Practical dynamic de Bruijn graphs

Alan Kuhnle, Victoria G. Crawford, Christina Boucher, Rayan Chikhi, Travis Gagie

Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Practical Dynamic de Bruijn Graphs

Alan Kuhnle [1,*], Victoria Crawford [1], Christina Boucher [1], Rayan Chikhi [2], and Travis Gagie [3]

[1]Computer & Information Science & Engineering, University of Florida, Gainesville, 32306
[2] Filled in by Rayan [3] Universidad Diego Portales and CeBiB, Santiago, Chile

*To whom correspondence should be addressed.

Abstract

As datasets of DNA reads grow rapidly, it becomes more and more important to represent de Bruijn graphs compactly while still supporting fast assembly. Previous implementations have not supported edge deletion, however, which is important for pruning spurious edges from the graph. Belazzougui et al. Belazzougui *et al.* (2016b) recently proposed a compact and fully dynamic representation, which supports exact membership queries and insertions and deletions of both nodes and edges. In this paper we give a practical implementation of their data structure, supporting exact membership queries and insertions and deletions of edges only, and demonstrate experimentally that its performance is comparable to that of state-of-the-art implementations based on Bloom filters. Our source-code is publicly available at `https://github.com/csirac/kbf` under an open-source license.

1 Introduction

Since ?? **??** introduced them to bioinformatics in the 1990s, de Bruijn graphs have come to dominate *de novo* genome assembly. In the kth-order de Bruijn for a set of reads, the nodes are the set of all $(k-1)$-mers in the reads and there is an edge from node u to node v if and only if there is a k-mer in the reads with prefix u and suffix v. Short-read assemblers typically use the Eulerian approach, meaning they look for paths of nodes with in- and out-degree 1, corresponding to contiguous sequences (*contigs*) in the genome. Some assemblers use a single value of k, such as ABySS **?** and Velvet **?**, while other use several values of k in turn, such as IDBA **?** and SPAdes **?**. Aside from assembly, de Bruijn graphs are also used for read correction **?** and variant discovery , for example.

Due to their widespread use and the large size of modern datasets, it is important that we use compact representations of de Bruijn graphs that can still be built quickly and support fast assembly. Several authors have proposed representations built on Bloom filters Bloom (1970), which is a space-efficient probabilistic data structure built on multiple hash functions that is used to test whether an element is in a set, with the possibility of false positives. For example, by storing each k-mer in a Bloom filter, ??? Pell *et al.* (2012) were able to represent the graph using as little as 4 bits per k-mer, albeit not with complete accuracy due to the probabilistic nature of the Bloom filter. ??? Chikhi and Rizk (2013) proposed using a Bloom filter with an additional structure to detect false positives, and was able to perform a complete *de novo* assembly of a human genome using 5.7GB of memory. ??? Salikhov *et al.* (2014) used cascading Bloom filters with false-positive detection and further reduced the memory usage by 30-40%.

By their nature, inserting an element into a Bloom filter is usually fairly easy, but deleting one is usually difficult or impossible. Other compact representations of de Bruijn graphs **?** usually do not allow even easy insertions. Two exceptions that we know of are data structures by Belazzougui and two sets of coauthors Belazzougui *et al.* (2016a); **?**, both of which support insertions and deletions both of nodes and of edges: the first is based on an extension by Bowe et al. Bowe *et al.* (2012) of FM-indexes **?**, which was further extended by Boucher et al. Boucher *et al.* (2015), and stores a pointer-based representation of recently updated nodes and edges that is incorporated into the main representation through periodic rebuilding; the second is based on a combination of Karp-Rabin hashing **?** and minimal perfect hashing **?** but still supports exact membership queries. It is not clear that a complete implementation of either of these data structures would be practical but, fortunately, the second data structure becomes much simpler if we concern ourselves with only exact membership queries and insertions and deletions of edges, not nodes. Edge deletions in particular are interesting because sequencing errors give rise to spurious nodes and edges, which it is often useful to prune from the graph before assembly. Node deletions can be simulated by edge deletions, since removing all the edges incident to a node effectively deletes it.

Our contribution. In this paper we implement the simple version of Belazzougui et al.'s hash-based data structure mentioned above, and demonstrate experimentally that its performance is comparable to that of state-of-the-art implementations based on Bloom filters. In Section 2 we review Belazzougui et al.'s design. We describe our implementation in Section **??**, and the results of our experiments in Section **??**. Finally, we give our conclusions in Section **??**.

2 Design

Belazzougui et al.'s data structure for a kth-order de Bruijn graph G is most easily described in layers. At the base is a Karp-Rabin hash function **?**, which can take either a $(k-1)$-mer and return an integer hash value in $\mathcal{O}(k)$ time, or take a $(k-1)$-mer v, its hash value and a character c, and return in $\mathcal{O}(1)$ time the hash value of the $(k-1)$-mer obtained from v by deleting its first character and appending c. Storing this function takes asymptotically negligible space.

