`Minia` — Short manual

R. Chikhi & G. Rizk
rayan.chikhi@ens-cachan.org

July 3, 2013

Abstract

Minia is a software for ultra-low memory DNA sequence assembly. It takes as input a set of short genomic sequences (typically, data produced by the Illumina DNA sequencer). Its output is a set of contigs (assembled sequences), forming an approximation of the expected genome. Minia is based on a succinct representation of the de Bruijn graph. The computational resources required to run Minia are significantly lower than that of other assemblers.

Contents

1 Installation

To install Minia, just type `make` in the Minia folder. Minia has been tested on Linux and MacOS systems. To run Minia, type `./minia`.

2 Parameters

The following options need to be specified, in the following order:

1. `input_file` – the input file

2. `kmer_size` – k-mer length
3. `min_abundance` – filters out k-mers seen less than the specified number of times
4. `estimated_genome_size` – rough estimation of the size of the genome to assemble, in base pairs.
5. `prefix` – any prefix string to store unique temporary files for this assembly

Minia now uses the Cascading Bloom filters improvement (http://arxiv.org/abs/1302.7278) by default, thanks to Gustavo Sacomoto for the implementation in Minia. Launch Minia with the `--original` option to revert to the original data structure.

3 Explanation of parameters

`kmer_size` The k-mer length is the length of the nodes in the de Bruijn graph. It strongly depends on the input dataset. A typical value to try for short Illumina reads (read length above 50) is 27. For longer Illumina reads ($\approx$ 100 bp) with sufficient coverage ($>$ 40x), we had good results with $k = 43$.

`min_abundance` The `min_abundance` is used to remove erroneous, low-abundance k-mers. This parameter also strongly depends on the dataset. It corresponds to the smallest amount of times a correct k-mer appears in the reads. A typical value is 3. Setting it to 1 is not recommended[1]. If the dataset has high coverage, try larger values.

`estimated_genome_size` The estimated genome size parameter only controls the memory usage during the first phase of Minia (graph construction). *It has no impact on the assembly.*

`prefix` The `prefix` parameter is any arbitrary file name prefix, for example, `test_assembly`.

4 Input

FASTA/FASTQ

Minia assembles any type of Illumina reads, given in the FASTA or FASTQ format. Paired or mate-pairs reads are OK, but keep in mind that Minia discards pairing information.

Multipe Files

Minia can assemble reads from multiple input files. The list of read files is specified in the input file of Minia (the first parameter), one file name per line. Therefore the input file is either (i) the read file itself (in fasta or fastq) or (ii) a file containing a list of file names.

[1] as no erroneous k-mer will be discarded, which will likely result in a very large memory usage

line format

In FASTA files, each read can be split into multiple lines, whereas in FASTQ, each read sequence must be in a single line.

gzip compression

Minia can direclty read files compressed with gzip. Compressed files should end with '.gz'. Input files of different types can be mixed (i.e. gzipped or not, in FASTA or FASTQ)

5 Output

The output of Minia is a set of contigs in the FASTA format, in the file `[prefix].contigs.fa`.

6 Memory usage

We estimate that the memory usage of Minia is roughly 2 GB of RAM per gigabases in the target genome to assemble. It is independent of the coverage of the input dataset, provided that the `min_abundance` parameter is correctly set. For example, a human genome was assembled in 5.7 GB of RAM. This was using the original data structure; the current implementation relies on Cascading Bloom filters and should use $\approx 1-2$ GB less memory. A better estimation of the memory usage can be found in the Appendix.

7 Larger k-mer lengths

Minia supports arbitrary large k-mer lengths. To compile Minia for k-mer lengths up to, say, 100, type:

```
make clean && make k=100
```

8 Appendixes

The rest of this manual describes the data structure used by Minia. The first text is from an original research article published at WABI 2012. The second text is an improvement made and implemented by other authors, published at WABI 2013.

Space-efficient and exact de Bruijn graph representation based on a Bloom filter

Rayan Chikhi[1] and Guillaume Rizk[2]

[1] Computer Science department, ENS Cachan/IRISA, 35042 Rennes, France
[2] Algorizk, 75013 Paris, France

Abstract. The de Bruijn graph data structure is widely used in next-generation sequencing (NGS). Many programs, e.g. *de novo* assemblers, rely on in-memory representation of this graph. However, current techniques for representing the de Bruijn graph of a human genome require a large amount of memory ($\geq$ 30 GB).
We propose a new encoding of the de Bruijn graph, which occupies an order of magnitude less space than current representations. The encoding is based on a Bloom filter, with an additional structure to remove critical false positives. An assembly software implementing this structure, Minia, performed a complete *de novo* assembly of human genome short reads using 5.7 GB of memory in 23 hours.

1 Introduction

The de Bruijn graph of a set of DNA or RNA sequences is a data structure which plays an increasingly important role in next-generation sequencing applications. It was first introduced to perform *de novo* assembly of DNA sequences [**?**]. It has recently been used in a wider set of applications: *de novo* mRNA [**?**] and metagenome [**?**] assembly, genomic variants detection [**?**,**?**] and *de novo* alternative splicing calling [**?**]. However, an important practical issue of this structure is its high memory footprint for large organisms. For instance, the straightforward encoding of the de Bruijn graph for the human genome ($n \approx 2.4 \cdot 10^9$, k-mer size $k = 27$) requires 15 GB ($n \cdot k/4$ bytes) of memory to store the nodes sequences alone. Graphs for much larger genomes and metagenomes cannot be constructed on a typical lab cluster, because of the prohibitive memory usage.

Recent research on de Bruijn graphs has been targeted on designing more lightweight data structures. Li *et al.* pioneered minimum-information de Bruijn graphs, by not recording read locations and paired-end information [**?**]. Simpson *et al.* implemented a distributed de Bruijn graph to reduce the memory usage per node [**?**]. Conway and Bromage applied sparse bit array structures to store an implicit, immutable graph representation [**?**]. Targeted methods compute local assemblies around sequences of interest, using negligible memory, with greedy extensions [**?**] or portions of the de Bruijn graph [**?**]. Ye *et al.* recently showed that a graph roughly equivalent to the de Bruijn graph can be obtained by storing only one out of g nodes ($10 \leq g \leq 25$) [**?**].

Conway and Bromage observed that the self-information of the edges is a lower bound for exactly encoding the de Bruijn graph [**?**]:

$$\log_2(\binom{4^{k+1}}{|E|}) \text{ bits},$$

where $k+1$ is the length of the sequence that uniquely defines an edge, and $|E|$ is the number of edges. In this article, we will consider for simplicity that a de Bruijn graph is fully defined by its nodes. A similar lower bound can then be derived from the self-information of the nodes. For a human genome graph, the self-information of $|N| \approx 2.4 \cdot 10^9$ nodes is $\log_2(\binom{4^k}{|N|}) \approx 6.8$ GB for $k = 27$, i.e. ≈ 24 bits per node.

A very recent article [**?**] from Pell *et al.* introduced the *probabilistic de Bruijn graph*, which is a de Bruijn graph stored as a Bloom filter (described in the next section). It is shown that the graph can be encoded with as little as 4 bits per node. An important drawback of this representation is that the Bloom filter introduces false nodes and false branching. However, they observe that the global structure of the graph is approximately preserved, up to a certain false positive rate. Pell *et al.* did not perform assembly directly by traversing the probabilistic graph. Instead, they use the graph to partition the set of reads into smaller sets, which are then assembled in turns using a classical assembler. In the arXiv version of [**?**] (Dec 2011), it is unclear how much memory is required by the partitioning algorithm.

In this article, we focus on encoding an exact representation of the de Bruijn graph that efficiently implements the following operations:

1. For any node, enumerate its neighbors
2. Sequentially enumerate all the nodes

The first operation requires random access, hence is supported by a structure stored in memory. Specifically, we show in this article that a probabilistic de Bruijn graph can be used to perform the first operation exactly, by recording a set of troublesome false positives. The second operation can be done with sequential access to the list of nodes stored on disk. One highlight of our scheme is that the resulting memory usage is approximated by

$$1.44 \log_2(\frac{16k}{2.08}) + 2.08 \text{ bits}/k\text{-mer}.$$

For the human genome example above and $k = 27$, the size of the structure is 3.7 GB, i.e. 13.2 bits per node. This is effectively below the self-information of the nodes. While this may appear surprising, this structure does not store the precise set of nodes in memory. In fact, compared to a classical de Bruijn graph, the membership of an arbitrary node cannot be efficiently answered by this representation. However, for the purpose of many applications (e.g. assembly), these membership queries are not needed.

