

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

Rayan Chikhi, Guillaume Rizk

▶ **To cite this version:**

Rayan Chikhi, Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. WABI 2012, Sep 2012, Ljubljana, Slovenia. pp 236-248, 10.1007/978-3-642-33122-0_19 . hal-00753930

HAL Id: hal-00753930
https://hal.science/hal-00753930

Submitted on 19 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 [5]. It has recently been used in a wider set of applications: *de novo* mRNA [4] and metagenome [13] assembly, genomic variants detection [14,6] and *de novo* alternative splicing calling [17]. 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 [9]. Simpson *et al.* implemented a distributed de Bruijn graph to reduce the memory usage per node [18]. Conway and Bromage applied sparse bit array structures to store an implicit, immutable graph representation [3]. Targeted methods compute local assemblies around sequences of interest, using negligible memory, with greedy extensions [19] or portions of the de Bruijn graph [15]. 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 ($10 \leq g \leq 25$) [20].

Conway and Bromage observed that the self-information of the edges is a lower bound for exactly encoding the de Bruijn graph [3]:

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

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 [12] 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 [12] (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** [5], 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** [8] 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 [8]:

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

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 [12]. 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

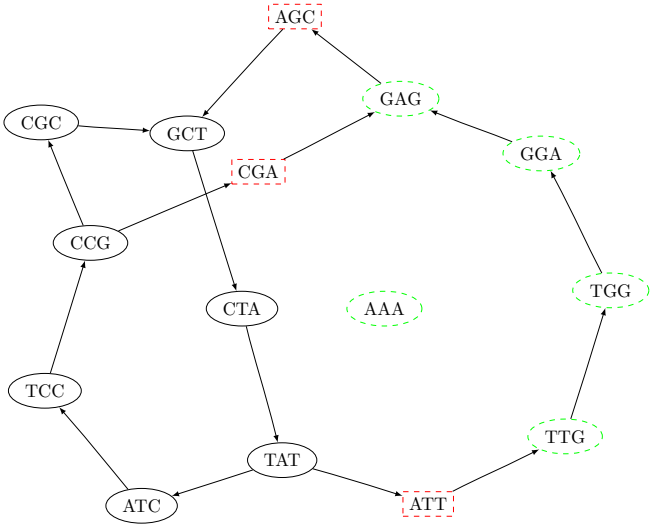
Our contribution is a mechanism that avoids false branching. Specifically, we propose to detect and store false positive elements which are responsible for false branching, in a separate structure. To this end, we introduce the cFP structure of *critical False Positives* k-mers, implemented with a standard 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

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

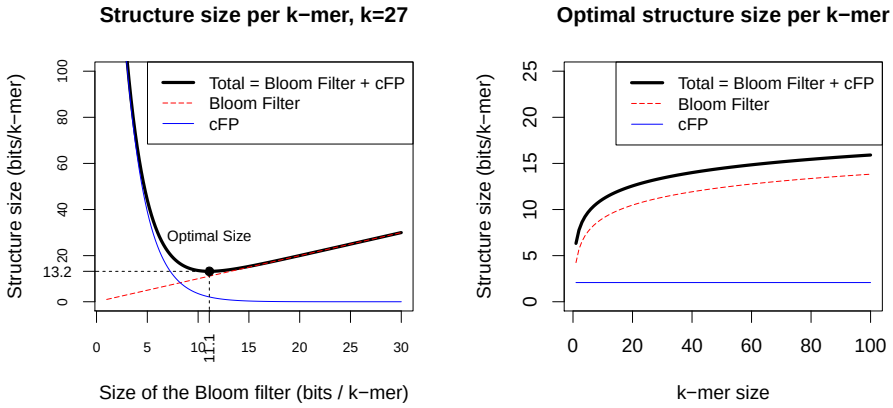
For illustration, Figure 2-(a) shows the size of the structure for various Bloom filter sizes and $k = 27$. For this value of k, the optimal size of the Bloom filter is 11.1 bits per k-mer, and the total structure occupies 13.2 bits per k-mer. Figure 2-(b) shows that k has only a modest influence on 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 [11] and transcriptomes [4], and *de novo* variant detection [17], 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 [2].

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 [7].



