Compressed Weighted de Bruijn Graphs

Giuseppe F. Italiano
Luiss University, Rome, Italy
Erable, INRIA Grenoble Rhône-Alpes, France

Nicola Prezza
DAIS, Ca' Foscari University of Venice, Italy

Blerina Sinaimeri
Luiss University, Rome, Italy
Erable, INRIA Grenoble Rhône-Alpes, France

Rossano Venturini
Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Abstract

We propose a new compressed representation for weighted de Bruijn graphs, which is based on the idea of delta-encoding the variations of k-mer abundances on a spanning branching of the graph. Our new data structure is likely to be of practical value: to give an idea, when combined with the compressed *BOSS* de Bruijn graph representation, it encodes the weighted de Bruijn graph of a 16x-covered DNA read-set (60M distinct k-mers, $k = 28$) within 4.15 bits per distinct k-mer and can answer abundance queries in about 60 microseconds on a standard machine. In contrast, state of the art tools declare a space usage of at least 30 bits per distinct k-mer for the same task, which is confirmed by our experiments. As a by-product of our new data structure, we exhibit efficient compressed data structures for answering partial sums on edge-weighted trees, which might be of independent interest.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Data structures design and analysis

Keywords and phrases weighted de Bruijn graphs, k-mer annotation, compressed data structures, partial sums

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.16

Supplementary Material The code is written in C++ and is available at
Software (Source Code): `https://github.com/nicolaprezza/cw-dBg`

Funding *Giuseppe F. Italiano*: Partially supported by MUR, the Italian Ministry for University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).
Rossano Venturini: Partially supported by MUR, the Italian Ministry for University and Research, under PRIN 2017 Project 2017K7XPAN *Algorithms, Data Structures and Combinatorics for Machine Learning* and Università di Pisa Project PRA_2020-2021_26.

1 Introduction

A DNA resequencing experiment consists of reconstructing the nucleotide sequence of large collections of short DNA fragments sampled from a reference genome [16, 18]. Depending on the genome's length and on the number of sampled fragments, each genome position is typically covered several times (up to a few hundred on average) by different fragments. One of the most common (lossy) representations of such a dataset is a de Bruijn graph of order k (k is usually chosen around 30) introduced in [31] and used by the majority of genome and transcriptome assemblers [2, 24, 20, 37]: this is a combinatorial structure storing all k-mers occurring in the DNA fragments, and connecting two k-mers with an edge when their length-$(k+1)$ concatenation also occurs in the dataset. While such a representation is already

useful to discover important features of the sequenced genome (for example, single-point mutations correspond to bubbles in De Bruijn graphs, see e.g., [35, 36]), a much more useful representation should also record the *abundance* of each k-mer, i.e., how often it appears in the dataset. This augmented structure takes the name *weighted de Bruijn graph* and has many applications in both genome and transcriptome analysis. For instance, in *de novo* genome projects it can be used for error correction [22, 37] or to infer genome characteristics as repeat structures or rate of heterozygosity [21]. In transcriptome projects the role of k-mer abundances is even more important as the abundance of a gene does not only reflect its copy-number in the genome, but also and mostly its expression level. Hence, k-mer abundances are crucial for quantifying the expression level of transcripts [26, 30] but also for distinguishing SNPs from sequencing errors, or between different types of alternative splicing events [17, 23].

Due to the massive size of sequencing data, the main challenge is to design data structures for weighted de Bruijn graphs, which can scale up to very large instances such as genomes, genome collections or metagenomics datasets (see for example [24]). In this paper, we tackle exactly this problem and present new space-efficient data structures for storing weighted de Bruijn graphs.

1.1 State of the art

As far as (unweighted) de Bruijn graphs are concerned, Bowe et al. in [3] presented a space-efficient data structure – *BOSS* in the following – based on the concept of *prefix sorting*. Their idea is to sort co-lexicographically the k-mers appearing in the dataset and to append their outgoing labels to a sequence. It turns out that this sequence is a generalization to de Bruijn graphs of the celebrated Burrows-Wheeler transform [4], the text permutation at the core of the FM-index [12]. In its most basic form, the BOSS representation requires just 4 bits per edge to be stored and supports fast navigation queries on the de Bruijn graph.

It would be highly desirable to achieve the same rate of space efficiency also for weighted de Bruijn graphs. A naive strategy to extend the BOSS representation to weighted de Bruijn graphs could be to associate to each node a counter explicitly storing k-mer abundances. The large variance of these counters (that is, difference between the largest and mean abundance), however, requires some compression strategy in order to save space with respect to a straightforward uncompressed solution.

Although many data structures already present in the literature could support k-mer quantification, their representations are not space efficient [24]. For instance an extension of Bloom filters, known as counting Bloom filters [19], allocates a fixed number of bits per k-mer to store its count. This is clearly not space efficient for datasets where most of the k-mers have low abundance. To deal with this, variable-length counters have been proposed. Among them the counting quotient filter has been introduced in [28] and based on it two tools have been proposed: deBGR [27] and Squeakr [29]. Both tools are *approximate* in the sense that the returned counters could be wrong with low probability, but can also store exact counts at the price of a higher space usage. On a RNA-seq Human experiment composed of 1.4 billion distinct k-mers ($k = 28$) with average abundance per k-mer equal to 27, deBGR [27] reports a space usage of 37 bit/k-mer, while Squeakr [29] uses 79.8 bit/k-mer. Another experiment on k-mer counting tools [29] reports that, on a dataset composed of 6.6 billion distinct k-mers ($k = 28$) with average abundance per k-mer equal to 14.9, the RAM usage of Squeakr [29] is of 77.8 bit/k-mer, the RAM usage of KMC2 [8] (signature-based) is of 139 bit/k-mer, and the RAM usage of Jellyfish2 [25] (hash-based) is of 80 bit/k-mer.

Finally, in [33] an approximate data structure called Set-Min sketch has been proposed. This method takes advantage of the power-law distribution of the k-mer counts to reduce the error rate of the returned results. Set-Min sketch is implemented in the tool fress which achieves better memory consumption but does not guarantee exact counts. For a dataset of 6.6 billion distinct k-mers ($k = 28$) with average abundance per k-mer equal to 14.9, the declared RAM usage of fress is 16.9 bit/k-mer with an error rate of 0.01.

1.2 Our contribution

In this paper we propose a new representation for weighted de Bruijn graphs, which is based on the idea of delta-encoding the abundance variations on a spanning branching of the graph. To show the usefulness of our representation we test it on different real datasets obtaining each time a significant space improvement with respect to the state of the art. As an example, when combined with the compressed BOSS de Bruijn graph representation [3], our new data structure stores the weighted de Bruijn graph of a 16x-covered DNA read-set (60M distinct k-mers, $k = 28$) within 4.15 bits per distinct k-mer: just 1.4 bits per k-mer on top of the BOSS representation[1] [3].

