ . Otherwise, append the suffix of the reverse complement of  $label(v_i)$  whose length is  $|label(v_i)| - (k - 1)$  to the end of  $s$ .
4. Return  $s$ .

Figure 2 shows an example of compacted de Bruijn graphs along with a path in it.



■ **Figure 2** The compacted de Bruijn graph built from the de Bruijn graph illustrated in Figure 1 along with a path in it that represents the string “CGAACCT”.

## 2.4 Spectrum-preserving string sets (SPSS)

In this section, the concept of SPSS (Spectrum-Preserving String Sets) [16] is described.

Let  $S$  be a set of canonical  $k$ -mers. A set of strings  $X$  is SPSS of  $S$  if the following conditions are satisfied.

- For each  $x \in X$ ,  $|x| \geq k$ .
- For each  $s \in S$ , one of the following is satisfied.
  - There exists  $x \in X$  such that  $s$  appears as a substring of  $x$ .
  - There exists  $x \in X$  such that the reverse complement of  $s$  appears as a substring of  $x$ .
- $\sum_{x \in X} |x| = |S| + (k - 1)|X|$

Note that it is possible to reconstruct  $S$  from  $X$  by finding all the  $k$ -mers in  $X$  and getting the canonical form of each.

Let  $\sum_{x \in X} |x|$  be the weight of SPSS  $X$ . As it is possible to reconstruct  $S$  from  $X$ , making small-weight SPSS corresponds to compressing a  $k$ -mer set.

## 3 Related work

[16] introduced SPSS, a representation of a single  $k$ -mer set. In addition, the paper proposed an algorithm to build efficient SPSS from a compacted de Bruijn graph. The algorithm, UST, works by heuristically finding a small-size path cover in a compacted de Bruijn graph and listing up the strings represented by each path. Specifically, it repeats the following procedure. First, a vertex that is not visited is arbitrarily picked. Second, a path from that vertex is constructed using vertices that are not visited. During this step, if multiple choices (of edges) are available, one of them is arbitrarily picked. Also, the path in construction will be merged to a previously-constructed path if possible.

Simplitigs [5] are a concept that is mostly equivalent to SPSS. The paper also proposes a heuristic algorithm like UST for constructing simplitigs from a de Bruijn graph.

Both SPSS and simplitigs are concepts for representing a single  $k$ -mer set. Although [5] uses pan-genomes in some of its experiments, the combination of simplitigs (or SPSS) and a technique to exploit similarities in multiple  $k$ -mer sets for better compression has been left as a future work.

Mantis [13] is one of the recent colored de Bruijn graph representations. It represents multiple  $k$ -mer sets with a CQF-based color table that maps  $k$ -mers to color IDs and a RRR-based color class table that maps color IDs to its existences in each  $k$ -mer set. Also, color IDs are picked so that a color that appears frequently gets a smaller ID.

Our approach is to use SPSS to compress multiple  $k$ -mer sets using the technique called “Iterative SPSS Decomposition”. This is a novel technique to use the power of SPSS along with a method to utilize similarities in multiple  $k$ -mer sets. This paper includes promising experimental results of the approach with a comparison to Mantis, an existing technique to efficiently represent multiple  $k$ -mer sets.

## 4 Methods

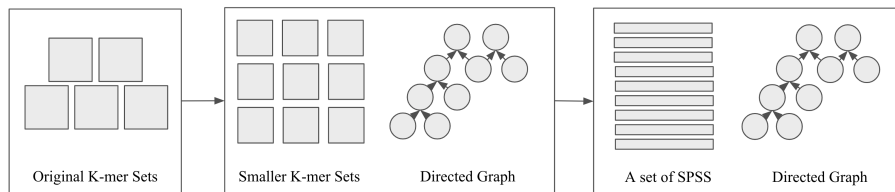
In this chapter, a compression algorithm for multiple  $k$ -mer sets is described along with the corresponding decompression algorithm. The overall algorithm is first described, followed by implementation techniques to perform the compression efficiently in terms of time and memory. Also, a parallel algorithm for SPSS construction, which can be used by itself and also for the speed-up of the compression algorithm, is introduced.

### 4.1 Compression of multiple $k$ -mer sets by iterative decomposition

Given  $N$   $k$ -mer sets  $S_1, S_2, \dots, S_N$ , the following procedure compresses them and saves the result to disk, split to multiple files.

1. Initialize a variable  $n$  with  $N$ .
2. Create a directed graph  $G$  with  $n$  vertices  $v_1, v_2, \dots, v_n$ .
3. Repeat the following operations.
  - a. Find  $1 \leq i < j \leq n$  such that  $|S_i \cap S_j|$  is maximized.
  - b. Create a set  $S_{n+1} = S_i \cap S_j$ .
  - c. Update  $S_i$  and  $S_j$  as follows:  $S_i \leftarrow S_i \setminus S_{n+1}, S_j \leftarrow S_j \setminus S_{n+1}$ .
  - d. Add a vertex  $v_{n+1}$  to  $G$ .
  - e. Add two edges in  $G$ : one from  $v_i$  to  $v_{n+1}$  and one from  $v_j$  to  $v_{n+1}$ .
  - f. Update  $n$  as follows:  $n \leftarrow n + 1$ .
4. For each  $i = 1, 2, \dots, n$ , construct SPSS from  $S_i$  and save it to disk.
5. Save  $G$  to disk.