We apply this representation to perform *de novo* assembly by traversing the graph. In our context, we refer by traversal to any algorithm which visits all the

nodes of the graph exactly once (e.g. a depth-first search algorithm). Thus, a mechanism is needed to mark which nodes have already been visited. However, nodes of a probabilistic de Bruijn graph cannot store additional information. We show that recording only the visited complex nodes (those with in-degree or out-degree different than one) is a space-efficient solution. The combination of (i) the probabilistic de Bruijn graph along with the set of critical false positives, and (ii) the marking scheme, enables to perform very low-memory *de novo* assembly.

In the first Section, the notions of de Bruijn graphs and Bloom filters are formally defined. Section 3 describes our scheme for exactly encoding the de Bruijn graph using a Bloom filter. Section 4 presents a solution for traversing our representation of the de Bruijn graph. Section 6 presents two experimental results: (i) an evaluation of the usefulness of removing false positives and (ii) an assembly of a real human dataset using an implementation of the structure. A comparison is made with other recent assemblers based on de Bruijn graphs.

2 de Bruijn graphs and Bloom filters

The **de Bruijn graph** [**?**], for a set of strings S, is a directed graph. For simplicity, we adopt a node-centric definition. The nodes are all the k-length substrings (also called k-*mers*) of each string in S. An edge $s_1 \rightarrow s_2$ is present if the $(k-1)$-length suffix of s_1 is also a prefix of s_2. Throughout this article, we will indifferently refer to a node and its k-mer sequence as the same object.

A more popular, edge-centric definition of de Bruijn graphs requires that edges reflect consecutive nodes. For k'-mer nodes, an edge $s_1 \rightarrow s_2$ is present if there exists a $(k'+1)$-mer in a string of S containing s_1 as a prefix and s_2 as a suffix. The node-centric and edge-centric definitions are essentially equivalent when $k' = k-1$ (although in the former, nodes have length k, and $k-1$ in the latter).

The **Bloom filter** [**?**] is a space efficient hash-based data structure, designed to test whether an element is in a set. It consists of a bit array of m bits, initialized with zeros, and h hash functions. To insert or test the membership of an element, h hash values are computed, yielding h array positions. The insert operation corresponds to setting all these positions to 1. The membership operation returns *yes* if and only if all of the bits at these positions are 1. A *no* answer means the element is definitely not in the set. A *yes* answer indicates that the element may or may not be in the set. Hence, the Bloom filter has one-sided errors. The probability of false positives increases with the number of elements inserted in the Bloom filter. When considering hash functions that yield equally likely positions in the bit array, and for large enough array size m and number of inserted elements n, the false positive rate $\mathcal{F}$ is [**?**]:

$$\mathcal{F} \approx \left(1 - e^{-hn/m}\right)^h = \left(1 - e^{-h/r}\right)^h \tag{1}$$

where $r = m/n$ is the number of bits per element. For a fixed ratio r, minimizing Equation 1 yields the optimal number of hash functions $h \approx 0.7r$, for which $\mathcal{F}$ is

approximately 0.6185^r. Solving Equation 1 for m, assuming that h is the optimal number of hash function, yields $m \approx 1.44 \log_2(\frac{1}{\mathcal{F}})n$.

The **probabilistic de Bruijn graph** is obtained by inserting all the nodes of a de Bruijn graph (i.e all k-mers) in a Bloom filter [**?**]. Edges are implicitly deduced by querying the Bloom filter for the membership of all possible extensions of a k-mer. Specifically, an *extension* of a k-mer v is the concatenation of either (i) the $k-1$ suffix of v with one of the four possible nucleotides, or (ii) one of the four nucleotides with the $k-1$ prefix of v. The probabilistic de Bruijn graph holds an over-approximation of the original de Bruijn graph. Querying the Bloom filter for the existence of an arbitrary node may return a false positive answer (but never a false negative). This introduces false branching between original and false positive nodes.

3 Removing critical false positives

3.1 The *cFP* structure

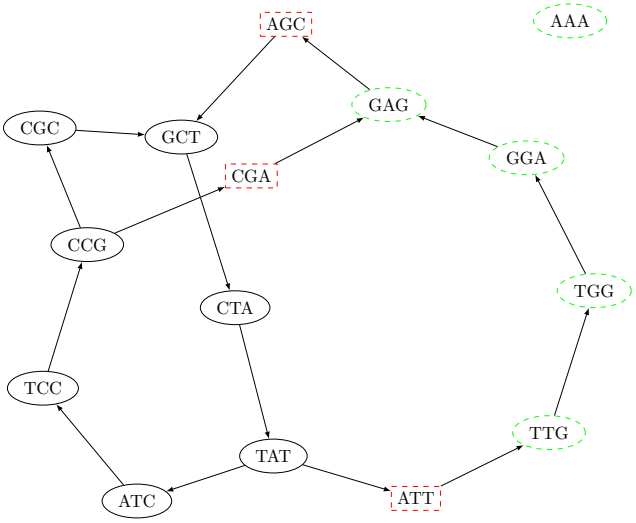
Our contribution is a mechanism that avoids false branching. Specifically, we propose to detect and store false positive elements which are responsible for false branching, in a separate structure. To this end, we introduce the cFP structure of *critical False Positives* k-mers, implemented with a standard set allowing fast membership test. Each query to the Bloom filter is modified such that the *yes* answer is returned if and only if the Bloom filter answers *yes* and the element is not in cFP.

Naturally, if cFP contained all the false positives elements, the benefits of using a Bloom filter for memory efficiency would be lost. The key observation is that the k-mers which will be queried when traversing the graph are not *all* possible k-mers. Let $\mathcal{S}$ be the set of true positive nodes, and $\mathcal{E}$ be the set of extensions of nodes from $\mathcal{S}$. Assuming we only traverse the graph by starting from a node in $\mathcal{S}$, false positives that do not belong to $\mathcal{E}$ will never be queried. Therefore, the set cFP will be a subset of $\mathcal{E}$. Let $\mathcal{P}$ be the set of all elements of $\mathcal{E}$ for which the Bloom filter answers *yes*. The **set of critical false positives *cFP*** is then formally defined as $cFP = \mathcal{P} \setminus \mathcal{S}$.

Figure 1 shows a simple graph with the set $\mathcal{S}$ of correct nodes in regular circles and cFP in dashed rectangles. The exact representation of the graph is therefore made of two data structures: the Bloom filter, and the set cFP of critical false positives. The set cFP can be constructed using an algorithm that limits its memory usage, e.g. to the size of the Bloom filter. The set $\mathcal{P}$ is created on disk, from which cFP is then gradually constructed by iteratively filtering $\mathcal{P}$ with partitions of $\mathcal{S}$ loaded in a hash-table.

3.2 Dimensioning the Bloom filter for minimal memory usage

The set cFP grows with the number of false positives. To optimize memory usage, a trade-off between the sizes of the Bloom filter and cFP is studied here.



(a)

$a_1 \ldots a_k$	$\sum_{i=1}^{k} a_i^i \bmod 10$
ATC	0
CCG	0
TCC	5
CGC	6
...	...

(b)

Bloom filter
1
0
0
0
0
1
1
0
0
0

(c)

Nodes self-information:

$$\lceil log_2 \binom{4^3}{7} \rceil = 30 \text{ bits}$$

Structure size:

$$\underbrace{10}_{\text{Bloom}} + \underbrace{3 \cdot 6}_{\text{False positives}} = 28 \text{ bits}$$

(d)

Fig. 1: A complete example of removing false positives in the probabilistic de Bruijn graph. (a) shows $\mathcal{S}$, an example de Bruijn graph (the 7 non-dashed nodes), and $\mathcal{B}$, its probabilistic representation from a Bloom filter (taking the union of all nodes). Dashed rectangular nodes (in red in the electronic version) are immediate neighbors of $\mathcal{S}$ in $\mathcal{B}$. These nodes are the critical false positives. Dashed circular nodes (in green) are all the other nodes of $\mathcal{B}$; (b) shows a sample of the hash values associates to the nodes of $\mathcal{S}$ (a toy hash function is used); (c) shows the complete Bloom filter associated to $\mathcal{S}$; incidentally, the nodes of $\mathcal{B}$ are exactly those to which the Bloom filter answers positively; (d) describes the lower bound for exactly encoding the nodes of $\mathcal{S}$ (self-information) and the space required to encode our structure (Bloom filter, 10 bits, and 3 critical false positives, 6 bits per 3-mer).