Our data structure does not make any assumption on the distribution of the k-mer counts, which makes it appropriate to any dataset for both DNA and RNA. As a by-product of our data structure, we exhibit efficient compressed data structures for answering partial sums on edge-weighted trees, which might be of independent interest.

2 Notation

Logarithms are to the base 2. To simplify notation, we take $\log 1 = 1$. Notation $[n]$ denotes the set $\{1, 2, \ldots, n\}$. Throughout the paper, we work with the DNA alphabet $\Sigma = \{A, C, G, T\}$. A k-mer is a string from Σ^k. If $w \in \Sigma^*$ is a string, then $w[i, j]$ denotes substring $w[i] \cdot w[i+1] \cdots w[j]$, where the symbol "$\cdot$" represents the concatenation of characters.

De Bruijn graphs. A de Bruijn graph is a directed graph $G = (\mathcal{V}, \mathcal{E})$ whose nodes are in bijection with the k-mers appearing in the dataset. To distinguish between k-mers and the corresponding nodes of G, we use the notation $\hat{s}$ to denote the node of G corresponding to k-mer $s \in \Sigma^k$.

Let $s, s' \in \Sigma^k$ be two k-mers. We say that s *is adjacent to* s' if $s[2, k] = s'[1, k-1]$ and the $(k+1)$-mer $s \cdot s'[k]$ occurs in the dataset. The directed edges of G are in bijection with all pairs of adjacent k-mers: $(\hat{s}, \hat{s}')$ belongs to $\mathcal{E}$ if and only if s is adjacent to s'. A *weighted* de Bruijn graph associates an integer positive weight $c(\hat{s})$ to each of its nodes. Weight $c(\hat{s})$ (equivalently, $c(s)$ when we refer to the corresponding k-mer) is called the *abundance* of s and corresponds to the number of times that k-mer s appears in the dataset.

3 Compressed Weighted de Bruijn Graphs

To compress the weights (abundances) of a de Bruijn graph, one could exploit the following observation: since consecutive genomic positions generate adjacent k-mers, weights of adjacent nodes in the de Bruijn graph are likely to be very similar. Let $s, s' \in \Sigma^k$ be such that s is

[1] Note that we divide the total space by the number of *distinct* k*-mers* rather than the number of de Bruijn graph edges as done in [3]. More details on this in Section 4.

adjacent to s'. Recall that with $c(s)$ we denote the k-mer's abundance and we extend this notation to the nodes of G as $c(\hat{s}) = c(s)$. Let $(\hat{s}, \hat{s}')$ be the corresponding edge of G, and let $w(\hat{s}, \hat{s}') = c(\hat{s}) - c(\hat{s}')$ be the difference between the weights of $\hat{s}$ and $\hat{s}'$. Note that $w(\hat{s}, \hat{s}')$ might be negative.

A first idea could be to store (i) the compressed integer $w(\hat{s}, \hat{s}')$ on each edge, and (ii) the explicit value $c(\hat{u})$ whenever $\hat{u}$ has in-degree equal to 0. In this case, we say that $\hat{u}$ is a *sampled node*. At this point, one could retrieve $c(\hat{s})$ also for any non-sampled node $\hat{s}$ by summing $c(\hat{u})$ to the weights of the edges on a path from $\hat{s}$ to $\hat{u}$, where $\hat{u}$ is a sampled node. This solution has two main drawbacks: it is not time-efficient (in the worst case we need to visit the whole graph G to compute one single weight) and it stores one integer per edge, whereas the original graph contained one integer per node (the abundance).

In fact, all we need is a *spanning branching*[2] $T = (V, E)$ of G. The idea is to store (i) the compressed weight $w(\hat{s}, \hat{s}')$ only for the edges of T and (ii) the weights $c(\hat{r}_1), \ldots, c(\hat{r}_t)$ for the t roots $\hat{r}_1, \ldots, \hat{r}_t$ of T. To simplify the description, in the following we will assume that $c(\hat{r}_i) = 0$ for $1 \leq i \leq t$. This information is sufficient to reconstruct the weight of each node of T, but still leaves us with a few issues:

(1) We would like to design a fast data structure to compute the *partial sum* of the values $w(\hat{s}, \hat{s}')$ on the edges of an arbitrary node-to-root path in T;
(2) We have to decide how to encode each $w(\hat{s}, \hat{s}')$ on the edges of T. This may affect the computation of a branching that minimizes such cost;
(3) We do not know to which k-mer (node of G) each node of T corresponds to.

Issues (1) and (2) can be solved with a compressed data structure for answering *partial sums on trees*. In Section 3.1 we first discuss our new compressed solution to partial sums on the special case of arrays, which may be of independent interest. Then, in Section 3.2 we generalize this solution to partial sums on trees. The solution will introduce a cost in bits for encoding each weight $w(\hat{s}, \hat{s}')$. The branching $T = (V, E)$ will then be chosen so as to minimize the sum of the costs on its edges. Issue (3) requires us to keep a mapping between the nodes of G and T. In Section 3.3 we show how to solve this problem with a simplified (and practical) variant of the structure of Section 3.2. Finally, in Section 4 we present experimental results on real DNA datasets, comparing the space usage of our data structure with the state of the art.

3.1 Partial Sums on Arrays

Given a (static) sequence $s[1, n]$ of **positive** integers such that $u = \sum_{i=1}^{n} s[i]$, we would like to design a data structure to support partial sum operations, i.e., to answer queries $\mathsf{sum}(i) = \sum_{j=1}^{i} s[j]$, for any $i \in [n]$. Note that there are succinct data structures that are able to support sum queries in constant time [10, 11, 32] within the information-theoretic lower bound of $\lceil \log \binom{u}{n} \rceil = n \log \frac{u}{n} + O(n)$ bits. These classic results are summarized in the following theorem.

▶ **Theorem 1** ([10, 11, 32]). *Given a (static) sequence $s[1, n]$ of positive integers such that $u = \sum_{i=1}^{n} s[i]$, there exists a data structure that can answer any* sum *query in $O(1)$ time using $n \log(u/n) + O(n)$ bits of space.*

[2] In this paper *spanning branching* stands for *spanning forest of arborescences*, that is, a collection of disjoint directed trees spanning the de Bruijn graph. We note that a spanning forest of *undirected* trees would work as well; however, as we discuss in Section 3.3 this introduces some technical complications.

