Bifrost – Highly parallel construction and indexing of colored and compacted de Bruijn graphs

Guillaume Holley[1]*, Páll Melsted[1]
[1]Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland, Reykjavík, Iceland

August 13, 2019

Abstract

Motivation: De Bruijn graphs are the core data structure for a wide range of assemblers and genome analysis software processing High Throughput Sequencing datasets. For population genomic analysis, the colored de Bruijn graph is often used in order to take advantage of the massive sets of sequenced genomes available for each species. However, memory consumption of tools based on the de Bruijn graph is often prohibitive, due to the high number of vertices, edges or colors in the graph. In order to process large and complex genomes, most short-read assemblers based on the de Bruijn graph paradigm reduce the assembly complexity and memory usage by compacting first all maximal non-branching paths of the graph into single vertices. Yet, de Bruijn graph compaction is challenging as it requires the uncompacted de Bruijn graph to be available in memory.

Results: We present a new parallel and memory efficient algorithm enabling the direct construction of the compacted de Bruijn graph without producing the intermediate uncompacted de Bruijn graph. Bifrost features a broad range of functions such as sequence querying, storage of user data alongside vertices and graph editing that automatically preserve the compaction property. Bifrost makes full use of the dynamic index efficiency and proposes a graph coloring method efficiently mapping each k-mer of the graph to the set of genomes in which it occurs. Experimental results show that our algorithm is competitive with state-of-the-art de Bruijn graph compaction and coloring tools. Bifrost was able to build the colored and compacted de Bruijn graph of about 118,000 Salmonella genomes on a mid-class server in about 4 days using 103 GB of main memory.

Availability: https://github.com/pmelsted/bifrost available with a BSD-2 license
Contact: guillaumeholley@gmail.com

*To whom correspondence should be addressed

1 Introduction

The de Bruijn graph is an abstract data structure with a rich history in computational biology as a tool for genome assembly (Pevzner *et al.*, 2001; Idury and Waterman, 1995). With the advent of High Throughput Sequencing (HTS), the Overlap Layout Consensus (OLC) framework frequently used to assemble Sanger sequencing data (Yang *et al.*, 2011) was progressively replaced in favor of de Bruijn graph based methods. Since 2008, a wide range of genome assemblers based on the de Bruijn graph have been released (Chaisson and Pevzner, 2008; Zerbino and Birney, 2008; Simpson *et al.*, 2009; Luo *et al.*, 2012; Chikhi and Rizk, 2013; Bankevich *et al.*, 2012; MacCallum *et al.*, 2009). Although SMS (Single Molecule Sequencing) technologies (Rang *et al.*, 2018; Rhoads and Au, 2015) have re-introduced the OLC framework as the method of choice to assemble long and erroneous reads (Koren *et al.*, 2017; Li, 2016; Chin *et al.*, 2016; Kamath *et al.*, 2017), de Bruijn graph based methods are nonetheless used to assemble and correct long reads (Salmela and Rivals, 2014; Ruan and Li, 2019). Overall, the de Bruijn graphs have found widespread use for a variety of problems such as de novo transcriptome assembly (Robertson *et al.*, 2010), variant calling (Uricaru *et al.*, 2015), short read compression (Benoit *et al.*, 2015), short read correction (Limasset *et al.*, 2019), long read correction (Salmela and Rivals, 2014) and short read mapping (Liu *et al.*, 2016) to name a few. The colored de Bruijn graph is a variant of the de Bruijn graph which keeps track of the source of each vertex in the graph (Iqbal *et al.*, 2012). The initial application was for assembly and genotyping but it has also found use in pan-genomics (Zekic *et al.*, 2018), variant calling (Fang *et al.*, 2016) and transcript quantification methods (Bray *et al.*, 2016).

Despite serving as a building block for many methods in computational biology, the de Bruijn graph adoption is hindered by two factors. First, the memory usage and computational requirements for building de Bruijn graphs from raw sequencing reads are considerable compared to alignment to a reference genome while only a handful of tools have focused on de Bruijn graph compaction (Minkin *et al.*, 2016; Chikhi *et al.*, 2016; Marcus *et al.*, 2014; Baier *et al.*, 2016; Minkin *et al.*, 2013). Second, de Bruijn graph construction usually requires tight integration with the code. In the best case, software libraries for building and manipulating de Bruijn graphs are used (Drezen *et al.*, 2014; Crusoe *et al.*, 2015) but in most cases, data structures to index the de Bruijn graph are re-implemented. Those downsides are intensified in the colored de Bruijn graph for which the memory consumption of colors rapidly overtakes the vertices and edges memory usage (Almodaresi *et al.*, 2017). For this reason, a lot of attention has been given to succinct data structures for building the colored de Bruijn graph (Marcus *et al.*, 2014; Holt and McMillan, 2014; Holley *et al.*, 2015; Baier *et al.*, 2016; Muggli *et al.*, 2017; Almodaresi *et al.*, 2017, 2018; Muggli *et al.*, 2019) and data structures for multi-set k-mer indexing (Solomon and Kingsford, 2016; Sun *et al.*, 2018; Solomon and Kingsford, 2018; Pandey *et al.*, 2018; Yu *et al.*, 2018; Bradley *et al.*, 2019).

In this paper, we present Bifrost, a software for efficiently constructing the colored and compacted de Bruijn graph, both in terms of runtime and memory usage. The data structures and algorithms implemented in Bifrost are specifically tailored for fast and lightweight construction, querying and dynamic manipulation of compacted de Bruijn graphs, both regular and colored. The software is designed to take advantage of multiple cores and modern processors instruction sets (SIMD operations). Bifrost is also available as a C++11 software library with minimal external dependencies and allows developers to build on top of an efficient de Bruijn graph engine by using the Bifrost API.

2 Definitions

A string s is a sequence of symbols drawn from an alphabet $\mathcal{A}$. The length of s is denoted by $|s|$. A substring of s is a string occurring in s: it has a starting position i, a length l and is denoted by $s(i, l)$. A substring of length l is also denoted an l-mer. In the following, we assume $\mathcal{A}$ is the DNA alphabet $\mathcal{A} = \{A, C, G, T\}$ for which symbols have complements: (A, T) and (C, G) are the complementing pairs. The reverse-complemented string $\overline{s}$ is the reverse sequence of complemented symbols in s. The canonical string $\hat{s}$ is the lexicographically smallest of s and its reverse-complement $\overline{s}$. The minimizer (Roberts *et al.*, 2004; Grabowski *et al.*, 2015) of an l-mer x is a g-mer y occuring in x such that $g < l$ and y is the lexicographically smallest of all the g-mers in x. The lexicographical order can be cumbersome to use since poly-A g-mers naturally occurs in sequencing data and is often replaced by a random order. The simplest way to obtain a random order is to compute a hash-value for each g-mer in x and select the g-mer with the smallest hash-value as the minimizer. In this work, we will only consider minimizers generated by random orderings.

