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 $\{h_1, \dots, h_p\}$, where $h_i : U \mapsto \{0, \dots, m-1\}$. 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 $h_1(x), \dots, h_p(x)$ 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 $rN + 6rNc^r + rNc^r + 6rNc^{2r} + rNc^{2r} + \ldots$ bits. The number of bits per k-mer is then

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

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 $rN + 6rNc^r + rNc^r + \ldots$, 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).

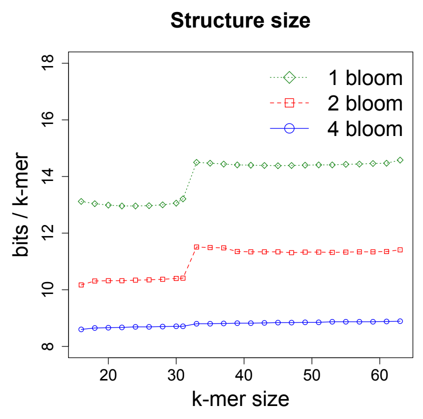
5.3 *E. coli* dataset, varying coverage

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.

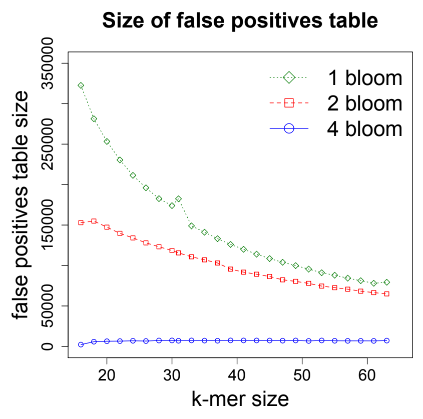
5.4 Human dataset

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.

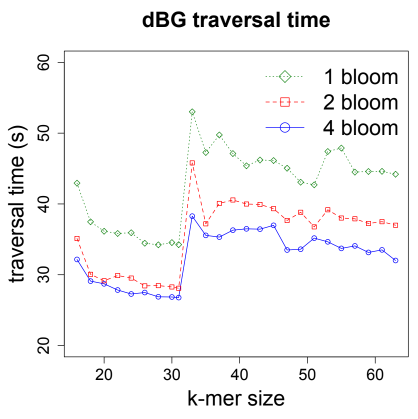
The results are in general consistent with the previous tests on *E.coli* datasets. There is an improvement of 34% (21%) for the 4 Bloom (2 Bloom) in the size of the structure. The graph traversal is also 26% faster in the 4 Bloom version. However, in contrast to the previous results, the graph construction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



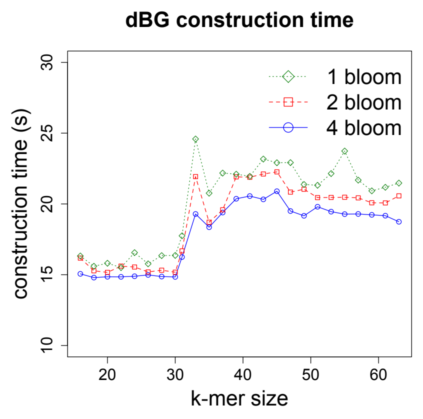
(a)



(b)

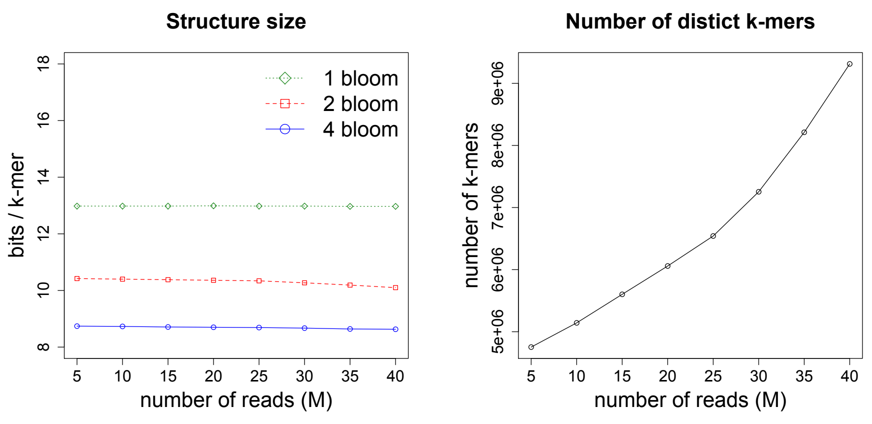


(c)



(d)

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.