For the purposes of this paper, the next level is a minimal perfect hash function that in $\mathcal{O}(1)$ time maps integers to values in $\{0, \ldots, n-1\}$, where n is the number of nodes in G. When restricted to the Karp-Rabin hash values of the nodes in G, the minimal perfect hash function is bijective. Storing this function takes $O(n+\log k+\log|\Sigma|)$ bits, where Σ is the alphabet (i.e., $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ in this paper). The construction algorithm is Las-Vegas randomized: any function it returns has these properties and with high probability it returns a function in $O(kn)$ time, with the probability taken over the random bits.

If we wanted to support insertions and deletions of nodes as well as of edges, we would instead use a dynamic minimal perfect hash function. This would require more space than a static minimal perfect hash function, however, and the construction algorithm can return a faulty function with low probability.

Let $f : \Sigma^{k-1} \to \{0, \ldots, n-1\}$ be the composition of the minimal perfect hash function and the Karp-Rabin hash function. Notice that, once we have computed $f(v)$ for a node v in G, we can compute $f(w)$ in $\mathcal{O}(1)$ time, for any neighbour w of v. Since f maps any $(k-1)$-mer to a a number between 0 and $n-1$, however, by itself it is not enough for us to navigate in G.

The third layer is a pair of n-by-$|\Sigma|$ binary arrays `IN` and out `OUT` that indicate which neighbours each node has. Specifically, for each node v and each character c, if $f(v) = i$ and j is 1 less than c's lexicographic rank in the alphabet, then $\texttt{IN}[i][j] = 1$ if and only if there is a directed edge to v from a $(k-1)$-mer that starts with c; symmetrically, $\texttt{OUT}[i][j] = 1$ if and only if there is a directed from v to a $(k-1)$-mer that ends with c. These arrays take $|\Sigma|$ bits per node, i.e., $4n$ bits in this paper.

Suppose v and w are $(k-1)$-mers such that $f(v) = i$, v starts with the lexicographically $(j+1)$st character in the alphabet, $f(w) = i'$, w ends with the lexicographically $(j'+1)$st character in the alphabet, and the last $k-2$ characters in v are the first $k-2$ characters of w. Belazzougui et al. showed that, if $\texttt{OUT}[i][j] = 1$ and $\texttt{IN}[i'][j'] = 1$ then either both v and w are in G or neither are. That is, assuming either v or w is in G, if $\texttt{OUT}[i][j] = 1$ and $\texttt{IN}[i'][j'] = 1$ then the edge (v, w) is also in G. Of course, if either if $\texttt{OUT}[i][j] = 0$ or $\texttt{IN}[i'][j'] = 0$, then the edge (v, w) is not in G. Similar ideas have been used previously, e.g., the implementation of SPAdes **?**.

Using `IN` and `OUT`, if we start at a node v we think is in G, we can explore its entire connected component in the underlying undirected graph (i.e., all the nodes from which v is reachable in G and all the nodes which can be reached from v). If we ever encounter a discrepancy between `IN` and `OUT`— i.e., an edge (u, w) that `IN` says is incident to w but `OUT` says is not incident to u, or vice versa — then we can deduce that v was in fact not in G. Unfortunately, the absence of such a discrepancy does not confirm that v is in G.

To be able to verify whether nodes are in G, Belazzougui et al. use a fourth layer, consisting of a forest of shallow rooted trees. The edges in this forest are a subset of the edges in the undirected graph underlying G. We choose the trees to have height between $k \lg \sigma$ and $3k \lg \sigma$, except that we allow a tree to be shorter than $k \lg \sigma$ when it covers an entire connected component in the underlying undirected graph. We store the $(k-1)$-mers that are the roots of the trees (in order by their hash values), mark the numbers between 0 and $n-1$ to which f maps those $(k-1)$-mers and, for each non-root node v, we mark the edge incident to v that leads to v's parent in the forest. This takes $2n|\Sigma| + n\lceil \lg|\Sigma| \rceil + 2n$ bits, so $10n$ bits in this paper, plus possibly $k\lceil \lg|\Sigma| \rceil$ bits for each connected component in the underlying undirected graph.

Given a node v, if $f(v)$ is marked as being the hash value of a root, then we can check in $\mathcal{O}(k)$ time whether v is in G by comparing it to the $(k-1)$-mer we have stored for that root. Otherwise, we assume v is in G, follow the edge to its parent u (checking there is no discrepancy between `IN` and `OUT`), and check that u is in G. If v really is in G, then in $\mathcal{O}(k)$ time we reach the root and verify it, thus also verify v (and all its ancestors). If v is not in G, then either we will take too many steps trying to reach a root, or we will find a discrepancy between `IN` and `OUT` along the way, or when we reach a root we will find the $(k-1)$-mer we are trying to check there is not the one we have stored.