A de Bruijn graph (dBG) is a directed graph $G = (V, E)$ in which each vertex $v \in V$ represents a k-mer. A directed edge $e \in E$ from vertex v to vertex v' representing k-mers x and x', respectively, exists if and only if $x(2, k-1) = x'(1, k-1)$. Each k-mer x has $|\mathcal{A}|$ possible successors $x(2, k-1) \odot a$ and $|\mathcal{A}|$ possible predecessors $a \odot x(1, k-1)$ in G with $a \in \mathcal{A}$ and $\odot$ as the concatenation operator. Note that in the original combinatorial definition of the dBG, all possible k-mers for an alphabet $\mathcal{A}$ are present in the graph, whereas in computational biology, the definition is restricted to a subset of the de Bruijn graph representing the k-mers in the input. A path in the graph is a sequence of distinct and connected vertices $p = (v_1, ..., v_m)$. We say that the path p is *non-branching* if all its vertices have an in- and out-degree of one with exception of the head vertex v_1 which can have more than one incoming edge and the tail vertex v_m which can have more than one outgoing edge. A non-branching path is maximal if he cannot be extended in the graph without being branching. A compacted de Bruijn graph (cdBG) merges all maximal non-branching paths of η vertices from the dBG into single vertices, called unitigs, representing words of length $k + \eta - 1$. Minimal examples of dBG and cdBG are provided in Figures 1a and 1b respectively. A colored de Bruijn graph is a graph $G = (V, E, C)$ in which (V, E) is a dBG and C is a set of colors such that each vertex $v \in V$ maps to a subset of C, we extend the definition of a cdBG to a colored compacted de Bruijn Graph (ccdBG) to be a graph $G = (V, E, C)$, where (V, E) is a cDBG, so the vertices represent unitigs, and each k-mer of a unitig maps to a subset of C.

Introduced by Bloom (1970), the Bloom filter (BF) is a space and time efficient data structure that records the approximate membership of elements in a set. The BF is represented as a bitmap B of m bits initialized with 0s, coupled with a set of f hash functions $h_1, ..., h_f$. Inserting and querying an element e into B is performed with the functions

$$\mathsf{Insert}(e, B) : B[h_i(e)] \leftarrow 1 \text{ for all } i = 1, ..., f$$

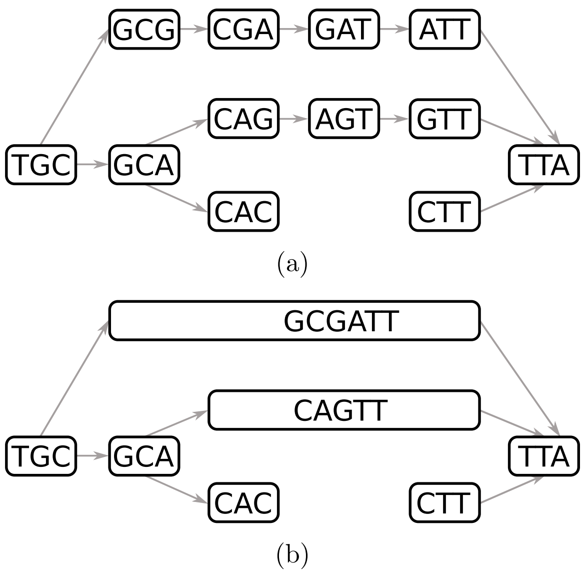

Figure 1: A de Bruijn graph in (a) and its compacted counterpart in (b) using 3-mers. For simplicity, reverse-complements are not considered.

and

$$\mathsf{MayContain}(e, B) : \bigwedge_{i=1}^{f} B[h_i(e)],$$

respectively, in which $\bigwedge$ is the logical conjunction operator. Those functions require $\mathcal{O}(1)$ time. The function MayContain may report false positives when querying for elements which were never inserted but are present in B as a result of independent insertions. Given n elements to insert, the optimal number of hash functions to use (Kirsch and Mitzenmacher, 2006) is $f = \frac{m}{n}\ln(2)$, for an approximate false positive rate of

$$\varphi \approx \left(1 - e^{\frac{-fn}{m}}\right)^{f} \approx 0.7^{\frac{m}{n}}$$

Hence, the BF trades off memory usage and time complexity with a decreased false positive rate.

In order to accelerate BFs, (Kirsch and Mitzenmacher, 2006) demonstrated that two hash functions combined in a double hashing technique can be applied in order to simulate more than two hash functions and obtain similar hashing performance. One main drawback of BFs is their poor data locality as bits corresponding to one element are scattered over B, resulting in several CPU cache-misses when inserting and querying. This issue was addressed in (Putze *et al.*, 2009), which presented the Blocked Bloom Filter (BBF), an array of smaller BFs individually fitting into one or multiple cache lines. To insert or look-up an element, a supplementary hash function is used to determine which BF to load. While BBFs are fast, their false positives ratios are usually higher than regular BFs due to the unbalanced load of each BF in the array.

As minimizers are used extensively throughout Bifrost, we use an efficient rolling hash function based on the work of Lemire and Kaser (2010) to select a g-mer as the minimizer within a single k-mer. Since overlapping k-mers are likely to share minimizers, we use an ascending minima approach (Harter, 2009) to recompute minimizers with amortized $O(1)$ costs, so that iterating over minimizers of adjacent k-mers in a sequence is linear in the length of the sequence. Another optimization is to restrict the computation of minimizers to a subset of g-mers of a k-mer; namely, we exclude the first and last g-mer as a candidate for being a minimizer. This ensures that for a given k-mer, all of its forward, respectively backward, adjacent k-mers necessarily share the same minimizer. While it is likely that a k-mer x and its neighbor x' share a minimizer, this neighbor hashing trick (Holley *et al.*, 2015) guarantees that when searching all forward, respectively backward, neighbors of x, they will all have the same minimizer and will be stored within the same block of a BBF, thus minimizing cache misses.

3 Methods

To build the cdBG, Bifrost builds first an approximation of the dBG using BBFs to filter out sequencing errors from the reads. The BBF containing the k-mers to insert in the graph is then used to build the exact cdBG. Finally, k-mers contained in the unitigs are mapped to their colors representing the input sources in which they occur.

The algorithms developed are described in this section. First Section 3.1 describes how an approximation of the true dBG is built from a set of sequencing reads, Algorithm 1 shows how to remove the majority of k-mers occuring only once and Algorithm 2 details the construction of a multithreaded BBF. Section 3.2 shows how the approximate cdBG is built from the BBF, Algorithm 3 details how the cdBG data structure indexes unitigs and Algorithm 4 shows how the data structure is queried for k-mers. Algorithm 5 shows how the unitigs of the approximate cdBG are discovered and Algorithm 6 constructs the approximated cdBG from a set of reads. Finally the approximate cdBG is cleaned up and converted to an exact cdBG using Algorithm 7. Section 3.3 shows how a ccdBG can be built efficiently on top of a cdBG with Algorithm 8.