Figure 3 illustrates the idea of the algorithm.



■ **Figure 3** Illustration of the compression algorithm.

In each iteration (step 3),  $\sum_{1 \leq x \leq n} |S_x|$ , which corresponds to the number of  $k$ -mers that have to be saved to disk, decreases by  $\max_{1 \leq i < j \leq n} |S_i \cap S_j|$ . If they were to be saved to disk naively (e.g., without using SPSS), it is apparent that the required disk usage would decrease as well. However, the construction of efficient (small-weight) SPSS gets harder when the size

of a  $k$ -mer set is small. The main question, which will be answered in the later chapter by showing that the compression actually works with real data, is whether the decrease in the number of total  $k$ -mers can trump the growing difficulty of making efficient SPSS.

In our implementation, when saving SPSS to disk (step 4), bzip2 was used to further compress the strings, the alphabet of which is  $\{A, C, G, T\}$ . Speaking of  $G$ , its adjacency list representation was saved to disk.

### Time complexity

Suppose that we can check the equality of two  $k$ -mers in constant time and that we can calculate a hash value from a  $k$ -mer in constant time. By representing a  $k$ -mer set with a hash table, we can perform the iteration of all the  $k$ -mers in linear time and the existence check of a  $k$ -mer in constant time. This is a reasonable assumption because  $k$  is usually small.

Let  $C$  be  $\max_{1 \leq i \leq N} |S_i|$  where  $S_1, S_2, \dots, S_N$  are the input data. In each iteration, the calculation of  $|S_x \cap S_y|$  ( $1 \leq x < y \leq n$ ) can be done in  $O(C)$  time. The selection of  $i$  and  $j$  to maximize  $|S_i \cap S_j|$  (step 3a), if implemented naively, will take  $O(C(N+M)^2)$  time where  $M$  is the number of iterations in step 3. Then, the time complexity for all the iterations will be  $O(CM(N+M)^2)$ .

This can be improved by using the binary heap [18]. Before the first iteration,  $|S_i \cap S_j|$  is calculated for each  $1 \leq i < j \leq N$ , and the heap is made out of the values. These can be done in  $O(CN^2)$  time and  $O(N^2)$  time, respectively. In each iteration, the minimum element in the heap is looked up in  $O(1)$  time, and  $O(N+M)$  elements are updated. The calculation of the updated values will take  $O(C(N+M))$  time, and the update of the heap will take  $O((N+M)\log(N+M))$  time. The resulting time complexity for all the iterations will be  $O(CN^2 + M(C(N+M) + (N+M)\log(N+M)))$ . Further speed-up is possible with multiple threads. Let  $T$  be the number of threads available. The total time complexity would then be  $O(\frac{CN^2}{T} + N^2 + M(\frac{C(N+M)}{T} + (N+M)\log(N+M)))$ . Note that multiple threads do not speed up the construction of and the updates of the heap.

Another possible approach is to use a hash table to keep track of  $|S_i \cap S_j|$  for each  $1 \leq i < j \leq N$ . This leads to  $O(\frac{CN^2 + M(C(N+M) + (N+M)^2)}{T})$  time because it requires  $O(\frac{(N+M)^2}{T})$  time for finding  $i$  and  $j$  to maximize  $|S_i \cap S_j|$  in each iteration instead of the construction of the heap ( $O(N^2)$  time) and the update of the heap ( $O((N+M)\log(N+M))$  time in each iteration). Because of its simplicity and the fact that  $C$  is often much larger than others, this approach was used in our implementation.

### Selection of iteration size

The number of iterations  $M$  is a parameter in the algorithm. The more the iterations, the smaller each  $k$ -mer set ( $S_1, S_2, \dots, S_n$ ) becomes, but it does not necessarily mean that it leads to smaller resulting file size, or better compression. This is because it gets harder to make efficient (small-weight) SPSS when the input  $k$ -mer set is small.

We chose to dynamically select  $M$  by monitoring the total weight of SPSS during the iterations of step 3. The value, which can be calculated by constructing SPSS to represent each  $k$ -mer set and summing up the lengths of all the elements (strings), can be thought of as an approximate of the final file size. In our implementation, the value was monitored in a regular interval, and when its decrease was below a threshold, the loop was stopped. Specifically, the interval was set to  $\lfloor N/8 \rfloor + 1$ , and the threshold was set to 1.25 %.



## 4.2 Decompression

Given the compressed data saved to disk and  $i$  ( $1 \leq i \leq N$ ), it is possible to reconstruct the original value of  $S_i$  with the following procedure.

1. Load  $G$  from disk.
2. Find vertices  $v_{x_1}, v_{x_2}, \dots, v_{x_m}$  in  $G$  that are reachable from  $v_i$  using breadth-first search.
3. Obtain  $S_{x_1}, S_{x_2}, \dots, S_{x_m}$  by loading files from disk and getting  $k$ -mer sets from SPSS.
4. Return  $S_{x_1} \cup S_{x_2} \cup \dots \cup S_{x_m}$ .