If we insert an edge between two nodes in the same connected component of the underlying undirected graph, or insert an edge between two connected components each larger than $k \lg|\Sigma|$, or delete an edge that is not in the forest, then we can simply update `IN` and `OUT` without updating the forest. Next, we describe the last layer of Belazzougui et al.'s data structure: how to update the forest when an edge is inserted between two connected components, at least one of which is smaller than $k \lg|\Sigma|$, or an edge in the forest is deleted.

First, suppose we add an edge between two connected components. If neither component is larger than $k \lg|\Sigma|$, we may merge the two corresponding trees into a single tree by discarding the old roots, sampling a new root, and reversing the direction of forest edges as necessary to ensure we have a single tree. If exactly one component is larger than $k \lg|\Sigma|$, we consider the depth in its tree of the node u in the larger component incident with the inserted edge. If u is of depth less than $2k \lg|\Sigma|$, we can simply reverse the necessary forest edges in the smaller component and discard its root to add it into the larger component. Otherwise, if the depth of u is at least $2k \lg|\Sigma|$, we traverse upwards in its tree $k \lg|\Sigma|$ steps to node v and break off the subtree rooted at v, storing v as the new root. Again, the root of the smaller component is discarded and forest edges are reversed as necessary to ensure that we have a tree.

Next, suppose we delete an edge (u, v) that is in the forest, so v had been the parent of u. We check the subtree below u and the size of the tree containing v. If both are larger than $k \lg|\Sigma|$, we store u as a root. Otherwise, if one of these trees is smaller than $k \lg|\Sigma|$, we search for any edge in the De Bruijn graph incident with a node w in this tree and a node x in an adjacent tree. If such an edge (w, x) or (x, w) is found, we make x the parent of w and reverse forest edges as necessary to append the tree of w below the adjacent tree; the resulting tree may exceed height $3k \lg|\Sigma|$; if it does, this implies that x is of depth at least $2k \lg|\Sigma|$. In this case, we traverse up $k \lg|\Sigma|$ steps from x and break off the subtree at that node by storing it as a root.

By these procedures, we maintain that each tree in the forest is of size at least $k \lg|\Sigma|$ and height at most $3k \lg|\Sigma|$, unless a tree is in a connected component of size less than $k \lg|\Sigma|$. Furthermore, both procedures for edge addition and deletion as described above take at most $O(k \lg|\Sigma|)$ time, since each change to the data structure takes constant time, computing hash values of u, v requires $O(k)$ time, and there are at most a bounded number of partial tree traversals in each case, where each tree traversal explores at most $2k \lg|\Sigma|$ elements.

We note as an aside that, since in this paper we are concerned only with insertions and deletions of edges and not nodes, we could further simplify Belazzougui et al.'s design. To do this, we would conceptually overlay on G another graph H with the same set of nodes, but with all possible edges present. We would then build a forest on H instead of on G, discard from H all the edges not in the forest, and build binary arrays

$\texttt{IN}_H$ and $\texttt{OUT}_H$. We could then verify that nodes are in G using the forest on H, which is static. We do not follow this approach because it uses more space, diverges significantly from Belazzougui et al.'s description, and would make it much more difficult to extend our implementation in the future to include insertions and deletions of nodes as well.

The above discussion is summarized in the following Lemma and Theorem.

LEMMA 1 (Belazzougui *et al.* (2016b)). *Given a set E of n k-mers over an alphabet Σ of size σ, with high probability in $O(kn)$ expected time we can build a function $f : \Sigma^k \to \{0, \ldots, m-1\}$ with the following properties:*

- *when its domain is restricted to E, f is bijective;*
- *we can store f in $O(n + \log k + \log \sigma)$ bits;*
- *given a k-mer v, we can compute $f(v)$ in $\mathcal{O}(k)$ time;*
- *given u and v such that the suffix of u of length $k-1$ is the prefix of v of length $k-1$, or vice versa, if we have already computed $f(u)$ then we can compute $f(v)$ in $\mathcal{O}(1)$-time.*

THEOREM 1 (Belazzougui *et al.* (2016b)). *Given a de Bruijn graph G with n nodes, with high probability in $\mathcal{O}(kn + n\sigma)$ expected time we can store G in $\mathcal{O}(\sigma n)$ bits plus $\mathcal{O}(k \log \sigma)$ bits for each connected component in the underlying undirected graph, such that checking whether a node is in G takes $\mathcal{O}(k \log \sigma)$ time, listing the edges incident to a node we are visiting takes $\mathcal{O}(\sigma)$ time, and crossing an edge takes $\mathcal{O}(1)$ time.*

3 Implementation

Our implementation takes as input a value $k \leq 32$[1] and a fasta file containing reads from which to contruct the De Bruijn graph. The implementation next constructs the dynamic data structure for efficient representation of the De Bruijn graph described in the previous section. In particular, the data structure is composed of the following: a hash function f that takes $k-1$-mers to $\{0, ..., n-1\}$ where n is the number of nodes in the de Bruijn graph, the matrices IN and OUT that encode the edges of the de Bruijn graph, and a forest covering the nodes of the de Bruijn graph. We describe the construction of each of these in Sections 3.1 to 3.3. In addtion, our implementation allows for membership queries and dynamic edge removal and addition, which are described in Sections 3.4 and 3.5 respectively.

