Disk Compression of k-mer Sets

Amatur Rahman
Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA
aur1111@psu.edu

Rayan Chikhi
Department of Computational Biology, C3BI USR 3756 CNRS, Institut Pasteur, Paris, France
rayan.chikhi@pasteur.fr

Paul Medvedev
Pennsylvania State University, University Park, PA, USA
pzm11@psu.edu

Abstract

K-mer based methods have become prevalent in many areas of bioinformatics. In applications such as database search, they often work with large multi-terabyte-sized datasets. Storing such large datasets is a detriment to tool developers, tool users, and reproducibility efforts. General purpose compressors like gzip, or those designed for read data, are sub-optimal because they do not take into account the specific redundancy pattern in k-mer sets. In our earlier work (Rahman and Medvedev, RECOMB 2020), we presented an algorithm UST-Compress that uses a spectrum-preserving string set representation to compress a set of k-mers to disk. In this paper, we present two improved methods for disk compression of k-mer sets, called ESS-Compress and ESS-Tip-Compress. They use a more relaxed notion of string set representation to further remove redundancy from the representation of UST-Compress. We explore their behavior both theoretically and on real data. We show that they improve the compression sizes achieved by UST-Compress by up to 27 percent, across a breadth of datasets. We also derive lower bounds on how well this type of compression strategy can hope to do.

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases de Bruijn graphs, compression, k-mer sets, spectrum-preserving string sets

Digital Object Identifier 10.4230/LIPIcs.WABI.2020.16

Supplementary Material Software available at `http://github.com/medvedevgroup/ESSCompress`.

Funding PM and AR were supported by NSF awards 1453527 and 1439057.
Amatur Rahman: AR is supported by NIH Computation, Bioinformatics, and Statistics training program.
Rayan Chikhi: INCEPTION project (PIA/ANR-16-CONV-0005).

1 Introduction

Many of today's bioinformatics analyses are powered by tools that are k-mer based. These tools first reduce the input sequence data, which may be of various lengths and type, to a set of short fixed length strings called k-mers. K-mer based methods are used in a broad range of applications, including genome assembly [4], metagenomics [38], genotyping [36, 14], variant calling [34], and phylogenomics [25]. They have also become the basis of a recent wave of database search tools [32, 33, 35, 15, 7, 5, 26, 13, 23], surveyed in [22]. K-mer based methods are not new, but only recently they have started to be applied to terabyte-sized datasets. For example, the dataset used to test the BIGSI database search index, which is composed of 31-mers from 450,000 microbial genomes [7], takes about 12 TB to store in compressed form.

Storing such large datasets is a detriment to tool developers, tool users, and reproducibility efforts. For tool developers, development time is significantly increased when having to manage such large files. Due to the iterative nature of the development process, these files do not typically just sit in one place, but instead get created/moved/recreated many times. For tool users, the time it takes for the tools to write these files to disk and load them into memory is non-negligible. In addition, as we scale to even larger datasets, storage costs start to play a larger factor. Finally, for reproducibility efforts, storing and moving terabytes of data across networks can be detrimental.

To minimize these negative effects, disk compression of k-mer sets is a natural solution. By disk compression, we refer to a compressed representation that, while supporting decompression, does not support any other querying of the compressed data. Compressed representations that allow for membership queries [10] are important in their own right, but are sub-optimal when only storage is required. Most k-mer sets are currently stored on disk in one of two ways. In the situation where the set of k-mers comes from k-mer counting reads, one can simply compress the reads themselves using one of many read compression tools [17, 16, 24]. This approach requires the substantial overhead of running a k-mer counter as part of decompression, but it is often used in the absence of better options. The second approach is to gzip/bzip the output of the k-mer counter [19, 30, 21, 27, 37]. As we showed in [29], both of these approaches are space-inefficient by at least an order-of-magnitude. This is not surprising, as neither of these approaches was designed specifically for disk compression of k-mer sets.

Disk compression tailor-made for k-mer sets was first considered in our earlier work [29]. The idea was based on the concept of *spectrum-preserving string sets (SPSS)*, introduced in [9, 29, 8]. In [8], the concept of SPSS is introduced under the name *simplitigs*. A set of strings S is said to be a SPSS representation of a set of k-mers K iff 1) the set of k-mers contained in S is exactly K, 2) S does not contain duplicate k-mers, and 3) each string in S is of length $\geq k$. The weight of an SPSS is the number of characters it contains. For example, if $K = \{ACG, CGT, CGA\}$, then $\{ACGT, CGA\}$ would be an SPSS of weight 7; also K itself would be an SPSS of K of weight 9. On the other hand, $\{CGACGT\}$ is not an SPSS, because it contains $GAC \notin K$. Intuitively, a low weight SPSS can be constructed by gluing together k-mers in K, with each glue operation reducing the weight by $k-1$. In [29], we proposed the following simple compression strategy, called UST-Compress. We find a low-weight SPSS S, using a greedy algorithm called UST, and compress S to disk using a generic nucleotide compression algorithm (e.g. MFC [28]). UST-Compress achieved significantly better compression sizes than the two approaches mentioned above.

UST-Compress was not designed to be the best possible disk compression algorithm but only to demonstrate one of the possible applications of the SPSS concept. When the goal is specifically disk compression, we are no longer bound to store a set of strings with exactly the same k-mers as K, as long as a decompression algorithm can correctly recover K. The main idea of this paper is to replace the SPSS with a more relaxed string set representation, over the alphabet $\{A, C, G, T, [,], +, -\}$. Our approach is loosely inspired by the notion of elastic-degenerate strings [18]. It attempts to remove even more duplicate $(k-1)$-mers from the representation than SPSS does, using the extra alphabet characters as placeholders for nearby repetitive $(k-1)$-mers. For the above example, our representation would be $ACG[+A]T$, where the " + " is interpreted as a placeholder for the $k-1$ characters before the open bracket (i.e. CG). After replacing the " + ", we get $ACG[CGA]T$ and we split the string by cleaving out the substring within brackets; i.e., we get $ACGT$ and CGA.

Based on this idea, we present two algorithms for the disk compression of k-mer sets, ESS-Compress and ESS-Tip-Compress. We explore the behavior of these algorithms both theoretically and on real data. We give a lower bound on how well this type of algorithm can compress. We show that they improve the compression sizes achieved by UST-Compress by 10-27% across a breadth of datasets. The two algorithms present a trade-off between time/memory and compression size, which we explore in our results. The two algorithms are freely available open source tools on `http://github.com/medvedevgroup/ESSCompress`.

2 Preliminaries

2.1 Basic definitions

Strings: The *length* of string x is denoted by $|x|$. A string of length k is called a *k-mer*. We assume k-mers are over the DNA alphabet. A string over the alphabet $\{A, C, G, T, [,], +, -\}$ is said to be *enriched*. We use $\cdot$ as the string concatenation operator. For a set of strings S, $weight(S) = \sum_{x \in S} |x|$ denotes the total count of characters. We define $suf_k(x)$ (respectively, $pre_k(x)$) to be the last (respectively, first) k characters of x. We define $cutPre_k(x) = suf_{|x|-k}(x)$ as x with the prefix removed. When the subscript is omitted from pre, suf, and $cutPre$, we assume it is $k-1$. A string x is *canonical* if it is the lexicographically smaller of x and its reverse complement.

