*Bioinformatics*, 33(24), 2017, 4024–4032
doi: 10.1093/bioinformatics/btw609
Advance Access Publication Date: 21 September 2016
Original Paper

Sequence analysis

# TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes

**Ilia Minkin[1], Son Pham[2] and Paul Medvedev[1,3,4,*]**

[1]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA, [2]BioTuring Inc., San Diego, CA 92121, USA, [3]Department of Biochemistry and Molecular Biology and [4]Genomic Sciences Institute of the Huck, The Pennsylvania State University, University Park, PA 16802, USA

*To whom correspondence should be addressed.
Associate Editor: Alfonso Valencia

Received on April 3, 2016; revised on September 1, 2016; accepted on September 16, 2016

## Abstract

**Motivation:** de Bruijn graphs have been proposed as a data structure to facilitate the analysis of related whole genome sequences, in both a population and comparative genomic settings. However, current approaches do not scale well to many genomes of large size (such as mammalian genomes).

**Results:** In this article, we present TwoPaCo, a simple and scalable low memory algorithm for the direct construction of the compacted de Bruijn graph from a set of complete genomes. We demonstrate that it can construct the graph for 100 simulated human genomes in less than a day and eight real primates in $< 2$ h, on a typical shared-memory machine. We believe that this progress will enable novel biological analyses of hundreds of mammalian-sized genomes.

**Availability and Implementation:** Our code and data is available for download from github.com/medvedevgroup/TwoPaCo.

**Contact:** ium125@psu.edu

**Supplementary information:** Supplementary data are available at *Bioinformatics* online.

## 1 Introduction

The study of related features across different genomes is fundamental to many areas of biology, such as pan-genome analysis and comparative genomics. These studies often start with a representation of the relationship between genomes as a multiple alignment (Gusfield, 1997) or as a graph (Lee *et al.*, 2002). With the ubiquity of cheap sequencing, the number of genome sequences available for these studies has expanded tremendously (Haussler *et al.*, 2008; Jarvis *et al.*, 2014; Koepfli *et al.*, 2015). The type of genomes available has also expanded: we have whole genomes, as opposed to only genic sequences, and we now have many mammalian sized (~3 Gb) genomes. In addition, novel long-read sequencing technologies like Oxford Nanopore promise to make such genomes even easier to obtain. Thus, we expect to have hundreds of whole mammalian genome sequences for comparison, in both the population and comparative genomic settings. However, our current computational ability to analyze such large datasets is, at best, limited.

A major bottleneck toward the goal of comparing hundreds of whole mammalian genomes is scalability issues due to the problem of repeats. Multiple alignment is a computationally hard problem due to the presence of high copy-count repeats, which are absent in many lower-order species but cover roughly half of a mammalian genome. For example, the human genome contains over a million ALU repeats. Most multiple alignment methods mask repeats due to the computational challenge of handling them, resulting in a loss of important features. Without masking repeats, most approaches do not scale well to modern data, both in terms of computation time and memory usage. A competition of whole-genome aligners demonstrated that some recent tools are able to handle larger datasets; however, these were still limited to $\leq 20$ genomes of length $< 200$ Mb (Earl *et al.*, 2014).

As an alternative to multiple alignment, de Bruijn (or the closely related A-Bruijn) graph approaches for comparing whole genome sequences have been proposed (Minkin *et al.*, 2013a, b; Pham and Pevzner, 2010; Raphael *et al.*, 2004). de Bruijn graphs have traditionally been used for *de novo* assembly (Miller *et al.*, 2010; Schatz *et al.*, 2010), but in the case of already assembled genomes, they are built from a few long sequences, as opposed to billions of short reads. In the setting of population genomics, a de Bruijn graph representation of closely related genomes can be used to discover polymorphism in a population (Dilthey *et al.*, 2015; Iqbal *et al.*, 2012). In metagenomics, the de Bruin graph was used as a reference representation for read mapping (Wang *et al.*, 2012; Ye and Tang, 2015) and to predict virulence in bacteria (Bradley *et al.*, 2015). In the comparative genomics setting, a de Bruijn graph representation can be used to detect synteny blocks (Minkin *et al.*, 2013b; Pham and Pevzner, 2010). Other graph representations besides the de Bruijn graph have also been proposed (Dilthey *et al.*, 2015; Ernst and Rahmann, 2013).

The use of de Bruijn or related graphs brings up a host of algorithmic questions that have been studied: how to construct those graphs efficiently (Baier *et al.*, 2015; Beller and Ohlebusch, 2015; Ben-Bassat and Chor, 2014; Cazaux *et al.*, 2014; Chikhi *et al.*, 2014; Marcus *et al.*, 2014; Simpson and Durbin, 2010), how to design fast querying indices (Beller and Ohlebusch, 2016; Holley *et al.*, 2015; Sirén *et al.*, 2014), how to align read data to such graphs (Huang *et al.*, 2013; Limasset *et al.*, 2016; Paten *et al.*, 2014), and how to efficiently represent them in memory. Proposed representations of the de Bruijn graph include succinct (Belk *et al.*, 2016; Bowe *et al.*, 2012), compacted (Baier *et al.*, 2015; Beller and Ohlebusch, 2015; Cazaux *et al.*, 2014; Marcus *et al.*, 2014; Minkin *et al.*, 2013b), and Bloom filter based (Chikhi and Rizk, 2013; Salikhov *et al.*, 2014).

In this article, we study the efficient construction of the compacted de Bruijn graph. In a compacted de Bruijn graph, non-branching paths are replaced by single edges, which results in an equivalent, but smaller graph. The construction of such a graph is a resource intensive step and often poses the major bottleneck in applications. There have been recent tools to tackle the problem of efficiently constructing the compacted graph in the whole genome sequence setting: Sibelia (Minkin *et al.*, 2013b), SplitMEM (Marcus *et al.*, 2014), and the tools of Beller and Ohlebusch (2015) and Baier *et al.* (2015). The fastest algorithm to date was able to process seven whole mammalian genomes in under 8 h (Baier *et al.*, 2015). However, constructing the compacted graph is still prohibitive for larger inputs.

In this article, we present TwoPaCo, a novel algorithm for constructing the compacted de Bruijn graphs from whole genome sequences. We demonstrate that it can construct the graph for 100 human genomes in less than a day and eight primates in <2 h, on a typical shared-memory machine. TwoPaCo is based on the following key insight. We start with a basic naive algorithm, which has a prohibitively large memory usage but has the benefit that it is easily parallelizable. We then create a two pass algorithm that uses the naive one as a subroutine. In the first pass, we use a probabilistic data structure to drastically reduce the size of the problem, and in the second pass, we run the naive algorithm on the reduced problem. One of our key design principles was to make the algorithm simple and embarrassingly parallelizable, in order to take advantage of multi-thread support of most shared-memory servers. We also developed a procedure that splits the input into subsets that can be processed independently. As a result, TwoPaCo can trade-off memory usage for the running time, enabling processing large datasets on machines with small memory. The result is a simple and scalable low memory algorithm for the direct construction of the compacted de Bruijn graph for a set of complete genomes.