This space bound is always at most the space required by writing down all partial sums explicitly, i.e., $n\lceil \log u \rceil$ bits. However, it is possible to get a better space bound in terms of a well-known data-aware measure called *gap* measure [15], where the gap measure of the sequence s is defined as $\mathsf{gap}(s) = \sum_{i=1}^{n} \lceil \log(s[i]+1) \rceil$ bits. Notice that in general it is not possible to represent each $s(i)$ with $\lceil \log(s(i)+1) \rceil$ bits, and thus data structures must incur a space overhead on top of the gap measure. Note that the gap measure is always at most the information-theoretic lower bound of $\lceil \log \binom{u}{n} \rceil$ bits: this is because the gap measure is maximised when $s[i] = u/n$ for every i. However, in practice the gap measure could be much smaller than the information-theoretic lower bound.

Gupta et al. [15] designed a data structure that is able to answer each sum query in $O(\log \log u)$ time and uses $\mathsf{gap}(s) + O(n \log \log(u/n) + n \log(u/n)/\log n)$ bits of space. Delpratt et al. [7] defined the delta measure $\Delta(s) = \sum_{i=1}^{n} |\delta(s[i])|$, where $|\delta(x)|$, for $x \geq 1$, is the size in bits of encoding the number x with Elias' δ coding. They present a data structure that answers each sum query in $O(\log \log u)$ time using $\Delta(s) + o(n)$ bits of space. Since for any x we have $\delta(x) = \log x + 2 \log \log x + O(1)$ bits, $\Delta(s) \leq \mathsf{gap}(s) + 2n \log \log(u/n) + O(n)$ bits [7]. This improves over Gupta et al. due to the lower order term in the space bound.

In the remainder of this section we prove the following theorem. Even if our result on de Bruijn graphs requires partial sums on trees instead of arrays, we present this result on arrays here mainly for two reasons. First, it will be used to solve partial sums on trees in Section 3.1. Second, we believe it can be of independent interest since it provides different query-time/space trade-offs for partial sums on arrays.

▶ **Theorem 2.** *Given a (static) sequence $s[1,n]$ of positive integers such that $u = \sum_{i=1}^{n} s[i]$, there exists a data structure that answer sum queries in $O(\frac{1}{\epsilon})$ time using $(1+\epsilon)\mathsf{gap}(s) + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$.*

We first present a solution using standard ideas and tools. This solution will be then refined to obtain the claimed result. We start by concatenating in a vector D the Elias' δ encoding of each value of s. Elias' γ and δ codes [10] are two standard encodings for positive integers. Notice that binary encoding could also suffice however we believe that the proof is clearer using the Elias's one. Elias' $\gamma(x)$ of an integer $x > 0$ is obtained by writing $\lfloor \log x \rfloor$ in unary, followed by the value $x - 2^{\lfloor \log x \rfloor}$ written with $\lfloor \log x \rfloor$ bits. Elias' δ instead encodes x by writing $\lfloor \log x \rfloor + 1$ with Elias' γ, followed by the value $x - 2^{\lfloor \log x \rfloor}$ written with $\lfloor \log x \rfloor$ bits. We would like also to be able to compute the starting position in D of the encoding of any integers of s. This can be easily done by representing the sequence $L[1,n]$ with the data structure of Theorem 1. The entry $L[i]$ equals the length of the encoding of $s[i]$. This way, the starting position of the encoding of $s[i]$ in D is $\mathsf{sum}(i)$ in L. Note that D and L suffice to decompress any value of the original sequence s in constant time. The space required by these two vectors is bounded by $\mathsf{gap}(s) + O(n \log \log(u/n))$ bits. In order to support sum queries, we can partition s into blocks of size $b = \lceil \frac{1}{\epsilon} \rceil$ each. We store in an array $B = [1, n/b]$ the partial sums in s up to the beginning of each block. This way we can easily support any $\mathsf{sum}(i)$ in $O(\frac{1}{\epsilon})$ time. We first get from B the sum up to the block, say j, that contains the ith element of s. Then, we decompress the elements in the jth blocks one after the other up to the element $s[i]$. This clearly requires $O(1/\epsilon)$ time per operation. However, the space needed by B is $\epsilon n \log u$ bits. This gives an overall space usage of $\mathsf{gap}(s) + \epsilon n \log u + O(n \log \log(u/n) + n)$ bits, which is worse than what is claimed in Theorem 2, since the term $\epsilon n \log u$ may be potentially larger than the term $\epsilon \mathsf{gap}(s)$.

To improve the space bound, we use a different partitioning strategy. The goal is to partition s into variable size blocks such that the cost of encodings the values in each block (but the last one) is between $\frac{\log u}{\epsilon}$ bits and $\frac{3 \log u}{\epsilon}$ bits. This guarantees that there are at most $\frac{\epsilon \mathsf{gap}(s)}{\log u} + 1$ blocks and, thus, the cost of storing the vector B is at most $\epsilon \mathsf{gap}(s) + \log u$ bits.

We observe that a partition with the above characteristics is always possible. For example we can use the following greedy algorithm. We start with an empty block and we process the sequence from left to right. While processing an element we have two possibilities: we either include the element in the current block or we create a new block with this element. We take the former decision only if the overall cost of encoding the elements in the block is less than $\frac{\log u}{\epsilon}$ bits. As the code of an element is at most $\log u + O(\log \log u)$ bits, the above partitioning strategy gives blocks of size between $\frac{\log u}{\epsilon}$ bits and $\frac{3 \log u}{\epsilon}$ bits as claimed.

As now we have variable-size blocks, we need to store a binary vector V of size n bits to keep track of blocks boundaries: an entry $V[i]$ is 1 if and only if the ith element of s is the first element in a block. We use a data structure to support rank/select operations on V in constant time [32].

A query $\mathsf{sum}(i)$ is answered as follows. We first use rank/select operations on V to compute the block j of position i and its offset p within this block. We now need to decode the first p elements of the jth block and sum them to the value $B[j]$. Unfortunately, there may be $\Theta(\log u)$ elements encoded in a block and, thus, we cannot use a trivial decoding. Instead, we conceptually split each block into subblocks such that each subblock is either (i) formed of elements whose overall encoding sizes is no more than $\frac{1}{2} \log u$ bits, or (ii) a single element. This split into subblocks can be done by an easy variant of the greedy partitioning strategy above. Note that there are $O(\frac{1}{\epsilon})$ subblocks per block. We use a table of size $O(\sqrt{u} \log u)$ bits to precompute the sum of any prefix of any subblock given its encoding. This way, computing the sum of the first p elements in jth block costs $O(\frac{1}{\epsilon})$ time as required. Note that the splitting above can be changed to use $\frac{1}{c} \log u$ bits, for any constant $c \geq 1$ instead $\frac{1}{2} \log u$ bits. This way we need a table of size $O(2^{\frac{1}{c} \log u} \log u) = O(u^{\frac{1}{c}} \log u)$ bits and the query time remains $O(\frac{1}{\epsilon})$. By adjusting infinitesimally the constant c, the table takes $O(u^{\frac{1}{c}})$ bits of space.