(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 [10], 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 [2]) is performed to traverse short, locally complex regions. Specifically, the traversal ignores tips (dead-end paths) shorter than $2k + 1$ nodes. It

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

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. [18]), 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 [12], which observed that when

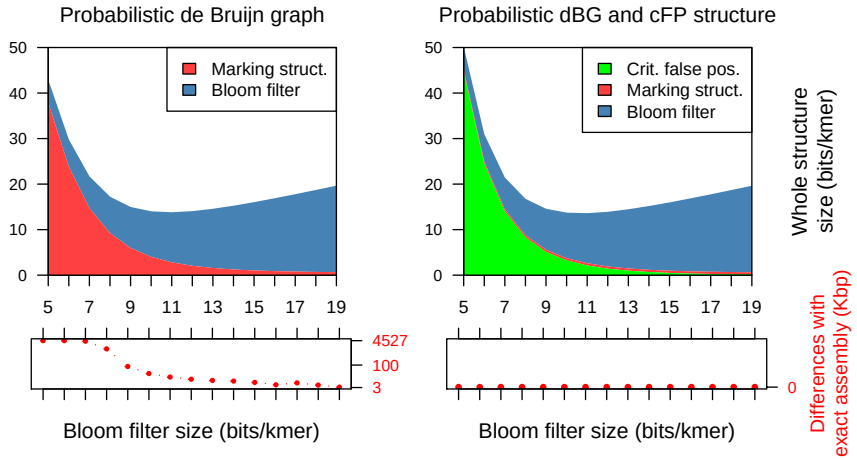


Fig. 3: Whole structures size (Bloom filter, marking structure, and cFP if applicable) of the probabilistic de Bruijn graph with (top right) and without the cFP structure (top left), for an actual dataset (E. coli, $k = 23$). All plots are in function of the number of bits per k-mer allocated to the 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.

the 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 $\approx$ 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 [18], SOAPdenovo [9], and the prototype assembler from Conway and Bromage [3]. 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 [16]. 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 [18].

To test another recent assembler, SparseAssembler [20], 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.

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).

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 [3], ABySS [18], and SOAPdenovo [9]. Contigs shorter than 100 bp were discarded. Assemblies were made without any pairing information.

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 [14,17]. 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 [1] 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).

References

1. Chazelle, B., Kilian, J., Rubinfeld, R., Tal, A.: The bloomier filter: an efficient data structure for static support lookup tables. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 30–39. SIAM (2004)
2. Chikhi, R., Lavenier, D.: Localized genome assembly from reads to scaffolds: practical traversal of the paired string graph. Algorithms in Bioinformatics pp. 39–48 (2011)
3. Conway, T.C., Bromage, A.J.: Succinct data structures for assembling large genomes. Bioinformatics 27(4), 479 (2011)
4. Grabherr, Manfred G, e.a.: Full-length transcriptome assembly from RNA-Seq data without a reference genome. Nat Biotech 29(7), 644–652 (Jul 2011),
5. Idury, R.M., Waterman, M.S.: A new algorithm for DNA sequence assembly. Journal of Computational Biology 2(2), 291–306 (1995)
6. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de bruijn graphs. Nature Genetics (2012)
7. Kingsford, C., Schatz, M.C., Pop, M.: Assembly complexity of prokaryotic genomes using short reads. BMC bioinformatics 11(1), 21 (2010)
8. Kirsch, A., Mitzenmacher, M.: Less hashing, same performance: Building a better bloom filter. AlgorithmsESA 2006 pp. 456–467 (2006)
9. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K.: De novo assembly of human genomes with massively parallel short read sequencing. Genome research 20(2), 265 (2010)
10. Marais, G., Kingsford, C.: A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics 27(6), 764–770 (2011),
11. Miller, J.R., Koren, S., Sutton, G.: Assembly algorithms for next-generation sequencing data. Genomics 95(6), 315–327 (2010)
12. Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J.M., Brown, C.T.: Scaling metagenome sequence assembly with probabilistic de bruijn graphs. Arxiv preprint arXiv:1112.4193 (2011)
13. Peng, Y., Leung, H.C.M., Yiu, S.M., Chin, F.Y.L.: Meta-IDBA: a de novo assembler for metagenomic data. Bioinformatics 27(13), i94–i101 (2011)
14. Peterlongo, P., Schnel, N., Pisanti, N., Sagot, M.F., Lacroix, V.: Identifying SNPs without a reference genome by comparing raw reads. In: String Processing and Information Retrieval. pp. 147–158. Springer (2010)
15. Peterlongo, P., Chikhi, R.: Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer. BMC Bioinformatics (1), 48 (2012)
16. Rizk, G., Lavenier, D.: GASSST: global alignment short sequence search tool. Bioinformatics 26(20), 2534 (2010)
17. Sacomoto, G., Kielbassa, J., Chikhi, R., Uricaru, R., Antoniou, P., Sagot, M., Peterlongo, P., Lacroix, V.: KISSPLICE: de-novo calling alternative splicing events from RNA-seq data. BMC Bioinformatics 13(Suppl 6), S5 (2012),
18. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J.M., Birol, .: ABySS: a parallel assembler for short read sequence data. Genome Research 19(6), 1117–1123 (2009),
19. Warren, R.L., Holt, R.A.: Targeted assembly of short sequence reads. PloS one 6(5), e19816 (2011)
20. Ye, C., Ma, Z., Cannon, C., Pop, M., Yu, D.: Exploiting sparseness in de novo genome assembly. BMC Bioinformatics 13(Suppl 6), S1 (2012),