As can be seen, by separating compressed data into multiple files, it was made possible to efficiently reconstruct the original  $k$ -mer sets. That is, we do not necessarily have to load all of  $S_1, S_2, \dots, S_n$  to memory when reconstructing one  $k$ -mer set that is requested.

The proof of correctness of this algorithm is shown in the appendix.

## 4.3 Implementation

The compression algorithm, if implemented naively, requires a lot of time and memory, even with the methods described in the “Time complexity” section. This section describes techniques that can be further employed to make it fast and memory efficient.

The compression algorithm, combined with these techniques, is named “Iterative SPSS Decomposition”.

### Representation of $k$ -mer sets

$S_1, S_2, \dots, S_n$  are primarily represented by SPSS in favor of its low memory usage. Specifically, the SPSS represented by  $\{s_1, s_2, \dots, s_m\}$  is saved to a bit vector whose length is  $2(|s_1| + |s_2| + \dots + |s_m|)$ , along with an array of non-negative integers  $\{|s_1| - k, |s_2| - k, \dots, |s_m| - k\}$ . It can be more memory efficient than having  $m$  bit vectors, as having multiple bit vectors entails keeping multiple pointers, whose sizes cannot be ignored. Furthermore, as the array of integers often contains small numbers, the variable-length integer encoding can be used to reduce its memory usage. In our implementation, StreamVByte [10] was used because it supports fast compression and decompression with SIMD instructions.

### Update of $k$ -mer sets in each iteration

In each iteration of the compression algorithm (step 3), two  $k$ -mer sets are updated, and one  $k$ -mer set is created. Here, a method to efficiently perform the operations is described.

First, it uses a bucketed structure to represent a  $k$ -mer set internally. When we have a  $k$ -mer, it can be represented with  $2k$  bits (e.g., by representing A, C, G, and T with 00, 01, 10, and 11, respectively). In the bucketed structure,  $k$ -mers are divided into  $2^h$  buckets.  $h$  higher bits out of the  $2k$  bits are used to select buckets, and  $2k - h$  lower bits are saved to each bucket. Figure 4 illustrates the bucketed structure with example data.



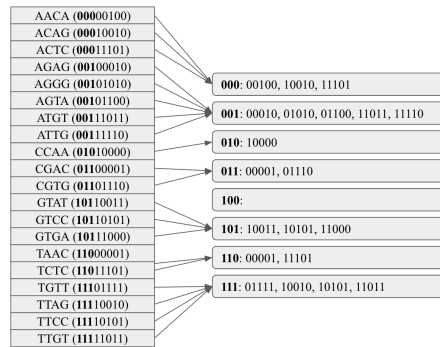


Figure 4 Illustration of the bucketed structure with example data ( $k = 4, h = 3, 20$   $k$ -mers).

With the bucketed structure, it is possible to calculate the intersection or the differences between two  $k$ -mer sets efficiently with multiple threads by dividing and assigning buckets to multiple threads. It is also possible to efficiently construct the bucketed structure from a  $k$ -mer set represented by SPSS with multiple threads, by dividing the strings into chunks, assigning chunks to multiple threads, making a bucketed  $k$ -mer set for each chunk, and merging them using  $2^h$  mutex locks.

When two  $k$ -mer sets  $S_i, S_j$  represented by SPSS are given, SPSS to represent  $S_i \cap S_j, S_i \setminus S_j,$  and  $S_j \setminus S_i$  can be constructed as follows. First, the bucketed structures are created for  $S_i$  and  $S_j$ . Second, the calculations of the intersection and the differences are performed on them. Finally, the SPSS for each of the three obtained  $k$ -mer sets are constructed. This procedure is illustrated in Figure 5. All of the operations can be done efficiently with multiple threads by utilizing the bucketed structure and by using the SPSS construction algorithm that will be introduced later.