We conclude by showing how to adapt any solution for the partial sums problem to the variant of the problem in which the sequence s has both positive and negative integers. Let us define $\mathsf{gap}^{\pm}(s) = \sum_{i=1}^{n} \lceil \log(|s[i]| + 1) \rceil$

This variant can be easily reduced to two instances of the original problem as follows. We first use a binary vector $S[1, n]$ that records the signs of the values in s, i.e., $S[i] = 1$ if and only if $s < 0$, $S[i] = 0$ otherwise. We use the data structure by Raman et al. [32] to support constant time rank/select operation on S using $\log \binom{n}{p} + o(n)$ bits of space, where p is the number of positive integers in s. Then, we create two sequences $s^+[1, p]$ and $s^-[1, n - p]$ that store positive and negative integers of s, respectively. Any partial sum query can now be answered with two partial sums queries on s^+ and s^-. Combining this reduction with Theorem 2 yields the following corollary.

▶ **Corollary 3.** *Given a (static) sequence $s[1, n]$ of positive and negative integers such that $u = \sum_{i=1}^{n} |s[i]|$, there exists a data structure that answer each* sum *operation in $O(\frac{1}{\epsilon})$ time using $(1 + \epsilon)\mathsf{gap}^{\pm}(s) + \log \binom{n}{p} + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$, where p is the number of positive integers.*

3.2 Partial Sums on Trees

Let $T = (V, E)$ be a rooted weighted tree and let $w(i, j)$ denote the (possibly negative) integer weight of edge $(i, j) \in E$. We would like to build a space-efficient data structure that answers partial sum queries of the form $\mathsf{sum}(v) = \sum_{(i,j) \in \Pi(v)} w(i, j)$ on paths, i.e., that reports the sum of the weights on the path $\Pi(v)$ connecting node v to the root, where v is represented in pre-order (the solution for post-order is symmetrical and we do not discuss it).

Chan et al. [5] tackle this problem in a more general framework, where weights belong to a semigroup of size γ. Let $n = |V|$ be the number of nodes. In this model, they provide a data structure taking $n \log \gamma + o(n \log \gamma) + 2n$ bits and answering queries in $O(\alpha(n))$ (inverse Ackermann) time. Note that this space is succinct but does not achieve compression: each weight $w(i, j)$ is stored in $\log \gamma$ bits, independently of its magnitude. Typically, one would like a data structure using much less space: ideally, close to $\mathsf{gap}^{\pm}(T) = \sum_{(i,j)\in E} \lceil \log(|w(i, j)|+1) \rceil$ bits of space.

A folklore approach can reduce partial sums on trees to partial sums on arrays. The idea consists of linearizing the weights according to a Euler tour of the tree. Recall that tree edges are directed towards the root and note that the Euler tour visits each tree edge (i, j) twice: the first time in the direction (j, i) and the second time in the direction (i, j). If $(i, j) \in E$, we assign to the reverse edge (j, i) weight $w(j, i) = -w(i, j)$. We initialize an empty sequence W and, for each edge (i, j) visited along the tour (that is, in the direction $i \to j$), append $-w(i, j)$ at the end of W. Then, it holds that $\mathsf{sum}(v) = \sum_{j=1}^{t} W[j]$, where t is the index corresponding to the first time we see node v in the Euler tour (i.e., $W[t]$ contains the weight $w(\pi(v), v)$, where $\pi(v)$ is the parent of v). This equality holds because, in $W[1], \ldots, W[t]$, all values that correspond to edges not belonging to the path from v to the root appear two times with opposite signs and thus they cancel out.

We can use the data structure of Corollary 3 on the sequence W to have $O(\frac{1}{\epsilon})$ query time. The space is $(2 + \epsilon)\mathsf{gap}^{\pm}(s) + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits because every gap occurs twice in W.

Another folklore approach uses the Heavy-Light decomposition of the dynamic trees of Sleator and Tarjan [34]. The idea is to split the tree into heavy paths and use a data structure for prefix sums on each of these heavy paths. By virtues of the decomposition, any root-to-a-node path crosses $O(\log n)$ heavy paths and, thus, a query can be answered with $O(\log n)$ prefix sums on arrays. This solution has a better space bound as every gap is represented exactly once, but its query time is logarithmic.

In the remainder of this section we prove the following theorem, which improves over the two folklore solutions above.

▶ **Theorem 4.** *Given a (static) rooted tree $T = (V, E)$ with positive integer weights $w(i, j)$ associated with each edge $(i, j) \in E$ such that $u = \sum_{(i,j)\in E} w(i, j)$, there exists a data structure that answers* sum *queries in $O(\frac{1}{\epsilon})$ time using $(1+\epsilon)\mathsf{gap}(T)+O(n \log \log(u/n)+u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$.*

The following corollary generalizes the solution to trees with positive and negative weights.

▶ **Corollary 5.** *Given a (static) rooted tree $T = (V, E)$ with positive and negative integer weights $w(i, j)$ associated with each edge $(i, j) \in E$ such that $u = \sum_{(i,j)\in E} |w(i, j)|$, there exists a data structure that solves queries* sum *in $O(\frac{1}{\epsilon})$ time using $(1 + \epsilon)\mathsf{gap}^{\pm}(T) + \log \binom{n}{p} + O(n \log \log(u/n) + u^{\frac{1}{c}})$ bits of space, for any $0 < \epsilon \leq 1$ and any constant $c \geq 1$, where p is the number of positive integers.*

These results are obtained by adapting to trees the solutions of Theorem 2 and Corollary 3, respectively.

The high level ideas behind our approach are as follows. We first partition the tree into subtrees. Subtrees are either disjoint or intersect only at their common root. Then, we store the sums on the paths from the tree's root to each subtree root. The sum $\mathsf{sum}(v)$ is computed by taking the sum up to the root of v's subtree and summing up the sum of the path within the subtree. The sum within each subtree is computed by using an approach similar to the one in the previous subsection.

The partitioning of the tree is crucial for the space and time efficiency of the solution. For this aim we use a weighted variant of the tree covering procedure described by Geary et al. [14, Sec. 2.1] to decompose T into subtrees. Given any parameter $M > 2$, the original tree covering procedure as described in [14, Sec. 2.1] is used to decompose T into $\Theta(n/M)$ sub-trees containing $O(M)$ nodes each. Two subtrees are either disjoint or intersect only at their common root. The covering is built by visiting the tree in post-order and by grouping nodes into components. The procedure lets the current component grow until its size falls in the range $[M, 3M-4]$. When a component reaches the required size, a new empty component is created. This greedy procedure guarantees that every component (except for the one containing the root) has between M and $3M-4$ nodes. See Geary et al. [14, Sec. 2.1] for more details.