Using the same notations as in the definition of the Bloom filter, given that $n = |\mathcal{S}|$, the size of the filter m and the false positive rate $\mathcal{F}$ are related through Equation 1. The expected size of cFP is $8n \cdot \mathcal{F}$, since each node only has eight possible extensions, which might be false positives. In the encoding of cFP, each k-mer occupies $2 \cdot k$ bits. Recall that for a given false positive rate $\mathcal{F}$, the expected optimal Bloom filter size is $1.44n \log_2(\frac{1}{\mathcal{F}})$. The total structure size is thus expected to be

$$\underbrace{1.44n \log_2 \left(\frac{1}{\mathcal{F}}\right)}_{\text{Bloom filter}} + \underbrace{(16 \cdot \mathcal{F}nk)}_{cFP} \text{ bits} \tag{2}$$

The size is minimal for $\mathcal{F} \approx (16k/2.08)^{-1}$. Thus, the minimal number of bits required to store the Bloom filter and the set cFP is approximately

$$n \cdot (1.44 \log_2(\frac{16k}{2.08}) + 2.08). \tag{3}$$

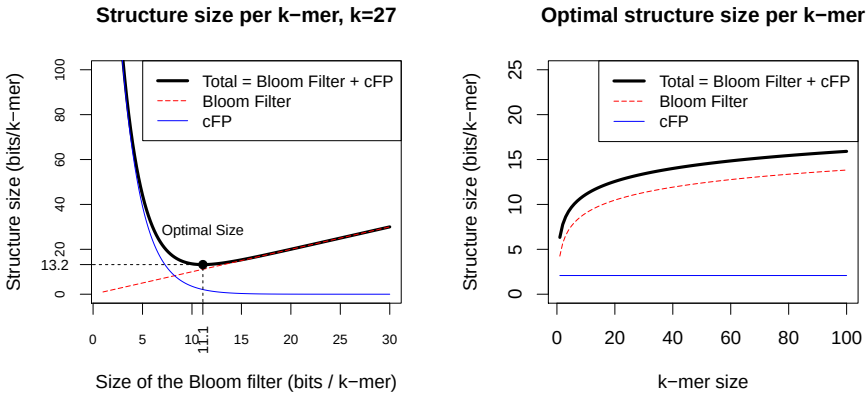
For illustration, Figure 2-(a) shows the size of the structure for various Bloom filter sizes and $k = 27$. For this value of k, the optimal size of the Bloom filter is 11.1 bits per k-mer, and the total structure occupies 13.2 bits per k-mer. Figure 2-(b) shows that k has only a modest influence on the optimal structure size. Note that the size of the cFP structure is in fact independent of k.

In comparison, a Bloom filter with virtually no critical false positives would require $\mathcal{F} \cdot 8n < 1$, i.e. $r > 1.44 \log_2(8n)$. For a human genome ($n = 2.4 \cdot 10^9$), r would be greater than 49.2, yielding a Bloom filter of size 13.7 GB.

4 Additional marking structure for graph traversal

Many NGS applications, e.g. *de novo* assembly of genomes [**?**] and transcriptomes [**?**], and *de novo* variant detection [**?**], rely on (i) simplifying and (ii) traversing the de Bruijn graph. However, the graph as represented in the previous section neither supports (i) simplifications (as it is immutable) nor (ii) traversals (as the Bloom filter cannot store an additional visited bit per node). To address the former issue, we argue that the simplification step can be avoided by designing a slightly more complex traversal procedure [**?**].

We introduce a novel, lightweight mechanism to record which portions of the graph have already been visited. The idea behind this mechanism is that not every node needs to be marked. Specifically, nodes that are inside simple paths (i.e nodes having an in-degree of 1 and an out-degree of 1) will either be all marked or all unmarked. We will refer to nodes having their in-degree or out-degree different to 1 as *complex* nodes. We propose to store marking information of complex nodes, by explicitly storing complex nodes in a separate hash table. In de Bruijn graphs of genomes, the complete set of nodes dwarfs the set of complex nodes, however the ratio depends on the genome complexity [**?**].



(a) (b)

Fig. 2: (a) Structure size (Bloom filter, critical false positives) in function of the number of bits per k-mer allocated to the Bloom filter (also called ratio r) for $k = 32$. The trade-off that optimizes the total size is shown in dashed lines. (b) Optimal size of the structure for different values of k.

The memory usage of the marking structure is $n_c C$, where n_c is the number of complex nodes in the graph and C is the memory usage of each entry in the hash table ($C \approx 2k + 8$).

5 Implementation

The de Bruijn graph structure described in this article was implemented in a new *de novo* assembly software: Minia[3]. An important preliminary step is to retrieve the list of distinct k-mers that appear in the reads, i.e. true graph nodes. To discard likely sequencing errors, only the k-mers which appear at least d times are kept (*solid* k-mers). We experimentally set d to 3. Classical methods that retrieve solid k-mers are based on hash tables [**?**], and their memory usage scale linearly with the number of distinct k-mers. To avoid using more memory than the whole structure, we implemented a constant-memory k-mer counting procedure (manuscript in preparation). To deal with reverse-complementation, k-mers are identified to their reverse-complements.

We implemented in Minia a graph traversal algorithm that constructs a set of contigs (gap-less sequences). The Bloom filter and the cFP structure are used to determine neighbors of each node. The marking structure records already traversed nodes. A bounded-depth, bounded-breadth BFS algorithm (following Property 2 in [**?**]) is performed to traverse short, locally complex regions. Specifically, the traversal ignores tips (dead-end paths) shorter than $2k + 1$ nodes.

[3] Source code available at http://minia.genouest.org/

It chooses a single path (consistently but arbitrarily), among all possible paths that traverse graph regions of breadth ≤ 20, provided these regions end with a single node of depth ≤ 500. These regions are assumed to be sequencing errors, short variants or short repetitions of length ≤ 500 bp. The breadth limit prevents combinatorial blowup. Note that paired-end reads information is not taken into account in this traversal. In a typical assembly pipeline (e.g. [**?**]), a separate program (*scaffolder*) can be used to link contigs using pairing information.

6 Results

Throughout the Results section, we will refer to the N50 metric of an assembly as the longest contig size, such that half the assembly is contained in contigs longer than this size.

6.1 On the usefulness of removing critical false positives

To test whether the combination of the Bloom filter and the cFP structure offers an advantage over a plain probabilistic de Bruijn graph, we compared both structures in terms of memory usage and assembly consistency. We retrieved 20 million *E. coli* short reads from the Short Read Archive (SRX000429), and discarded pairing information. Using this dataset, we constructed the probabilistic de Bruijn graph, the cFP structure, and marking structure, for various Bloom filter sizes (ranging from 5 to 19 bits per k-mer) and $k = 23$ (yielding 4.7 M solid k-mers).

We measured the memory usage of both structures. For each, we performed an assembly using Minia with exactly the same traversal procedure. The assemblies were compared to a reference assembly (using MUMmer), made with an exact graph. The percentage of nucleotides in contigs which aligned to the reference assembly was recorded.

Figure 3 shows that both the probabilistic de Bruijn graph and our structure have the same optimal Bloom filter size (11 bits per k-mer, total structure size of 13.82 bits and 13.62 per k-mer respectively). In the case of the probabilistic de Bruijn graph, the marking structure is prominent. This is because the graph has a significant amount of complex k-mers, most of them are linked to false positive nodes. For the graph equipped with the cFP structure, the marking structure only records the actual complex nodes; it occupies consistently 0.49 bits per k-mer. Both structures have comparable memory usage.

However, Figure 3 shows that the probabilistic de Bruijn graph produces assemblies which strongly depend on the Bloom filter size. Even for large sizes, the probabilistic graph assemblies differ by more than 3 Kbp to the reference assembly. We observed that the majority of these differences were due to missing regions in the probabilistic graph assemblies. This is likely caused by extra branching, which shortens the lengths of some contigs (contigs shorter than 100 bp are discarded).