3.1 Approximating the de Bruijn graph

The k-mers extracted from the reads will be inserted into two BBFs: BBF_1 will contain all k-mers occurring at least once in the input read sets while BBF_2 will contain all k-mers occurring twice or more often. This separation allows us to filter out unique k-mers which are likely to be sequencing errors (Melsted and Pritchard, 2011). Algorithm 1 starts by iterating over the reads and extracts all the canonical k-mers. BBF_1 is queried for the presence of each such k-mer and k-mers already present in BBF_1 are inserted into BBF_2. Finally, BBF_1 is discarded as the cdBG will be built from the k-mers of BBF_2.

In order to accelerate the insertions into the BBFs, the minimizer hash-value of each k-mer is used to determine the BBF block in which the k-mer is inserted. This guarantees that overlapping k-mers sharing the same minimizer position within a read are inserted into the same BBF block, thus improving the cache efficiency of BBFs. Furthermore, the neighbor hashing of the minimizers guarantees that all predecessors and successors of a k-mer are hashing to the same block, thus improving graph traversal for the exact cdBG construction step. Finally, the BBFs in Bifrost use 2-choice hashing (Azar *et al.*, 1999) to balance the number of insertions per block and reduce the number of false positives.

Algorithm 1 Construct Blocked Bloom Filters

```
Input: Read set $F$
1: function FILTER($F$)
2:     $BBF_1, BBF_2 \leftarrow$ empty Blocked Bloom filters
3:     for each read $r \in F$ do
4:         for each canonical $k$-mer $x \in r$ do
5:             $b \leftarrow \mathsf{MayContain}(x, BBF_1)$
6:             if $b$ is true then $\mathsf{Insert}(x, BBF_2)$
7:             else $\mathsf{Insert}(x, BBF_1)$
8:     return $BBF_2$
```

Instead of selecting a single BBF block when inserting a k-mer, two blocks are selected. If none of the two blocks already contains the k-mer, it is inserted into the block which has the fewest number of bits set. To enable parallel insertions, each BBF block is equipped with a spinlock to avoid multiple threads inserting at the same time within the same block. Algorithm 2 refines the insertion function introduced in Section 2 to enable 2-choice hashing and spinlocks usage with BBFs. Bifrost can make use of modern processors instruction sets to query simultaneously up to 16 bits within a block using AVX instructions.

Algorithm 2 Insert a k-mer into a BBF

```
Input: $k$-mer $x$, Blocked Bloom Filter $BBF$
1: function INSERT($x$, $BBF$)
2:     $y \leftarrow x.\mathsf{getMinimizer}()$                ▷ Minimizer of $x$
3:     $b \leftarrow |BBF|$                                  ▷ Number of blocks
4:     $b_1 \leftarrow BBF.h_1(y) \mod b$                    ▷ First block ID
5:     $b_2 \leftarrow BBF.h_2(y) \mod b$                    ▷ Second block ID
6:     if $b_2 < b_1$ then $\mathsf{swap}(b_1, b_2)$          ▷ Avoid deadlock
7:     $BBF[b_1].\mathsf{lock}()$                            ▷ Lock the first block
8:     if $\mathsf{MayContain}(x, BBF[b_1])$ is false then
9:         $BBF[b_2].\mathsf{lock}()$                        ▷ Lock the second block
10:        if $\mathsf{MayContain}(x, BBF[b_2])$ is false then
11:            $w_1 \leftarrow \mathsf{HammingWeight}(BBF[b_1])$
12:            $w_2 \leftarrow \mathsf{HammingWeight}(BBF[b_2])$
13:            if $w_1 < w_2$ then $\mathsf{Insert}(x, BBF[b_1])$
14:            else $\mathsf{Insert}(x, BBF[b_2])$
15:        $BBF[b_2].\mathsf{unlock}()$                      ▷ Unlock the second block
16:    $BBF[b_1].\mathsf{unlock}()$                          ▷ Unlock the first block
```

3.2 Constructing the compacted de Bruijn graph

The following section describes the data structure indexing the unitigs. Section 3.2.2 details the unitig extraction procedure from the BBF and the insertion of unitigs into the cdBG data structure.

3.2.1 Data structure

The cdBG data structure $D = (U, M)$ is composed of a unitig array U and a hash-table of minimizers M. A unitig u is first inserted into U and gets a unique identifier id_u. Unitig u is then decomposed into its set of constituent k-mers from which minimizers are extracted. Each minimizer is identified by a position p_m in u. While there can be as many minimizer positions as there are k-mers in the unitig, it is likely that multiple overlapping k-mers share the same minimizer position. The canonical g-mers corresponding to the minimizers are inserted into M and associated with their position p_m in u and the identifier id_u. Note that a minimizer might have multiple occurrences, either within a unitig or in different unitigs of the graph. The cdBG data structure D is illustrated in Figure 2. Algorithm 3 details the insertion of a unitig u in the cdBG data structure. Note that removing a unitig from the graph can be done in a reversed-fashion to Algorithm 3: The tuples associated with unitig u are removed from M and unitig u is removed from U.

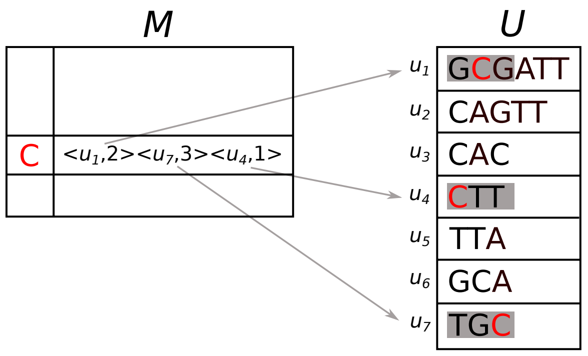

Figure 2: Data structure of a cdBG composed of a hash-table M and a unitig array U. Unitigs are composed of 3-mers and are indexed using minimizers of length 1. For simplicity, a lexicographic ordering of minimizers is here used and only one minimizer is shown.