3.1 Hash function

The data structure relies upon a hash function f to map $k-1$-mers to $\{0, \ldots, n-1\}$ where n is the number of nodes in the de Bruijn graph. Let N be the set of $k-1$-mers from the input reads. The hash function in our implementation is a composition $h \circ g$ that is bijective on N, where g is a Karp-Rabin hash function (Karp and Rabin, 1987) that is injective on N, and h is a minimal perfect hash function (Hagerup and Tholey, 2001) on $g(N)$. We next provide definitions of Karp-Rabin and minimal perfect hash functions.

DEFINITION 1. ***(Karp-Rabin**) Suppose we have a subset S of the universe U of all possible strings of length k over an alphabet $\Sigma = \{0, \ldots, \sigma-1\}$. Given a prime P and base $r \in [0, P-1]$, a Karp-Rabin hash function g is a function defined over U such that $g(x_1 ... x_k) = (\sum_{i=1}^{n} x_i r^i) \mod P$.*

[1] k must be at most 32 because the k-mers are stored in 64 bits

DEFINITION 2. **Minimal perfect hash** *A minimal perfect hash function h for a set of size n, S, is a function defined on the universe such that h is one-to-one on S and the range is $\{0, ..., n-1\}$.*

First, we discuss the procedure to generate the Karp-Rabin hash function g; the minimal perfect hash function h is then constructed using the computed Karp-Rabin values. The prime P is chosen to be the smallest prime greater than $(k-1)n^2$ in order to give a high probability of injectivity. Next, r is chosen from a uniform distribution on 0 to $P-1$; after choosing r, we have a valid candidate for function g. The powers of $r \mod P$ and $r^{-1} \mod P$ are precomputed; to compute $r^{-1} \mod P$, it is necessary to employ the generalized Euclidean algorithm. Next, g is tested for injectivity. The above process repeats until g is injective. After g has been generated, the minimal perfect hash h on the image of g is constructed using the library BBHash (Limasset *et al.* (2017)). After this construction, the image of g is discarded, and only the precomputed powers of $r \mod P$, the value of $r^{-1} \mod P$, prime P itself, and g are stored.

The hash value of $k-1$-mer $a = a_1 \ldots a_{k-1}$ may be found by first computing the sum $\sum_{i=1}^{k-1} a_i r^i \mod P$ using the stored powers of r, which can be done in $O(k)$ time. Once the Karp-Rabin value $g(a)$ is computed, we use the perfect hash function h to find $h(g(a))$. In the case that we have the Karp-Rabin value $g(a')$ of a $k-1$-mer a' that is a neighbor of a in the De Bruijn graph, then we can update the value of $g(a')$ and get $g(a)$ in $O(1)$ time. For example, suppose a' is an out neighbor of a, i.e. $a' = a_2 \ldots a_k$. Then $g(a) = (g(a') - a_k \cdot r^{k-1}) \cdot r + a_1 \cdot r \mod P$. Similarly, if a' is an in neighbor of a, $a' = a_0 \ldots a_{k-2}$, then $g(a) = (g(a') - a_0 \cdot r) \cdot r^{-1} + a_{k-1} \cdot r^{k-1} \mod P$.

3.2 IN and OUT

The edges of the de Bruijn graph G are stored in two binary matrices, IN and OUT, each having n rows and 4 columns. The rows correspond to $k-1$-mers, while the columns correspond to letters A, G, T, and C, respectively.

To construct IN and OUT, first all k-mers are extracted from the input reads; IN and OUT are initialized to 0. For each k-mer, which represents an edge in the de Bruijn graph, we compute the hash value of the prefix $k-1$-mer and then use the hash value update described in Section 3.1 in order to find the hash value of the suffix $k-1$-mer. The corresponding entries of IN and OUT are then updated to 1. This process takes $O(km)$ time where m is the number of edges in the de Bruijn graph. Notice that an improvement on the construction time could be made if the $k-1$-mers were read in order of their appearance in each input read, since the hash value update could be used for all but the first $k-1$-mer in each read.