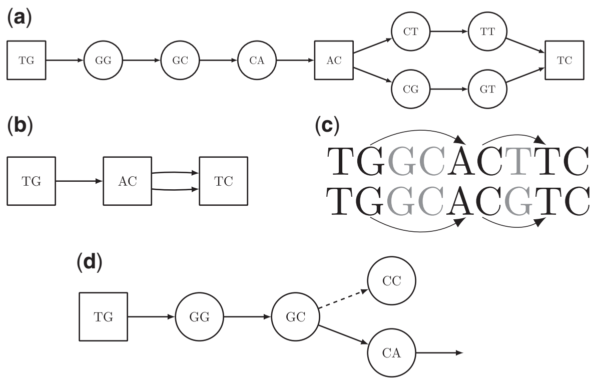

**Fig. 1.** The de Bruijn graph and its compacted version. (**a**) An example of an ordinary de Bruijn graph built from the genomes $S = \{"TGGCACGTC", "TGGCACTTC"\}$ and $k = 2$. Junctions are indicated by square vertices. (**b**) Graph obtained after compaction. (**c**) The two genomes that generate the graph, with the junction $k$-mers in bold; the arrows between them indicate edges in the compacted graph and non-branching paths in the ordinary graph. The strings between them label the edges in the compacted graph. (**d**) If we store edges in a Bloom filter, we may observe false edges (dotted line) in the ordinary graph; this can lead to detection of false junctions, like the vertex 'GC' in this case

## 2 Preliminaries

For a string $x$, we denote by $x[i..j]$ the substring from positions $i$ to $j$, inclusive of the endpoints. We say that a string $x$ is the *prefix* of a string $y$, if $x$ constitutes the first $|x|$ characters of $y$, where $|x|$ is the length of $x$. A string $x$ is the *suffix* of a string $y$, if $x$ constitutes the last $|x|$ characters of $y$. At first, we define the *de Bruijn graph* built from a single string. For a string $s$ and an integer $k$, we designate the de Bruijn graph as $G(s, k)$. Its vertex set consists of all substrings of $s$ of length $k$, called *k-mers*. Two vertices $u$ and $v$ are connected with a directed edge $u \to v$ if $s$ contains a substring $e$, $|e| = k + 1$ such that $u$ is the prefix of $e$ and $v$ is the suffix of $e$. We will use terms '$k$-mer' and 'vertex' interchangeably, as well as '$(k+1)$-mer' and 'edge'. For clarity of presentation, we have defined the de Bruijn graph as a simple graph, but we in fact store it as a multi-graph.

Now we define the de Bruijn graph for multiple strings. The union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the graph $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. For a collection of strings $S = \{s_1, s_2, \ldots, s_n\}$ and an integer $k$, the de Bruijn graph is the union of the graphs constructed from individual strings, i.e. $G(S, k) = G(s_1, k) \cup G(s_2, k) \cup \ldots \cup G(s_n, k)$. Figure 1a shows an example of a graph built from two strings. Recall that a *path* through a graph is a sequence of adjacent vertices where the only repeated vertices may be the first and last one, whereas a *walk* can repeat both vertices and edges. We say that a walk or path $p$ in the de Bruijn graph $G(S, k)$ *spells* a string $t$ if $G(t, k) = p$. We say that a vertex $v$ is a *bifurcation* if at least one of the following holds (i) $v$ has more than one incoming edge and (ii) $v$ has more than one outgoing edge. A vertex $v$ is a *sentinel* if it is a first or last $k$-mer of an input string. We call a vertex a *junction* if it is a bifurcation, or a sentinel, or both. The set $J(s, k)$ is the set of positions $i$ of the string $s$ such that the $k$-mer $s[i..i + k - 1]$ is a junction. For a collection of strings $S$ the set $J(S, k)$ is defined analogously.

A de Bruijn graph can be compacted by collapsing non-branching paths into single edges. More precisely a *non-branching path* in an ordinary de Bruijn graph is a path $u \rightsquigarrow v$ (a path from $u$ to $v$) such that the only junction vertices on this path are possibly $u$ or $v$. The *compaction* of a non-branching path $p = u \rightsquigarrow v$ is removal of edges of $p$ and replacing it with an edge $u \to v$ (the graph is output such that for each vertex we maintain a list of its occurrences in the input so that one can easy reconstruct the original path). A *maximal* non-branching path is a non-branching path that cannot be extended by adding an edge. The *compacted graph* $G_c(S, k)$ is the graph obtained from $G(S, k)$ by compaction of all its maximal non-branching paths. This graph is sometimes referred to as the compressed graph in the literature (Beller and Ohlebusch, 2015). It is easy to see that the vertex set of $G_c(S, k)$ is the set of junctions of the graph $G(S, k)$ and two vertices $u$ and $v$ of $G_c(S, k)$ are connected if there is a non-branching path $u \rightsquigarrow v$ in $G(S, k)$. Figure 1b shows an example of a compacted de Bruijn graph. Note that a compacted graph is a multi-graph: after compaction a pair of vertices can be connected by edges going in the same direction that corresponded to different paths in the ordinary graph.

Graph compaction is the first step of most algorithms working with de Bruijn graphs, since it drastically reduces the number of vertices. It can be obtained from the ordinary graph in linear time by a simple graph traversal. However, building and storing the ordinary graph takes lots of space, which we seek to avoid in our algorithm by constructing the compacted graph directly.

A *Bloom filter* is a space efficient data structure for representing sets that supports two operations: storing an element in the set and checking if an element is in the set (Bloom, 1970). A Bloom filter offers improvements in space usage but can generate false positives during membership queries. Bloom filters have previously been successfully applied to assembly (Chikhi and Rizk, 2013; Heo *et al.*, 2014; Melsted and Pritchard, 2011; Salikhov *et al.*, 2013) and to indexing and compression of whole genomes as well as large RNA-seq datasets (Holley *et al.*, 2015; Rozov *et al.*, 2014; Solomon and Kingsford, 2016). In particular, they have been applied to the closely related problem of constructing and compacting a de Bruijn graph from short read sequences. While this article addresses the whole genome setting, we find that the Bloom filter remains useful to represent a set of $k$-mers.

## 3 Reduction to the problem of finding junction positions

TwoPaCo is based on the observation that there is a bijection between maximal non-branching paths of the de Bruijn graph and substrings of the input whose junctions are exactly the two flanking $k$-mers (Observation 1 below). This observation reduces the problem of graph compaction to finding the set of junction positions $J(S, k)$, as follows. The vertex set of the compacted graph is the set of all $k$-mers located at positions $J(S, k)$. To construct the edges, we need to find substrings flanked by junctions. To do this, we can traverse positions of $J(S, k)$ in the order they appear in the input. For every two consecutive junction positions $i$ and $j$, we record an edge between the $k$-mer at $i$ and the $k$-mer at $j$. Figure 1c shows an example of how sequences of junctions generate non-branching paths in the ordinary graph and edges in the compacted one.