In our solution we use a simple variant of this covering procedure. We fix M to be $\frac{\log u}{\epsilon}$ and we use the greedy procedure above with the only difference that every node i has an encoding cost which equals the size of encoding $w(\pi(i), i)$, where $\pi(i)$ is the parent of i. We build our components to have a cost bounded by M. This way, we can control the size of the encoding of each subtree, which is between $\frac{\log u}{\epsilon}$ bits and $\frac{3\log u}{\epsilon} + 2\log u$ bits. We now need to δ-encode the weights w of a subtree one after the other by following an ordering (specified later) which is suitable for solving queries efficiently. Our goal is, given a node v represented in pre-order, to compute the sum from tree's root to node v. This is solved by summing up: i) the sum from the tree's root to the root of the subtree containing v; ii) the sum of the weights on the path from the subtree's root to node v. Geary et al. [14, Sec. 4.3] show how to find the pre-order number of the root of the subtree containing v in constant time and $o(n)$ bits of space. This index can be used to access a vector containing the sum mentioned at point (i). Point (ii) is computed in $O(\frac{1}{\epsilon})$ time in a way similar to what we have done for arrays. We split the representation of a subtree into $O(\frac{1}{\epsilon})$ subblocks and use tables storing precomputed answers to all the possible subblocks representations. However, the solution here is more involved than the one we used for arrays because we need to compute the sum only of some of the elements of a subblock, i.e., those elements that belong to the query path, and exclude the other elements. This can be done by using a mask that marks the elements belonging to the query path. Given the representation of a subblock and a mask, a precomputed table returns the sum of the marked elements only. The main issue is now to compute the required mask. This can be done by splitting the subtree into subsubtrees whose encoding takes between $\frac{1}{12}\log u$ bits and $\frac{1}{4}\log u$ bits. This is done by using once again the covering procedure described above. We δ-encode the elements in these subsubtrees by visiting nodes in BFS order. We also need to store the topology of each subsubtree. To do that, we can use a balanced sequence representation of each subsubtree so that we need $2n + o(n)$ bits overall. Observe that each subsubtree has at most $\frac{1}{4}\log u$ nodes. Thus, its balanced parentheses representation always takes at most $\frac{1}{2}\log u$ bits and we can use a table of size $O(u^{\frac{1}{2}}\log^2 u)$ bits that, given a subsubtree topology ($\frac{1}{2}\log u$ bits) and a node represented as a pre-order position in the subsubtree topology ($O(\log\log u)$ bits), returns the required mask, i.e., a mask of $\frac{1}{4}\log u$ bits that marks only nodes on the path from the root to the node. Another table of size $O(u^{\frac{1}{2}}\log u)$ bits is used to, given any possible combination of the delta-encoded weights of a subsubtree (at most $\frac{1}{4}\log u$ bits) and any possible mask (at most $\frac{1}{4}\log u$ bits), return the sum of the marked elements only. Note that to perform the above operations we need, given a pre-order node v, to obtain (a) the packed balanced parentheses sequence representation of the subsubtree containing v and (b) the pre-order position of v within its subsubtree. Information (a) can be obtained as done above by using the procedure described by Geary et al. [14, Sec. 4.3]: we locate the pre-order number of the root of the subsubtree and use it as index in an array containing the packed balanced

sequence representations of the subsubtrees ($O(n)$ bits of space). In the same section, Geary et al. [14, Sec. 4.3] show also how to obtain information (b) in constant time and $o(n)$ bits of additional space. Finally note that, as done in the previous section, we can replace the size $\frac{1}{4}\log u$ of the subsubtrees by $\frac{1}{c}\log u$ for any constant $c \geq 1$ and obtain the claimed space bound.

3.3 Adding the dBg Topology

In this section we discuss how to combine a simplified (and practical) variant of the data structure of Corollary 5 with the BOSS de Bruijn graph representation of Bowe et al. [3]. Our final data structure represents a weighted de Bruijn graph in compressed space and supports computing the abundance of any given input k-mer. In Section 4 we present experimental results based on our data structure. Our implementation is available at `https://github.com/nicolaprezza/cw-dBg/`.

Let n and m be the number of nodes and edges of the input de Bruijn graph G, respectively. First of all, we compute a branching T of G minimizing measure $\mathtt{gap}^{\pm}(T)$. This can be achieved using Gabow et al.'s optimized version [13] of Edmonds' algorithm [9], running in $O(m + n\log n)$ time.

The second step is to observe that, by definition of branching, its topology is embedded in the topology of the de Bruijn graph. Consider the list $\hat{s}_1, \ldots, \hat{s}_n$ of the n nodes of the de Bruijn graph sorted by the co-lexicographic order of their corresponding k-mers. Note that this is precisely the order in which nodes are stored in the BOSS representation [3]. Let $e_i^1, \ldots, e_i^{t_i}$ be the t_i incoming edges of node $\hat{s}_i$, and consider the list $e_1, \ldots, e_m = \langle e_i^1, \ldots, e_i^{t_i}\rangle_{i=1,\ldots,n}$ of all the m edges in the graph, sorted by their target node. We keep one bitvector $\mathtt{IN\_B}[1, m]$ marking the edges, in the above order, that are included in the branching. It is clear that the topology of the de Bruijn graph, combined with bitvector `IN_B`, fully specifies the topology of the branching. Importantly, we observe that de Bruijn graphs are usually very sparse, i.e., $m \approx n$. This implies that `IN_B` is composed mostly of bits equal to 1, thus its zero-order entropy $\lceil \log \binom{m}{n} \rceil$ is likely to be very small. We thus store `IN_B` with the zero-order compressed representation of Raman et al. [32], supporting constant-time rank and select queries.

The third step is to build a simplified version of the structure of Corollary 5 over each arborescence of the branching. We do this as follows. Given a parameter ρ (the *sample rate*), $1 \leq \rho \leq n$, we use the tree covering procedure of Geary et al. [14, Sec. 2.1] to decompose each arborescence into $\Theta(n'/\rho)$ subtrees of $O(\rho)$ nodes each, where n' is the number of nodes of the arborescence. We furthermore explicitly store the abundances of the subtree's roots in a vector, sorting them by the order in which nodes appear in the BOSS data structure (that is, by the co-lexicographic order of their corresponding k-mers). The sampled abundances take $O((\frac{n}{\rho} + t)\log u)$ bits, where t is the number of arborescences. An additional zero-order compressed bitvector marks sampled nodes. Using the representation of Raman et al. [32], this bitvector takes $O((\frac{n}{\rho} + t)\log \rho) + o(n)$ bits and supports access, rank, and select operations in constant time.