Algorithm 3 Insert a Unitig into a cdBG

```
Input: Unitig $u$, cdBG data structure $D$
1: function INSERT($u$, $D = (U, M)$)
2:     $id_u \leftarrow Insert(u, U)$                      ▷ Unitig $u$ is inserted and gets an identifier
3:     $p'_m \leftarrow -1$                                ▷ Initialize position of previous minimizer
4:     $i \leftarrow 1$
5:     while $i \leq |u| - k + 1$ do                       ▷ Iterate over $k$-mer positions of $u$
6:         $x \leftarrow u(i, k)$                          ▷ $k$-mer at position $i$ in $u$
7:         for each minimizer positions $p_m \in x$ do
8:             if $p_m + i > p'_m$ then                    ▷ New minimizer
9:                 $y \leftarrow x(p_m, g)$                ▷ Minimizer of $x$
10:                $\mathcal{T} \leftarrow \mathsf{Find}(\hat{y}, M)$   ▷ Find occurrences of $\hat{y}$ in $U$
11:                if $\mathcal{T} = \varnothing$ then $\mathsf{Insert}(\{\hat{y}, \langle id_u, p_m + i\rangle\}, M)$
12:                else $\mathcal{T} \leftarrow \mathcal{T} \cup \langle id_u, p_m + i\rangle$
13:                $p'_m = p_m + i$
14:        $i \leftarrow i + 1$
```

Looking-up a k-mer x in the cdBG data structure is similar to inserting a unitig. The canonical g-mer corresponding to the minimizer of x is extracted and used to query M. If the g-mer is not in M, x does not occur in a unitig of the cdBG. However, if the g-mer is present, the identifiers of the unitigs containing the g-mer and the g-mer positions within those unitigs are returned. K-mer x and its reverse-complement $\overline{x}$ are then anchored in those unitigs at the given minimizer positions and compared. If the comparison is positive, a tuple with the unitig identifier and the k-mer position in the unitig is returned. Algorithm 4 shows how to look-up D for a k-mer.

Algorithm 4 Find a k-mer in a Unitig in a cdBG

```
Input: $k$-mer $x$, cdBG data structure $D$
1:  function FIND($x$, $D = (U, M)$)
2:      for each minimizer positions $p_m \in x$ do
3:          $y \leftarrow x(p_m, g)$
4:          $\mathcal{T} \leftarrow$ Find($\hat{y}$, $M$)
5:          if $\mathcal{T} \neq \varnothing$ then
6:              for each tuple $\langle p'_m, id \rangle \in \mathcal{T}$ do
7:                  $u \leftarrow U[id]$
8:                  $p_k \leftarrow p'_m - p_m + 1$                    ▷ Possible position of $x$ on $u$
9:                  if $1 \leq p_k \leq |u| - k + 1$ and $u(p_k, k) = x$ then
10:                     return $\langle p_k, id \rangle$
11:                 $p_k \leftarrow p'_m - k + g + p_m$                ▷ Possible position of $\overline{x}$ on $u$
12:                 if $1 \leq p_k \leq |u| - k + 1$ and $u(p_k, k) = \overline{x}$ then
13:                     return $\langle p_k, id \rangle$
14:     return $\langle -1, -1 \rangle$                                ▷ No matches found
```

3.2.2 Unitig extraction

The BBF returned by Algorithm 1 represents an approximation of the dBG: It contains the true positive k-mers composing the unitigs of the cdBG and false positive k-mers to exclude from the cdBG construction. The false positives k-mers are either artifacts of BBF_2 or single occurrence k-mers that should have been filtered out by Algorithm 1 but were inserted into BBF_2 as a result of their false occurrences in BBF_1. Although BBFs are efficient data structures, they do not allow to iterate over the contents. To get around this limitation, we iterate over the original set of reads and query BBF_2 to identify k-mers that are present.

Given a k-mer x, Algorithm 5 extracts from the BBF the unitig from which x is a substring, conditioned upon the presence of x in the BBF. K-mer x is extended forward, respectively backward, by reconstructing iteratively the prefix, respectively suffix, of the unitig using function Extend. Note that a backward extension is performed by extending forward from the reverse-complement of x and the extracted suffix is reverse-complemented to obtain the unitig prefix. Forward extensions are made with function ExtendForward which iteratively concatenate the last character from the next k-mer in the extension until no more k-mer is found or the extracted k-mer creates a cycle. Finally, k-mer x is extended with x' using function ExtendKmer if the two k-mers belong to the same maximal non-branching path, i.e, if x' is the only successor of x in the BBF and x is the only predecessor of x' in the BBF,

Algorithm 5 Unitig extraction from a BBF

```
Input: k-mer $x$, Blocked Bloom Filter $BBF$
 1: function EXTEND($x$, $BBF$)
 2:     $s_f \leftarrow \mathsf{ExtendForward}(x, BBF)$                ▷ Forward extension
 3:     $s_b \leftarrow \mathsf{ExtendForward}(\overline{x}, BBF)$     ▷ Backward extension
 4:     $s \leftarrow \overline{s_b} \odot x \odot s_f$                ▷ Unitig
 5:     return $s$

Input: k-mer $x$, Blocked Bloom Filter $BBF$
 1: function EXTENDFORWARD($x$, $BBF$)
 2:     $s \leftarrow \varepsilon$                                     ▷ String for extension
 3:     $x_e \leftarrow x$                                             ▷ Previous extended k-mer
 4:     $x'_e \leftarrow \mathsf{ExtendKmer}(x_e, BBF)$                ▷ New extended k-mer
 5:     while $x'_e \neq \varepsilon$ do                               ▷ While extending is possible
 6:         if $x'_e = x$ then $x'_e \leftarrow \varepsilon$           ▷ Cycle
 7:         else if $x'_e = x_f$ then $x'_e \leftarrow \varepsilon$    ▷ Self-loop k-mer
 8:         else if $x'_e = \overline{x_e}$ then $x'_e \leftarrow \varepsilon$   ▷ Self-loop k-mer
 9:         else
10:             $s \leftarrow s \odot x'_e(k, 1)$                      ▷ Extend string
11:             $x_e \leftarrow x'_e$
12:             $x'_e \leftarrow \mathsf{ExtendKmer}(x_e, BBF)$        ▷ Extend previous k-mer
13:     return $s_f$

Input: k-mer $x$, Blocked Bloom Filter $BBF$
 1: function EXTENDKMER($x$, $BBF$)
 2:     $i \leftarrow 0$
 3:     $x' \leftarrow x(2, k-1)$                                      ▷ Prefix of all successors of $x$
 4:     $x_e \leftarrow \varepsilon$                                   ▷ Extended k-mer
 5:     for each $a \in \mathcal{A}$ do
 6:         if $\mathsf{MayContain}(x' \odot a)$ is true then          ▷ BBF has successor of $x$
 7:             $i \leftarrow i + 1$                                   ▷ Increment count of successors
 8:             $x_e \leftarrow x' \odot a$                            ▷ Save last successor found
 9:     if $i = 1$ then                                                ▷ If $x$ has only one successor
10:         $i \leftarrow 0$
11:         $x' \leftarrow x_e(1, k-1)$                                ▷ Suffix of all predecessors of $x_e$
12:         for each $a \in \mathcal{A}$ do
13:             if $\mathsf{MayContain}(a \odot x')$ is true then $i \leftarrow i + 1$
14:     if $i \neq 1$ then $x_e \leftarrow \varepsilon$
15:     return $x_e$
```

Given the read set, the BBF containing the filtered k-mers and an empty cdBG data structure, Algorithm 6 extracts the unitigs from the BBF and inserts them into the cdBG data structure. The algorithm iterates over the k-mers of the reads and query the BBF for their presence. A missing k-mer in the BBF indicates the k-mer was filtered out by Algorithm 1 and will not be part of a unitig, in which case the next k-mer in the read is queried. However, in case of the k-mer presence in the BBF, the cdBG is searched for the unitig containing this k-mer using Algorithm 4. If the k-mer is missing from the unitigs present in the cdBG data structure, it means its unitig has not been extracted yet from the BBF. The extraction using Algorithm 5 takes place and the extracted unitig is inserted into the cdBG data structure with Algorithm 3.