For x and y with $suf(x) = pre(y)$, we define *gluing* x and y as $x \odot y = x \cdot cutPre(y)$. For $s \in \{0, 1\}$, we define $orient(x, s)$ to be x if $s = 0$ and to be the reverse complement of x if $s = 1$. We say that x_0 and x_1 have a *(s_0, s_1)-oriented-overlap* if $suf(orient(x_0, 1 - s_0)) = pre(orient(x_1, s_1))$. Intuitively, such an overlap exists between two strings if we can orient them in such a way that they are glueable. For example, AAC and TTG have a $(0, 0)$-oriented overlap.

Bidirected de Bruijn graphs: A *bidirected graph* G is a pair (V, E) where the set V are called vertices and E is a set of edges. An edge e is a set of two pairs, $\{(u_0, s_0), (u_1, s_1)\}$, where $u_i \in V$ and $s_i \in \{0, 1\}$, for $i \in \{0, 1\}$. Note that this differs from the notion of an edge in an undirected graph, where $E \subseteq V \times V$. Intuitively, every vertex has two sides, and an edge connects to a side of a vertex (see Figure 1 for examples). An edge is a *loop* if $u_0 = u_1$. Given a non-loop edge e that is incident to a vertex u, we denote $side(u, e)$ as the side of u to which it is incident. We say that a vertex u is a *dead-end* if it has exactly one side to which no edges are incident. A *bidirected DNA graph* is a bidirected graph G where every vertex u has a string label $lab(u)$, and for every edge $e = \{(u_0, s_0), (u_1, s_1)\}$, there is a (s_0, s_1)-oriented-overlap between $lab(u_0)$ and $lab(u_1)$ (see Figure 1 for examples). G is said to be *overlap-closed* if there is an edge for every such overlap. Let K be a set of canonical k-mers. The node-centric *bidirected de Bruijn graph*, denoted by $dBG(K)$, is the overlap-closed bidirected DNA graph where the vertices and their labels correspond to K. In this paper, we will assume that $dBG(K)$ is not just a single cycle; such a case is easy to handle in practice but is a space-consuming corner-case in all the analyses.

Paths and spellings: A sequence $p = (u_0, e_1, u_1, \ldots, e_n, u_n)$ is a *path* iff 1) for all $1 \leq i \leq n$, e_i is incident to u_{i-1} and to u_i, 2) for all $1 \leq i \leq n-1$, $side(u_i, e_i) = 1 - side(u_i, e_{i+1})$, and 3) all the u_is are different. A path can also be any single vertex. Vertices $u_1, \ldots, u_{n-1}$ are called *internal* and u_0 and u_n are called *endpoints*. We call u_0 to be the *initiator* vertex of p. We say that p is *normalized* if for every e_i, $side(u_{i-1}, e_i) = 1$ and $side(u_i, e_i) = 0$; intuitively, the path uses edges like in a directed graph. The *spelling* of a normalized path p is defined as $spell(p) = lab(u_0) \odot \cdots \odot lab(u_n)$. If P is a set of normalized paths, then $spell(P) = \bigcup_{p \in P} spell(p)$.

Unitigs and the compacted de Bruijn graph: A path in $dbG(K)$ is a *unitig* if all its vertices have in- and out-degrees of 1, except that the first vertex can have any in-degree and the last vertex can have any out-degree. A single vertex is also a unitig. A unitig is *maximal* if it is not a sub-path of another unitig. It was shown in [12] that if $dBG(K)$ is not a cycle, then the set of maximal unitigs forms a unique decomposition of the vertices in $dBG(K)$ into vertex-disjoint paths. The bidirected *compacted de Bruijn graph* of K, denoted by $cdBG(K)$, is the overlap-closed bidirected DNA graph where the vertices are the maximal unitigs of $dBG(K)$, and the labels of the vertices are the spellings of the unitigs. In practice, this graph can be efficiently constructed from K using the BCALM2 tool [12, 11].

Spanning out-forest: Given a directed graph D, an *out-tree* is a subgraph in which every vertex except one, called the root, has in-degree one, and, when the directions on the edges are ignored, is a tree. An *out-forest* is a collection of vertex-disjoint out-trees. An out-forest is *spanning* if it covers all the vertices of D.

2.2 Path covers and UST-Compress

A *vertex-disjoint normalized path cover* Ψ of $cdBG(K)$ is a set of normalized paths such that every vertex is in exactly one path and no path visits a vertex more than once; we will sometimes use the shorter term *path cover* to mean the same thing. There is a close relationship between SPSS representations of K and path covers, shown in [29]. In particular, a path cover Ψ induces the SPSS $spell(\Psi)$. An example of a path cover is one where every vertex of $cdBG(K)$ is in its own path, and the corresponding SPSS is the set of all maximal unitig sequences. Figures 1 and 2 show examples of path covers. The number of paths in Ψ (denoted as $|\Psi|$) and the weight of the induced SPSS is closely related:

$$weight(spell(\Psi)) = |K| + |\Psi|(k-1) \tag{1}$$

This relationship also translates to the number of edges in Ψ; by its definition, the number of edges in Ψ is simply the number of vertices in $cdBG(K)$ minus $|\Psi|$.

The idea of our previous algorithm UST-Compress [29] is to find a path cover Ψ_{UST} with as many edges as possible. Having more edges reduces the number of paths, which in turn reduces the weight of the corresponding SPSS and the size of the final compressed output. We can understand this intuitively as follows. Edges in $cdBG(K)$ connect unitigs whose endpoints have the same $(k-1)$-mer (after accounting for reverse complements). For every edge we add to our path cover, we glue these two unitigs and remove one duplicate instance of the $(k-1)$-mer from the corresponding SPSS. Note however that Ψ_{UST} does not remove all duplicate $(k-1)$-mers from the SPSS, because Ψ can only have two edges incident on a vertex, one from each side, and hence a unitig can only be glued at most twice. If a unitig has edges to more than two other unitigs, then some of the adjacent unitigs would include the duplicate $(k-1)$-mer in the SPSS. The idea of our paper is to exploit the redundancy due to those remaining edges an thus further reduce the size of the representation.

3 ESS-Compress

3.1 Main algorithm

Our starting point is a set of canonical k-mers K, the graph $cdBG(K)$, and a vertex-disjoint normalized path cover Ψ of $cdBG(K)$ returned by UST.[1] To develop the intuition for our

[1] Though we did not explain it in [29], UST always returns normalized paths. It flips any vertex that is in the wrong orientation on its path, by reverse complementing its label, without affecting anything else.

algorithm, we first consider a simple example (Figure 1A). In this example, we see a vertex-disjoint path cover Ψ composed of two paths, ψ^p and ψ^c. There is an edge between an internal vertex (=unitig [2]) u^p of ψ^p and the initiator vertex u^c of ψ^c. Such an edge is an example of an absorption edge. ESS-Compress constructs an enriched string representation of K, as shown in the figure. The basic idea is that u^p and u^c share a common $(k-1)$-mer (i.e. GT). We can cut out this common portion from the string representing u^c and replace it with a special marker character "+". We can then include u^c inside of the representation of u^p by surrounding u^c with brackets. The marker character " + " is a placeholder for the $k-1$ nucleotides right before the opening bracket. To decompress the enriched string, we first replace the marker to get $TCGT[GTAA]T$ and then cleave out the bracketed string to get $\{TCGTT, GTAA\}$. This exactly recovers the SPSS representation of ψ^p and ψ^c.