We store the weight of each edge of T (that is, the difference between the abundances of its endpoint k-mers) using a version of Elias' gamma encoding supporting fast random access queries. First, we convert each (possibly negative) weight w into a positive integer $w' \geq 1$ using the formula

$$w' = \begin{cases} -2w, & \text{if } w < 0 \\ 2w + 1, & \text{otherwise} \end{cases}$$

In practice, using the above conversion, rather than storing explicitly the sign of w in a separate bitvector, has an important practical advantage, as the minimum branching algorithm compresses also the sign of the weights. Indeed, we observed experimentally that this choice compresses each sign to about 0.3 bits on average (rather than 1 bit per sign required when storing them in a plain bitvector). The random access gamma-compressed weights are implemented as follows. We concatenate the binary representation of each w' (devoid of its most significant bit) in an uncompressed bitvector X, and its length in unary in a bitvector Y (for example: $w' = 27$ would be encoded as 1011 in X and 10000 in Y). We compress Y using the representation of Raman et al. [32]. The i-th weight can be extracted from this representation in $O(1)$ time with one select operation on Y and one packed access operation on X. Since we compress Y, it is not hard to see that our representation takes at most $\mathsf{gap}^{\pm}(T) + O(n \log\log(u/n))$ bits of space. Finally we mention that, in order to achieve further compression, we used the compressed bitvector of Raman et al. [32] also to implement the components of the BOSS representation [3].

Let s be a k-mer. To retrieve its abundance, we do the following:

1. We use the de Bruijn graph representation to retrieve the node $\hat{s}$ (in the BOSS representation, a position in the Burrows-Wheeler transform of the graph) corresponding to s. This step takes $O(k)$ time via the backward search algorithm (see [3] for full details) since we assume constant-sized DNA alphabet $\Sigma = \{A, C, G, T\}$.
2. Starting from $\hat{s}$, we move upward towards the root of the tree containing $\hat{s}$ in the branching, stopping as soon as a sampled node $\hat{s}'$ is found (that is, a node whose abundance has been stored explicitly). Each move in the tree is implemented with a constant-time application of the FL function [3]. Along the walk, we sum the abundance of the sampled node $\hat{s}'$ to the weights of the edges on the path connecting $\hat{s}'$ with $\hat{s}$. Overall, this step takes $O(\rho)$ time.

Observe that Step 1 can be avoided if we are already given the representation of a node in the de Bruijn graph (for example, if we are navigating it). Now we can also explain why we use a branching instead of a minimum spanning undirected forest. With a branching, the parent of node $\hat{s}$ (in its corresponding arborescence) is always one of its incoming edges (precisely, the one marked in bitvetor `IN_B`). With an undirected spanning forest, instead, the parent of node $\hat{s}$ could be one of its outgoing edges; this would force us to use an additional bitvector, increasing the overall space and slowing down operations.

To sum up, our implementation offers the following trade-offs. Let G be a de Bruijn graph with m edges and n nodes, $w(\hat{i}, \hat{j}) = c(\hat{i}) - c(\hat{j})$ be the weight associated with each edge $(\hat{i}, \hat{j})$ of G, where $c(\hat{i})$ is the abundance of node $\hat{i}$, $T = (V, E)$ be a branching of G with t connected components minimizing measure $\mathsf{gap}^{\pm}(T) = \sum_{(\hat{i},\hat{j})\in E} \lceil \log(|w(\hat{i}, \hat{j})| + 1) \rceil$, $u = \sum_{(\hat{i},\hat{j})\in E} |w(\hat{i}, \hat{j})|$, and $1 \leq \rho \leq n$ be the user-defined sample rate. Our data structure uses $\mathsf{gap}^{\pm}(T) + O\left(n \log\log(u/n) + \left(\frac{n}{\rho} + t\right) \log u\right) + \log \binom{m}{n}$ bits on top of the compressed BOSS representation [3] and allows us to retrieve:

(1) the abundance of a given k-mer $s \in \Sigma^k$ in $O(k + \rho)$ time, and
(2) the abundance of a given node in the de Bruijn graph (represented as a position in BOSS) in $O(\rho)$ time.

Note that, in practice, ρ is usually chosen to be $\rho \gg k$ (in our experiments, $\rho = 64$ and $k = 28$) so the two query times are not expected to differ much in practical applications.

Our experiments (see next section) highlight that, in practice, the number t of arborescences is negligible compared to the size of the graph.

4 Experiments

In order to get a feeling of its practical value, we implemented a first preliminary version of our compressed representation of weighted de Bruijn graphs. In the following, we refer to this implementatio as cw-dBg; its code is available at `https://github.com/nicolaprezza/cw-dBg`.

Tools. We ran cw-dBg on four datasets and compared the results obtained against the state-of-the-art algorithms for this problem: Squeakr [29], deBGR [27], and the very recent tool fress [33] which appeared online in November 2020. Both Squeakr and deBGR are based on the so-called *counting quotient filter (CQF)* data structure [28]. Squeakr supports two modes: approximate and exact. When run in exact mode, the k-mers are inserted in the CQF using an invertible $2k$-bit hash function. In approximate mode, Squeakr uses a p-bit hash function, with $p \leq 2k$, and thus can have false positives with an error rate depending on the chosen p. deBGR is based on Squeakr and builds an approximate data structure with smaller error rate by increasing the space complexity of approximate Squeakr by just $18\% - 20\%$. Finally, fress [33] implements an approximate data structure called *Set-Min sketch*, inspired by the *Count-Min sketch* data structure [6], that takes advantage of the power-law distribution of the k-mer counts to reduce both the error rate of the returned results and the space usage (by an entropy-compression mechanism). Similarly to Count-Min sketch, Set-Min sketch uses several hash functions to map a given k-mer to sketches of its abundance; at query time, the abundance of the k-mer is computed by combining the retrieved sketches. As shown by the authors, Set-Min sketch is more space-efficient than minimal perfect hash functions and provides better error guarantees compared to equally-dimensioned Count-Min sketches.

Datasets. We selected datasets to cover applications that could be as widely different as possible. In particular, we tested the above tools against the following datasets:

- IAVs Inf. 1 ($1.2Gbp$) and IAVs Inf. 2 ($578Mbp$) are samples from the dataset presented in [1] of human lung cells infected by Influenza A viruses (IAVs) and correspond to a sample of 10 millions reads and a sample of $4,814,148$ reads from chromosome 2, respectively.
- E. coli ($2.3Gbp$) consists of $22,720,100$ Illumina reads of *E. Coli* K-12 strain MG1655 (available in the ENA repository under the following study: PRJEB2323, `https://www.ebi.ac.uk/ena/browser/view/ERR022075`).
- Human ($6.5Gbp$) consists of $63,917,134$ Illumina reads of human RNA (available in the ENA repository under the following study: PRJNA609878, `https://www.ebi.ac.uk/ena/browser/view/SRX7829390`).