Algorithm 6 Initial cdBG Construction

```
Input: Read set $F$, Blocked Bloom Filter $BBF$, cdBG data structure $D$
1:  function INSERT($F$, $BBF$, $D = (U, M)$)
2:      for each read $r \in F$ do
3:          $i \leftarrow 1$
4:          while $i \leq |r| - k + 1$ do                    ▷ For all $k$-mer positions in $r$
5:              $x \leftarrow r(i, k)$                                ▷ Get $k$-mer
6:              if MayContain($x, BBF$) is true then                  ▷ If $x$ in BBF
7:                  $t \leftarrow$ Find($x, D$)          ▷ Search unitig associated to $x$
8:                  if $t = \langle -1, -1 \rangle$ then              ▷ If unitig not found
9:                      $u \leftarrow$ Extend($x, BBF$)        ▷ Extract unitig from BBF
10:                     Insert($u, D$)               ▷ Insert unitig in data structure
11:                     $t \leftarrow$ Find($x, D$)
12:                 $p, id \leftarrow t$
13:                 $r_s \leftarrow r(i + k, |r| - i - k)$       ▷ Suffix of $r$ starting at $x$
14:                 $u_s \leftarrow u(p + k, |u| - p - k)$       ▷ Suffix of $u$ starting at $x$
15:                 $l \leftarrow$ Lcp($r_s, u_s$)   ▷ Longest Common Prefix of $u_s$ and $r_s$
16:                 $i \leftarrow i + |l|$          ▷ Move iterator forward of $|l|$ positions
17:             else $i \leftarrow i + 1$
```

3.2.3 Eliminating the false positive k-mers

The cdBG constructed by Algorithm 6 is not exact as it contains false positive k-mers of BBF_2. Those false positive k-mers create two types of errors in the graph:

- False connection: A false positive k-mer connects a unitig with no successors to a unitig with no predecessors. Hence, one unitig is extracted from the BBF instead of two.
- False branching: A false positive k-mer connects as a successor, respectively predecessor, to a true positive k-mer which already has a successor, respectively predecessor. Hence, three unitigs are extracted from the BBF instead of one.

An example of a cdBG containing the two types of errors is illustrated in Figure 3: K-mer 'CCG' creates a false branching and 'ACT' creates a false connection.

In order to distinguish false positive from true positive k-mers, a counter is maintained on each k-mer of the unitigs and Algorithm 6 is modified to increment the counters of the

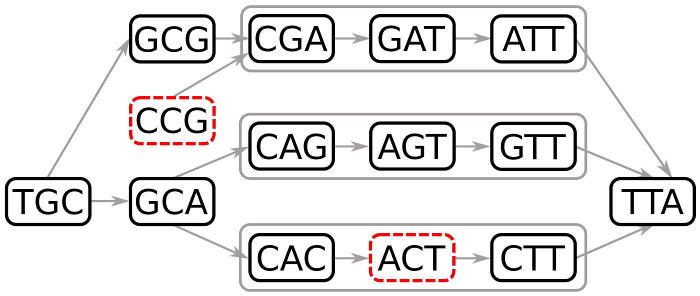

Figure 3: A compacted de Bruijn graph containing false positive 3-mers. Errors are represented in red dashed line vertices: K-mer 'CCG' creates a false branching and 'ACT' creates a false connection. K-mers that are compacted in a unitig are grouped in a grey line box.

k-mers occurring in the reads. Hence, false positive k-mers with no or one single occurrence are deleted from the graph. In the case of a false connection k-mer, deleting the k-mer splits a unitig. In case of a false branching, deleting the k-mer joins one or multiple unitigs.

3.2.4 Ghost k-mers

The false positive rate of the BBF will affect the length of the unitigs extracted by Algorithm 5. Consider a unitig of length $k+\eta-1$ in the true cdBG, consisting of η k-mers. For each internal k-mer, the algorithm makes 8 queries to the BBF, two of which will return true and 6 of which should return false. If the BBF has a false positive rate of p, the algorithm will advance to the next k-mer with probability $(1-p)^6 \approx 1-6p$ and stop prematurely with probability $\approx 6p$. The number of k-mers in the extracted unitig will then be limited by η on one hand and a geometric distribution with probability $6p$, whose expected value is $\frac{1}{6p}$. When $p = 10^{-3}$ this would lead to an average unitig length of 167. While these errors are fixed with Algorithm 7, this leads to an increased memory usage. One way to increase the length would be to use more memory in the BBF which would reduce the false positive rate. However we observe that the most likely configuration is that a single false positive k-mer x', adjacent to a real k-mer x in the unitig, causes a premature halt to the extraction of the true unitig. When x' has no other neighbor in the BBF except for x, we call it a ghost k-mer, insert it into a hash table to keep track of it in case we observe it later but do not stop the extraction of the unitig. In the rare case that x' turns out to belong to the true cdBG, we identify the unitig containing x' and fix the mistake. The probability that we halt can now be approximated as $42p^2$, since this would require two adjacent false positive k-mers to occur in the BBF. The use of ghost k-mers greatly reduces fragmentation which improves memory usage and running time.