The observation follows in a straight-forward way from the definitions, but we state and prove it here for completeness.

**Observation 1.** *Let s be an input string and P be the set of maximal non-branching paths of the graph G(s, k). Let T be the set of substrings of s such that each* $t \in T$ *starts and ends with a junction of G(s, k) and does not contain junctions in between. Then there exists a bijective function* $g : T \rightarrow P$.

**Proof.** Let $g$ be the function mapping substrings of $s$ to walks in $G(s, k)$, where $g$ maps a substring to the vertices corresponding to its constituent $k$-mers. To prove that $g$ is a bijection when restricted to $T$, we have to show that it is both an injection and surjection. Note that $g$ is injective by construction, that is, any walk is spelled by a unique string. To prove that it is surjective, we need to show that for any maximal non-branching path $p = u \rightsquigarrow v$, there is a $t \in T$ such that $g(t) = p$. That is, $p$ is spelled by a string in $T$. Since the walk $g(s)$ must traverse all vertices in the graph, and the internal vertices of $p$ have in- and out-degrees equal to one, the walk $g(s)$ must contain $p$ as a subwalk. Hence, the string $t$ spelled by $p$ must be a substring of $s$, i.e. $g(t) = p$. The internal $k$-mers of $t$ are non-junctions because $p$ is non-branching, and the first and last $k$-mers of $t$ are junctions because $p$ is maximal. Hence, $t \in T$.

Generalization of the observation to the case of multiple strings is straightforward.

## 4 Single round algorithm

In the previous section, we reduced the problem of constructing a compacted de Bruijn graph to that of finding the locations in the genome where junction vertices are located. We will now present our algorithm for finding junction positions, in increasing layers of complexity. First, we will describe Algorithm 1, which can already be used as a naive algorithm to identify the junctions. However, Algorithm 1 alone has a prohibitively large memory footprint. To address this, we will present Algorithm 2, which uses Algorithm 1 as a subroutine but reduces the memory requirements. In cases of very large inputs, even Algorithm 2 can exceed the available memory. In Section 5, we finally present Algorithm 3, which addresses this limitation. It limits memory usage, at the expense of time, by calling Algorithm 2 over several rounds. We refer to this final algorithm (Algorithm 3) as TwoPaCo.

In Algorithm 1, we start with a candidate set $C$ of junction positions in the genomes. A set of positions $C$ is called a *candidate set* if $C \supseteq J(S, k)$ and any two positions that start with the same $k$-mer can be either both present or both absent from $C$. $C$ is represented using Boolean flags which mark every position of the genomes which is present in the set. If Algorithm 1 is used naively, it would be called with every position marked; in general, however, we can use $C$ to capture the fact that the unmarked positions have been previously eliminated from consideration as junctions.

First, we store all edges of the ordinary de Bruijn graph in a set $E$. We do this by a linear scan and for a $(k+1)$-mer at position $i$ in a string $S$, if either of the $k$-mers at positions $i$ or $i+1$ are marked, we insert the $(k+1)$-mer into the set $E$ (Lines 1–5). Second, we again scan through the genomes and consider the $k$-mer $v$ at every marked position. We use $E$ to check how many edges in $G(S, k)$ enter and leave $v$ (Lines 9–15). Since the DNA alphabet is finite, we can do this by merely considering all eight possible $(k+1)$-mers—four entering, and four leaving—and checking whether they are in $E$. If the in- and out-degrees do not satisfy the definition of a junction, we unmark position $i$; otherwise, we leave it marked.

Algorithm 1 can be used naively to find all junction positions, by initially marking every position as a potential junction. Storing the set $E$ in memory, however, is infeasible for large datasets. To reduce the space requirements, we develop the two pass Algorithm 2. In the first pass, we run Algorithm 1, but use a Bloom filter to store the set $E$ instead of a hash table. A Bloom filter takes significantly less space than a hash table; however, the downside is that it can generate false positives during membership queries. That is, when we check if a $(k+1)$-mer is present in $E$ (Lines 12 and 14 in Algorithm 1) we may receive an answer that it is present, when it is in reality absent. The effect is that the calculated in- and out-degrees may be inflated and we may leave non-junctions marked (Line 17), see Figure 1d. Nevertheless, the marked positions still represent a candidate set of

**Algorithm 1.** *Filter-Junctions*

**Input:** strings $S = \{s_1, \dots, s_n\}$, integer $k$, and an empty set data structure $E$. A candidate set of marked junction positions $C \supseteq J(S, k)$ is also given. When the algorithm is run naively, all the positions would be marked.
**Output:** a reduced candidate set of junction positions.

```
1: for $s \in S$ do
2:     for $1 \leq i < |s| - k$ do
3:         if $C[s, i]$ = marked then            ▷ Insert the two $(k+1)$-mers containing the $k$-mer at $i$ into $E$.
4:             Insert $s[i..i+k]$ into $E$.
5:             Insert $s[i-1..i-1+k]$ into $E$.
6: for $s \in S$ do
7:     for $1 \leq i < |s| - k$ do
8:         if $C[s, i]$ = marked and $s[i..i+k-1]$ is not a sentinel then
9:             $in \leftarrow 0$                        ▷ Number of entering edges
10:            $out \leftarrow 0$                       ▷ Number of leaving edges
11:            for $c \in \{A, C, G, T\}$ do            ▷ Consider possible edges and count how many of them exist
12:                if $v \cdot c \in E$ then            ▷ The symbol · depicts string concatenation
13:                    $out \leftarrow out + 1$
14:                if $c \cdot v \in E$ then
15:                    $in \leftarrow in + 1$
16:            if $in = 1$ and $out = 1$ then           ▷ If the $k$-mer at $i$ is not a junction.
17:                $C[s, i] \leftarrow$ Unmarked
18: return $C$
```

**Algorithm 2.** Filter-Junctions-Two-Pass

**Input:** strings $S = \{s_1, \dots, s_n\}$, integer $k$, a candidate set of junction positions $C_{\text{in}}$, integer $b$
**Output:** a candidate set of junction positions $C_{\text{out}}$

```
1: $F \leftarrow$ an empty Bloom filter of size $b$
2: $C_{\text{temp}} \leftarrow Filter - Junctions(S, k, F, C_{\text{in}})$        ▷ The first pass
3: $H \leftarrow$ an empty hash table
4: $C_{\text{out}} \leftarrow Filter - Junctions(S, k, H, C_{\text{temp}})$   ▷ The second pass
5: return $C_{\text{out}}$
```

junctions, since a junction will never be unmarked. Thus, running Algorithm 1 with the Bloom filter reduces memory but does not always unmark non-junction positions. In order to eliminate these marks, we run Algorithm 1 again, using the positions marked in the first pass as a starting point, but this time using a hash table to store $E$ (Line 4 in Algorithm 2). This second pass will unmark all remaining marked non-junction positions. Since the set of candidate marks has been significantly reduced after the first pass, the memory use of the hash table is no longer prohibitive. As with Algorithm 1, Algorithm 2 can be used to find all junction positions by initially marking every position as a potential junction.