Formally, we say that an edge in $cdBG(K)$ is an *absorption edge* iff 1) it connects two unitigs u^p and u^c, on two distinct paths ψ^p and ψ^c, respectively, 2) u^p is an internal vertex, and 3) u^c is an initiator vertex. We refer to u^p and ψ^p as *parents* and u^c and ψ^c as *children*; we also say that ψ^p and u^p absorb ψ^c and u^c. [3]

Figure 1B-D shows the other cases, corresponding to the possible orientation of the absorption edge. The logic is the same, but we need to introduce a second marker character " − " that is a placeholder for the reverse complement of the last $k-1$ characters right before the opening bracket. In each of these cases, we add 3 extra characters (two brackets and one marker) and remove $k-1$ nucleotide characters.

Next, observe that a single parent path can absorb multiple children paths, as illustrated in Figure 2A. Also, observe that a single parent unitig can absorb more than one child path, as shown in Figure 2B. As in the previous example, we save $k-1-3=k-4$ characters for every absorbed edge.

These absorptions can be recursively combined, as shown in Figure 2C. Because we require a parent unitig to be an internal vertex and a child unitig to be an initiator vertex, the same unitig cannot be both parent and child. Therefore, ESS-Compress can construct a representation recursively, without any conflicts. The recursion tree is reflected in the nesting structure of the brackets in the enriched string.

However, we must be careful to avoid cycles in the recursion. We define the *absorption digraph* D_A as the directed graph whose vertex set is the set of paths Ψ and an edge $(\psi^p \to \psi^c)$ if ψ^p absorbs ψ^c. For every edge in D_A, we also associate the corresponding bidirected edge between u^p and u^c in $cdBG(K)$. We would like to select a subset of edges F along which to perform absorptions, so as to avoid cycles in D_A and to make sure a path cannot be absorbed by more than one other path. We would also try to choose as many edges as possible, since each absorption saves $k-4$ characters. To achieve these goals, ESS-Compress defines F as a spanning out-forest in D_A with the maximum number of edges. We postpone the algorithm to find F to Section 3.2.

The high-level pseudo-code of ESS-Compress is shown in Algorithm 1 and illustrated in Figure 3. The recursive algorithm to create the enriched representation using F as a guide is shown in Algorithm 2. It follows the intuition we just developed. It starts from the paths that will not be absorbed (i.e. the roots in F). For a path ψ^p, it first computes the enriched representations of all the child paths (Lines 3 to 9). It then integrates them into

[2] Note that the vertices of this graph (i.e. $cdBG(K)$) correspond to maximal unitigs in the non-compacted graph (i.e. $dBG(K)$). We will generally use "vertex" and "unitig" interchangeably, to refer to a vertex in $cdBG(K)$. We never use "unitig" to refer to a type of path in $cdBG(K)$.

[3] In our code, we actually allow a slightly broader definition of absorption. In particular, we also allow an edge to be absorbing if u^p is an initiator and $s^p=1$, or if u^p is an initiator and $|lab(u^p)| \geq 2k-2$. For the sake of simplicity, we do not consider this edge case in the paper.

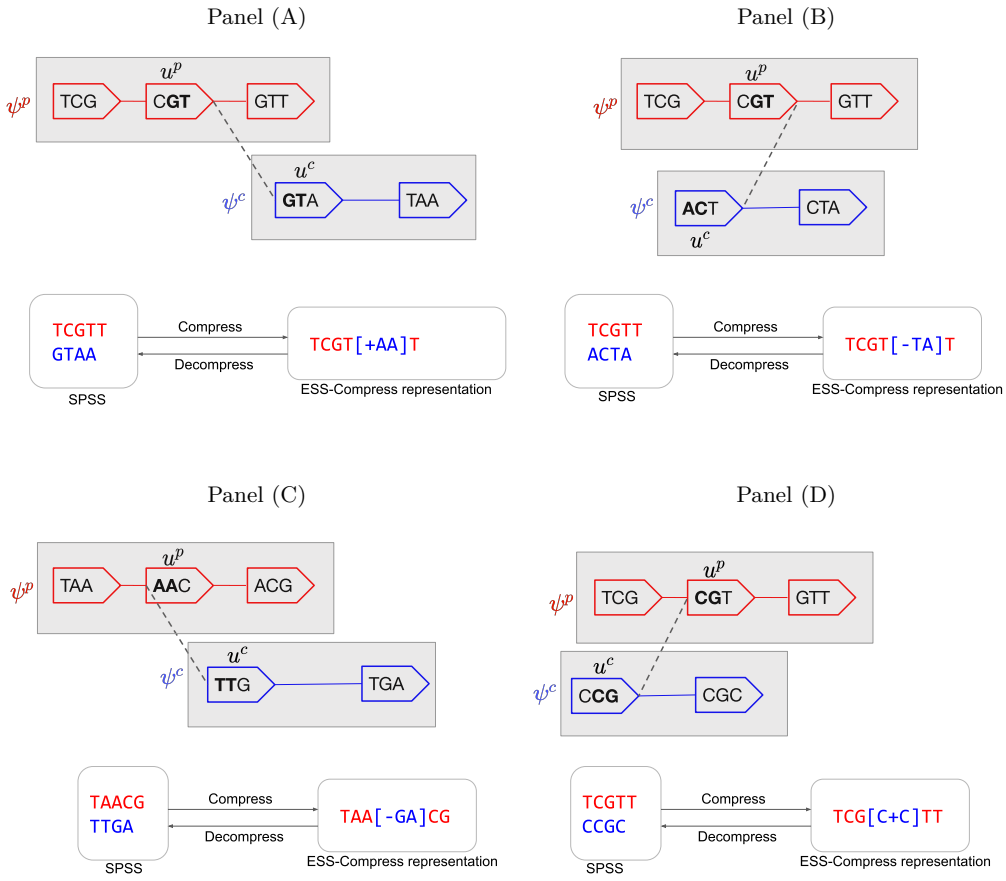

Figure 1 Examples of the four types of absorption. Each panel shows the edges along two paths: ψ^p (red vertices inside a shaded rectangle) and ψ^c (blue vertices inside a shaded rectangle) and an absorption edge $e = \{(u^p, s^p), (u^c, s^c)\}$ (dashed line) between the parent unitig u^p and the child unitig u^c. The graph being shown in each panel is $cdBG(K)$, but only the absorption edge and the edges of ψ^p and ψ^c are shown. In this simple example, the unitigs of $dBG(K)$ are just paths made of single vertices, and hence the vertices of $cdBG(K)$ have labels of length $k = 3$. Each vertex is shown as a pointed rectangle with its label inside; we use the convention that the "zero" side of a vertex is the flat side on the left, and the "one" side is the pointy side on the right. At the bottom left of each panel, we show the spectrum-preserving string set (SPSS) $spell(\{\psi^p, \psi^c\})$. At the bottom right, we show the enriched representation generated by our algorithm. Depending on the value of s^p and s^c, four different cases can arise. When $s^p = 1, s^c = 0$ (shown in (A)), $pre(lab(u^c))$ is replaced with marker "+", as it is same as $suf(lab(u^p))$. When $s^p = 1, s^c = 1$ (shown in (B)), $pre(lab(u^c))$ is replaced by "−", as it is same as the reverse complement of $suf(lab(u^p))$. When $s^p = 0, s^c = 0$ (shown in (C)), $pre(lab(u^c))$ is replaced with "−", as it is the same as the reverse complement of $pre(lab(u^p))$. When $s^p = 0, s^c = 1$ (shown in (D)), $suf(lab(u^c))$ is replaced with "+", as it is the same as $pre(lab(u^p))$.