Below ≈ 9 bits per k-mer, probabilistic graph assemblies significantly deteriorate. This is consistent with another article [**?**], which observed that when the

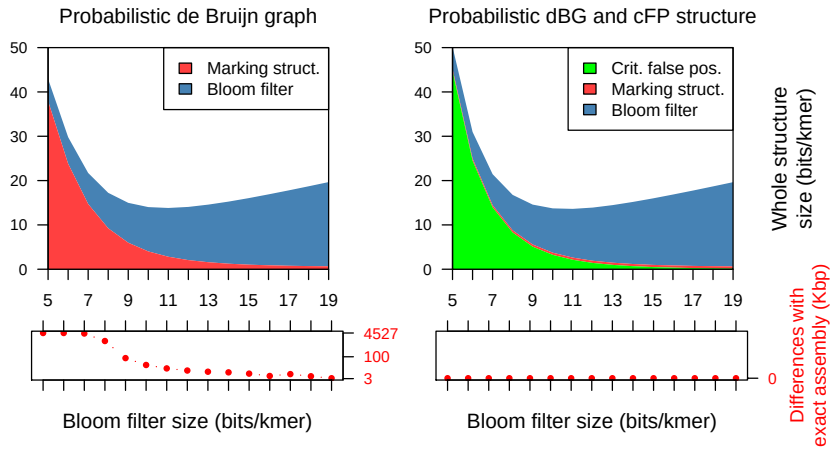


Fig. 3: Whole structures size (Bloom filter, marking structure, and cFP if applicable) of the probabilistic de Bruijn graph with (top right) and without the cFP structure (top left), for an actual dataset (E. coli, $k = 23$). All plots are in function of the number of bits per k-mer allocated to the Bloom filter. Additionally, the difference is shown (bottom left and bottom right) between a reference assembly made using an exact de Bruijn graph, and an assembly made with each structure.

false positive rate is over 18% (i.e., the Bloom filter occupies ≤ 4 bits per k-mer), distant nodes in the original graph become connected in the probabilistic de Bruijn graph. To sum up, assemblies produced by the probabilistic de Bruijn graph are prone to randomness, while those produced by our structure are exact.

6.2 *de novo* assembly

We assembled a complete human genome (NA18507, SRA:SRX016231, 142.3 Gbp of unfiltered reads of length ≈ 100 bp, representing 47x coverage) using Minia. After k-mer counting, 2,712,827,800 solid k-mers ($d = 3$) were inserted in a Bloom filter dimensioned to 11.1 bits per solid k-mer. The cFP structure contained 78,762,871 k-mers, which were stored as a sorted list of 64 bits integers, representing 1.86 bits per solid k-mer. A total of 166,649,498 complex k-mers (6% of the solid k-mers) were stored in the marking structure using 4.42 bits per solid k-mer (implementation uses $8\lceil \frac{k}{32} \rceil$ bytes per k-mer). Table 1 shows the time and memory usage required for each step in Minia.

We compared our results with assemblies reported by the authors of ABySS [**?**], SOAPdenovo [**?**], and the prototype assembler from Conway and Bromage [**?**]. Table 2 shows the results for four classical assembly quality metrics, and the time and peak memory usage of the compared programs. We note that Minia has the lowest memory usage (5.7 GB), seconded by the assembler from Conway and Bromage (32 GB). The wall-clock execution time of Minia (23 h) is comparable to the other assemblers; note that it is the only single-threaded assembler. The N50 metric of our assembly (1.2 Kbp) is slightly above that of the other assemblies (seconded by SOAPdenovo, 0.9 Kbp). All the programs except one assembled 2.1 Gbp of sequences.

We furthermore assessed the accuracy of our assembly by aligning the contigs produced by Minia to the GRCh37 human reference using GASSST [**?**]. Out of the 2,090,828,207 nucleotides assembled, 1,978,520,767 nucleotides (94.6%) were contained in contigs having a full-length alignment to the reference, with at least 98% sequence identity. For comparison, 94.2% of the contigs assembled by ABySS aligned full-length to the reference with 95% identity [**?**].

To test another recent assembler, SparseAssembler [**?**], the authors assembled another dataset (NA12878), using much larger effective k values. SparseAssembler stores an approximation of the de Bruijn graph, which can be compared to a classical graph for $k' = k + g$, where g is the sparseness factor. The reported assembly of the NA12878 individual by SparseAssembler ($k + g = 56$) has a N50 value of 2.1 Kbp and was assembled using 26 GB of memory, in a day. As an attempt to perform a fair comparison, we increased the value of k from 27 to 51 for the assembly done in Table 2 ($k = 56$ showed worse contiguity). The N50 obtained by Minia (2.0 Kbp) was computed with respect to the size of SparseAssembler assembly. Minia assembled this dataset using 6.1 GB of memory in 27 h, a 4.2× memory improvement compared to SparseAssembler.

Step	Time (h)	Memory (Gb)
k-mer counting	11.1	Constant (set to 4.0)
Enumerating positive extensions	2.8	3.6 (Bloom filter)
Constructing cFP	2.9	Constant (set to 4.0)
Assembly	6.4	5.7 (Bloom f.+ cFP + mark. struct.)
Overall	23.2	5.7

Table 1: Details of steps implemented in Minia, with wall-clock time and memory usage for the human genome assembly. For constant-memory steps, memory usage was automatically set to an estimation of the final memory size. In all steps, only one CPU core was used.

Method	**Minia**	C. & B.	ABySS	SOAPdenovo
Value of k chosen	27	27	27	25
Number of contigs (M)	3.49	7.69	4.35	-
Longest contig (Kbp)	18.6	22.0	15.9	-
Contig N50 (bp)	1156	250	870	886
Sum (Gbp)	2.09	1.72	2.10	2.08
Nb of nodes/cores	1/1	1/8	21/168	1/16
Time (wall-clock, h)	23	50	15	33
Memory (sum of nodes, GB)	**5.7**	**32**	**336**	**140**

Table 2: *de novo* human genome (NA18507) assemblies reported by our assembler (Minia), Conway and Bromage assembler [**?**], ABySS [**?**], and SOAPdenovo [**?**]. Contigs shorter than 100 bp were discarded. Assemblies were made without any pairing information.

7 Discussion

This article introduces a new, space-efficient representation of the de Bruijn graph. The graph is implicitly encoded as a Bloom filter. A subset of false positives, those which introduce false branching from true positive nodes, are recorded in a separate structure. A new marking structure is introduced, in order for any traversal algorithm to mark which nodes have already been visited. The marking structure is also space-efficient, as it only stores information for a subset of k-mers. Combining the Bloom filter, the critical false positives structure and the marking structure, we implemented a new memory-efficient method for *de novo* assembly (Minia).

To the best of our knowledge, Minia is the first method that can create contigs for a complete human genome on a desktop computer. Our method improves the memory usage of de Bruijn graphs by two orders of magnitude compared to ABySS and SOAPdenovo, and by roughly one order of magnitude compared to succinct and sparse de Bruijn graph constructions. Furthermore, the current implementation completes the assembly in 1 day using a single thread.

De Bruijn graphs have more NGS applications than just *de novo* assembly. We plan to port our structure to replace the more expensive graph representations in two pipelines for reference-free alternative splicing detection, and SNP detection [**?**,**?**]. We wish to highlight three directions for improvement. First, some steps of Minia could be implemented in parallel, e.g. graph traversal. Second, a more succinct structure can be used to mark complex k-mers. Two candidates are Bloomier filters [**?**] and minimal perfect hashing.

Third, the set of critical false positives could be reduced, by exploiting the nature of the traversal algorithm used in Minia. The traversal ignores short tips, and in general, graph regions that are eventually unconnected. One could then define n-th order critical false positives (n-cFP) as follows. An extension of a true positive graph node is a n-cFP if and only if a breadth-first search from the true positive node, in the direction of the extension, has at least one node of depth $n + 1$. In other words, false positive neighbors of the original graph which are part of tips, and generally local dead-end graph structures, will not be flagged as critical false positives. This is an extension of the method presented in this article which, in this notation, only detects 0-th order critical false positives.

Acknowledgments

The authors are grateful to Dominique Lavenier for helpful discussions and advice, and Aurélien Rizk for proof-reading the manuscript. This work benefited from the ANR grant associated with the MAPPI project (2010-2014).

Using cascading Bloom filters to improve the memory usage for de Brujin graphs

Kamil Salikhov[1], Gustavo Sacomoto[2,3], and Gregory Kucherov[4,5]

[1] Lomonosov Moscow State University, Moscow, Russia, salikhov.kamil@gmail.com
[2] INRIA Grenoble Rhône-Alpes, France, gustavo.sacomoto@inria.fr
[3] Laboratoire Biométrie et Biologie Evolutive, Université Lyon 1, Lyon, France
[4] Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel
[5] Laboratoire d'Informatique Gaspard Monge, Université Paris-Est & CNRS, Marne-la-Vallée, Paris, France, Gregory.Kucherov@univ-mlv.fr

Abstract. De Brujin graphs are widely used in bioinformatics for processing next-generation sequencing (NGS) data. Due to the very large size of NGS datasets, it is essential to represent de Bruijn graphs compactly, and several approaches to this problem have been proposed recently. In this work, we show how to reduce the memory required by the algorithm of Chikhi and Rizk (WABI, 2012) that represents de Brujin graphs using Bloom filters. Our method requires 30% to 40% less memory with respect to their method, with insignificant impact to construction time. At the same time, our experiments showed a better query time compared to their method. This is, to our knowledge, the best *practical* representation for de Bruijn graphs.