Our implemented algorithms also handle the reverse complementarity of DNA, using standard techniques. We summarize this briefly for the sake of completeness. For a string $s$, let $\bar{s}$ be its reverse complement, and define the comprehensive de Bruijn Graph as the graph $G_{\text{comp}}(s, k) = G(s, k) \cup G(\bar{s}, k)$; the graph for multiple strings and the compacted graph is defined analogously. To build the compacted comprehensive graph, we have to modify Algorithm 1 so that $E$ represents each $k$-mer and its reverse complement jointly. For example, this can be done by always storing the *canonical* form of a $k$-mer, which is the lexicographically smallest string between the $k$-mer and its reverse complement (Chikhi *et al.*, 2014). Similarly, we have to be careful when we make membership queries to $E$ in Algorithm 1, so that we are always querying canonical $k$-mers.

## 5 Multiple rounds: dealing with memory restrictions

While Algorithm 2 significantly reduces the memory usage, it is still possible that the hash table in the second pass may not fit into the main memory, for some very large inputs. To deal with this issue, we develop Algorithm 3, which splits the input $k$-mers into $\ell$ parts and runs Algorithm 2 in $\ell$ rounds. In each round, Algorithm 3 will consider only approximately $1/\ell$ of the $k$-mers to check if they are junctions. Each round processes only one part, thus decreasing memory usage. When $\ell = 1$, Algorithm 3 reduces to Algorithm 2 and does not limit its memory use, but when $\ell$ is increased, the peak memory usage decreases at the expense of more rounds and hence longer running time.

Suppose that the set of $k$-mers is partitioned into $\ell$ classes $V_1, \dots, V_\ell$. Then, in round $i$, our algorithm begins by marking the positions whose $k$-mers are in class $V_i$ (Line 15). Note that each position is considered in exactly one round. We then call Algorithm 2, which unmarks those positions which are not junctions. After all the rounds are complete, the junction vertices are exactly those that remain marked (Line 17).

To obtain the partition classes $V_i$, we first note that the maximum memory usage of Algorithm 3 is minimized when the partition leads to an equally sized hash table in every round. To achieve this, we would like the sizes of sets $V_i$ to be as equal as possible. We are not concerned with obtaining an optimal partition, since a small discrepancy in the memory in each round is permissible. Also note that we must partition the $k$-mers, which is different from partitioning the positions. In particular, if two different positions have the same $k$-mer, they must belong to the same class; hence, we cannot simply divide our strings into chunks.

Our idea is based on a two-step process. First, we partition the universe of all $k$-mers into $q \gg \ell$ parts (e.g. $q = 2^{32}$). This is done implicitly by defining a uniform hash function $f$ over the universe of $k$-mers with range $[0, q)$. A $k$-mer $h$ belongs to part $i$ if its hash value $f(h) = i$. We then obtain the number of input $k$-mers that belong to

**Algorithm 3.** TwoPaCo

```
Input: strings $S = \{s_1, \ldots, s_n\}$, integer $k$, integer $\ell$, integer $b$
Output: the compacted de Bruijn graph $G_c(S, k)$
1: Initialize counters $c_0, \ldots, c_{q-1}$ to zeroes
2: $F \leftarrow$ an empty Bloom filter of size $b$
3: for $s \in S$ do
4:     for $1 \le i \le |s| - k + 1$ do
5:         $h \leftarrow s[i..i + k - 1]$
6:         if $h$ not in $F$ then
7:             Insert $h$ into $F$
8:             $c_{f(h)} \leftarrow c_{f(h)} + 1$
9: $T \leftarrow \sum_{0 \le t < q} c_t/\ell$                ▷ Mean number of $k$-mers per partition
10: $p_0 \leftarrow 0, p_\ell \leftarrow q$
11: for $1 \le i < \ell$ do
12:     $p_i \leftarrow$ biggest integer larger than $p_{i-1}$ such that $(\sum_{p_{i-1} \le j < p_i} c_j) \le T$, or $\min\{\ell, p_{i-1} + 1\}$ if it does not exist.
13: $C_{\text{init}} \leftarrow$ Boolean array with every position unmarked
14: for $1 \le i \le \ell$ do
15:     $C_i \leftarrow$ mark every position of $C_{\text{init}}$ that starts a $k$-mer $h$ with hash value $p_{i-1} \le f(h) < p_i$
16:     $C'_i \leftarrow$ Filter − Junctions − Two − Pass$(S, k, b, C_i)$
17: $C_{\text{final}} = \cup C'_i$
18: return Graph implied by $C_{\text{final}}$, as described in Section 3.
```

each part, $c_0, \ldots, c_{q-1}$, as follows (Lines 1–8). We make a pass through the input and use a Bloom filter to store all the $k$-mers. For every $k$-mer, if it is not already present in the Bloom filter, we increase the corresponding counter. This way, we try to count only unique $k$-mers, though the count can be slightly inflated by false positives. Due to the heuristic nature of our partitioning process, we can tolerate slightly inaccurate counts. Notice that the partition defined by $f$ is not necessarily balanced when applied to the input $k$-mers, i.e. there might be a big difference between $c_i$ and $c_j$, for some $i$ and $j$. However, this initial partition is much more fine grained then we need, which makes it a useful starting point.

Second, we obtain our desired partition into $V_1, \ldots, V_\ell$ by agglomerating consecutive parts of our fine-grained partition. Specifically, we implicitly define $V_1, \ldots, V_\ell$ using a sequence of integers $0 = p_0 \le p_1 \le \cdots \le p_{\ell-1} \le p_\ell = q$, where a $k$-mer $h$ belongs to $V_i$ if $p_{i-1} \le f(h) < p_i$. To find a sequence $p_0, \ldots, p_\ell$ that would create a balanced partition, we use a greedy heuristic (Lines 10–12). It makes a linear scan through the counts $c_0, \ldots, c_{q-1}$, and fills the current partition with as many $k$-mers as possible, until the number of $k$-mers exceeds the $1/\ell$ of the total $k$-mers.

## 6 Parallelization scheme

We designed our algorithm so that it can be effectively parallelized on a multi-processor shared memory machine. The bulk of the computation happens in Algorithm 1, which consists of two parts. Each part is a loop over all the positions in the input, Lines 1–5 in the first part and Lines 6–17 in the second. The first loop is embarrassingly parallelizable as long as the data structure representing the set $E$ supports concurrent writes. We use a lock-free Bloom filter when Algorithm 1 is called during the first pass of Algorithm 2, and a concurrent hash table when it is called during the second pass. The second loop is trivially parallelizable: threads will get non-overlapping portion of genomes, hence the synchronization on $C$ is not needed. A synchronization barrier separates the two loops. The compacted edge generation step that we discussed in the Section 3 is embarrassingly parallelizable as well.

We implement the parallelization using the standard single producer/multiple consumer pattern (Oaks and Wong, 2004). According to this design pattern, we create (i) a single reader thread that splits the input into equal sized substrings and puts them into worker queues and (ii) many worker threads that dequeue and process the substrings. We utilized parallel programming primitives from the Intel's Threading Building Blocks library (Reinders, 2007). Note that this way we store only part of the input and the corresponding array $C$ in the input to save memory.