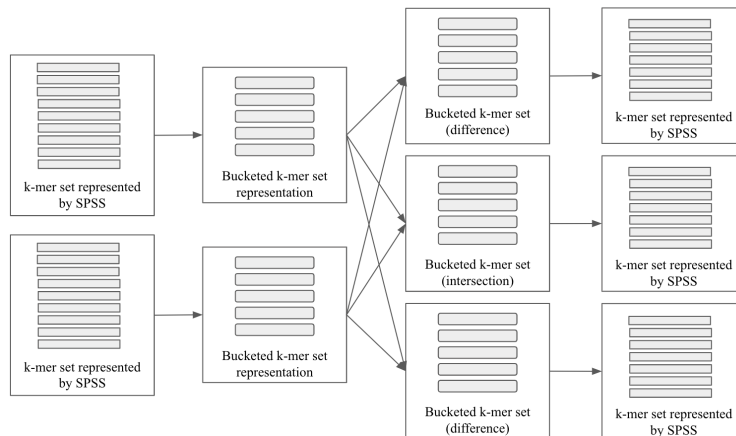


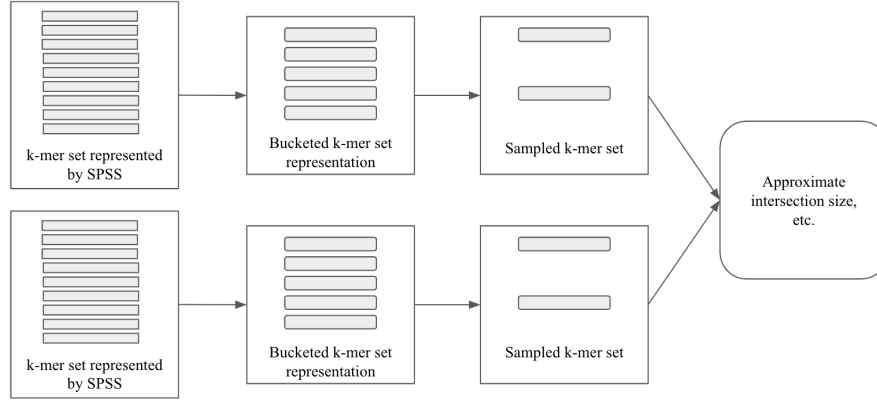
Figure 5 Illustration of the update of  $k$ -mer sets in each iteration of the compression algorithm (step 3). All the calculations can be done efficiently with multiple threads, using the feature of the bucketed structure and the SPSS construction algorithm that will be introduced later.

### Calculation of intersection size

Calculation of the intersection size between two  $k$ -mer sets is repeated during the compression. Here, a method to perform the operation with small memory usage is discussed.

## 12:10 Compression of Multiple $k$ -Mer Sets

We take the approach of approximating intersection sizes by “sampling”  $k$ -mers from each  $k$ -mer set. The sampling is performed by constructing the bucketed structure from a  $k$ -mer set (represented by SPSS) and randomly deleting buckets. Here, the same set of buckets gets deleted for each  $k$ -mer set. If  $x\%$  of the buckets are left after the sampling, we can calculate an estimate of the intersection size between the original two  $k$ -mer sets by calculating the intersection size between the sampled  $k$ -mer sets and by multiplying the value by  $\frac{100}{x}$ . This idea of using sampled  $k$ -mer sets to calculate intersection sizes is illustrated in Figure 6.



■ **Figure 6** Illustration of the calculation of an approximate intersection size between 2  $k$ -mer sets.

In our implementation, as constructing the bucketed structure and performing the sampling take some time, we kept the sampled  $k$ -mer sets for each  $k$ -mer set on memory during the iterations, and they got updated when the underlying  $k$ -mer sets were updated or when a new  $k$ -mer set was created. Speaking of the sampling size,  $x = 2$  was used.

### 4.4 Fast construction of small-weight SPSS

The small-weight SPSS construction algorithm presented in [16], UST, does not support parallel processing. Here, we introduce a parallel small-weight SPSS construction algorithm.

Let  $G = (V, E)$  be a compacted de Bruijn graph. The following procedure creates small-weight SPSS for the corresponding  $k$ -mer set.

1. Create a graph  $G'$  whose vertex set is the same as  $G$ . Each vertex in  $G'$  has two sides as in  $G$ .
2. Prepare  $n$  mutex locks  $mu_1, mu_2, \dots, mu_n$ .
3. For each vertex  $v$  in  $G'$ , perform the following operations in parallel.
  - a. If there is a vertex  $v'$  such that  $v' \neq v$  and there is an edge between the right side of  $v$  and the left side of  $v'$  in  $G$ , perform the following operations.
    - i. Acquire the locks  $mu_{id(v)\%n}$  and  $mu_{id(v')\%n}$ . Here,  $id$  is a bijective function from  $V$  to integers from 0 to  $|V| - 1$ .
    - ii. If there are no edges incident to the right side of  $v$  and there are no edges incident to the left side of  $v'$  in  $G'$ , add an edge in  $G'$  to connect the right side of  $v$  and the left side of  $v'$ .
    - iii. Release the locks  $mu_{id(v')\%n}$  and  $mu_{id(v)\%n}$ .
  - b. If there is a vertex  $v'$  such that  $v' \neq v$  and there is an edge between the right side of  $v$  and the right side of  $v'$  in  $G$ , perform the similar operations as (a).

- c. If there is a vertex  $v'$  such that  $v' \neq v$  and there is an edge between the left side of  $v$  and the right side of  $v'$  in  $G$ , perform the similar operations as (a).
- d. If there is a vertex  $v'$  such that  $v' \neq v$  and there is an edge between the left side of  $v$  and the left side of  $v'$  in  $G$ , perform the similar operations as (a).
4. Divide  $V$  to  $V_1, V_2, \dots, V_m$  so that the elements of each are connected and each is maximal.
5. For  $i = 1, 2, \dots, m$ , perform the following operation in parallel.
  - a. If both the sides of  $v$  have edges incident to it for all  $v \in V_i$ , remove one arbitrary edge that connects two vertices in  $V_i$ .
6. Make a set of vertices  $V'$  such that for each  $v \in V'$ , there is no edges incident to the left side of  $v$  or there is no edges incident to the right side of  $v$ .
7. Prepare a set of strings  $S$ .
8. For each vertex  $v$  in  $V'$ , perform the following operations in parallel.
  - a. Find the longest path from  $v$  in  $G'$ . It is uniquely determined.
  - b. Let the endpoint of the path be  $v'$ . If the length of the path is 1 or  $label(v) < label(v')$ , add to  $S$  the string represented by the path. Here, the dictionary order of strings is used to compare strings.
9. Return  $S$ .