3.2.5 Recurrent minimizers

Even in the case of a minimizer random ordering as described in Section 2, some minimizers are expected to occur more often in unitigs than others, due to indels occurring in homopolymer and tandem repeat sequences. Those minimizers are likely to increase the running time as their lists of tuples in the minimizer hash-table M will be much longer than for the other minimizers. We define a minimizer as *recurrent* if it occurs t times or more in the unitigs of the cdBG. In order to limit the impact of recurrent minimizers on

Algorithm 7 Removal of False Positives

```
Input: cdBG data structure $D$
 1: function RemoveFP($D = (U, M)$)
 2:     $\mathcal{X} \leftarrow \varnothing$
 3:     for each unitig $u \in U$ do
 4:         $i \leftarrow 1$
 5:         while $i \leq |u| - k + 1$ do                          ▷ For all $k$-mer positions in $u$
 6:             if getCounter$(i, u) < 2$ then                       ▷ $k$-mer is a false positive
 7:                 $\mathcal{X} \leftarrow \mathcal{X} \cup u(i, k)$
 8:             $i \leftarrow i + 1$
 9:     for each $k$-mer $x \in X$ do                               ▷ For all false positive $k$-mers
10:         $p, id \leftarrow$ Find$(x, D)$                          ▷ Find unitig containing FP
11:         $u \leftarrow U[id]$                                     ▷ Unitig containing FP
12:         Remove$(u, D)$                                           ▷ Remove $u$ from graph
13:         if $p = 1$ then Join$(\overline{u(1, k)}, D)$            ▷ Fix false connection
14:         else Insert$(u(1, p + k - 2), D)$                        ▷ Insert prefix of $u$
15:         if $p = |u| - k + 1$ then Join$(u(|u| - k + 1, k), D)$
16:         else Insert$(u(p + 1, |u| - p), D)$                      ▷ Insert suffix of $u$

Input: $k$-mer $x$, cdBG data structure $D$
 1: function Join($x$, $D = (U, G)$)
 2:     for each $a \in \mathcal{A}$ do
 3:         $x_s \leftarrow x(2, k - 1) \odot a$                     ▷ Possible successor of $x$
 4:         $p_s, id_s \leftarrow Find(x_s, D)$                      ▷ Find possible successor
 5:         if $p_s \neq -1$ then                                    ▷ Successor $x_s$ is found
 6:             $x_p \leftarrow$ ExtendKmer$(\overline{x_s}, D)$
 7:             if $x_p \neq \varepsilon$ then                       ▷ Unitigs of $x_s$ and $x_p$ can be joined
 8:                 $p_p, id_p \leftarrow Find(x_p, D)$
 9:                 $u_p \leftarrow U[id_p]$                         ▷ Unitigs of $x_p$
10:                 $u_s \leftarrow U[id_s]$                         ▷ Unitigs of $x_s$
11:                 if $x_p \neq u(|u| - k + 1, k)$ then $u_p \leftarrow \overline{u_p}$
12:                 if $x_s \neq u_s(1, k)$ then $u_s \leftarrow \overline{u_s}$
13:                 $u \leftarrow u_p \odot u_s(k, |u_s| - k + 1)$
14:                 Remove$(u_p, D)$                                 ▷ Remove $u_p$
15:                 Remove$(u_s, D)$                                 ▷ Remove $u_s$
16:                 Insert$(u, D)$                                   ▷ Insert joined unitig $u$
```

the graph construction, list of tuples in M have a maximum length t. When a k-mer x and its corresponding minimizer y must be inserted into the cdBG data structure, the length of the list associated with y in M is verified first. If the length is greater or equals to t, y is a recurrent minimizer. In such case, a non-recurrent minimizer $y' > y$ is extracted from x and inserted into M. If x does not contain a non-recurrent minimizer y', the recurrent minimizer y is inserted into M instead. Whenever k-mer x is searched, the list of tuples associated with its minimizer y is traversed and x is anchored on the instances of y in the unitigs of the graph until a match is found, as described in Algorithm 4. However, if no match is found for x and the list of tuples associated with y contains t or more tuples, the non-recurrent minimizer y' is extracted from x and the search continues using minimizer y'.

3.3 Coloring

We denote as D' the data structure of a ccdBG: It is composed of a unitig array U, a minimizer hash-table M, an array O of color containers, an array H of hash functions and a hash table K of k-mers.

3.3.1 Container representation

In Bifrost, a color is represented by an integer from 1 to $|C|$. A unitig u composed of $\eta = |u| - k + 1$ k-mers is associated with a binary matrix of size $\eta \times |C|$: rows represent the different k-mer positions in u and columns represent the colors from C. A bit set at row $1 \leq i \leq \eta$ and column $1 \leq j \leq |C|$ indicates that k-mer $u(i, k)$ occurs in dataset j. In order to limit the memory usage of colors, multiple compressed index are used to represent these binary matrices depending on their sparsity:

- A 64 bits word that can be either a tuple $\langle \text{position } i, \text{color } j \rangle$ or a binary matrix of size $\eta \times |C| \leq 62$ (2 bits are reserved for the meta-data)
- A compressed bitmap adapted from a Roaring bitmap container (Chambi *et al.*, 2016). This compressed bitmap stores up to 65488 tuples $\langle \text{position } i, \text{color } j \rangle$ and uses a maximum of 8 KB of memory. This container has 3 representations of the tuples it indexes: bit vector, sorted list of tuples and run-length encoded list of sorted tuples. Compared to a Roaring bitmap, this compressed bitmap uses less memory for its meta-data and requires less cache-miss to access the tuples.
- A Roaring bitmap (Chambi *et al.*, 2016) to store more than 65488 tuples. Roaring bitmaps are SIMD accelerated and propose numerous functions to manipulate bitmaps such as set intersection and union.

Those representations have a logarithmic worst-case time look-up and insertion.

3.3.2 Container indexing

Color containers can become substantially large and in order to avoid costly data transfer operations when the ccdBG data structure D' is modified, color containers are not associated directly to unitigs in D'. Instead, a solution derived from the MPHF (Minimal Perfect Hash Function) library BBHash (Limasset *et al.*, 2017) is used to link unitigs of array U to color containers of array O. The benefit of such a method is that operations which affects only the structure of the graph do not move the color containers in memory. Algorithm 8 describes how color containers are associated to their respective unitigs.

4 Results

To compare the performance of Bifrost, we benchmarked against state-of-the-art software on publicly available dataset. We focus on two representative problems for which the de

Algorithm 8 ccdBG Construction

```
Input: ccdBG data structure $D'$, set of hash functions $\mathcal{H}$
1:  function AssociateColors($D' = (U, M, O, H, K)$, $\mathcal{H}$)
2:      $B \leftarrow$ binary array of length $|U|$ initialized with 0s
3:      $O \leftarrow$ array of empty color containers of length $|U|$
4:      $h_e \leftarrow$ empty hash function $f\colon A \rightarrow \varnothing$
5:      $H \leftarrow$ array of hash functions of length $|U|$ initialized with $h_e$
6:      $id_u \leftarrow 1$
7:      $i_B \leftarrow 1$
8:      while $id_u \leq |U|$ do                              ▷ For each unitig
9:          $x \leftarrow U[id_u](1, k)$                         ▷ First $k$-mer of unitig
10:         for each hash function $h \in \mathcal{H}$ do
11:             $v \leftarrow h(x) \mod |U|$
12:             if $B[v] = 0$ then                           ▷ Color container in $C[v]$ is free
13:                 $B[v] \leftarrow 1$                      ▷ Color container in $C[v]$ is linked to $id_u$
14:                 $H[id_u] \leftarrow h$                   ▷ Unitig $id_u$ must be hashed with $h$
15:                 break
16:         if $H[id_u] = h_e$ then                          ▷ No color container was free for $id_u$
17:             while $i_B \leq |U|$ do                      ▷ Search next free color container
18:                 if $B[i_B] = 0$ then
19:                     break                                ▷ Color container at pos. $i_B$ is free
20:                 $i_B \leftarrow i_B + 1$
21:             $B[i_B] = 1$                                 ▷ Color container at pos. $i_B$ is reserved
22:             Insert($\{x, i_B\}$, $K$)
23:         $id_u \leftarrow id_u + 1$
```

Bruijn graph has been employed, namely whole genome assembly of HTS short reads and pan-genome analysis of assembled genomes. All experiments were run of a server with an 16-core Intel Xeon E5-2650 processor and 256G of RAM. Running time was measured as wall clock time using the `time` command, peak memory was measured by `ps`. All programs that support multithreading were run with 16 threads.