3.3 Forest

In this section we summarize the procedure to construct the forest, which is a division of the directed de Bruijn graph into undirected trees of bounded height, where only the $k-1$-mer of the root of each tree is stored. In our implementation σ is 4, and hence the tree heights are bounded by 3α and the minimum size of a tree is α, where $\alpha = (k-1) \lg |\Sigma| = 2(k-1)$.

The forest is constructed within a single Breadth-First-Search (BFS) through the undirected graph underlying the de Bruijn graph. The following process is performed for each connected component. We first choose a starting node for the BFS, s, in the component. s is set as a root in the forest, and its $(k-1)$-mer is stored. As we visit each node n, we set n's parent in the forest to be its parent in the BFS by storing the following 3 bits: 1 bit to indicate whether the parent is accessed via IN or OUT, and 2 bits to indicate which of the 4 letters. In addition, we store 1 bit to indicate whether n is a root in the forest or not. In order to

ensure that every tree has a height in the appropriate range, we also store new roots as we go along using the following process. For each node n in the component, we define a node $r(n)$ that is an ancestor of n in the BFS representing the root of the forest tree that n is in. Initially, $r(n) = s$ for all n. Once we have reached a node n that is of height greater than 2α from $r(n)$, we set $r' = n$, and reset the height of n to 0; that is, we remember n as a potential root and start measuring the height below this potential root. Once we have reached a node n that is of height greater than α, we store r' as a root and set $r(n) = r'$. This way, both the new tree with root r' and the tree we have broken off from are both of height at least α and at most 3α; since a tree of height α contains at least α nodes, this procedure ensures the minimum size of each tree as well. The only exception is if a connected component is of smaller size than α; in this case, a single tree is created by the above procedure that spans the connected component.

3.4 Membership Query

Given a $k-1$-mer x, one can query the data structure for membership of x in the nodes of the de Bruijn graph. We describe the implementation of this membership query in this section.

Whether a $(k-1)$-mer x is a member of the data structure can be found by travelling up towards a root in the forest. First, we hash x, and we find the node in the forest corresponding to this hash value. Using the data stored for that forest node and x's $k-1$-mer, the parent's (supposed) $k-1$-mer is computed. The parent's hash value is then found by using the hash update procedure described in Section 3.1. We then verify that such an edge exists between the two $k-1$-mers in the de Bruijn graph by checking `IN` and `OUT`. If x, the $(k-1)$-mer that we are querying for membership, is not a $k-1$-mer in the graph, it may be the case that `IN` and `OUT` contradict the existence of an edge between the nodes. We can therefore eliminate the possibility of membership for some $k-1$-mers and return false through this check. While the above test has not failed, we repeat the process with the parent's $k-1$-mer until we have reached a forest root or we have moved up $6(k-1)$ times, that being the maximum tree height. In the latter case, the membership of x is returned false. Otherwise, if a root is reached, the $k-1$-mer of the root computed from travelling up the tree can be compared to the stored $k-1$-mer of the root. In this case, whether x is a member depends on whether the two are equal.

3.5 Updating for dynamic graphs

The data structure is dynamic with respect to edge addition and removal. In this section, we describe the procedure for updating the data structure. Both edge addition and edge removal use a tree merging procedure, which we describe first.

Merge trees procedure The merge trees procedure takes as input an ordered pair of nodes (u, v) such that edge (u, v) or (v, u) is in the De Bruijn graph and merges their respective trees T_u, T_v into a single tree. The procedure to merge the trees works as follows. First, we reverse all of the forest edges from u to its root r_u, ensuring that all of T_u is below u. Next, we unstore r_u as a root. Finally, we update the forest edge of u to ensure that its parent is v.

Edge addition procedure When an edge (u, v) is added between two k-mers, `IN` and `OUT` may be updated in constant time. Suppose u belongs to tree T_u and v belongs to tree T_v. If neither T_u nor T_v is below the minimum size, or if $T_u = T_v$, the procedure exits since no update to the forest is necessary. Otherwise, suppose both T_u and T_v are below the minimum size α. In this case, the merge trees procedure is called with input (u, v). If exactly one of the trees is below the minimum size, let $s, l \in \{u, v\}$ be the nodes corresponding to the smaller, larger trees, respectively. Then, if the depth of l is at most 2α, we simply call the merge trees procedure with pair (s, l), and the smaller tree is merged into the larger. If the depth of l exceeds 2α, we simply travel up α steps from l and store that node as a root and then call merge trees (s, l), and the smaller tree is merged into the new tree created in which l has depth α.