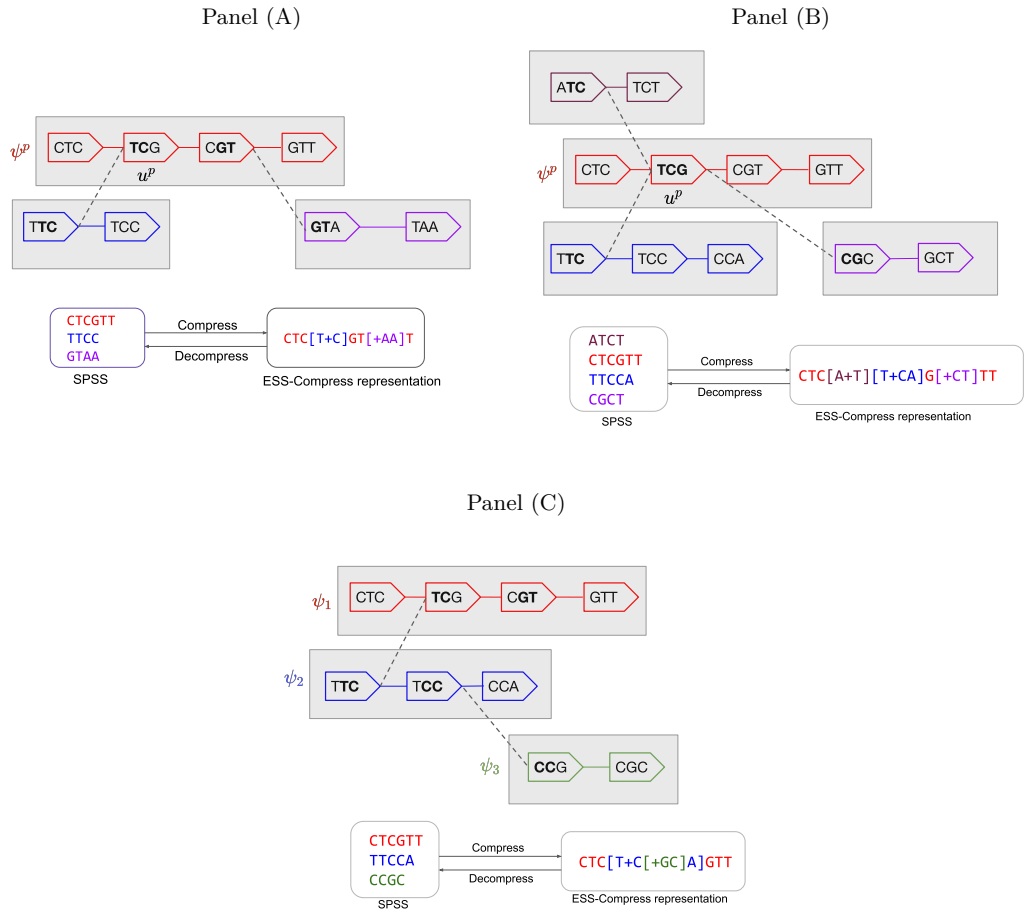

Figure 2 More complex absorption examples. In (A), one path absorbs multiple paths. In (B), one unitig u^p absorbs multiple paths. In (C), one path (ψ_1) absorbs another (ψ_2) which itself absorbs another (ψ_3). This is a recursive absorption, showing how a path can be both a child and a parent.

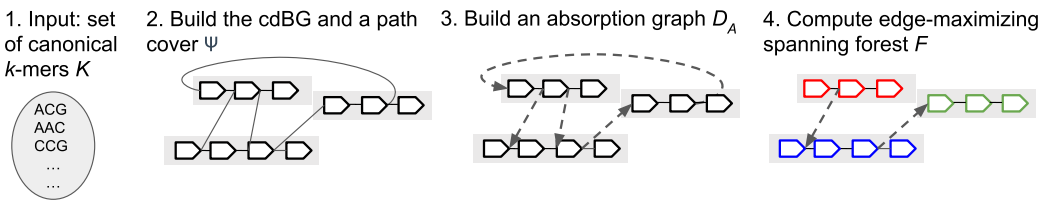

Figure 3 Visual overview of the steps in Algorithm 1.

Algorithm 1 ESS-Compress (K)
Input: a set of canonical k-mers K
Output: a set of enriched strings X.

1: Construct $cdBG(K)$
2: Run UST to get a path cover Ψ
3: Run DFS algorithm to get F, a spanning out-forest of the absorption graph D_A
4: $X \leftarrow \emptyset$
5: **for** each path ψ which is a root in F **do**
6: add Spell-Path-Enrich (ψ, null) to X
7: **end for**
8: **return** X

the appropriate locations of $spell(\psi^p)$ (Lines 10 to 14). It then uses a marker to replace the redundant sequence in the spelling of ψ^p, with respect to ψ^p's own parent (Lines 17 to 31). To decide which marker to use, it receives as a parameter the absorption edge e_D that was used to absorb ψ^p.

Decompression is done by a recursive algorithm DEC that takes as input an enriched string x and a $(k-1)$-mer called $markerReplacement$. Initially, DEC is called independently on every enriched string $x \in$ ESS-Compress(K), with $markerReplacement = null$. We call the characters of x which are not enclosed within brackets *outer*. The brackets themselves are not considered outer characters. DEC first replaces any occurrence of an outer " + " (respectively, " − ") with $markerReplacement$ (respectively, the reverse complement of $markerReplacement$). It then outputs all the outer characters as a single string. Then, for every top-level open/close bracket pair in x, it calls DEC recursively on the sequence in between the brackets, and passes as $markerReplacement$ the rightmost $k-1$ outer characters to the left of the open bracket.

3.2 Algorithm to choose absorption edges

Let D be any directed graph and consider the problem of finding a spanning out-forest with the maximum number of edges. We call this the problem of finding an *edge-maximizing spanning out-forest*. This problem is a specific instance of the maximum weight out-forest problem [3], which allows for weights to be placed on the edges. As we show in this section, there is an optimal algorithm for our problem that is simpler than the algorithm for arbitrary weights described in [3].

Our algorithm first decomposes D into strongly connected components, and builds $SC(D)$, the strongly connected component digraph of D. In $SC(D)$, the vertices are the strongly connected components of D, and there is an edge from component c_1 to c_2 if there is an edge in D from some vertex in c_1 to some vertex in c_2. For every component that is a source in $SC(D)$, we pick an arbitrary vertex from it (in D) and put it into a "starter" set. Then, we perform a depth-first search (DFS) traversal of D, but whenever we start a new tree, we initiate it with a vertex from the starter set, if one is available. We remove the vertex from the starter set once it is used to initiate a tree. We then output the DFS forest F.

We will prove that F is a spanning out-forest of D with the maximum number of edges.