Experimental settings. The experiments were carried out on a single core of an 8-core Intel i9-9900K server with 64 GB of RAM and running Linux 5.4.0, 64 bits. Our code is written in C++ and compiled with `gcc` 9.3.0.

As deBGR assumes k-mers of size 28 (which cannot be changed), for consistency we also used this value in all our experiments for all tools. Table 1 reports some characteristics of the datasets. Notice that, as it was previously mentioned, the number of arborescences is very small, when compared to the size of the graph, and thus negligible in practice. We used sample rate $\rho = 64$ for cw-dBg.

deBGR and Squeakr (the latter both in its approximate and exact version) were run using only 1 thread. When running deBGR, the user needs to specify two parameters related to the log of the number of slots in the counting quotient filter. The *CQF size* argument is estimated

Table 1 Characteristics of the de Bruijn graph resulted from each of the dataset for k-mers of size 28. The columns correspond to: number of bases and number of distinct k-mers in the dataset, number of edges and number of arborescences in the de Bruijn graph, average and max abundance of the k-mers and the total number of distinct abundances in the dataset (which influences fress' space).

Dataset	# bases	# k-mers	# edges	# Arb.	Avg ab.	Max ab.	# ab.
IAVs Inf. 1	1,200,000,000	57,629,309	65,880,010	259	16.14	607,028	13,452
IAVs Inf. 2	577,697,760	34,914,869	41,295,094	100	12.82	55,110	9,007
E. coli	2,317,450,200	87,414,957	281,845,871	1	19.49	8,520,260	2,657
Human	6,455,630,534	362,818,017	463,413,958	1226	13.04	11,054,076	21,472

using the lognumslots.sh script provided by the authors and corresponds approximately to the log of the number of k-mers. The *exact CQF size* argument is estimated as 20% less then the *CQF size* (personal communication with the authors of deBGR).

Finally, fress is an approximate tool and allows to specify a parameter related to the error rate: in our experiments, we ran it using an error rate of $\epsilon = 0.01$, as used in fress' paper [33]. Notice that the error rate estimates the expected number of collision which could possibly lead to output a wrong abundance. An error rate of 0.01 means that in N queries, the number of expected collisions in the hash table is ϵN.

Experimental results. In Table 2 we present the detailed space required by our representation for each of the datasets considered, while Table 3 reports the space usage of all the tools considered.

The bit/k-mer of each component in Tables 2 and 3 were computed using the number of distinct k-mers over the alphabet $\{A, C, G, T\}$ present in the original dataset. We note that this is not the same calculation performed by Bowe et al. [3] when estimating the size of their BOSS representation: in that case, the authors divide by the number of *edges* in the de Bruijn graph when also including dummy (that is, left-padded) k-mers, obtaining roughly 4 bits per edge. In fact, the two values (number of edges and number of k-mers) are close to each other only when the number of dummy k-mers is small (which is not always the case, as explained below). We believe that dividing the total bit-size by the number of distinct k-mers appearing in the dataset is more general, since it applies to any k-mer counting data structure like the ones shown in Table 3. Notice that for this reason in Table 2 the space reported for the BOSS representation of the E. coli dataset is larger than the typical 4 bit/edge reported by the authors of BOSS [3] as the number of dummy k-mers for this dataset is significantly large (approximately equal to the number of distinct k-mers).

The results in Table 2 and Table 3 show that our representation uses at most 4.75 bit/k-mer (and as little as 1.42 bit/k-mer) to encode the abundances. Even when adding the BOSS representation (required to map k-mers to their corresponding abundances), we obtain at most 12.64 bit/k-mer (and as little as 4.14 bit/k-mer) while the exact version of Squeakr uses 83.26 bit/k-mer in the best scenario. In almost all our experiments cw-dBg required substantially much less space, an by several orders of magnitude, than all the other tools. The only exception occured in the E. coli dataset, where fress performed slightly better than cw-dBg by using 10.95 bit/k-mer (versus 12.64 bit/k-mer of cw-dBg).

We stress out, however, that fress achieves this slight space saving at the cost of returning a wrong abundance 1% of the times in the worst case. Indeed, if its error rate is lowered from 0.01 to 0.005, fress uses 14.6 bit/k-mer on this dataset. The efficiency of fress on this dataset

Table 2 The space required for each dataset by our representation of the weighted de Bruijn graph. The columns correspond to: the size of the file containing the final representation, the space required by the BOSS representation of the de Bruijn graph; the space required by the compressed abundances divided in the space required by the delta-compressed weights (stored on the edges of the branching) and the one for the branching topology and the sampled abundances on the root of each subtree; the total space required by our representation and the average query time in microseconds for retrieving the abundance of a given k-mer (represented as a string). Space is measured in bits per distinct k-mer (see Table 1) and we used sample rate $\rho = 64$.

Dataset	Final size (MB)	BOSS (bit/k-mer)	Compressed ab. (bit/k-mer)			Total size (bit/k-mer)	Query time (μs/query)
			weights	branching	total		
IAVs Inf. 1	29	2.73	0.95	0.48	1.43	4.15	47.56
IAVs Inf. 2	20	2.87	1.19	0.53	1.72	4.59	44.24
E. coli	132	7.88	2.93	1.82	4.75	12.64	62.34
Human	223	3.14	1.40	0.61	2.01	5.14	67.42

Table 3 For each dataset we report the space required for the representation of the weighted de Bruijn graph by each of the considered tools.

	IAVs Inf. 1		IAVs Inf. 2		E. coli		Human	
	MB	bit/k-mer	MB	bit/k-mer	MB	bit/k-mer	MB	bit/k-mer
cw-dBg	29	4.15	20	4.59	132	12.64	223	5.14
Squeakr approx.	325	47.19	179	42.80	325	31.11	1126	24.05
Squeakr exact	965	140.40	499	119.75	965	92.57	3686	83.26
deBGR	912	132.46	376	89.55	912	87.32	5529	119.00
fress	733	106.56	135	32.35	115	10.95	733	16.90

is explained by the fact that the number of distinct k-mer abundances is much smaller than in the other three datasets (see Table 1). This is clearly the case where we expect fress to work well, as it fully takes advantage of the power-law distribution of the abundances by entropy-compressing their values.

Finally, we estimated the average query time of cw-dBg, by querying a set of $5,000$ randomly chosen 28-mers. The largest measured query time for cw-dBg on our machine was 67.42 μs/query on the Human dataset (see Table 2 for the query times on all datasets). As expected, hash-based data structures have better query time: on the same Human dataset, Squeakr answers queries in average 0.19 μs/query using 1 thread. However, as we discussed before, hash-based structures have a much higher space usage, in addition to being approximate. The exact variant of Squeakr uses orders of magnitude more space than cw-dBg.