Edge removal procedure The edge removal procedure takes as input edge (u, v) to be deleted from the De Bruijn graph. First, `IN` and `OUT` are updated. Next, we check the forest edges of u, v to see if one contains the other as its parent in its tree. If so, the forest is modified as follows. First, the child node c is stored as a root, breaking off its subtree as a new tree in the forest. We then look at the trees T_p containing the former parent; we check if T_p is below the minimum size α. If it is, we examine each node x in T_p and look for an edge in the De Bruijn graph that is incident with both x and T_x, a tree such that $T_p \neq T_x$. If a tree T_x is found, let $y \in T_x$ such that edge (x, y) or (y, x) is in the De Bruijn graph. Then the merge trees procedure is called with pair (x, y) to merge T_p into T_x. However, the resulting tree T' may violate the height constraint, so we get check the depth d_x of x in T'. Then, we find the deepest node below x in T' (there are at most α nodes to check). If the deepest node below x is of depth greater than 3α, we create a new tree by traveling up 2α steps from the deepest node and breaking off the subtree below the resulting node by storing it as a root. After this is done, the tree containing p has been fixed so that, if possible, the minimum size and maximum height conditions are satisfied. Finally, the tree T_c containing c is checked to see if it is below the minimum size, and if it is, exactly the same procedure as above is run with T_c.

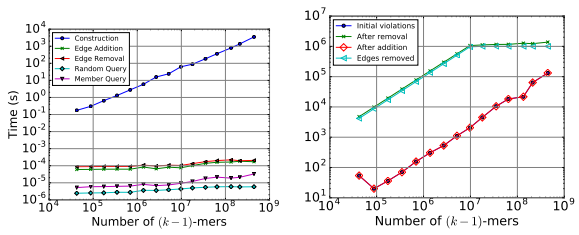
Table 1. Datasets used in the experimental evaluation.

N_{reads}	$N_{31\text{-mer}}$	Size (MB)	$\bar{\tau}$
632	42390	0.084	56
1264	87883	0.17	58
2528	174083	0.34	58
5055	347052	0.69	58
10111	692367	1.37	57
20222	1379165	2.72	58
40444	2690286	5.30	56
80888	5155725	10.1	61
161775	9751629	19.0	63
323551	17878199	34.7	75
647101	35277985	68.5	71
1294202	64285446	124	69
2588494	134080658	260	66
5176809	211650847	408	63
10353618	444356484	858	58

4 Results

We evaluate our data structure using read data from *E. coli* K-12 substr. MG1655, consisting of 27 million paired-end 100 sequence reads (NCBI SRA accession ERA000206) generated from an Illumina Genome Analyzer II. To create datasets of varying sizes, we partitioned the read data into disjoint sets of reads. For each dataset, the columns of Table 1 show the number of reads N_{reads}, the number of nodes in the De Bruijn graph $N_{31\text{-mer}}$, the size in megabytes (MB) of our constructed data structure, and the average tree height $\bar{\tau}$ in the covering forest.

To construct the De Bruijn graph, we set $k = 32$. We performed all evaluations on a server with Intel(R) Xeon(R) CPU E5-2667 @ 2.90GHz (12 cores) with 256 GB RAM.



(a) (b)

Fig. 1. (a): Time vs number of $k-1$-mers for various graph operations. (b): Tree size violations as described in the text.

Construction Time and Space In Figure 1(a) (Construction), we show the time required to construct the de Bruijn graph data structure vs. the number of $k-1$-mers in the graph. The construction time is the total time required to construct the hash function, `IN` and `OUT`, and partition the graph into the forest. In our evaluation, the construction time scaled linearly with the number of $k-1$-mers, as shown in Fig. 1(a), as predicted by the discussion before Theorem 1.

As shown in Table 1, the memory required by the data structure also scaled linearly with the number of $k-1$-mers, also in agreement with Theorem 1.

Membership Query Time Next, we tested the average time required for the data structure to answer membership queries: whether $k-1$-mer u is present in the De Bruijn graph. First, we generated 10^6 random $k-1$-mers and show the mean query time in Fig. 1(a) (Random Query) – the mean query time remains on the order of a few microseconds as the number of $k-1$-mers in the graph increases. However, since these $k-1$-mers were generated uniformly randomly, most of these $k-1$-mers were not in the De Bruijn graph. Often, the data structure is able to detect that a $k-1$-mer is not a member of the graph without a full tree traversal up to the root. Therefore, we performed a second test of the mean query time, where each $k-1$-mer is selected randomly from the set of $k-1$-mers known to be in the graph. Thus, each one of these queries requires a full tree traversal to the root node. Result are shown in Fig. 1(a) (Member Query). As expected, the member queries take slightly longer than random queries – however, even on the largest graph tested, the average time for querying a $k-1$-mer in the graph is at most a few tens of microseconds.

Dynamic Edge Deletion and Addition In order to evaluate the dynamic aspect of our data structure, we report the average time required for an edge removal and an edge addition to the De Bruijn graph. For edge removal, $\min\{10^6, 0.1m\}$ edges originally present in the De Bruijn graph were uniformly randomly selected for removal. After an edge is removed, the forest is updated as described above. After all edges were removed, we reinserted all removed edges back into the De Bruijn graph. The respective average times for this edge removal and addition process is shown in Fig 1(a). The time required to update the data structure is drastically lower than the time required to construct the structure from scratch, always by more than three orders of magnitude.