Step 4 can be done efficiently if we use a disjoint-set data structure that can be used by multiple threads. In our experiments, an implementation based on [3] was used.

The proof of correctness of this algorithm is shown in the appendix.

## 5 Experiments and Results

In this chapter, the effectiveness of the compression algorithm is shown along with the effects of the number of input files and the comparison to the compression method based on Mantis [13, 1], one of the state-of-the-art colored de Bruijn graph representations. The performance gain of the proposed SPSS construction algorithm in multi-thread environments is also shown along with the comparison to the existing method, UST [16].

### 5.1 Compression of multiple $k$ -mer sets

The proposed algorithm to compress multiple  $k$ -mer sets was tested with 3292 E. coli whole-genome sequencing data and 2555 human RNA-Seq data. The former is the data that was accessible for Project PRJNA292667 on Sequence Read Archive [9] on 12/4/2020. The latter is the data used in [17], but some files were omitted because 5 files were corrupted (SRR346129, SRR346128, SRR346127, SRR448331, and SRR448330) and some files contained no reads whose length is at least  $k$ . For the E. coli data, we used  $k = 23$  and the cutoff value of 4, i.e.,  $k$ -mers that appeared 3 times or less were ignored to leave out erroneous data. For the RNA-Seq data, we used  $k = 23$ , and different cutoff values were used for different input files, following the procedures in [17]. The reason we did not apply the dynamic cutoff setting to the E. coli data is that the file sizes did not vary that much in the E. coli data compared to the human RNA-Seq data. 32-core Intel Cascade Lake machines running at 2800.262 MHz were used.

#### Results

As a result, the E. coli genome data could be compressed to 2.42% in size from the naive representation of the  $k$ -mers (the representation in which each  $k$ -mer uses  $2k$  bits), whereas the compression rate was 6.03 % with zero iterations, which corresponds to skipping step 3 of the compression algorithm and individually compressing  $k$ -mer sets with SPSS and bzip2.

## 12:12 Compression of Multiple $k$ -Mer Sets

This shows that by repeating the iterations, the algorithm successfully reduced the file size, thereby illustrating the algorithm’s effectiveness. The effectiveness was also shown with the human RNA-Seq data, and the results are summarized in Table 1.

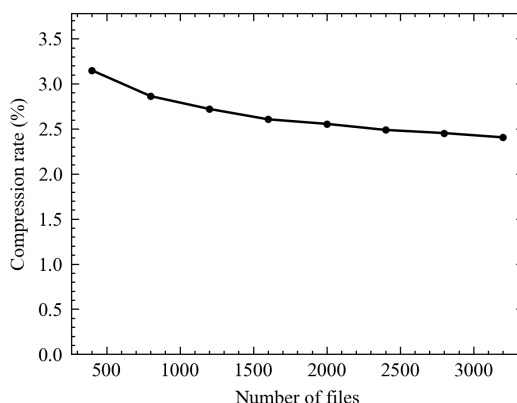
■ **Table 1** Comparison between the compression ratio that could be achieved without iterations, which corresponds to individually compressing each  $k$ -mer set with SPSS and bzip2, and the one that could be achieved with iterations, which corresponds to the actual result by the proposed compression algorithm.

	Without iterations	Actual result
E. coli genome (3292 files)	6.03%	<b>2.42%</b>
Human RNA-Seq (2555 files)	8.06%	<b>6.04%</b>

### Effects of the number of input files

We conducted an experiment to see how the compression capability is affected by the number of input files.

First, the 3292 E. coli data were shuffled randomly. Then, we applied our algorithm with the first 400, 800, ..., and 3200 files. The compression rates for each setting are shown in Figure 7. We can see that the more files there are, the better compression it can achieve.



■ **Figure 7** Compression rates that could be achieved with our proposed algorithm with 400, 800, ..., and 3200 files of E. coli data.

### Comparison to existing methods

As we saw in the earlier chapter, colored de Bruijn graphs can be used to represent multiple  $k$ -mer sets efficiently. In this section, one of the recent colored de Bruijn graph implementations, Mantis [13, 1], is compared with our proposed method in terms of compression capability, required time, and memory usage.

To see the compression capability of Mantis, the following steps were taken. First, we used Squeakr [14] to pre-process the 3292 E. coli data or the 2555 human RNA-Seq data (with “-n” option) and used “mantis build” command (with “-s 30” option) followed by “mantis mst” command to build the colored de Bruijn graph. For “mantis mst”, we specified to use 32 threads, whereas we did not have the multi-threading option for “mantis build”. And boundaries.bv, deltas.bv, parents.bv, meta\_info.json, and dbg\_cqf.ser were further compressed with bzip2 and their total file size was measured. The applications were run on machines with the same CPU specifications as the ones that were used in the experiments for our method.