4.1 cDBG construction for Genome assembly

We constructed the compacted de Bruijn graph using a human genome short read dataset from the NA12878 sample from Genome In A Bottle consortium (Zook *et al.*, 2016), downsampled from 300-fold coverage to 30-fold coverage to reflect normal sequencing depth. The input is about 696 million 150 bp paired-end sequences with about 30x coverage. We compared Bifrost to the pipeline BCALM2 (Chikhi *et al.*, 2016) and Blight (Marchet *et al.*, 2019). BCALM2 is a disk-based compaction tool for short read data and assembled genomes while Blight is a non-dynamic data structure for indexing unitigs.

BCALM2 can be configured for different memory usage where a lower memory usage results in a longer running time. Hence, BCALM2 was configured in Table 1 with a maximum memory usage of 45 GB, similar to the Bifrost memory usage for the same dataset. We also ran BCALM2 with a 100 GB of memory, resulting in a running time of 4.91 hours

	Running time (h)	RAM peak (GB)	Disk peak (GB)
Bifrost	5.51	**41.5**	**0**
BCALM2+Blight	**4.78** (3.16+1.61)	46.6 (46.6,10.4)	129 (129,1)

Table 1: Running time, memory usage, and external disk space used for de Bruijn graph building and indexing. Numbers in parenthesis show the respective use of BCALM2 and Blight: The running time was added while the maximum was taken for memory and disk.

and memory usage of 88 GB with no external disk used.

4.2 Pan-genome analysis

We constructed colored and compacted de Bruijn graphs for a maximum of 117,913 assembled genomes of *Salmonella*. The input represents all publicly available *Salmonella* assemblies from the database Enterobase (Zhou *et al.*, 2019) as of August 2018. This is a 7.3x increase in the number of colors compared to the work of Muggli *et al.* (2019) who reported the construction for 16,000 *Salmonella* strains. We compared Bifrost to VARI-merge (Muggli *et al.*, 2019) as both tools can construct the colored de Bruijn graph and update it without reconstructing the graph entirely. The main differences between the two tools is that VARI-merge is mainly a disk-based method that produces a non-compacted colored de Bruijn graph. We only benchmarked VARI-merge as it is currently the state-of-the-art for colored de Bruijn graph construction. A comparison of VARI-merge to other colored de Bruijn graph construction tools is given in (Muggli *et al.*, 2019). Results are given in Table 2 for a variable number of strains. Note that the reported VARI-merge time includes the time spent by KMC2 (Deorowicz *et al.*, 2015) to compute the k-mers required in input of VARI-merge.

	Number of strains	Running time (h)	RAM peak (GB)	Disk peak (GB)
Bifrost	100	**0.016**	**0.16**	**0**
VARI-merge		0.33	5.1	17
Bifrost	400	**0.05**	**0.29**	**0**
VARI-merge		1.016	15.4	51
Bifrost	1600	**0.38**	**2.4**	**0**
VARI-merge		4.86	56.9	228
Bifrost	4000	**1.66**	**3.7**	**0**
VARI-merge		12.35	138	449
Bifrost	117913	**93.35**	**102.74**	**0**
VARI-merge		N/A	N/A	N/A

Table 2: Running time, memory usage, and external disk usage for constructing the colored de Bruijn graphs of an increasing number of *Salmonella* strains. N/A indicates the result is unavailable.

In Muggli *et al.* (2019), the authors process 16,000 strains in batches of 4,000, merging the batches to produce a colored de Bruijn graph of all strains. This required 254Gb of memory and 2.34Tb of external disk, with a total running time of 69 hours. In comparison, Bifrost processed 117,913 strains using about 103Gb of memory, no external disk usage and a total running time of 93.35 hours. While the running time is not directly comparable across different machines due to different processors, this is in line with Bifrost being about eight times faster than VARI-merge.

5 Discussion

We present Bifrost, a method for constructing compacted de Bruijn graphs, both regular and colored, with minimal memory requirements. Bifrost is competitive with the state-of-the-art de Bruijn graph construction method BCALM2 and the unitig indexing tool Blight with the advantage that Bifrost is dynamic. For colored de Bruijn graphs, Bifrost is about eight times faster than VARI-merge and uses about 20 times less memory with no external disk. The query capabilities of Bifrost are both for identifying colors for a given k-mer as well as navigating the de Bruijn graph. The software was developed with the intention of being usable as a tool or a library wherever large de Bruijn graphs are needed with minimal external dependencies.

Acknowledgements

The authors would like to thank Nina Luhmann, Birte Kehr and Thomas Krannich for their helpful feedback during the development of the software and Trausti Sæmundsson for work on an early draft of the software.

Funding

This work was supported by the Icelandic Research Fund Project grant number 152399-053.

Competing Interests

The authors declare that they have no competing interests.

References

Almodaresi, F. *et al.* (2017). Rainbowfish: A Succinct Colored de Bruijn Graph Representation. In *Proc. of the 17th Workshop on Algorithms in Bioinformatics (WABI'17)*, volume 88, pages 18:1–18:15.

Almodaresi, F. *et al.* (2018). An Efficient, Scalable and Exact Representation of High-Dimensional Color Information Enabled via de Bruijn Graph Search. *bioRxiv*.

Azar, Y. *et al.* (1999). Balanced Allocations. *SIAM J. Comput.*, **29**(1), 180–200.

Baier, U. *et al.* (2016). Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. *Bioinformatics*, **32**(4), 497–504.

Bankevich, A. *et al.* (2012). SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *J. Comput. Biol.*, **19**(5), 455–477.

Benoit, G. *et al.* (2015). Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC Bioinform.*, **16**(1), 288.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*, **13**(7), 422–426.

Bradley, P. *et al.* (2019). Ultrafast search of all deposited bacterial and viral genomic data. *Nat. Biotechnol.*, **37**, 152–159.

Bray, N. L. *et al.* (2016). Near-optimal probabilistic RNA-seq quantification. *Nat. Biotechnol.*, **34**, 525–527.