## 7 Theoretical analysis and comparison

In this section, we will analyze the running time and memory usage of our algorithm, and compare it with that of other algorithms. Suppose that the de Bruijn graph $G(S, k)$ has $E$ edges, $J$ junctions and $L$ non-junctions that we call *links*. First, we will analyze the number of false positive junctions. A false positive junction is a link whose positions in $S$ are incorrectly left marked at the end of the first pass. We assign an indicator variable $I_\ell$ to each link $\ell$, $I_\ell = 1$ if the link $\ell$ is a false positive junction and $I_\ell = 0$ otherwise. This way, the total number of false positive junctions is $FP = \sum_{1 \le \ell \le L} I_\ell$. Let the probability that a link is a false positive junction be $p$. By linearity of expectation, we have $\mathbb{E}[FP] = \mathbb{E}[\sum_{1 \le \ell \le L} I_\ell] = Lp$. To calculate the probability $p$, note that each link has exactly one incoming and one outgoing true edge. Hence, querying the Bloom filter in Line 12 and Line 14 of Algorithm 1 may discover at most six false edges: three incoming and three outgoing ones. At least one false positive from those six queries results in the link misclassified as a junction. Mitzenmacher and Upfal (2005) show that the probability of a single false positive resulting from querying a Bloom filter is $q = (1 - e^{-hE/b})^h$. where $h$ is the number of hash functions used by the Bloom filter and $b$ is the number of bits in the filter. Assuming that queries are independent, $p = 1 - (1 - q)^6 = 1 - (1 - (1 - e^{hE/b})^h)^6$.

Now we will analyze the running time. Let $m$ be the total length of the input strings. First, note that storing and querying $k$-mers with the Bloom filter requires calculation of $h$ hash values for each operation. We use a family of sliding window hash functions, so both filling and querying the Bloom filter in the first pass takes $O(mh)$ operations. In the second pass the algorithm employs a hash table to store and query $(k+1)$-mers. Denote by $M$ the number of marks left in the array $C$ after the first pass. The expected running time is then $O(mh + Mk)$, since each hash table operation takes $k$ time and there are $O(M)$ operations total. To calculate $M$, let us assume that the average number of times a false positive junction occurs in all the input strings is given by $r$. Then, the expected value of $M$ is $|G_c| + Lpr$, where $|G_c|$ is the number of edges in the compacted de Bruijn multi-graph. The expected running time is then $O(mh + (|G_c| + Lpr)k)$

To calculate the memory usage, note that the first pass allocates $b$ bits of memory for the Bloom filter and the second pass uses a hash table that contains at most $8(J + FP)$ elements. Hence, the expected memory usage is $O(\max[b, (J + Lp)k])$. The array $C$ of marks is accessed sequentially by the algorithm and can be stored in the external memory without loss of performance. As discussed in Section 6, at each moment the memory contains only a constant amount of characters of the input strings, so the input length does not contribute to the asymptotic bound.

Table 1 contains asymptotic upper bounds on memory usage and running times of different algorithms for constructing the compacted de Bruijn graph from multiple complete genomes. The performance of TwoPaCo depends highly on the number of junctions present. On practical instances of related genomes datasets, there is a lot of shared sequence and the number of junctions is low. Unlike other algorithms, our expected memory usage depends only on the structure of the input, but not directly on its size. At the same time, dependence on $k$ makes TwoPaCo less applicable in case of very large $k$.

**Table 1.** Running times and memory consumption of different algorithms for constructing the de Bruijn graph from multiple complete genomes

| Algorithm | Running time | Memory |
|---|---|---|
| Sibelia | $O(m)$ | $O(m)$ |
| SplitMEM | $O(m \log g)$ | $O(m + \lvert G_c \rvert)$ |
| bwt-based[a] | $O(m)$ | $O(m)$ |
| TwoPaCo | $O(mh + (\lvert G_c \rvert + Lpr)k)$ | $O(\max[b, (J + Lp)k])$ |

*Note*: For SplitMEM $g$ stands for the size of the largest genome in the input. An explanation of other variables is given in the Section 7.

[a]Baier *et al.* (2015).

## 8 Results

To evaluate the performance of TwoPaCo, we conducted several experiments. We compared its running time and memory footprint with other available implementations of de Bruijn graph compaction algorithms. We then ran TwoPaCo on a real dataset of biological interest as well as a large dataset of simulated data. We assessed the parallel scalability of our implementation and capabilities of running the algorithm on machines with limited memory using the round splitting procedure Finally, we evaluated the effects of input length and structure on the running time and memory usage.

First, we benchmarked TwoPaCo against Sibelia (Minkin *et al.*, 2013b), SplitMEM (Marcus *et al.*, 2014) and the bwt-based algorithm of Baier *et al.* (2015), using default parameters. As far as we understood, the algorithm in Beller and Ohlebusch (2015) was subsumed by Baier *et al.* (2015). There were two important caveats. First, in most genomics application, it is necessary to account for both strands in the de Bruijn graph. To make SplitMem and bwt-based work with both strands, we appended the reverse complements of the sequences to the input, as suggested by their authors. In our results, we show SplitMEM and the bwt-based in two versions: (i) considering only one strand and (ii) considering both strands. Second, both minia and Sibelia not only constructs the compacted graph but also modifies it after construction. We therefore ran these tools only in the mode where they construct the graph only (contrary to the bechmarks in Marcus *et al.*, 2014). In addition to whole genome tools, one can also apply tools from genome assembly to construct the compacted graph. In this case, one would run a $k$-mer counter on the genomes, and then run a graph compaction tool on the resulting $k$-mers. Note that graph compaction algorithms that start with a set of $k$-mers must build non-branching paths by combining appropriate $k$-mers, while in the whole genome setting the non-branching paths are contained in the input (per 1). We tested how well these pipelines would compare against specialized approaches for whole genomes. We tried two pipelines: minia 2.0.3 (Chikhi and Rizk, 2013) and the DSK 2.1.0 $k$-mer counter (Rizk *et al.*, 2013) followed by BCALM (Chikhi *et al.*, 2014) (a new version of BCALM that allows parallelism was very recently published but was not available at the time of writing, Chikhi *et al.*, 2016). We allowed them to use the maximum memory. Although minia has some parallelism, its graph compaction implementation is single-threaded. We allowed minia and DSK to use all 15 threads. BCALM is single-threaded. Supplementary material for the article contains the exact command lines that we used for benchmarking.

For benchmarking purposes, we used the following datasets: (i) 62 *Escherichiacoli* genomes (310 Mb) from Marcus *et al.* (2014) and (ii) seven human genomes (~21 Gb) used by Baier *et al.* (2015) which includes five different assemblies of the human reference genome and two paternal haplotypes of NA12878 (see Baier *et al.* (2015) for more details). We ran our experiments on the highest memory Amazon EC2 instance (r3.8xlarge): a server with 32 Intel Xeon E5-2670 processors and 244 GB of RAM. We set the default number of internal hash functions in the Bloom filters to four. We also verified the correctness of TwoPaCo by comparing its output to that of a naive compaction algorithm on feasible test cases. A direct comparison to the output of other tools is impractical since each algorithm handles edges cases differently (e.g. the presence of undetermined nucleotides (Ns) in the input).