1 Introduction

Modern next-generation sequencing (NGS) technologies generate huge volumes of short nucleotide sequences (*reads*) drawn from the DNA sample under study. The length of a read varies from 35 to about 400 base pairs (letters) and the number of reads may be hundreds of millions, thus the total volume of data may reach tens or even hundreds of Gb.

Many computational tools dealing with NGS data, especially those devoted to *genome assembly*, are based on the concept of a *de Bruijn graph*, see e.g. [8]. The nodes of the de Bruijn graph[1] are all distinct k-*mers* occurring in the reads, and two k-*mers* are linked by an arc if there is a suffix-prefix overlap of size $k-1$. The value of k is an open parameter, that in practice is chosen between 20 and 64. The idea of using de Bruijn graph for genome assembly goes back to the "pre-NGS era" [11]. Note, however, that *de novo* genome assembly is not the only application of those graphs when dealing with NGS data. There are several

[1] Note that this actually a *subgraph* of the de Bruijn graph under its classical combinatorial definition. However, we still call it de Bruijn graph to follow the terminology common to the bioinformatics literature.

others, including: *de novo* transcriptome assembly [5] and *de novo* alternative splicing calling [14] from transcriptomic NGS data (RNA-seq); metagenome assembly [10] from metagenomic NGS data; and genomic variant detection [6] from genomic NGS data using a reference genome.

Due to the very large size of NGS datasets, it is essential to represent de Bruijn graphs as compactly as possible. This has been a very active line of research. Recently, several papers have been published that propose different approaches to compressing de Bruijn graphs [4,15,3,2,9].

Conway and Bromage [4] proposed a method based on classical succinct data structures, i.e. bitmaps with efficient rank/select operations. On the same direction, Bowe *et al.* [2] proposed a very interesting succinct representation that, assuming only one string (read) is present, uses only $4m$ bits, where m is the number of arcs in the graph. The more realistic case, where there are M reads, can be easily reduced to the one string case by concatenating all M reads using a special separator character. However, in this case the size of the structure is $4m + O(M \log m)$ bits ([2], Theorem 1). Since the multiplicative constant of the second term is hidden by the asymptotic notation, it is hard to know precisely what would be the size of this structure in practice.

Ye at al. [15] proposed a different method based on a sparse representation of de Bruijn graphs, where only a subset of k-mers present in the dataset are stored. Pell et al. [9] proposed a method to represent it approximately, the so called *probabilistic de Bruijn graph*. In their representation a node have a small probability to be a false positive, i.e. the k-mer is not present in the dataset. Finally, Chikhi and Rizk [3] improved Pell's scheme in order to obtain an exact representation of the de Bruijn graph. This was, to our knowledge, the best *practical* representation of an exact de Bruijn graph.

In this work, we focus on the method proposed in [3] which is based on Bloom filters. They were first used in [9] to provide a very space-efficient representation of a subset of a given set (in our case, a subset of k-mers), at the price of allowing *one-sided errors*, namely *false positives*. The method of [3] is based on the following idea: if all queried nodes (k-mers) are only those which are reachable from some node known to belong to the graph, then only a fraction of all false positives can actually occur. Storing these false positives explicitly leads to an exact (false positive free) and space-efficient representation of the de Bruijn graph.

Our contribution is an improvement of this scheme by changing the representation of the set of false positives. We achieve this by iteratively applying a Bloom filter to represent the set of false positives, then the set of "false false positives" etc. We show analytically that this cascade of Bloom filters allows for a considerable further economy of memory, improving the method of [3]. Depending on the value of k, our method requires 30% to 40% less memory with respect to the method of [3]. Moreover, with our method, the memory grows very little as k grows. Finally, we implemented our method and tested it against [3] on real datasets. The tests confirm the theoretical predictions for the size of structure and show a 20% to 30% *improvement* in query times.

2 Preliminaries

A *Bloom filter* is a space-efficient data structure for representing a given subset of elements $T \subseteq U$, with support for efficient membership queries with one-sided error. That is, if a query for an element $x \in U$ returns *no* then $x \notin T$, but if it returns *yes* then x may or not belong to T, i.e. with small probability $x \notin T$ (false positive). It consists of a bitmap (array of bits) B with size m and a set of p distinct hash functions $\{h_1, \ldots, h_p\}$, where $h_i : U \mapsto \{0, \ldots, m-1\}$. Initially, all bits of B are set to 0. An insertion of an element $x \in T$ is done by setting the elements of B with indices $h_1(x), \ldots, h_p(x)$ to 1, i.e. $B[h_i(x)] = 1$ for all $i \in [1, p]$. The membership queries are done symmetrically, returning *yes* if all $B[h_i(x)]$ are equal 1 and *no* otherwise. As shown in [7], when considering hash functions that yield equally likely positions in the bit array, and for large enough array size m and number of inserted elements n, the false positive rate $\mathcal{F}$ is

$$\mathcal{F} \approx (1 - e^{-pn/m})^p = (1 - e^{-p/r})^p \tag{1}$$

where $r = m/n$ is the number of bits (of the bitmap B) per element (of T represented). It is not hard to see that this expression is minimized when $p = r \ln 2$, giving a false positive rate of

$$\mathcal{F} \approx (1 - e^{-p/r})^p = (1/2)^p \approx 0.6185^r. \tag{2}$$

A *de Bruijn graph*, for a given parameter k, of a set of reads (strings) $\mathcal{R} \subseteq \Sigma^* = \{A, C, T, G\}^*$ is entirely defined by the set $T \subseteq U = \Sigma^k$ of k-mers present in $\mathcal{R}$. The nodes of the graph are precisely the k-mers of T and for any two vertices $u, v \in T$, there is an arc from u to v if the suffix of u of size $k-1$ is equal to the prefix of v of the same size. Thus, given a set $T \subseteq U$ of k-mers we can represent its de Bruijn graph using a Bloom filter B. This approach has the disadvantage of having false positive nodes, as direct consequence of the false positive queries in the Bloom filter, which can create false connections in the graph (see [9] for the influence of false positive nodes on the topology of the graph). The naive way to remove those false positives nodes, by explicitly storing (e.g. using a hash table) the set of all false positives of B, is clearly inefficient, as the expected number of elements to be explicitly stored is $|U|\mathcal{F} = 4^k\mathcal{F}$.

The key idea of [3] is to explicitly store only a subset of all false positives of B, the so-called *critical false positives*. This is possible because in order to perform an exact (without false positive nodes) graph traversal, only potential neighbors of nodes in T are queried. In other words, the set of critical false positives consists of the potential neighbors of T that are false positives of B, i.e. the k-mers from U that overlap the k-mers from T by $k-1$ letters and are false positives of B. Thus, the size of the set of critical false positives is bounded by $8|T|$, since each node of T has at most $2|\Sigma| = 8$ neighbors (for each node, there are $|\Sigma|$ k-mers overlapping the $k-1$ suffix and $|\Sigma|$ overlapping the $k-1$ prefix). Therefore, the expected number of critical false positives can be upper-estimated by $8|T|\mathcal{F}$.

3 Cascading Bloom filter

Let $\mathcal{R}$ be a set of reads and T_0 be the set of occurring k-mers (nodes of the de Brujin graph) that we want to store. As stated in Section 2, the method of [3] stores T_0 via a bitmap B_1 using a Bloom filter, together with the set T_1 of critical false positives. T_1 consists of those k-mers which have a $k-1$ overlap with k-mers from T_0 but which are stored in B_1 "by mistake", i.e. belong[2] to B_1 but not to T_0. B_1 and T_1 are sufficient to represent the graph provided that the only queried k-mers are those which are potential neighbors of k-mers of T_0.

The idea we introduce in this work is to use this structure recursively and represent the set T_1 by a new bitmap B_2 and a new set T_2, then represent T_2 by B_3 and T_3, and so on. More formally, starting from B_1 and T_1 defined as above, we define a series of bitmaps $B_1, B_2, \ldots$ and a series of sets $T_1, T_2, \ldots$ as follows. B_2 stores the set of false positives T_1 using another Bloom filter, and the set T_2 contains the critical false positives of B_2, i.e. "true nodes" from T_0 that are stored in B_2 "by mistake" (we call them **false**[2] positives). B_3 and T_3, and, generally, B_i and T_i are defined similarly: B_i stores k-mers of T_{i-1} using a Bloom filter, and T_i contains k-mers stored in B_i "by mistake", i.e. those k-mers that do not belong to T_{i-1} but belong to T_{i-2} (we call them **false**^{i} positives). Observe that $T_0 \cap T_1 = \emptyset$, $T_0 \supseteq T_2 \supseteq T_4 \ldots$ and $T_1 \supseteq T_3 \supseteq T_5 \ldots$.