As you can see in Table 2, our proposed method performed better than Mantis in terms of compression capacity. We can also see that our implementation used more time than Mantis but used less memory than Mantis.

■ **Table 2** Comparison between our method and Mantis [13, 1].

Data	Metric	Mantis	Proposed
E. coli genome (3292 files)	Time	<b>2h24m10s</b>	3h59m08s
	Memory	33.5GB	<b>26.1GB</b>
	Compression	3.70%	<b>2.42%</b>
Human RNA-Seq (2555 files)	Time	<b>1h14m54s</b>	2h44m21s
	Memory	27.4GB	<b>24.8GB</b>
	Compression	6.97%	<b>6.04%</b>

Note that our algorithm and Mantis have different strengths other than the differences in compression capability, time, and memory. The structure of Mantis, once loaded to memory, achieves fast responses to queries necessary for tasks like genotyping, for which our method cannot be used. On the other hand, with our proposed method, it is not necessary to load all the data to memory for decompression, whereas we have to do so with Mantis.

## 5.2 Fast construction of small-weight SPSS

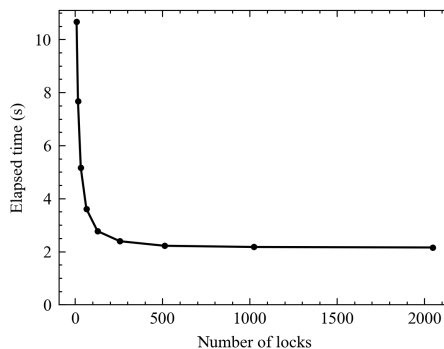
We introduced a small-weight SPSS construction algorithm that can be parallelized. In this section, the performance of the proposed algorithm is measured, and it is compared with the performance of the existing non-parallelizable algorithm, UST [16].

For the experiments, we obtained a set of canonical  $k$ -mers from whole-genome sequencing data of *E. coli* (SRR3160259 from Sequence Read Archive [9]).  $k$  was set to 23, and the cutoff was set to 4. All the experiments were repeated 10 times on 16-core Intel Cascade Lake machines running at 2800.262 MHz, and the average values were considered.

### Effects of the number of locks

In this section, the effect of the number of locks, a parameter of the algorithm, is measured.

We fixed the number of threads to 16 and saw the effects of the number of mutex locks. Figure 8 shows the time required for construction of SPSS with 8, 16, 32, ..., or 2048 locks. We can see a downward trend with the number of locks, but it is almost flat after 512 locks. Based on this result, we chose to use 1024 mutex locks in the rest of the experiments.



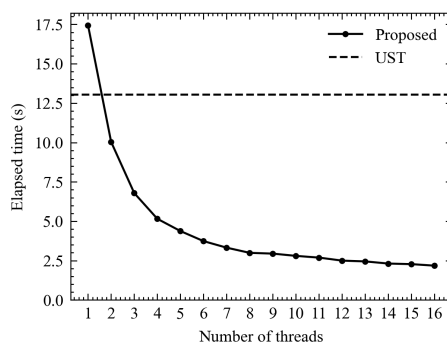
■ **Figure 8** Required time for construction of SPSS using 8, 16, 32, ..., or 2048 mutex locks.

### Comparison to existing methods

Here, we compare the performance of the proposed SPSS construction algorithm with the existing algorithm, UST [16], and show that our algorithm performs well with multiple threads whereas UST does not support parallelization.

We varied the number of threads and measured the proposed algorithm’s performance. Figure 9 shows the required time with 1, 2, ..., or 16 threads. It took 17.4 seconds with 1 thread, 10.0 seconds with 2 threads, and down to 2.18 seconds with 16 threads.

With UST, the operation took 13.0 seconds. So, our method performed worse than UST with a single thread, but the runtime could be reduced by 83.2% with 16 threads compared to UST with a single thread. Note that UST cannot utilize multiple threads.



■ **Figure 9** Required time for construction of SPSS using 1, 2, ..., or 16 threads with the proposed algorithm. The baseline value (required time with UST [16]) is also shown.

## 6 Conclusion and Future Work

In this paper, we introduced a compression algorithm for multiple  $k$ -mer sets that combines the power of SPSS along with a method to utilize similarities in  $k$ -mer sets, along with techniques to make the compression algorithm efficient, including a small-weight SPSS construction algorithm that runs fast in multi-core environments.

With the compression algorithm, it was possible to compress 3292 *E. coli* whole-genome sequencing data and 2555 human RNA-Seq data with better compression rate than Mantis [13, 1], one of the state-of-the-art colored de Bruijn graph representations.

In addition to the high compression capability, our compression algorithm has some other benefits. First, the performance of the compression algorithm improves well with multiple threads. This is especially important on machines where many cores are available, which is often the case these days. Second, the proposed method requires small memory usage, as shown in the comparison to Mantis. It indicates that our method can be applied to larger files. Third, with the proposed algorithm, it is not necessary to load all the compressed data to memory when doing the decompression, thereby achieving smaller memory usage on decompression, which will not be the case with methods based on colored de Bruijn graphs.