The results are shown in the first four rows of Table 2. For seven human genomes, TwoPaCo was at least 7 times faster than the second best algorithm, when we used 15 threads. When only a single thread was used, TwoPaCo was still slightly better than the second best DSK + BCALM for $k = 25$, and 2.5–3.4 times faster than the second best bwt-based on $k = 100$.

We also assessed TwoPaCo's ability to handle (i) large numbers of long closely-related genomes and (ii) more divergent genomes. To do so, we generated 93 human genomes using the FIGG genome simulator (Killcoyne and del Sol, 2014) and 'normal' simulation parameters. The FIGG genome simulator generates complete sequences based on a reference genome and variations' frequencies extracted from the datasets from projects like 1000 Genomes Project Consortium *et al.* (2010) and Gibbs *et al.* (2003). The mutations comprise single-nucleotide alterations as well as indels and structural variations of larger size. We ran TwoPaCo on two datasets: (i) 43 simulated genomes plus the seven used in Table 2 and (ii) 93 simulated human genomes plus the seven. The results are shown in the last five rows of Table 2. We construct the graph for 100 human genomes in 23 h using 77 GB of RAM and 15 threads. For eight primates, we used under 2 h and 34–62 GB of RAM on 15 threads.

**Table 2.** Benchmarking comparisons

| | DSK+BCALM | Minia | Sibelia | SplitMem | bwt-based from Baier *et al.* (2015) | | TwoPaCo | |
|---|---|---|---|---|---|---|---|---|
| | | | | Single strand | Single strand | Both strands | 1 thread | 15 threads |
| 62 *E.coli* ($k = 25$) | 6 (1.57) | 151 (0.9) | 10 (12.2) | 70 (178.0) | 8 (0.85) | 12 (1.7) | 4 (0.16) | 2 (0.39) |
| 62 *E.coli* ($k = 100$) | 13 (2.50) | 114 (1.9) | 8 (7.6) | 67 (178.0) | 8 (0.50) | 12 (1.0) | 4 (0.19) | 2 (0.39) |
| 7 humans ($k = 25$) | 444 (22.44) | 968 (48.09) | – | – | 867 (100.30) | 1605 (209.88) | 436 (4.40) | 63 (4.84) |
| 7 humans ($k = 100$) | 1347 (221.65) | 1857 (222.0) | – | – | 807 (46.02) | 1080 (92.26) | 317 (8.42) | 57 (8.75) |
| 8 primates ($k = 25$) | 2088 (85.62) | – | – | – | – | – | 914 (34.36) | 111 (34.36) |
| 8 primates ($k = 100$) | – | – | – | – | – | – | 756 (56.06) | 101 (61.68) |
| (43 + 7) humans ($k = 25$) | – | – | – | – | – | – | | 705 (69.77) |
| (43 + 7) humans ($k = 100$) | – | – | – | – | – | – | | 927 (70.21) |
| (93 + 7) humans ($k = 25$) | – | – | – | – | – | – | | 1383 (77.42) |

*Note*: Each cell shows the running time in minutes and the memory usage in parenthesis in gigabytes. TwoPaCo was run using just one round, with a Bloom filter size $b = 0.13$ GB for *E.coli*, 4.3 GB for 7 humans with $k = 25$, $b = 8.6$ GB with $k = 100$, $b = 34$ GB for primates, and $b = 69$ GB for (43 + 7) and larger human dataset. A dash in the SplitMem and bwt-based columns indicates that they ran out of memory, a dash in the Sibelia column indicates that it could not be run on such large inputs, a dash in the minia column indicates that it did not finish in 48 h, a dash in the BCALM column indicates that it ran out of disk space (4 TB). A double dash indicates that the software had a segmentation fault. An empty slot indicates that the experiment was not done.

To measure the parallel scalability of TwoPaCo, we fixed a dataset consisting of five simulated human genomes. Figure 2 shows scaling results for 1–32 worker threads. The first pass of Algorithm 2, and the conversion of junction vertices to the graph (as described in Section 3), scale almost linearly up to 16 threads. The second pass does not scale past four worker threads, due to what we believe is the limited parallel performance of the concurrent hash table, which we plan to improve in the future.

Next, we evaluated the performance of TwoPaCo under memory restrictions. For each run, we set a different memory threshold and checked how many rounds were necessary so that TwoPaCo did not exceed the threshold (Table 3). This experiment illustrates that TwoPaCo is capable of constructing the compacted graph for a dataset of five human genomes under memory restrictions commensurate with a low-end laptop.

For the benchmarks and real datasets in the experiments above, we recorded the number of marks that Algorithm 2 left in the array $C$ after each stage (Table 4). We did not record those numbers for the larger datasets due to the associated cost restrictions of re-running the larger experiments.

Our last experiment assessed the effects of the input size and structure (number of junctions and number of distinct $k$-mers) on running time and memory consumption (Fig. 3). As expected from the theoretical analysis, the running time depends both on the input size and structure, while memory consumption depends only on structure. For example, consider the dataset from Baier *et al.* (2015), which has highly similar genomes. As a result, the number of distinct $k$-mers and junctions is nearly constant even as the number of genomes increases. This dataset has the lowest running time, and the amount of memory TwoPaCo uses does not increase with the number of genomes. Unlike the memory usage, the running time does see a dominant effect of the input size, as the running time increases with the number of genomes for this dataset. On the other hand, consider the primates dataset, which is more variable and contains more distinct $k$-mers and junctions than the simulated human dataset. As a result, TwoPaCo takes a longer time and has larger memory consumption.

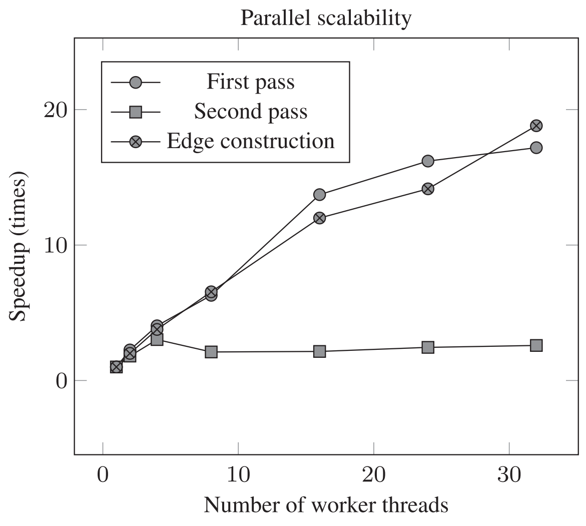

**Fig. 2.** Parallel speedup of the different parts of TwoPaCo. Edge constructions refers to the conversion of junction positions to the compacted graph, as described in Section 3. The Bloom filter was 8.58 GB and used eight internal hash functions. We set $k = 25$

## 9 Conclusion

In this article, we gave an efficient algorithm for constructing the compacted de Bruijn graph for a collection of complete genomic sequences. It is based on identifying the positions of the genome which correspond to vertices of the compacted graph. TwoPaCo works by narrowing down the set of candidates using a probabilistic data structure, in order to make the deterministic memory-intensive approach feasible. We note that the effectiveness of the algorithm relies on having whole genome sequences, making it inapplicable to the case when genomes are represented as shorts read fragments. Parallel speedup of the second pass of Algorithm 2 is an important direction of the future work that we are going to pursue.