The following lemma shows that the construction is correct, that is it allows one to verify whether or not a given k-mer belongs to the set T_0.

Lemma 1. *Given a k-mer (node) K, consider the smallest i such that $K \notin B_{i+1}$ (if $K \notin B_1$, we define $i = 0$). Then, if i is odd, then $K \in T_0$, and if i is even (including 0), then $K \notin T_0$.*

Proof. Observe that $K \notin B_{i+1}$ implies $K \notin T_i$ by the basic property of Bloom filters that membership queries have one-sided error, i.e. there are no false negatives. We first check the Lemma for $i = 0, 1$.

For $i = 0$, we have $K \notin B_1$, and then $K \notin T_0$.

For $i = 1$, we have $K \in B_1$ but $K \notin B_2$. The latter implies that $K \notin T_1$, and then K must be a false[2] positive, that is $K \in T_0$. Note that here we use the fact that the only queried k-mers K are either nodes of T_0 or their neighbors in the graph (see [3]), and therefore if $K \in B_1$ and $K \notin T_0$ then $K \in T_1$.

For the general case $i \geq 2$, we show by induction that $K \in T_{i-1}$. Indeed, $K \in B_1 \cap \ldots \cap B_i$ implies $K \in T_{i-1} \cup T_i$ (which, again, is easily seen by induction), and $K \notin B_{i+1}$ implies $K \notin T_i$.

Since $T_{i-1} \subseteq T_0$ for odd i, and $T_{i-1} \subseteq T_1$ for even i (for $T_0 \cap T_1 = \emptyset$), the lemma follows.

Naturally, the lemma provides an algorithm to check if a given k-mer K belongs to the graph: it suffices to check successively if it belongs to $B_1, B_2, \ldots$ until we encounter the first B_{i+1} which does not contain K. Then, the answer

[2] By a slight abuse of language, we say that "an element belongs to B_j" if it is accepted by the corresponding Bloom filter.

will simply depend on whether i is even or odd: K belongs to the graph if and only if i is odd.

In our reasoning so far, we assumed an infinite number of bitmaps B_i. Of course, in practice we cannot store infinitely many (and even simply many) bitmaps. Therefore, we "truncate" the construction at some step t and store a finite set of bitmaps $B_1, B_2, \ldots, B_t$ together with an explicit representation of T_t. The procedure of Lemma 1 is extended in the obvious way: if for all $1 \leq i \leq t$, $K \in B_i$, then the answer is determined by directly checking $K \in T_t$.

4 Memory and time usage

First, we estimate the memory needed by our data structure, under the assumption of an infinite number of bitmaps. Let N be the number of "true positives", i.e. nodes of T_0. As stated in Section 2, if T_0 has to be stored via a bitmap B_1 of size rN, the false positive rate can be estimated as c^r, where $c = 0.6185$. And, the expected number of critical false positive nodes (set T_1) has been estimated in [3] to be $8Nc^r$, as every node has eight extensions, i.e. potential neighbors in the graph. We slightly refine this estimation to $6Nc^r$ by noticing that for most of the graph nodes, two out of these eight extensions belong to T_0 (are real nodes) and thus only six are potential false positives. Furthermore, to store these $6Nc^r$ critical false positive nodes, we use a bitmap B_2 of size $6rNc^r$. Bitmap B_3 is used for storing nodes of T_0 which are stored in B_2 "by mistake" (set T_2). We estimate the number of these nodes as the fraction c^r (false positive rate of filter B_2) of N (size of T_0), that is Nc^r. Similarly, the number of nodes we need to put to B_4 is $6Nc^r$ multiplied by c^r, i.e. $6Nc^{2r}$. Keeping counting in this way, the memory needed for the whole structure is $rN + 6rNc^r + rNc^r + 6rNc^{2r} + rNc^{2r} + ...$ bits. The number of bits per k-mer is then

$$r + 6rc^r + rc^r + 6rc^{2r} + ... = (r + 6rc^r)(1 + c^r + c^{2r} + ...) = (1 + 6c^r)\frac{r}{1 - c^r}. \quad (3)$$

A simple calculation shows that the minimum of this expression is achieved when $r = 5.464$, and then the minimum memory used per k-mer is 8.45 bits.

As mentioned earlier, in practice we store only a finite number of bitmaps $B_1, \ldots, B_t$ together with an explicit representation (such as array or hash table) of T_t. In this case, the memory taken by the bitmaps is a truncated sum $rN + 6rNc^r + rNc^r + ..$, and a data structure storing T_t takes either $2k \cdot Nc^{\lceil \frac{t}{2} \rceil r}$ or $2k \cdot 6Nc^{\lceil \frac{t}{2} \rceil r}$ bits, depending on whether t is even or odd. The latter follows from the observations that we need to store $Nc^{\lceil \frac{t}{2} \rceil r}$ (or $6rNc^{\lceil \frac{t}{2} \rceil r}$) k-mers, each taking $2k$ bits of memory. Consequently, we have to adjust the optimal value of r minimizing the total space, and re-estimate the resulting space spent on one k-mer.

Table 1 shows estimations for optimal values of r and the corresponding space per k-mer for $t = 4$ and $t = 6$, and several values of k. The data demonstrates that even such small values of t lead to considerable memory savings. It appears that the space per k-mer is very close to the "optimal" space (8.45 bits)

obtained for the infinite number of filters. Table 1 reveals another advantage of our improvement: the number of bits per stored k-mer remains almost constant for different values of k.

k	optimal r for $t = 4$	bits per k-mer for $t = 4$	optimal r for $t = 6$	bits per k-mer for $t = 6$	bits per k-mer for $t = 1$ ([3])
16	5.777	8.556	5.506	8.459	12.078
32	6.049	8.664	5.556	8.47	13.518
64	6.399	8.824	5.641	8.49	14.958
128	6.819	9.045	5.772	8.524	16.398

Table 1. 1st column: k-mer size; 2nd and 4th columns: optimal value of r for Bloom filters (bitmap size per number of stored elements) for $t = 4$ and $t = 6$ respectively; 3rd and 5th columns: the resulting space per k-mer (for $t = 4$ and $t = 6$); 6th column: space per k-mer for the method of [3] ($t = 1$)

The last column of Table 1 shows the memory usage of the original method of [3], obtained using the estimation $(1.44 \log_2(\frac{16k}{2.08}) + 2.08)$ the authors provided. Note that according to that estimation, doubling the value of k results in a memory increment by 1.44 bits, whereas in our method the increment is of 0.11 to 0.22 bits.

Let us now estimate preprocessing and query times for our scheme. If the value of t is small (such as $t = 4$, as in Table 1), the preprocessing time grows insignificantly in comparison to the original method of [3]. To construct each B_i, we need to store T_{i-2} (possibly on disk, if we want to save on the internal memory used by the algorithm) in order to compute those k-mers which are stored in B_{i-1} "by mistake". The preprocessing time increases little in comparison to the original method of [3], as the size of B_i decreases exponentially and then the time spent to construct the whole structure is linear on the size of T_0.

The query time can be split in two parts: the time spent on querying t Bloom filters and the time spent on querying T_t. Clearly, using t Bloom filters instead of a single one introduces a multiplicative factor of t to the first part of the query time. On the other hand, the set T_t is generally much smaller than T_1, due to the above-mentioned exponential decrease. Depending on the data structure for storing T_t, the time saving in querying T_t vs. T_1 may even dominate the time loss in querying multiple Bloom filters. Our experimental results (Section 5.1 below) confirm that this situation does indeed occur in practice. Note that even in the case when querying T_t weakly depends on its size (e.g. when T_t is implemented by a hash table), the query time will not increase much, due to our choice of a small value for t, as discussed earlier.

4.1 Using different values of r for different filters

In the previous section, we assumed that each of our Bloom filters uses the same value of r, the ratio of bitmap size to the number of stored k-mers. However, formula (3) for the number of bits per k-mer shows a difference for odd and even filter indices. This suggests that using different parameters r for different filters, rather than the same for all filters, may reduce the space even further. If r_i denotes the corresponding ratio for filter B_i, then (3) should be rewritten to

$$r_1 + 6r_2c^{r_1} + r_3c^{r_2} + 6r_4c^{r_1+r_3} + ..., \tag{4}$$

and the minimum value of this expression becomes 7.93 (this value is achieved with $r_1 = 4.41; r_i = 1.44, i > 1$).