There are possible extensions to the compression algorithm. Currently, it only supports sets of unique  $k$ -mers, and does not support multi-sets of  $k$ -mers, or abundance of  $k$ -mers, which can preserve more information of original sequence data. Some researches have been done on methods to efficiently handle abundance of  $k$ -mers in a single data source, one of which is REINDEER data structure [12]. It is interesting to see whether our compression algorithm can be applied to multi-sets of  $k$ -mers.

The proposed compression algorithm can be helpful for researchers who want to work with multiple datasets, maintainers of genome-related databases, etc. Also, the proposed small-weight SPSS construction algorithm, which was faster by 83.2 % than existing methods in 16-core environments and was used in the implementation of the compression algorithm, will speed up SPSS-based programs in multi-core environments.

---

## References

- 1 Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An efficient, scalable and exact representation of high-dimensional color information enabled via de Bruijn graph search. In *Proceedings of the International Conference on Research in Computational Molecular Biology*, pages 1–18, 2019.
- 2 Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: A succinct colored de Bruijn graph representation. In *Proceedings of the International Workshop on Algorithms in Bioinformatics*, pages 18:1–18:15, 2017.
- 3 Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the ACM symposium on Theory of Computing*, pages 370–380, 1991.
- 4 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Proceedings of the International Workshop on Algorithms in Bioinformatics*, pages 225–235, 2012.
- 5 Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de bruijn graphs. *Genome biology*, 22(1):1–24, 2021.
- 6 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.
- 7 Ramana M. Idury and Michael S. Waterman. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2(2):291–306, 1995.
- 8 Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232, 2012.
- 9 Rasko Leinonen, Hideaki Sugawara, and Martin Shumway. The sequence read archive. *Nucleic Acids Research*, 39(suppl\_1):D19–D21, 2010.
- 10 Daniel Lemire, Nathan Kurz, and Christoph Rupp. Stream VByte: Faster byte-oriented integer compression. *Information Processing Letters*, 130:1–6, 2018.
- 11 Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, 2015.
- 12 Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. REINDEER: Efficient indexing of  $k$ -mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement\_1):i177–i185, 2020.
- 13 Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Systems*, 7(2):201–207, 2018.
- 14 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. Squeakr: An exact and approximate  $k$ -mer counting system. *Bioinformatics*, 34(4):568–575, 2018.
- 15 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An eulerian path approach to DNA fragment assembly. *National Academy of Sciences*, 98(17):9748–9753, 2001.
- 16 Amatur Rahman and Paul Medvedev. Representation of  $k$ -mer sets using spectrum-preserving string sets. In *Proceedings of the International Conference on Research in Computational Molecular Biology*, pages 152–168, 2020.
- 17 Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, 34(3):300–302, 2016.
- 18 John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.



## A Proof of correctness of the compression algorithm

In this chapter, we prove the correctness of the compression and decompression algorithm presented in 4.1 and 4.2.

We first consider the following condition.

► **Condition 1.** Let  $\{v_{x_1}, v_{x_2}, \dots, v_{x_{m_i}}\}$  be a set of vertices in  $G$  that are reachable from  $v_i$ .  $S_{x_1} \cup S_{x_2} \cup \dots \cup S_{x_{m_i}}$  is equal to the initial value of  $S_i$ .

Here, “the initial value of  $S_i$ ” refers to the value given as part of an input to the compression algorithm if  $1 \leq i \leq N$ , and it refers to the value initialized in step 3b of the compression algorithm if  $N < i$ .

We prove the correctness by showing the following theorem.

► **Theorem 2.** If Condition 1 holds true for  $i = 1, 2, \dots, n$  at the beginning of an iteration of step 3 in the compression algorithm, Condition 1 also holds true for  $i = 1, 2, \dots, n$  at the end of the iteration.

**Proof.** Let  $n'$  be the value of  $n$  at the beginning of the iteration, and suppose that  $v_j$  and  $v_k$  are selected in step 3a of the iteration.

If  $v_j$  and  $v_k$  cannot be reached from  $v_i$  ( $i \in \{1, 2, \dots, n'\}$ ), the set of vertices  $\{v_{x_1}, v_{x_2}, \dots, v_{x_{m_i}}\}$  that are reachable from  $v_i$  does not change during the iteration. As  $S_{x_1}, S_{x_2}, \dots, S_{x_{m_i}}$  do not change as well, the value of  $S_{x_1} \cup S_{x_2} \cup \dots \cup S_{x_{m_i}}$  gets unchanged, and Condition 1 holds true after the iteration.