A critical parameter of the TwoPaCo is the size of the Bloom filter ($b$). We recommend the user to set $b$ to be the maximum memory they wish to allocate to the algorithm. If the memory usage then exceeds $b$ (which would happen due to the size of the hash table), then the number of rounds should be increased until the memory usage falls below $b$. In future work, we plan to implement an algorithm to automatically select a value of $b$ that would minimize the maximum memory used by the algorithm. We also plan to automate the choice of the number of rounds, given a desired memory limit.

**Table 3.** The minimal number of rounds it takes for TwoPaCo to compress the graph without exceeding a given memory threshold, using five simulated human genomes

| Memory threshold | Used memory | Bloom filter size | Running time | Rounds |
|---|---|---|---|---|
| 10 | 8.62 | 8.59 | 259 | 1 |
| 8 | 6.73 | 4.29 | 434 | 3 |
| 6 | 5.98 | 4.29 | 539 | 4 |
| 4 | 3.51 | 2.14 | 665 | 6 |

*Note*: Memory quantities are in gigabytes and running times are in minutes. It was carried out on a machine with a Intel Xeon E7-8837 processor. We used $k = 25$ and ran the computation with eight worker threads. In each run, we used the largest possible Bloom filter size that fitted a given restriction (in our implementation the number of bits it has to be a power of two).

**Table 4.** Number of marks in the array *C* initially and after each pass of Algorithm 2

| Dataset | Initially (total positions) | First pass | Second pass |
|---|---|---|---|
| 62 *E.coli* ($k$=25) | 310 157 564 | 24 649 489 | 24 572 562 |
| 62 *E.coli* ($k$=100) | 310 157 489 | 22 848 018 | 9 492 091 |
| 7 humans ($k$=25) | 21 201 290 922 | 3 489 946 013 | 2 974 098 154 |
| 7 humans ($k$ = 100) | 21 201 290 847 | 1 374 287 870 | 188 224 214 |
| 8 primates ($k$=25) | 24 540 556 921 | 5 423 003 377 | 5 401 587 503 |
| 8 primates ($k$=100) | 24 540 556 846 | 1 174 160 336 | 502 441 107 |

The algorithm can also be used to construct a partially compacted graph by omitting the second pass of Algorithm 2. A partially compacted graph is one where some, but not necessarily all, of the non-branching paths have been compacted. Partially compacted graphs are faster to construct and can be useful in applications when the size of the graph is not critical or full compaction takes too much resources.

TwoPaCo makes significant progress in extending the number and size of genomes from which a compacted de Bruijn graph can be constructed. We believe that this progress will enable novel biological analyses of mammalian-sized genomes. In this article, we focus on the graph compaction algorithm only, and present a detailed analysis of its performance, both theoretical and experimental. However, future applications of the compacted de Bruijn graph is an exciting and important question. The de Bruijn graph can be the core of a tool to answer biological questions, however, such a tool must not only construct the graph but implement additional algorithms to analyze it. For example, the synteny block reconstruction tool Sibelia (Minkin *et al.*, 2013b) not only builds the compacted de Bruijn graph but also performs iterative graph simplification while increasing the value of *k*. Another very recent example is the use of the de Bruijn graph to construct the Burrows-Wheeler transform of many complete genomes (Liu *et al.*, 2016). TwoPaCo can also be useful in other applications, such as the representation of multiple reference genomes or variants between genomes. See Marschall *et al.* (2016) for a detailed discussion of applications of the de Bruijn graph in computational pan-genomics. Similar efforts are under way in the GA4GH technical group. TwoPaCo could be particularly useful on poorly assembled draft genomes, whose accurate alignment is especially challenging.

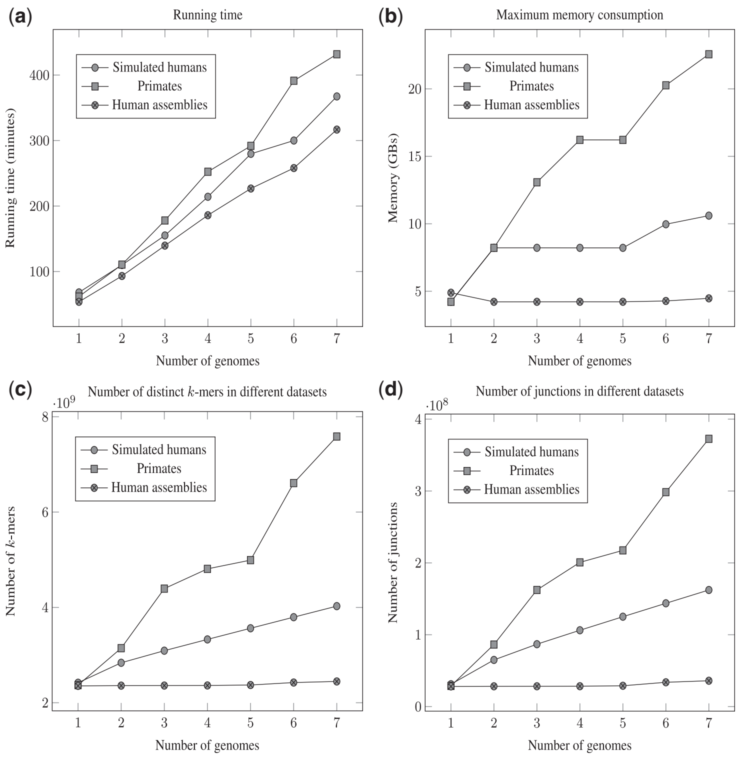

**Fig. 3.** Effects of the input length and structure on the memory and running time. Here we varied the number of input genomes from one to seven and recorded the running time (**a**) and memory usage (**b**). We also calculated the number of distinct *k*-mers (c) and junctions (d) in the input to illustrate their effect on the algorithm's performance. We used three datasets: simulated humans, primates, and seven human assemblies from Baier *et al.* (2015). The experiment was performed on a machine with a Intel Xeon E7-8837 processor. We used $k = 25$ and ran the computation with eight worker threads and a single round. For each run, we used the optimal Bloom filter size, i.e. the filter size that minimizes the maximum memory consumption. The number of distinct *k*-mers was computed using the KMC2 *k*-mer counter Deorowicz *et al.* (2015). In our implementation, the number of bits in the Bloom filter has to be a power of two, which leads to the non-smooth growth of the memory curve in (b)

## Acknowledgements

We would like to thank Daniel Lemire for modifying his hash function library (Lemire and Kaser, 2010) for the purpose of our algorithm.