In the same way, we can use different values of r_i in the truncated case. This leads to a small 2% to 4% improvement in comparison with case of unique value of r. Table 2 shows results for the case $t = 4$ for different values of k.

k	r_1, r_2, r_3, r_4	bits per k-mer different values of r	bits per k-mer single value of r
16	5.254, 3.541, 4.981, 8.653	8.336	8.556
32	5.383, 3.899, 5.318, 9.108	8.404	8.664
64	5.572, 4.452, 5.681, 9.108	8.512	8.824
128	5.786, 5.108, 6.109, 9.109	8.669	9.045

Table 2. Estimated memory occupation for the case of different values of r vs. single value of r, for 4 Bloom filters ($t = 4$). Numbers in the second column represent values of r_i on which the minimum is achieved. For the case of single r, its value is shown in Table 1.

5 Experimental results

5.1 Implementation and experimental setup

We implemented our method using the Minia software [3] and ran comparative tests for 2 and 4 Bloom filters ($t = 2, 4$). Note that since the only modified part of Minia was the construction step and the k-mer membership queries, this allows us to precisely evaluate our method against the one of [3].

The first step of the implementation is to retrieve the list of k-mers that appear more than d times using DSK [13] – a constant memory streaming algorithm to count k-mers. Each k-mer appearing more than d times (set T_0) is inserted into B_1. Next, all possible extensions of each k-mer in T_0 are queried

against B_1, and those which return true are written to the disk. Then, this set is traversed and only the k-mers absent from T_0 are kept. This results in the set T_1 of critical false positives, which is also kept on disk. Up to this point, the procedure is identical to that of [3].

Next, we insert all k-mers from T_1 into B_2 and to obtain T_2, we check for each k-mer in T_0 if a query to B_2 returns true. This results in the set T_2. Thus, at this point we have B_1, B_2 and T_2, a complete representation for $t = 2$. In order to build the data structure for $t = 4$, we continue this process, by inserting T_2 in B_3 and retrieving T_3 from T_1 (stored on disk). It should be noted that to obtain T_i we need T_{i-2}, and by always storing it on disk we guarantee not to use more memory than the size of the final structure. The set T_t (that is, T_1, T_2 or T_4 in our experiments) is stored as a sorted array and is searched by a binary search. We found this implementation more efficient than a hash table.

Assessing the query time is done through the procedure of graph traversal, as it is implemented in [3]. Since the procedure is identical and independent on the data structure, the time spent on graph traversal is a faithful estimator of the query time.

We compare three versions: $t = 1$ (i.e. the version of [3]), $t = 2$ and $t = 4$. For convenience, we define 1 Bloom, 2 Bloom and 4 Bloom as the versions with $t = 1, 2$ and 4, respectively.

5.2 *E.coli* dataset, varying k

In this set of tests, our main goal was to evaluate the influence of the k-mer size on principal parameters: size of the whole data structure, size of the set T_t, graph traversal time, and time of construction of the data structure. We retrieved 10M *E. coli* reads of 100bp from the *Short Read Archive* (ERX008638) without read pairing information and extracted all k-mers occurring at least two times. The total number of k-mers considered varied, depending on the value of k, from 6,967,781 ($k = 15$) to 5,923,501 ($k = 63$). We ran each version, 1 Bloom ([3]), 2 Bloom and 4 Bloom, for values of k ranging from 16 to 64. The results are shown in Fig. 1.

The total size of the structures in bits per stored k-mer, i.e. the size of B_1 and T_1 (respectively, B_1, B_2,T_2 or B_1, B_2, B_3, B_4,T_4) is shown in Fig. 1(a). As expected, the space for 4 Bloom filters is the smallest for all values of k considered, showing a considerable improvement, ranging from 32% to 39%, over the version of [3]. Even the version with just 2 Bloom filters shows an improvement of at least 20% over [3], for all values of k. Regarding the influence of the k-mer size on the structure size, we observe that for 4 Bloom filters the structure size is almost constant, the minimum value is 8.60 and the largest is 8.89, an increase of only 3%. For 1 and 2 Bloom the same pattern is seen: a plateau from $k = 16$ to 32, a jump for $k = 33$ and another plateau from $k = 33$ to 64. The jump at $k = 32$ is due to switching from 64-bit to 128-bit representation of k-mers in the table T_t.

The traversal times for each version is shown in Fig. 1(c). The fastest version is 4 Bloom, showing an improvement over [3] of 18% to 30%, followed by 2

Bloom. This result is surprising and may seem counter-intuitive, as we have four filters to apply to the queried k-mer rather than a single filter as in [3]. However, the size of T_4 (or even T_2) is much smaller than T_1, as the size of T_i's decreases exponentially. As T_t is stored in an array, the time economy in searching T_4 (or T_2) compared to T_1 dominates the time lost on querying additional Bloom filters, which explains the overall gain in query time.

As far as the construction time is concerned (Fig. 1(d)), our versions yielded also a faster construction, with the 4 Bloom version being 5% to 22% faster than that of [3]. The gain is explained by the time required for sorting the array storing T_t, which is much higher for T_0 than for T_2 or T_4. However, the gain is less significant here, and, on the other hand, was not observed for bigger datasets (see Section 5.4).

5.3 *E. coli* dataset, varying coverage

From the complete *E. coli* dataset (≈44M reads) from the previous section, we selected several samples ranging from 5M to 40M reads in order to assess the impact of the coverage on the size of the data structures. This strain *E. coli* (K-12 MG1655) is estimated to have a genome of 4.6M bp [1], implying that a sample of 5M reads (of 100bp) corresponds to ≈100X coverage. We set $d = 3$ and $k = 27$. The results are shown in Fig. 2. As expected, the memory consumption per k-mer remains almost constant for increasing coverage, with a slight decrease for 2 and 4 Bloom. The best results are obtained with the 4 Bloom version, an improvement of 33% over the 1 Bloom version of [3]. On the other hand, the number of distinct k-mers increases markedly (around 10% for each 5M reads) with increasing coverage, see Fig. 2(b). This is due to sequencing errors: an increase in coverage implies more errors with higher coverage, which are not removed by our cutoff $d = 3$. This suggests that the value of d should be chosen according to the coverage of the sample. Moreover, in the case where read qualities are available, a quality control pre-processing step may help to reduce the number of sequencing errors.

5.4 Human dataset

We also compared 2 and 4 Bloom versions with the 1 Bloom version of [3] on a large dataset. For that, we retrieved 564M Human reads of 100bp (SRA: SRX016231) without pairing information and discarded the reads occurring less than 3 times. The dataset corresponds to ≈17X coverage. A total of 2,455,753,508 k-mers were indexed. We ran each version, 1 Bloom ([3]), 2 Bloom and 4 Bloom with $k = 23$. The results are shown in Table 3.

The results are in general consistent with the previous tests on *E.coli* datasets. There is an improvement of 34% (21%) for the 4 Bloom (2 Bloom) in the size of the structure. The graph traversal is also 26% faster in the 4 Bloom version. However, in contrast to the previous results, the graph construction time increased by 10% and 7% for 4 and 2 Bloom versions respectively, when compared