Chaisson, M. J. and Pevzner, P. A. (2008). Short read fragment assembly of bacterial genomes. *Genome research*, **18**(2), 324–330.

Chambi, S. *et al.* (2016). Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience*, **46**(5), 709–719.

Chikhi, R. and Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms Mol. Biol.*, **8**(22).

Chikhi, R. *et al.* (2016). Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, **32**(12), i201–i208.

Chin, C.-S. *et al.* (2016). Phased diploid genome assembly with single-molecule real-time sequencing. *Nat. Methods*, **13**(12), 1050.

Crusoe, M. R. *et al.* (2015). The khmer software package: enabling efficient nucleotide sequence analysis. *F1000Research*, **4**.

Deorowicz, S. *et al.* (2015). KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, **31**(10), 1569–1576.

Drezen, E. *et al.* (2014). GATB: Genome Assembly & Analysis Tool Box. *Bioinformatics*, **30**(20), 2959–2961.

Fang, H. *et al.* (2016). Indel variant analysis of short-read sequencing data with Scalpel. *Nat. Protoc.*, **11**, 2529–2548.

Grabowski, S. *et al.* (2015). Disk-based compression of data from genome sequencing. *Bioinformatics*, **31**(9), 1389–1395.

Harter, R. (2009). The minimum on a sliding window algorithm. http://richardhartersworld.com/cri/2001/slidingmin.html. [Online; accessed 25-March-2019].

Holley, G. *et al.* (2015). Bloom Filter Trie–A Data Structure for Pan-Genome Storage. In *Proc. of the 15th Workshop on Algorithms in Bioinformatics (WABI'15)*, volume 9289, pages 217–230.

Holt, J. and McMillan, L. (2014). Merging of multi-string BWTs with applications. *Bioinformatics*, **30**(24), 3524–3531.

Idury, R. M. and Waterman, M. S. (1995). A new algorithm for DNA sequence assembly. *J. Comput. Biol.*, **2**(2).

Iqbal, Z. *et al.* (2012). De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet*, **44**, 226–232.

Kamath, G. M. *et al.* (2017). HINGE: long-read assembly achieves optimal repeat resolution. *Genome Res.*, pages gr–216465.

Kirsch, A. and Mitzenmacher, M. (2006). Less hashing, same performance: Building a better Bloom filter. In *Proc. of the European Symposium on Algorithms (ESA'06)*, volume 4168, pages 456–467.

Koren, S. *et al.* (2017). Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Res.*, **27**(5), 722–736.

Lemire, D. and Kaser, O. (2010). Recursive n-gram hashing is pairwise independent, at best. *Comput. Speech. Lang.*, **24**(4), 698–710.

Li, H. (2016). Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, **32**(14), 2103–2110.

Limasset, A. *et al.* (2017). Fast and scalable minimal perfect hashing for massive key sets. *arXiv*, **arXiv:1702.03154**.

Limasset, A. *et al.* (2019). Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs. *Bioinformatics*, **btz102**.

Liu, B. *et al.* (2016). deBGA: read alignment with de Bruijn graph-based seed and extension. *Bioinformatics*, **32**(21), 3224–3232.

Luo, R. *et al.* (2012). SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, **1**, 18.

MacCallum, I. *et al.* (2009). ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome Biol.*, **10**, R103.

Marchet, C. *et al.* (2019). Indexing De Bruijn graphs with minimizers. In *Proc. of the 23rd International Conference on Research in Computational Molecular Biology (RECOMB'19)*.

Marcus, S. *et al.* (2014). SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, **30**(24), 3476–3483.

Melsted, P. and Pritchard, J. K. (2011). Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinform.*, **12**(1), 333.

Minkin, I. *et al.* (2013). Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In *Proc. of the 13th Workshop on Algorithms in Bioinformatics (WABI'13)*, volume 8126, pages 215–229.

Minkin, I. *et al.* (2016). TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, page btw609.

Muggli, M. D. *et al.* (2017). Succinct colored de Bruijn graphs. *Bioinformatics*, **33**(20), 3181–3187.

Muggli, M. D. *et al.* (2019). Building Large Updatable Colored de Bruijn Graphs via Merging. *bioRxiv*.

Pandey, P. *et al.* (2018). Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Systems*, **7**(2), 201–207.e4.

Pevzner, P. A. *et al.* (2001). An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA*, **98**(17), 9748–9753.

Putze, F. *et al.* (2009). Cache-, hash- and space-efficient bloom filters. *ACM J. Exp. Algorithmic*, **14**, 9.

Rang, F. J. *et al.* (2018). From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy. *Genome Biol.*, **19**(1), 90.

Rhoads, A. and Au, K. F. (2015). PacBio sequencing and its applications. *Genomics, Proteomics & Bioinformatics*, **13**(5), 278–289.

Roberts, M. *et al.* (2004). Reducing storage requirements for biological sequence comparison. *Bioinformatics*, **20**(18), 3363–3369.

Robertson, G. *et al.* (2010). De novo assembly and analysis of RNA-seq data. *Nat. Methods*, **7**, 909–912.

Ruan, J. and Li, H. (2019). Fast and accurate long-read assembly with wtdbg2. *bioRxiv*.

Salmela, L. and Rivals, E. (2014). LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, **30**(24), 3506–3514.

Simpson, J. T. *et al.* (2009). ABySS: A parallel assembler for short read sequence data. *Genome Res.*, **19**(6), 1117–1123.

Solomon, B. and Kingsford, C. (2016). Fast search of thousands of short-read sequencing experiments. *Nat. Biotechnol.*, **34**, 300–302.

Solomon, B. and Kingsford, C. (2018). Improved Search of Large Transcriptomic Sequencing Databases Using Split Sequence Bloom Trees. *J. Comput. Biol.*, **25**(7).

Sun, C. *et al.* (2018). Allsome sequence bloom trees. *J. Comput. Biol.*, **25**(5).

Uricaru, R. *et al.* (2015). Reference-free detection of isolated snps. *Nucleic Acids Res.*, **43**(2), e11.

Yang, B. *et al.* (2011). Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in Functional Genomics*, **11**(1), 25–37.

Yu, Y. *et al.* (2018). Seqothello: querying rna-seq experiments at scale. *Genome Biol.*, **19**, 167.

Zekic, T. *et al.* (2018). Pan-Genome Storage and Analysis Techniques. In *Comparative Genomics*, pages 29–53. Springer.

Zerbino, D. R. and Birney, E. (2008). Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, **18**(5), 821–829.

Zhou, Z. *et al.* (2019). The user's guide to comparative genomics with EnteroBase. Three case studies: micro-clades within Salmonella enterica serovar Agama, ancient and modern populations of Yersinia pestis, and core genomic diversity of all Escherichia. *bioRxiv*, page 613554.

Zook, J. M. *et al.* (2016). Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Sci. Data*, **3**, 160025.