Tree violations Since $k-1=31$, the minimum size of a tree should be 62. However, due to small connected components, many trees are below the minimum size. Fig. 1(b) shows the number of size violations on each dataset initially, after the sequence of edge removals, and after all removed edges are reinserted; also shown is the number of edges removed on each dataset.

We verified that every tree below the minimum size is isolated; *i.e.*, each such tree spans its connected component and so the forest structure is optimal with respect to number of tree violations, and it remains optimal throughout the dynamic procedures. One interesting observation is that almost every edge removal created at least one small component, as shown in Fig. 1(b), where the number of tree violations (and hence small, connected components) is always greater than the number of edges removed. This fact suggests that many of the connected components of the underlying undirected graph corresponding to the De Bruijn graph are small, although above the minimum height. Also, this fact suggests that components are easily disconnected.

The mean tree height $\bar{\tau}$ in the covering forest for each dataset is shown in Table 1. Despite the inability to guarantee the theoretical minimum tree size from the presence of small connected components, the mean tree height is close to 62 for each dataset, as shown in Table 1. Therefore, the expected time and space required for the data structure still follow the results of Theorem 1 in our experimental evaluation.

5 Discussion and Conclusions

LEMMA 2. *(Belazzougui* et al. *(2016b)) If N is dynamic then we can maintain a function f as described in Lemma 1 except that:*

- *the range of f becomes $\{0,\ldots,3n-1\}$;*
- *when its domain is restricted to N, f is injective;*
- *our space bound for f is $\mathcal{O}(n(\log\log n+\log\log\sigma))$ bits with high probability;*
- *insertions and deletions take $\mathcal{O}(k)$ amortized expected time.*
- *the data structure may work incorrectly with very low probability (inversely polynomial in n).*

For node addition and removal, it is sufficient to consider addition and removal of isolated nodes. Thus, the representation of the nodes must be updated. This update requires a dynamic perfect hash function rather than a minimal perfect hash function to be implemented. Details of how to implement such a hash table can be found in Mortensen *et al.* (2005).

References

Belazzougui, D., Gagie, T., Maekinen, V., and Puglisi, S. J. (2016a). Bidirectional variable-order de bruijn graphs. In *LATIN 2016: Theoretical Informatics: 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*, volume 9644, page 164. Springer.

Belazzougui, D., Gagie, T., Maekinen, V., and Previtali, M. (2016b). Fully Dynamic de Bruijn Graphs. In *String Processing and Information Retrieval*, pages 145–152.

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

Boucher, C., Bowe, A., Gagie, T., Puglisi, S. J., and Sadakane, K. (2015). Variable-order de bruijn graphs. In *Data Compression Conference (DCC), 2015*, pages 383–392. IEEE.

Bowe, A., Onodera, T., Sadakane, K., and Shibuya, T. (2012). Succinct de bruijn graphs. In *International Workshop on Algorithms in Bioinformatics*, pages 225–235. Springer.

Chikhi, R. and Rizk, G. (2013). Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, **8**(1), 22.

Hagerup, T. and Tholey, T. (2001). Efficient minimal perfect hashing in nearly minimal space. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 317–326.

Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*,

31(2), 249–260.

Limasset, A., Rizk, G., Chikhi, R., and Peterlongo, P. (2017). Fast and scalable minimal perfect hashing for massive key sets. *ArXiv*.

Mortensen, C. W., Pagh, R., and Patraccu, M. (2005). On dynamic range reporting in one dimension. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 104–111. ACM.

Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J. M., and Brown, C. T. (2012). Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, **109**(33), 13272–13277.

Salikhov, K., Sacomoto, G., and Kucherov, G. (2014). Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms for Molecular Biology*, **9**(1), 2.

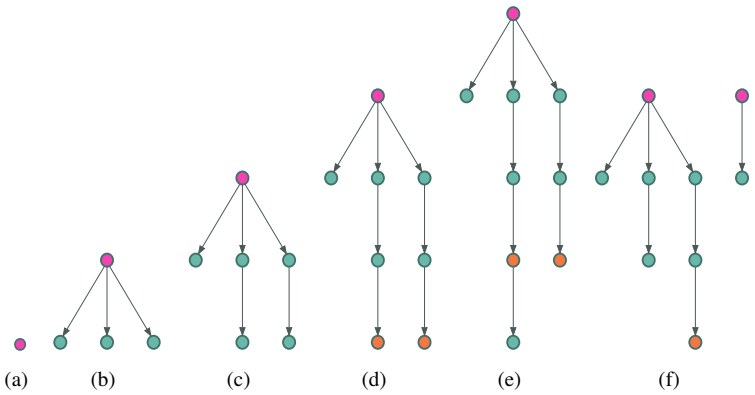


Fig. 2. Illustration of Alg. 1, with $\alpha = 1$. The pink nodes are roots that are stored in the forest. In stage (d), the orange nodes are at a height of $2\alpha + 1$, and so are potential roots. In stage (e), the node at the bottom is at a height of $3\alpha + 1$, and so the potential root is stored in the forest, forming a new tree as shown in stage (f).