## Funding

This work was supported by the National Science Foundation [DBI-1356529, CCF-1439057, IIS-1453527, IIS-1421908 to P.M.].

*Conflict of Interest*: None declared.

## References

1000 Genomes Project Consortium *et al.* (2010) A map of human genome variation from population-scale sequencing. *Nature*, **467**, 1061–1073.

Baier,U. *et al.* (2015) Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. *Bioinformatics*, **32**, 497–504.

Belk,K. *et al.* (2016) Succinct colored de Bruijn graphs. *bioRxiv*, doi: http://dx.doi.org/10.1101/040071.

Beller,T. and Ohlebusch,E. (2015) Efficient construction of a compressed de Bruijn graph for pan-genome analysis. In *Combinatorial Pattern Matching*. Springer, Cham, Switzerland, **9133**, pp. 40–51.

Beller,T. and Ohlebusch,E. (2016) A representation of a compressed de Bruijn graph for pan-genome analysis that enables search. *arXiv Preprint arXiv:1602.03333*, **11**, 1–17.

Ben-Bassat,I. and Chor,B. (2014) String graph construction using incremental hashing. *Bioinformatics*, **30**, 3515–3523.

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

Bowe,A. *et al.* (2012) Succinct de Bruijn graphs. In *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, pp. 225–235.

Bradley,P. *et al.* (2015) Rapid antibiotic-resistance predictions from genome sequence data for staphylococcus aureus and mycobacterium tuberculosis. *Nat. Commun.*, **6**, doi: 10.1038/ncomms10063.

Cazaux,B. *et al.* (2014) From indexing data structures to de Bruijn graphs. In *Combinatorial Pattern Matching*. Springer, Cham, Switzerland, **8486**, pp. 89–99.

Chikhi,R. and Rizk,G. (2013) Space-efficient and exact de Bruijn graph representation based on a bloom filter. *Algorithms Mol. Biol.*, **8**, 1.

Chikhi,R. *et al.* (2014) On the representation of de Bruijn graphs. In *Research in Computational Molecular Biology*. Springer, Cham, Switzerland, **8394**, pp. 35–55.

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

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

Dilthey,A. *et al.* (2015) Improved genome inference in the mhc using a population reference graph. *Nat. Genet.*, **47**, 682–688.

Earl,D. *et al.* (2014) Alignathon: a competitive assessment of whole-genome alignment methods. *Genome Res.*, **24**, 2077–2089.

Ernst,C. and Rahmann,S. (2013) Pancake: a data structure for pangenomes. In: Tim,B. *et al.* (eds.) *German Conference on Bioinformatics*, Vol. **34**, pp. 35–45.

Gibbs,R.A. *et al.* (2003) The international hapmap project. *Nature*, **426**, 789–796.

Gusfield,D. (1997) *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK.

Haussler,D. *et al.* (2008) Genome 10K: a proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J. Hered.*, **100**, 659–674.

Heo,Y. *et al.* (2014) BLESS: bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, **30**, 1354–1362

Holley,G. *et al.* (2015) Bloom filter trie–a data structure for pan-genome storage. In *Algorithms in Bioinformatics*. Springer, Berlin, Heidelberg, pp. 217–230.

Huang,L. *et al.* (2013) Short read alignment with populations of genomes. *Bioinformatics*, **29**, i361–i370.

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

Jarvis,E.D. *et al.* (2014) Whole-genome analyses resolve early branches in the tree of life of modern birds. *Science*, **346**, 1320–1331.

Killcoyne,S. and del Sol,A. (2014) FIGG: simulating populations of whole genome sequences for heterogeneous data analyses. *BMC Bioinformatics*, **15**, 149.

Koepfli,K.P. *et al.* (2015) The genome 10k project: a way forward. *Annu. Rev. Anim. Biosci*, **3**, 57–111.

Lee,C. *et al.* (2002) Multiple sequence alignment using partial order graphs. *Bioinformatics*, **18**, 452–464.

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

Limasset,A. *et al.* (2016) Read mapping on de Bruijn graphs. *BMC Bioinformatics*, **17**, 1–12.

Liu,B. *et al.* (2016) deBWT: parallel construction of Burrows–Wheeler Transform for large collection of genomes with de Bruijn-branch encoding. *Bioinformatics*, **32**, i174–i182.

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

Marschall,T. *et al.* (2016) Computational pan-genomics: status, promises and challenges. *bioRxiv*, doi: http://dx.doi.org/10.1101/043430.

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

Miller,J.R. *et al.* (2010) Assembly algorithms for next-generation sequencing data. *Genomics*, **95**, 315–327.

Minkin,I. *et al.* (2013a) C-sibelia: an easy-to-use and highly accurate tool for bacterial genome comparison. *F1000Research*, **2**, doi: 10.12688/f1000research.2-258.v1.

Minkin,I. *et al.* (2013b) Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, pp. 215–229.

Mitzenmacher,M. and Upfal,E. (2005) *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Cambridge, UK.

Oaks,S. and Wong,H. (2004) In Mike,L. and Debra,C. (eds.) *Java Threads*. O'Reilly Media, Sebastopol, CA.

Paten,B. *et al.* (2014) Mapping to a reference genome structure. *arXiv preprint arXiv:1404.5010*.

Pham,S.K. and Pevzner,P.A. (2010) DRIMM-Synteny: decomposing genomes into evolutionary conserved segments. *Bioinformatics*, **26**, 2509–2516.

Raphael,B. *et al.* (2004) A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Res.*, **14**, 2336–2346.

Reinders,J. (2007) *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Sebastopol, CA.

Rizk,G. *et al.* (2013) Dsk: k-mer counting with very low memory usage. *Bioinformatics*, **29**, 652–653.

Rozov,R. *et al.* (2014) Fast lossless compression via cascading Bloom filters. *BMC Bioinformatics*, **15**, 1.

Salikhov,K. *et al.* (2013) Using cascading Bloom filters to improve the memory usage for de Brujin graphs. In Darling, A. and Stoye, J. (eds) *Algorithms in Bioinformatics, Volume 8126 of Lecture Notes in Computer Science*. Berlin: Springer, pp. 364–376.

Salikhov,K. *et al.* (2014) Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms Mol. Biol.*, **9**, 1.

Schatz,M.C. *et al.* (2010) Assembly of large genomes using second-generation sequencing. *Genome Res.*, **20**, 1165–1173.

Simpson,J.T. and Durbin,R. (2010) Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, **26**, i367–i373.

Sirén,J. *et al.* (2014) Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, **11**, 375–388.

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

Wang,M. *et al.* (2012) A de Bruijn graph approach to the quantification of closely-related genomes in a microbial community. *J. Comput. Biol.*, **19**, 814–825.

Ye,Y. and Tang,H. (2015) Utilizing de Bruijn graph of metagenome assembly for metatranscriptome analysis. *Bioinformatics*, **32**, 1001–1008.