▶ **Lemma 1** (Correctness of edge-maximizing spanning out-forest algorithm). *Let D be a directed graph, let F be the spanning out-forest returned by our algorithm run on D, and let n_{sc} be the number of source components in $SC(D)$. Then, the number of out-trees in F is n_{sc} and this is the smallest possible for any spanning out-forest. Also, the number of edges in F is the maximum possible for any spanning out-forest.*

Algorithm 2 Spell-Path-Enrich(ψ,e_D)
Input: a path ψ corresponding to the sequence of unitigs $u_0, \ldots, u_n$. If ψ is itself absorbed, then the absorption edge e_D.
Output: an enriched string representation of ψ and all its descendent paths in F.

for $i = 0$ to n **do** ▷ for each unitig in ψ
  Use u^p to denote the i^{th} unitig of ψ.
  $ins_0 =$ "" ▷ absorbed enriched strings to insert at the end
  $ins_1 =$ "" ▷ absorbed enriched strings to insert after prefix
  for each unitig u^c absorbed by u^p in F **do**
    Let $e = \{(u^p, s^p), (u^c, s^c)\}$ be the corresponding absorption edge in $cdBG(K)$
    Let $\psi^c \in \Psi$ be the path containing u^c.
    $ins_{s^p} \leftarrow ins_{s^p} \cdot$ Spell-Path-Enrich(ψ^c, e)
  end for
  if $i = 0$ **then** ▷ if u^p is the first unitig in ψ
    $enrichedStr[i] \leftarrow pre(lab(u^p)) \cdot ins_0 \cdot cutPre(lab(u^p)) \cdot ins_1$
  else
    $enrichedStr[i] \leftarrow ins_0 \cdot cutPre(lab(u^p)) \cdot ins_1$
  end if
end for
$x \leftarrow$ concatenate $enrichedStr[i]$, in increasing order of i
if $e_D \neq null$ **then** ▷ if ψ is not a root in F
  /* Perform marker replacement, following Figure 1 */
  Let $\{(u^p, s^p), (u^c, s^c)\} = e_D$
  if $(s^p$ xor $s^c) = 1$ **then**
    $marker =$ " + "
  else
    $marker =$ " − "
  end if
  if $s^c = 1$ **then**
    In x, replace $suf(lab(u^c))$ with $marker$
  else
    In x, replace $pre(lab(u^c))$ with $marker$
  end if
  $x \leftarrow$ "[" $\cdot\, x \cdot$ "]"
end if
return x

Proof. Consider any spanning out-forest of D. If it has less than n_{sc} out-trees, then by the pigeonhole principle, there are two source components c_1 and c_2 with vertices v_1 and v_2, respectively, belonging to the same out-tree. This is a contradiction, since c_1 and c_2 are source components and hence there cannot be a path between them. Hence, any spanning out-forest must have at least n_{sc} out-trees. Now, consider F. Every vertex in D is reachable from one of the vertices in the starter set, by its construction. There are n_{sc} starter vertices, so F will have at most n_{sc} out-trees. Since any spanning out-forest must have at least n_{sc} out-trees, F will have n_{sc} out-trees and it will be the minimum achievable. Also, in any spanning out-forest, the number of edges is the number of vertices minus the number of out-trees; hence F will have the the maximum number of edges of any spanning out-forest. ◀

3.3 The weight of the ESS-Compress representation

In this section, we derive a formula for the weight of the ESS-Compress representation and explore the potential benefits of some variations of ESS-Compress.

▶ **Theorem 2.** *Let K be a set of canonical k-mers, and let Ψ be a vertex-disjoint normalized path cover of $cdBG(K)$ that is used by ESS-Compress(K). Let n_{sc} be the number of sources in the strongly connected component graph of the absorption graph D_A. Let X be the solution returned by ESS-Compress(K). Then*

$$weight(X) = |K| + 3|\Psi| + n_{sc}(k-4)$$

Proof. If we unroll the recursion of ESS-Compress, then there are exactly $|\Psi|$ runs of Spell-Path-Enrich, one for each $\psi \in \Psi$. For each call, we let n_ψ be the number of characters in the returned string that are added non-recursively (i.e. everything except ins_0 and ins_1). Considering the structure of the recursion and accounting for characters in this way, we have that $weight(X) = \sum_{\psi\in\Psi} n_\psi$.

Prior to marker replacement (Line 17, the non-recursive part of x is $spell(\psi)$). When ψ is a root in the absorption forest F, then the marker absorption stage is not executed and so $n_\psi = |spell(\psi)|$. Otherwise, the marker absorption phase (Lines 17 to 31) removes $k-1$ characters, adds 1 new marker character, and adds two new bracket characters. Hence, $n_\psi = |spell(\psi)| - (k-1) + 3 = |spell(\psi)| - (k-4)$. By Lem. 1, F contains n_{sc} roots. Hence,

$$\begin{aligned} weight(X) &= \sum_{\psi\in\Psi} n_\psi = \sum_{\psi \text{ is a root}} |spell(\psi)| + \sum_{\psi \text{ is not a root}} |spell(\psi)| - (k-4) \\ &= \sum_{\psi\in\Psi} |spell(\psi)| - (k-4)(|\Psi| - n_{sc}) = |K| + 3|\Psi| - n_{sc}(k-4) \end{aligned}$$

The last equality follows by applying Equation (1) from Section 2. ◀

We can use Thm. 2 to better understand ESS-Compress. The weight depends on the choice of Ψ. The Ψ returned by UST has, empirically, almost the minimum $|\Psi|$ possible [29]. This (almost) minimizes the $3|\Psi|$ term in Thm. 2. However, this may not necessarily lead to the lowest total weight, because there is an interplay between Ψ and n_{sc}, as follows. Let Ψ' be a vertex-disjoint normalized path cover with $|\Psi'| > |\Psi|$. Its paths are shorter, on average, than Ψ's. There may now be edges of $cdBG(K)$ that become absorption edges, that were not with Ψ. For example, an edge between two unitigs which are internal in Ψ is not, by our definition, an absorption edge. With the shorter paths in Ψ', one of these unitigs may become an initiator vertex, making the edge absorbing. This may in turn improve connectivity in D_A and decrease n_{sc}, counterbalancing the increase in $|\Psi'|$. Nevertheless, ESS-Compress does not consider alternative path covers and always uses the one returned by UST.

Another aspect of ESS-Compress that could be changed is the definition of absorption edge. We restrict absorption edges to be between an initiator unitig and an internal unitig; however, one could in principle also define ways to absorb between an endpoint unitig and an internal unitig, or between two internal unitigs. This could potentially decrease n_{sc} by increasing the number of absorption edges, though it would likely need more complicated and space-consuming encoding schemes.

How much could be gained by modifying the path cover and the absorption rules that ESS-Compress uses? We can answer this by observing that n_{sc} cannot be less than C, the number of connected components of the undirected graph underlying $cdBG(K)$. At the same time, in [29] we gave an algorithm to compute an instance-specific lower bound β on the number of paths in any vertex-disjoint path cover. Putting this together, we conclude that

regardless of which path cover is used and which subset of $cdBG(K)$ edges are allowed to be absorbing, the weight of a ESS-Compress representation cannot be lower than:

$$|K| + 3\beta + C(k-4) \tag{2}$$

As we will see in the results, the weight of ESS-Compress is never more than 2% higher than this lower bound, which is why we did not pursue these other possible optimizations to ESS-Compress. We note, however, that the above is not a general lower bound and does not rule out the possibility of lower-weight string set representations that beat ESS-Compress.