The construction of the `Forest` class takes place in function `construct_forest`, which is a member function of the `FDBG` class.

Algorithm 1: `construct_forest`(G, r)

Input: graph G, root r

AlgoLine1.1 $S = \emptyset, Q = \emptyset$;
AlgoLine1.2 $p(r) = NULL, p_1(r) = r, p_2(r) = r, h(r) = 0.$;
AlgoLine1.3 $store(r)$;
AlgoLine1.4 $Q.enqueue(r)$;
AlgoLine1.5 **while** $Q \neq \emptyset$ **do**
    AlgoLine1.6 $c = Q.dequeue()$;
    AlgoLine1.7 **for** $n \in N(c)$ **do**
        AlgoLine1.8 **if** $n \notin S$ **then**
            AlgoLine1.9 $Q.enqueue(n)$;
            AlgoLine1.10 $S = S \cup \{n\}$;
            AlgoLine1.11 $p(n) = c$;
            AlgoLine1.12 $h(n) = h(c) + 1$;
            AlgoLine1.13 **if** $h(n) \leq \alpha$ **then**
                AlgoLine1.14 $p_1(n) = p_1(c)$;
                AlgoLine1.15 $p_2(n) = p_2(c)$;
            AlgoLine1.16 **end**
            AlgoLine1.17 **if** $\alpha < h(n) \leq 2\alpha$ **then**
                AlgoLine1.18 $store(p_1(c))$;
                AlgoLine1.19 $p_1(n) = p_1(c)$;
                AlgoLine1.20 $p_2(n) = p_1(c)$;
            AlgoLine1.21 **end**
            AlgoLine1.22 **if** $h(n) = 2\alpha + 1$ **then**
                AlgoLine1.23 $h(n) = 0$;
                AlgoLine1.24 $p_1(n) = n$;
                AlgoLine1.25 $p_2(n) = p_1(c)$;
            AlgoLine1.26 **end**
        AlgoLine1.27 **end**
    AlgoLine1.28 **end**
AlgoLine1.29 **end**

LEMMA 3. *At termination of FOREST, associate each node u with root $p_1(u)$, if $p_1(u)$ is stored. Otherwise, associate node u with root $p_2(u)$. Then G is partitioned into forest in $O(n + m)$ time, where each node u is in a tree of height $\alpha \leq T(u) \leq 3\alpha$*

Proof sketch. Let $u \in G$, which is within a subtree rooted as described in statement of the lemma. Suppose the root is $p_1(u)$. Then u is a descendant of $p_1(u)$ of height at most 2α. Suppose the root is $p_2(u)$. Then u is descendant of $p_2(u)$ of height at most 3α.

Furthermore, suppose $u \in G$ is stored. Then there is descendant v of u with $p_1(v) = u$ of height $\alpha + 1$.

Algorithm 2: `CheckMembership`(x)

Input: `kmer_t` x
Result: True if x is a node in the De Bruijn graph, false otherwise

AlgoLine2.1x_{kr} = `get_karp_rabin`(x)
AlgoLine2.2x_{hash} = `get_hash`(x_{kr})
AlgoLine2.3**if** $x_{hash} \notin \{0, ..., n-1\}$ **then**
    AlgoLine2.4**return** $false$;
AlgoLine2.5**end**
AlgoLine2.6i = 1
AlgoLine2.7**while** $(is\_forest\_root(x_{hash}) == false)$ **do**
    AlgoLine2.8p, p_{hash}, p_{kr} = `get_parent`(x, x_{hash}, x_{kr})
    AlgoLine2.9**if** $verify\_edge(p, x, p_{hash}, x_{hash}) == false$ **then**
        AlgoLine2.10**return** $false$
    AlgoLine2.11**end**
    AlgoLine2.12$x, x_{hash}, x_{kr} = p, p_{hash}, p_{kr}$
    AlgoLine2.13i++
    AlgoLine2.14**if** $(x_{hash} \notin \{0, ..., n-1\}) \vee (i > 3\alpha)$ **then**
        AlgoLine2.15**return** $false$;
    AlgoLine2.16**end**
AlgoLine2.17**end**
AlgoLine2.18**if** p == $stored\_kmer(p_{hash})$ **then**
    AlgoLine2.19**return** $true$
AlgoLine2.20**end**
AlgoLine2.21**return** $false$