5 Conclusions and open problems

We propose a new compressed representation for weighted de Bruijn graphs based on the idea of delta-encoding the variations of k-mer abundances on a spanning branching of the graph. As a by-product of independent interest, we exhibit efficient compressed data structures for answering partial sums on edge-weighted trees.

We show both theoretically and experimentally that our approach uses significantly less memory than the one used by the state-of-the-art exact representations. Our de Bruijn graph representation is general, in other words it is not restricted by the application (e.g., variation finding or RNA-seq), and can be used as part of any algorithm that represents NGS data

with de Bruijn graphs. Future extensions will include implementing a strategy similar to the one used by fress in order to take advantage of the small number of distinct abundance observed in practice. The basic idea is to use a table storing all the distinct abundances sorted and, for each node, encode its abundance's offset in the table with our data structure (that is, storing the deltas of these offsets rather than of the original abundances). We also plan to integrate our structure in a usable bioinformatics tool.

References

1 Usama Ashraf, Clara Benoit-Pilven, Vincent Navratil, Cécile Ligneau, Guillaume Fournier, Sandie Munier, Odile Sismeiro, Jean-Yves Coppée, Vincent Lacroix, and Nadia Naffakh. Influenza virus infection induces widespread alterations of host cell splicing. *NAR Genomics and Bioinformatics*, 2(4), November 2020.

2 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, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. Spades: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012. PMID: 22506599. doi:10.1089/cmb.2012.0021.

3 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn Graphs. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, pages 225–235, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

4 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

5 Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct indices for path minimum, with applications. *Algorithmica*, 78(2):453–491, 2017. doi:10.1007/s00453-016-0170-7.

6 Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005*, pages 44–55, 2005. doi:10.1137/1.9781611972757.5.

7 O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Compressed prefix sums. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 4362 of *Lecture Notes in Computer Science*, pages 235–247. Springer, 2007. doi:10.1007/978-3-540-69507-3_19.

8 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, January 2015. doi:10.1093/bioinformatics/btv022.

9 Jack Edmonds. Optimum branchings. *Journal of Research of the national Bureau of Standards Section B*, 71(4):233–240, 1967.

10 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. doi:10.1145/321812.321820.

11 Robert Mario Fano. On the number of bits required to implement an associative memory. memorandum 61. *Computer Structures Group, Project MAC, MIT, Cambridge, Mass., nd*, 1971.

12 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

13 Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

14 Richard F Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms (TALG)*, 2(4):510–534, 2006.

15 Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007. doi:10.1016/j.tcs.2007.07.042.

16 David L. Hyten, Steven B. Cannon, Qijian Song, Nathan Weeks, Edward W. Fickus, Randy C. Shoemaker, James E. Specht, Andrew D. Farmer, Gregory D. May, and Perry B. Cregan. High-throughput snp discovery through deep resequencing of a reduced representation library to anchor and orient scaffolds in the soybean whole genome sequence. *BMC Genomics*, 11(1):38, 2010. doi:10.1186/1471-2164-11-38.

17 Katerina Kechris, Yee Hwa Yang, and Ru-Fang Yeh. Prediction of alternatively skipped exons and splicing enhancers from exon junction arrays. *BMC Genomics*, 9(1):551, 2008. doi:10.1186/1471-2164-9-551.

18 Ruiqiang Li, Yingrui Li, Xiaodong Fang, Huanming Yang, Jian Wang, Karsten Kristiansen, and Jun Wang. Snp detection for massively parallel whole-genome resequencing. *Genome Research*, 19(6):1124–1132, 2009. doi:10.1101/gr.088013.108.

19 Li Fan, Pei Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000. doi:10.1109/90.851975.

20 Leandro Lima, Blerina Sinaimeri, Gustavo Sacomoto, Hélène Lopez-Maestre, Camille Marchet, Vincent Miele, Marie-France Sagot, and Vincent Lacroix. Playing hide and seek with repeats in local and global de novo transcriptome assembly of short RNA-seq reads. *Algorithms Mol Biol*, 12, 2017.

21 Binghang Liu, Yujian Shi, Jianying Yuan, Xuesong Hu, Hao Zhang, Nan Li, Zhenyu Li, Yanxiang Chen, Desheng Mu, and Wei Fan. Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects. *arXiv preprint*, 2013. arXiv:1308.2012.

22 Yongchao Liu, Jan Schröder, and Bertil Schmidt. Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data. *Bioinformatics*, 29(3):308–315, November 2012. doi:10.1093/bioinformatics/bts690.

23 Hélène Lopez-Maestre, Lilia Brinza, Camille Marchet, Janice Kielbassa, Sylvère Bastien, Mathilde Boutigny, David Monnin, Adil El Filali, Claudia Marcia Carareto, Cristina Vieira, Franck Picard, Natacha Kremer, Fabrice Vavre, Marie-France Sagot, and Vincent Lacroix. SNP calling from RNA-seq data without a reference genome: identification, quantification, differential analysis and impact on the protein sequence. *Nucleic Acids Research*, 44(19):e148–e148, 2016. doi:10.1093/nar/gkw655.

24 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 data sets. *Genome Research*, 31(1):1–12, 2021.

25 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, January 2011. doi:10.1093/bioinformatics/btr011.

26 Ali Mortazavi, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature Methods*, 5(7):621–628, 2008. doi:10.1038/nmeth.1226.

27 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, 33(14):i133–i141, July 2017. doi:10.1093/bioinformatics/btx261.

28 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 775–787, New York, NY, USA, 2017. Association for Computing Machinery.

29 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, October 2017. doi:10.1093/bioinformatics/btx636.

30 Rob Patro, Geet Duggal, Michael I. Love, Rafael A. Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature methods*, 14(4):417–419, 2017. URL: https://pubmed.ncbi.nlm.nih.gov/28263959.

31 Pavel A. Pevzner. 1-tuple dna sequencing: Computer analysis. *Journal of Biomolecular Structure and Dynamics*, 7(1):63–73, 1989. PMID: 2684223. doi:10.1080/07391102.1989.10507752.

32 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

33 Yoshihiro Shibuya and Gregory Kucherov. Set-min sketch: a probabilistic map for power-law distributions with application to k-mer annotation. *SeqBIM 2020*, 2020. doi:10.1101/2020.11.14.382713.

34 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

35 Raluca Uricaru, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. Reference-free detection of isolated SNPs. *Nucleic Acids Research*, 43(2):e11–e11, November 2014. doi:10.1093/nar/gku1187.

36 Reda Younsi and Dan MacLean. Using $2k + 2$ bubble searches to find single nucleotide polymorphisms in k-mer graphs. *Bioinformatics*, 31(5):642–646, October 2014. doi:10.1093/bioinformatics/btu706.

37 Birney E. Zerbino DR. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18(5):821–9., 2008. PMID: 18349386. doi:doi:10.1101/gr.074492.107.