4 ESS-Tip-Compress: a simpler alternative

ESS-Compress is designed to achieve a low compression size but can require a large memory stack due to its recursive structure. The memory during compression and decompression is proportional to the depth of this stack, which is the depth of the out-forest F. If F were to be more shallow, then the memory would be reduced. In this section, we describe ESS-Tip-Compress, a simpler, faster, and lower-memory technique that can be used when compression speed/memory are prioritized. It is centered on dead-end vertices in the compacted graph, which usually correspond to tips in the uncompacted dBG and are typically due to sequencing errors, endpoints of transcripts, or coverage gaps. ESS-Tip-Compress is based on the observation that a large chunk of the graph is dead-end vertices (at least for sequencing data), and limiting absorption to only them can yield much of the benefits of a more sophisticated algorithm.

First, we find a vertex-disjoint normalized path cover Ψ that is forced to have each dead-end vertex in its own dedicated path (i.e. its path only contains the vertex itself). This can be done easily by running UST on the graph obtained from $cdBG(K)$ by removing all dead-end vertices. Next, we select the absorption forest F as follows. For each dead-end vertex v, we identify a non-dead-end vertex u which is connected to v via an edge e. In the rare case that such a u does not exist, we skip v. Otherwise, we add $(u \to v)$ to F. We can assume without loss of generality that $side(u, e) = 1 - side(v, e)$ because if that is not the case, than we can replace $lab(v)$ by its reverse complement and thereby change the side to which e is incident. For any paths that remain uncovered by F, we add them as roots of their own tree. Finally, we run a slightly modified version of Spell-Path-Enrich, using this Ψ and this F.

We modify Spell-Path-Enrich as follows. First, observe that F has max depth of 2 vertices. Hence, the parenthesis generated by Spell-Path-Enrich are never nested. Second, observe that the marker value is always " + ", because $side(u, e) = 1 - side(v, e)$ for all absorption edges in F. These observations allow us to reduce the number of extra characters we need for each absorption down to 2, instead of 3 (we omit the implementation details).

5 Empirical Results

We evaluated our methods on one small bacterial dataset, two metagenomic datasets from NIH human microbiome project, and RNA-seq reads from both human and plant (Table 1). To obtain the set of k-mers K from these datasets, we ran the DSK k-mer-counter [30] with $k = 31$ and filtered out low-frequency k-mers (<5 for whole human and <2 for the other datasets). We then constructed $cdBG(K)$ using BCALM2. The last three columns in Table 1 show the properties of the graph: number of vertices, number of dead-end vertices and total percentage of isolated vertices. We ran all our experiments single-threaded on

Table 1 Dataset characteristics.

Dataset	Source	Read Length (bp)	# reads	# distinct 31-mers	# unitigs	% dead-end unitigs	% isolated unitigs
R. sphaeroides	GAGE [31]	101	2,050,868	5,908,467	442,681	47%	8%
Human RNA-seq	SRR957915	101	49,459,840	101,017,526	7,665,682	40%	13%
Gingiva metagenome	SRS014473	101	55,419,548	101,872,420	5,678,516	36%	15%
Soybean RNA-seq	SRR11458718	125	83,594,116	111,206,789	3,659,969	28%	12%
Tongue metagenome	SRS011086	101	81,664,789	165,159,726	11,358,233	37%	11%
Whole human	ERR174310	101	207,579,467	2,319,022,432	51,094,913	14%	18%

a server with an Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz processor with 64 cores and 512 GB of memory. We used `/usr/bin/time` to measure time and memory. Detailed steps to reproduce our experiments are available at `https://github.com/medvedevgroup/ESSCompress/tree/master/experiments`.

The output of our tools was compressed with MFC. Note that MFC is not optimized for non-nucleotide characters, but such characters are rare in our string sets (< 0.1 bits per k-mer). We compared our tools against four other approaches. The first is UST-Compress, which we showed in our previous work to outperform other disk compressors [29]. The second is to strip the read FASTA files of all non-sequence information and compress them using MFC. The third is to simply write one distinct k-mer per line to a file and compress it using MFC. The fourth is the BOSS method, as implemented in [1]. BOSS is a succinct implementation of a de Bruijn graph [6]. Though it is designed to answer membership queries, it also achieved the closest compression size to UST-Compress in our previous study [29]. As in [29], we compressed BOSS's binary output using LZMA. We confirmed the correctness of all evaluated tools, including our own, on the datasets.

We did not explore the possibility of replacing UST in our pipeline with ProphAsm [2]. ProphAsm is an alternative algorithm to compute an SPSS called simplitigs, but we showed in [29] that the UST SPSS representation is nearly optimal, with only 2-3% difference to the lower bound of weight. Since ProphAsm computes the same kind of representation, it is impossible for it to improve result beyond 2-3%. We also did not compare against other k-mer membership data structures because in our previous paper [29], we showed that UST-Compress and BOSS achieve a better compression ratio on the tested datasets.

String set properties

We first measure the weights and sizes of our ESS-Compress and ESS-Tip-Compress, shown in Table 2. ESS-Compress uses 13-42% less characters than UST. ESS-Tip-Compress was worse than ESS-Compress (6-13% larger), but still better than UST-Compress (3-38% smaller). The lower bound computed by Eq. 2 is very close to the weight of ESS-Compress (within 1.7%, Table 2), indicating that the alternate strategies explored in Section 3.3 would not be useful on these datasets.

Compression size

Table 3 shows the final compression sizes, after the string sets are compressed with MFC. ESS-Compress outperforms the second best tool (which is usually UST-Compress) by 4-27%. It outperforms the naive strategies (i.e. read FASTA or one k-mer per line) by an order-of-magnitude. Interestingly, it outperforms ESS-Tip-Compress by only 1-8%; this can be attributed to the large number of dead-end vertices (Table 1).

Table 2 The weights and sizes of various string set representations. The rightmost column shows the lower bound computed by Equation (2) in Section 3.3. The weight of ESS-Compress was verified to be the same as predicted by Theorem 2.

	UST		ESS-Tip-Compress		ESS-Compress		Eq. 2 lower bound
Dataset	# strings	#char/ k-mer	# strings	#char/ k-mer	# strings	#char/ k-mer	#char/ k-mer
R. sphaeroides	240,562	2.22	61,909	1.38	36,456	1.29	1.28
Human RNA-seq	4,098,389	2.22	1,834,945	1.60	1,098,938	1.42	1.39
Gingiva metagenome	3,095,476	1.91	1,499,270	1.48	917,388	1.33	1.32
Soybean RNA-seq	1,806,078	1.49	1,137,350	1.32	515,244	1.17	1.17
Tongue metagenome	6,030,814	2.10	2,664,422	1.53	1,327,701	1.33	1.32
Whole human	22,072,219	1.32	21,320,263	1.28	10,321,275	1.15	1.14

Table 3 The compression sizes, as measured in bits per k-mer in the compressed output. All string representations (i.e. not BOSS) are compressed using MFC in the final step. Since BOSS is a binary representation, we use LZMA for the final compression step.

Dataset	Read FASTA	One k-mer per line	BOSS	UST-Compress	ESS-Tip-Compress	ESS-Compress
R. sphaeroides	45.4	28.4	6.55	3.93	2.90	2.87
Human RNA-seq	45.8	31.7	6.89	4.14	3.43	3.33
Gingiva metagenome	48.0	32.4	10.64	3.76	3.22	3.05
Soybean RNA-seq	43.0	33.1	5.97	2.83	2.66	2.55
Tongue metagenome	48.1	33.3	3.59	4.07	3.32	3.07
Whole human	31.9	48.2	4.65	2.49	2.46	2.40

We observe that our improvement in weight (Table 2) does not directly translate to improvement after compression with MFC (Table 3). For ESS-Compress, the average improvement in weight over UST is 30% but the improvement in bits is 17%. We attribute this to the fact that MFC works by exploiting redundant regions, based on their context. Thus, the redundant sequence that ESS-Compress removes is likely the sequence that was more compressible by MFC and hence MFC looses some of its effectiveness.