If  $v_j$  can be reached from  $v_i$  and  $v_k$  cannot be reached from  $v_i$  ( $i \in \{1, 2, \dots, n'\}$ ),  $v_{n'+1}$  gets added to the set of vertices that are reachable from  $v_i$ . If we suppose that the set of vertices that are reachable from  $v_i$  at the beginning is  $\{v_j, v_{x_1}, v_{x_2}, \dots, v_{x_{m_i}}\}$ , the set of vertices that are reachable from  $v_i$  at the end of the iteration will be  $\{v_j, v_{x_1}, v_{x_2}, \dots, v_{x_{m_i}}, v_{n'+1}\}$ . As  $S_j$  gets subtracted by  $S_{n'+1}$  and  $S_{x_1}, S_{x_2}, \dots, S_{x_{m_i}}$  do not change over the iteration, the value of  $S_j \cup S_{x_1} \cup S_{x_2} \cup \dots \cup S_{x_{m_i}}$  at the beginning of the iteration is equal to the value of  $S_j \cup S_{x_1} \cup S_{x_2} \cup \dots \cup S_{x_{m_i}} \cup S_{n'+1}$  at the end of the iteration. Hence, Condition 1 holds true after the iteration. If  $v_k$  can be reached from  $v_i$  and  $v_j$  cannot be reached from  $v_i$ , the same argument applies.

If  $v_j$  and  $v_k$  are reachable from  $v_i$  ( $i \in \{1, 2, \dots, n'\}$ ),  $v_{n'+1}$  gets added to the set of vertices that are reachable from  $v_i$ . If we suppose that the set of vertices that are reachable from  $v_i$  at the beginning is  $\{v_j, v_k, v_{x_1}, v_{x_2}, \dots, v_{x_{m_i}}\}$ , the set of vertices that are reachable from  $v_i$  at the end of the iteration will be  $\{v_j, v_k, v_{x_1}, v_{x_2}, \dots, v_{x_{m_i}}, v_{n'+1}\}$ . As  $S_j$  and  $S_k$  gets subtracted by  $S_{n'+1}$  and  $S_{x_1}, S_{x_2}, \dots, S_{x_{m_i}}$  do not change over the iteration, the value of  $S_j \cup S_k \cup S_{x_1} \cup S_{x_2} \cup \dots \cup S_{x_{m_i}}$  at the beginning of the iteration is equal to the value of  $S_j \cup S_k \cup S_{x_1} \cup S_{x_2} \cup \dots \cup S_{x_{m_i}} \cup S_{n'+1}$  at the end of the iteration. Hence, Condition 1 holds true after the iteration.

As  $v_{n'+1}$  does not have outgoing edges,  $v_{n'+1}$  itself is the only vertex that is reachable from  $v_{n'+1}$  after the iteration. As  $S_{n'+1}$  is initialized in the iteration, Condition 1 holds true after the iteration for  $i = n' + 1$ . ◀

By combining Theorem 1 with the fact that Condition 1 holds true for  $i = 1, 2, \dots, n$  before the iterations, we can prove that Condition 1 holds true after any number of iterations, thereby showing the correctness of the algorithm.

## **B** Proof of correctness of the SPSS construction algorithm

In this chapter, we prove the correctness of the SPSS construction algorithm presented in 4.4.

As shown in [16], a set of strings represented by the paths of a path cover on a compacted de Bruijn graph is SPSS of the corresponding  $k$ -mer set. We prove that the proposed algorithm produces SPSS by showing that step 8 considers all the vertices exactly once and that a path cover on the compacted de Bruijn graph is considered.

**Proof.** After step 3 of the algorithm, for each vertex  $v$  in  $G'$ , one of the following is true.

- There is one edge incident to the left side of  $v$  and there is one edge incident to the right side of  $v$ .
- There is one edge incident to the left side of  $v$  and there are no edges incident to the right side of  $v$ .
- There are no edges incident to the left side of  $v$  and there is one edge incident to the right side of  $v$ .
- There are no edges incident to  $v$ .

Hence, in step 5, for each  $i = 1, 2, \dots, m$ , one of the following is true.

- There is a path  $p$  in  $G'$  such that  $p$  contains all the vertices in  $V_i$  and either of  $p$ 's endpoint vertices has a side without edges.
- There is a loop such that it contains all the vertices in  $V_i$ . Here, a loop refers to a set of vertices such that they are connected and each of them has one edge incident to its left side and one edge incident to its right side.

Loops get removed during step 5, and the former of the above is true for each  $i = 1, 2, \dots, m$  after step 5.

If  $|V_i| = 1 (i \in \{1, 2, \dots, m\})$ , the vertex  $v \in V_i$  will be added to  $V'$  in step 6. The label of  $v$  will be considered once in step 8.

If  $|V_i| > 1 (i \in \{1, 2, \dots, m\})$ , there is a path  $\{v_1, v_2, \dots, v_{|V_i|}\}$  such that it contains all the vertices in  $V_i$ , one side of  $v_1$  does not have edges incident to it, one side of  $v_{|V_i|}$  does not have edges incident to it, and  $v_2, v_3, \dots, v_{|V_i|-1}$  have edges on both of their sides.  $v_1$  and  $v_{|V_i|}$  will be added to  $V'$  in step 6. And the path will be considered twice in step 8a. But as only one of  $label(v_1) < label(v_{V_i})$  and  $label(v_{V_i}) < label(v_1)$  can be true, the path is only considered once in step 8 as a whole. ◀