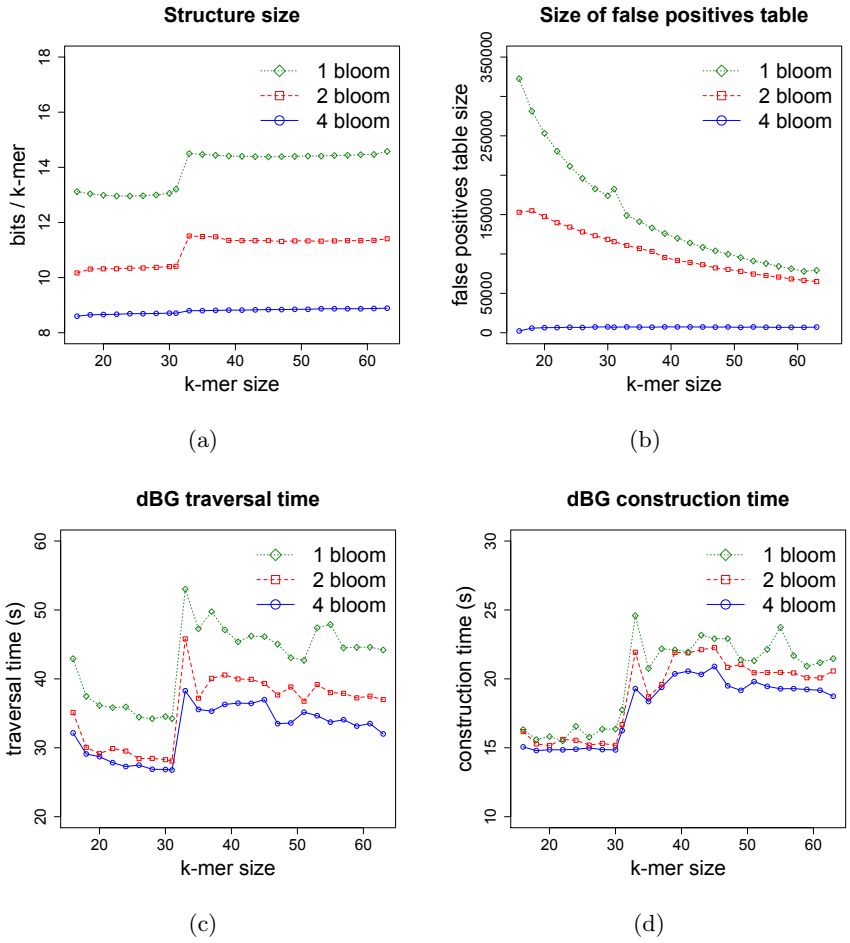
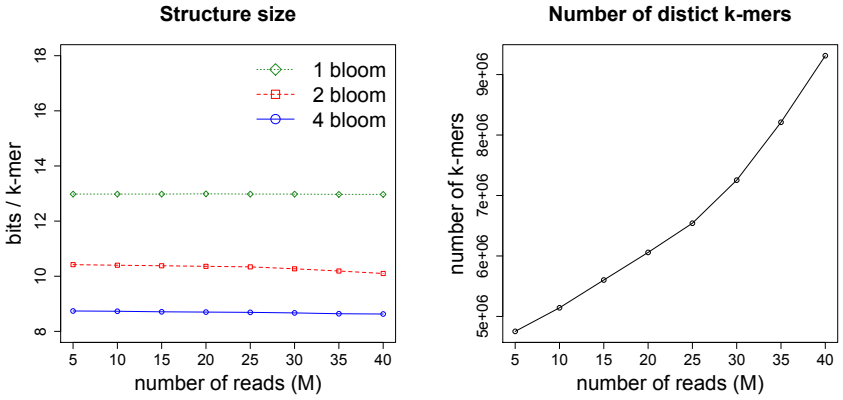


Fig. 1. Results for 10M E.coli reads of 100bp using several values of k. The *1 Bloom* version corresponds to the one presented in [3]. (a) Size of the structure in bits used per k-mer stored. (b) Number of false positives stored in T_1, T_2 or T_4 for 1, 2 or 4 Bloom filters, respectively. (c) De Bruijn graph construction time, excluding k-mer counting step. (d) De Bruijn graph traversal time, including branching k-mer indexing.



(a) (b)

Fig. 2. Results for *E.coli* reads of 100bp using $k = 27$. The *1 Bloom* version corresponds to the one presented in [3]. (a) Size of the structure in bits used per k-mer stored. (b) Number of distinct k-mers.

to the 1 Bloom version. This is due to the fact that disk writing/reading operations now dominate the time for the graph construction, and 2 and 4 Bloom versions generate more disk accesses than 1 Bloom. As stated in Section 5.1, when constructing the 1 Bloom structure, the only part written on the disk is T_1 and it is read only once to fill an array in memory. For 4 Bloom, T_1 and T_2 are written to the disk, and T_0 and T_1 are read at least one time each to build B_2 and B_3. Moreover, since the size coefficient of B_1 reduces, from $r = 11.10$ in 1 Bloom to $r = 5.97$ in 4 Bloom, the number of false positives in T_1 increases.

6 Discussion and Conclusions

Using cascading Bloom filters for storing de Bruijn graphs brings a clear advantage over the single-filter method of [3]. In terms of memory consumption, which is the main parameter here, we obtained an improvement of around 30%-40% in all our experiments. Our data structure takes 8.5 to 9 bits per stored k-mer, compared to 13 to 15 bits by the method of [3]. This confirms our analytical estimations. The above results were obtained using only four filters and are very close to the estimated optimum (around 8.4 bits/k-mer) produced by the infinite number of filters. An interesting characteristic of our method is that the memory grows insignificantly with the growth of k, even slower than with the method of [3]. Somewhat surprisingly, we also obtained a significant decrease, of order 20%-30%, of query time. The construction time of the data structure varied from being 10% slower (for the human dataset) to 22% faster (for the bacterial dataset).

Method	1 Bloom	2 Bloom	4 Bloom
Construction time (s)	40160.7	43362.8	44300.7
Traversal time (s)	46596.5	35909.3	34177.2
r coefficient	11.10	7.80	5.97
Bloom filters size (MB)	$B_1 = 3250.95$	$B_1 = 2283.64$ $B_2 = 323.08$	$B_1 = 1749.04$ $B_2 = 591.57$ $B_3 = 100.56$ $B_4 = 34.01$
False positive table size (MB)	$T_1 = 545.94$	$T_2 = 425.74$	$T_4 = 36.62$
Total size (MB)	3796.89	3032.46	2511.8
Size (bits/k-mer)	**12.96**	**10.35**	**8.58**

Table 3. Results of 1, 2 and 4 Bloom filters version for 564M Human reads of 100bp using $k = 23$. The *1 Bloom* version corresponds to the one presented in [3].

As stated previously, another compact encoding of de Bruijn graphs has been proposed in [2], however no implementation of the method was made available. For this reason, we could not experimentally compare our method with the one of [2]. We remark, however, that the space bound of [2] heavily depends on the number of reads (i.e. coverage), while in our case, the data structure size is almost invariant with respect to the coverage (Section 5.3).

An interesting prospect for further possible improvements of our method is offered by work [12], where an efficient replacement to Bloom filter was introduced. The results of [12] suggest that we could hope to reduce the memory to about 5 bits per k-mer. However, there exist obstacles on this way: an implementation of such a structure would probably result in a significant construction and query time increase.

Acknowledgements Part of this work has been done during the visit of KS to LIGM in France, supported by the CNRS French-Russian exchange program in Computer Science. GK has been partly supported by the ABS2NGS grant of the French gouvernement (program *Investissement d'Avenir*) as well as by a Marie-Curie Intra-European Fellowship for Carrier Development. GS was supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no. [247073]10

References

1. F. R. Blattner, G. Plunkett, and C.A. Bloch et al. The complete genome sequence of escherichia coli k-12. *Science*, 277(5331):1453–1462, 1997.

2. A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In B.J. Raphael and J.Tang, editors, *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer, 2012.
3. R. Chikhi and G. Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In Benjamin J. Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7534 of *Lecture Notes in Computer Science*, pages 236–248. Springer, 2012.
4. T.C. Conway and A.J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.
5. M. G. Grabherr, B. J. Haas, M. Yassour, J.Z. Levin, and et al. Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nat Biotech*, 29(7):644–652, July 2011.
6. Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, 44(2):226–232, Feb 2012.
7. A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, September 2008.
8. J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, Jun 2010.
9. J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc. Natl. Acad. Sci. U.S.A.*, 109(33):13272–13277, Aug 2012.
10. Y. Peng, H. C. M. Leung, S. M. Yiu, and F. Y. L. Chin. Meta-IDBA: a de novo assembler for metagenomic data. *Bioinformatics*, 27(13):i94–i101, 2011.
11. P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U.S.A.*, 98(17):9748–9753, Aug 2001.
12. E. Porat. An optimal Bloom filter replacement based on matrix solving. In *Computer Science - Theory and Applications, Fourth International Computer Science Symposium in Russia, CSR 2009, Novosibirsk, Russia, August 18-23, 2009. Proceedings*, volume 5675 of *Lecture Notes in Computer Science*, pages 263–273. Springer, 2009.
13. G. Rizk, D. Lavenier, and R. Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 2013.
14. G. Sacomoto, J. Kielbassa, R. Chikhi, and R. Uricaru et al. KISSPLICE: de-novo calling alternative splicing events from RNA-seq data. *BMC Bioinformatics*, 13(Suppl 6):S5, 2012.
15. C. Ye, Z. Ma, C. Cannon, M. Pop, and D. Yu. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, 13(Suppl 6):S1, 2012.