We also verified that ESS-Compress can successfully compress datasets of varying k-mer sizes (between 21 and 71) and low-frequency thresholds (2,3, and 4). Figure 4 shows compressed sizes of human RNA-seq data in bits/k-mer as well as their weights compared to the lower bounds. The weight of ESS-Compress closely matches the lower bound across all parameters ($< 2.4\%$ gap), but the weight and compression size increase for larger k and lower thresholds.

Decompression and compression time and memory

The cost of decompression is important since it is incurred every time the dataset is used for analysis. For both ESS-Compress and ESS-Tip-Compress, the decompression memory is < 1 GB (Table 5) the time is < 10 minutes for the large whole human dataset and < 1.5 minutes for the other datasets (Table 4). Both of these are dominated by the MFC portion.

Compression is typically done only once, but the time and memory use can still be important in some applications. Tables 5 and 6 show the compression time and memory

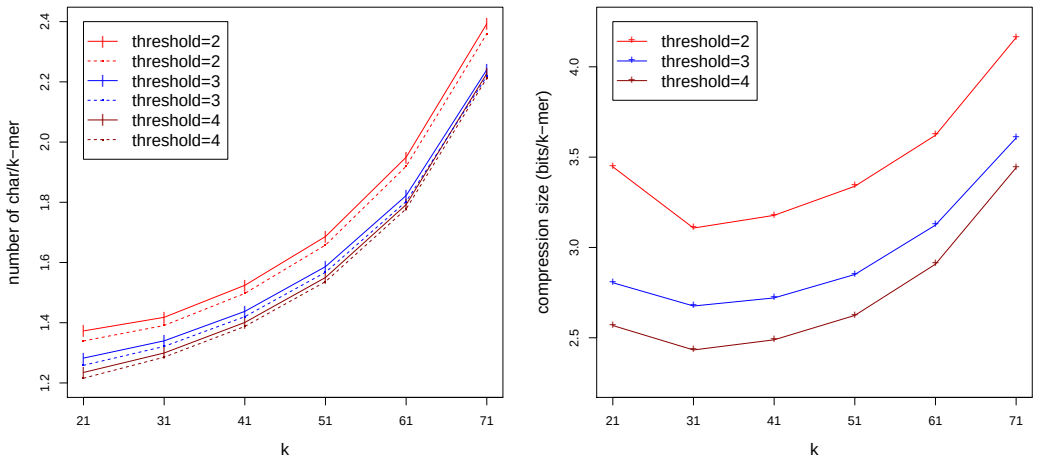

Figure 4 Compression performance of ESS-Compress when varying k and the low-frequency filter threshold, on Human RNA-seq dataset. In the left panel, solid lines represent the weight of the ESS-Compress representation, compared against the lower bound, represented by the dashed lines. In the right panel, compressed sizes are shown in bits/k-mer.

Table 4 Decompression time in *seconds*. The time is broken down into the portion taken by MFC to decompress the binary file into an enriched string set and the portion taken by our core algorithm to decompress the enriched string set into an SPSS. Note that BOSS does not implement decompression (because it is a membership data structure) so it is not included.

Dataset	UST-Compress	ESS-Tip-Compress			ESS-Compress		
	MFC-D	MFC-D	Core	Total	MFC-D	Core	Total
R. sphaeroides	3	2	1	4	2	1	3
Human RNA-seq	40	41	19	60	34	17	51
Gingiva metagenome	37	38	16	54	30	15	45
Soybean	31	33	13	46	29	13	42
Tongue metagenome	62	61	28	89	49	25	74
Whole human	302	337	259	596	303	250	553

usage. For UST-Compress, the time is dominated by the *cdBG* construction step (i.e. BCALM2). For ESS-Compress, the time and memory are significantly increased beyond that. Here, the advantage of ESS-Tip-Compress stands out. Its run time is nearly the same as UST-Compress, and its memory, while close to UST-Compress, is significantly lower than ESS-Compress.

Note that MFC is one of many DNA sequence compressors that can be used with our algorithms. MFC is known to achieve superior compression ratios but is slower for compression/decompression than other competitors [20]. We recommend using MFC since it was not the time or memory bottleneck during compression, in our datasets.

Table 5 Peak memory usage for compression and decompression. Decompression takes far less memory than compression, so compression memory is shown in *GB* and decompression memory in *MB*. Decompression memory is split in the same manner as the running time in Table 4.

Dataset	Compression (GB)				Decompression (MB)				
	BOSS	UST-Compress	ESS-Tip-Compress	ESS-Compress	UST-Compress	ESS-Tip-Compress		ESS-Compress	
					MFC-D	MFC-D	Core	MFC-D	Core
R. sphaeroides	2	3	3	3	509	513	3	513	4
Human RNA-seq	4	3	3	6	515	515	3	515	38
Gingiva metagenome	4	2	2	5	515	515	3	515	4
Soybean	4	2	2	3	515	515	3	515	12
Tongue metagenome	4	2	2	9	515	515	3	515	6
Whole human	5	12	11	42	515	515	3	515	735

Table 6 Compression time, measured in *minutes*. The column for BOSS includes the time for k-mer counting the reads using KMC [19], the time to run BOSS construction, and the time to run LZMA. The total time in UST-Compress, ESS-Tip-Compress and ESS-Compress include the time to compute *cdBG* from the reads using BCALM, which is same for all three. The columns labelled *core* refer to Algorithm 1. ESS-Tip-Compress core uses the specific instance of Algorithm 1 defined in Section 4.

Dataset	BOSS	BCALM	UST-Compress			ESS-Tip-Compress			ESS-Compress		
			UST	MFC	Total	Core	MFC	Total	Core	MFC	Total
R. sphaeroides	0.2	0.4	0.1	0.1	1	0.1	0.0	1	0.2	0.0	1
Human RNA-seq	4.0	6.6	1.6	0.8	9	1.3	0.7	9	5.0	0.6	12
Gingiva metagenome	4.3	5.5	1.2	0.7	7	1.0	0.7	7	3.4	0.6	10
Soybean	5.7	9.6	0.8	0.6	11	0.7	0.7	11	2.4	0.5	13
Tongue metagenome	7.4	8.7	1.6	0.8	11	1.9	1.1	12	7.6	0.9	17
Whole human	95	106	11	7	124	10	6	122	40	7	152

6 Discussion

In this paper, we presented a disk compression algorithm for k-mer sets called ESS-Compress. ESS-Compress is based on the strategy of representing a set of k-mers as a set of longer strings with as few total characters as possible. Once this string set is constructed, it is compressed using a generic nucleotide compressor such as MFC. On real data, ESS-Compress uses up to 42% less characters than the previous best algorithm UST-Compress. After MFC compression, ESS-Compress uses up to 27% less bits than UST-Compress.

We also presented a second algorithm ESS-Tip-Compress. It is simpler than ESS-Compress and does not achieve as good of compression sizes. However, the difference is less than 8% on our data, and it has the advantage of being about twice as fast and using significantly less memory during compression. For many users, this may be a desirable trade-off.

Our algorithms can also be used to compress information associated with the k-mers in K, such as their counts. Every k-mer in K corresponds to a unique location in the enriched string set. The counts can then be ordered sequentially, in the same order as the k-mers appear in the string set, and stored in a separate file. This file can then be compressed/decompressed separately using a generic compressor. After decompression of the enriched string set, the order of k-mers in the output SPSS will be the same as in the counts file.

We discussed several potential improvements to ESS-Compress, such as allowing more edges in the compacted de Bruijn graph to be absorbing or exploring the space of all path covers. We also gave a lower bound to what such improvements could achieve and showed

they cannot gain more than 2% in space on our datasets. This makes these improvement of little interest, unless we encounter datasets where the gap is much larger.

ESS-Compress works by removing redundant $(k-1)$-mers from the string set, but a more general strategy could be to somehow remove ℓ-mer duplicates, for all $\ell_{min} \leq \ell \leq k-1$. Such a strategy would require novel algorithms but would still be unable to reduce the characters per k-mer below one. On our datasets, this amounts to at most a 30% improvement in characters, which would be further reduced after MFC compression. It is also not clear if a 30% improvement in characters is even possible, since this kind of strategy would require a more sophisticated encoding scheme with more overhead.

Another direction to achieve lower compression sizes is to look beyond string set approaches. We observe, for example, that the large improvement of ESS-Compress compared to UST-Compress, measured in the weight of the string set, was significantly reduced when measured in bits after MFC compression. This indicates that some of the work done by ESS-Compress duplicates the work done by MFC on UST, which is itself designed to remove redundancy in the input. Thus, generic compressors such as MFC could potentially be modified to work directly on k-mer sets.

We believe that the biggest opportunity for improving the two algorithms of this paper are the compression time and memory. The time is dominated by the initial step of running BCALM2 to find unitigs. It may be possible to avoid this step by running UST directly on the non-compacted graph. Such an approach was taken in [8], and it would be interesting to see if it ends up improving on the memory and run-time of BCALM2. The memory usage, on the other hand, can likely be optimized with better software engineering. The current implementation of Algorithm 2 is done in a memoized bottom-up manner. Instead, a top down iterative implementation can reduce memory usage by directly writing to disk as soon as a vertex is processed. A "max-depth" option in Algorithm 2 could also be used to limit the depth of the recursion, thereby controlling memory at the cost of the compression ratio.

References

1 URL: `https://github.com/cosmo-team/cosmo/tree/VARI`.

2 URL: `https://github.com/prophyle/prophasm`.

3 Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

4 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.

5 Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. COBS: a compact bit-sliced signature index. *arXiv preprint arXiv:1905.09624*, 2019.

6 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *Proceedings of the 12th International Conference on Algorithms in Bioinformatics*, volume 7534 of *LNCS*, page 225–235. Springer, 2012.

7 Phelim Bradley, Henk C den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology*, 37(2):152, 2019.

8 Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *bioRxiv*, 2020. `doi:10.1101/2020.01.12.903443`.

9 Karel Břinda. *Novel computational techniques for mapping and classifying Next-Generation Sequencing data*. PhD thesis, Université Paris-Est, November 2016. `doi:10.5281/zenodo.1045317`.

10 Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent sets of k-long DNA sequences. *arXiv:1903.12312 [cs, q-bio]*, 2019.

11 Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. In *Research in Computational Molecular Biology, RECOMB 2014*, volume 8394 of *Lecture Notes in Computer Science*, pages 35–55. Springer, 2014.

12 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.

13 Temesgen Hailemariam Dadi, Enrico Siragusa, Vitor C Piro, Andreas Andrusch, Enrico Seiler, Bernhard Y Renard, and Knut Reinert. DREAM-Yara: An exact read mapper for very large databases with short update time. *Bioinformatics*, 34(17):i766–i772, 2018.

14 Luca Denti, Marco Previtali, Giulia Bernardini, Alexander Schönhuth, and Paola Bonizzoni. MALVA: genotyping by Mapping-free ALlele detection of known VAriants. *iScience*, 18:20–27, 2019.

15 R. S. Harris and P. Medvedev. Improved Representation of Sequence Bloom Trees. *Bioinformatics*, 36(3):721–727, 2020.

16 Mikel Hernaez, Dmitri Pavlichin, Tsachy Weissman, and Idoia Ochoa. Genomic Data Compression. *Annual Review of Biomedical Data Science*, 2, 2019.

17 Morteza Hosseini, Diogo Pratas, and Armando Pinho. A survey on data compression methods for biological sequences. *Information*, 7(4):56, 2016.

18 Costas S Iliopoulos, Ritu Kundu, and Solon P Pissis. Efficient pattern matching in elastic-degenerate texts. In *International Conference on Language and Automata Theory and Applications*, pages 131–142. Springer, 2017.

19 Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

20 Kirill Kryukov, Mahoko Takahashi Ueda, So Nakagawa, and Tadashi Imanishi. Nucleotide Archival Format (NAF) enables efficient lossless reference-free compression of DNA sequences. *bioRxiv*, page 501130, 2018.

21 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

22 Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k-mers for querying large collections of sequencing datasets. *bioRxiv*, page 866756, 2019.

23 Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. Reindeer: efficient indexing of k-mer presence and abundance in sequencing datasets. *bioRxiv*, 2020.

24 Ibrahim Numanagić, James K Bonfield, Faraz Hach, Jan Voges, Jörn Ostermann, Claudio Alberti, Marco Mattavelli, and S Cenk Sahinalp. Comparison of high-throughput sequencing data compression tools. *Nature methods*, 13(12):1005, 2016.

25 Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome biology*, 17(1):132, 2016.

26 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.

27 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2017.

28 Armando J Pinho and Diogo Pratas. MFCompress: a compression tool for FASTA and multi-FASTA data. *Bioinformatics*, 30(1):117–118, 2013.

29 Amatur Rahman and Paul Medvedev. Representation of k-mer sets using spectrum-preserving string sets. In *Research in Computational Molecular Biology - 24th Annual International Conference, RECOMB 2020, Padua, Italy, May 10-13, 2020, Proceedings*, volume 12074 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2020.

30 Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

31 Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

32 B. Solomon and C. Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology*, 34(3):300–302, 2016.

33 B. Solomon and C. Kingsford. Improved Search of Large Transcriptomic Sequencing Databases Using Split Sequence Bloom Trees. *Journal of Computational Biology*, 25(7):755–765, 2018.

34 Daniel S Standage, C Titus Brown, and Fereydoun Hormozdiari. Kevlar: a mapping-free framework for accurate discovery of de novo variants. *iScience*, 18:28–36, 2019.

35 Chen Sun, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. AllSome Sequence Bloom Trees. In *Research in Computational Molecular Biology - 21st Annual International Conference, RECOMB 2017, Hong Kong, China, May 3-7, 2017, Proceedings*, volume 10229 of *Lecture Notes in Computer Science*, pages 272–286, 2017.

36 Chen Sun and Paul Medvedev. Toward fast and accurate snp genotyping from whole genome sequencing data for bedside diagnostics. *Bioinformatics*, 35(3):415–420, 2018.

37 Isaac Turner, Kiran V Garimella, Zamin Iqbal, and Gil McVean. Integrating long-range connectivity information into de bruijn graphs. *Bioinformatics*, 34(15):2556–2565, 2018.

38 Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):R46, 2014.