Bioinformatics
doi.10.1093/bioinformatics/xxxxxx

Implementation and evaluation of an efficient dynamic data structure for De Bruijn graphs

Project Proposal

Alan Kuhnle [1], Victoria Crawford [1], Yuanpu Xie [2], Zizhao Zhang [1], Ke Bo [2]

[1] Computer & Information Science & Engineering, University of Florida, Gainesville, 32306 and
[2] Biomedical Engineering, University of Florida, Gainesville, 32603

Abstract

Compact data structures for the representation of de Bruijn graphs are important for genome sequence assembly. A data structure supporting efficient and exact membership queries for fully dynamic de Bruijn graphs was recently proposed in Belazzougui *et al.* (2016). For our project, we will implement this data structure and evaluate its practical performance as compared to popular implementations based upon bloom filters when used for de Bruijn construction from sequence data.

1 Introduction

The de Bruijn graph is a data structure which plays an important role in second-generation sequencing applications, which tend to yield a large number of short sequence fragments (Conway and Bromage, 2011). It was first introduced by Idury and Waterman (1995) for *de novo* assembly of DNA sequences. Given an alphabet set Σ of σ symbols and a set of sequences $\mathcal{S} := \{S_1, S_2, ..., S_t\}$, the de Bruijn graph of order k on $\mathcal{S}$ is defined as follows. Let M_k denote the sets of k-mers (sequence of length k) that occur as subsequence in S_i for some i. A directed graph with vertex set M_k is created, with directed edge e connecting node $s \in M_k$ to node $r \in M_k$ if and only there exists a subsequence of some S_i of length $k+1$ with s as a prefix and r as a suffix. An example is shown in Fig. 1.

For sequence assembly, the set S is a collection of overlapping short DNA fragments, called reads. The task is to find an ordering of these fragments such that a longer DNA sequence containing these fragments is revealed. The longer sequence should correspond to an Eulerian walk of the de Bruijn graph.

However, while methods of gathering sequence data has greatly improved with second-generation sequencing technology, *de novo* assembly of the large volumes of short reads from second generation sequencing remains difficult due to the computational complexity of the problem as well as memory requirements (Conway and Bromage, 2011). In particular, a key issue of using de Bruijn graph in practice is the high memory occupation for certain organisms. The human genome encoded in a de Bruijn graph with a k-mer size of 27 requires 15GB to store the node sequences (Chikhi and Rizk, 2013). Graphs for larger genomes cannot even be constructed on typical lab clusters as the total volume of data may approach hundreds of gigabytes (GB) (Salikhov *et al.*, 2014). Further, many *de novo* assemblers require that the graph have a representation in-memory, and therefore space-efficiency is vital (Chikhi and Rizk, 2013)

In Belazzougui *et al.* (2016), a data structure for de Bruijn graphs that is fully dynamic (nodes and edges may be inserted and deleted efficiently) is proposed. k-mers are represented by integers using a combination of Karp-Rabin hashing and minimal perfect hashing, thereby allowing the graph to occupy a greatly reduced amount of memory. A partition of the underlying undirected graph into a forest along with an encoding of edge information in matrices IN, OUT allows efficient membership queries with no error.

Proposed work For our project, we will implement and evaluate the data structure described in Belazzougui *et al.* (2016) to store dynamic De Bruijn graphs. In Section 1.1, we discuss related data structures for compact de Bruijn representation, while in Section 2 we describe the data structure and how we plan to implement the required hash function and tree partition. In Section 3, we discuss the particular metrics on which we will evaluate the data structure, and to which implementations of alternative data structures we will compare.

1.1 Related work

There are numerous other papers on data structures for de Bruijn graphs that decrease the memory needed to store the graph. Using hash functions in order to represent k-mers, which correspond to nodes in the graph, in a more compact form is one possibility. One approach is a Bloom filter based data structure. A Bloom filter (Bloom, 1970) is a space-efficient, probabilistic data structure built upon multiple hash functions that is used to test whether an element is in a set, with the possibility of false positives. Each k-mer of a de Bruijn graph may be stored in a Bloom filter as was done by Pell *et al.* (2012). They were able to store the graph in as little as 4 bits per k-mer, but not with exact accuracy due to the nature of a Bloom filter. Because of this inexactness it is possible that a Bloom filter introduce false nodes and branching, and so Chikhi and Rizk (2013) proposed encoding the de Bruijn graph by 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) also used Bloom filters with false positive detection but was able to further reduce the memory by 30-40% by using Cascading Bloom filters. One disadvantage is that de Bruijn graphs based upon Bloom filters usually are semi-dynamic at best, typically only supporting insertions (Belazzougui *et al.*, 2016).

Another direction is to represent the graph using succinct data structures. A succinct data structure is one that uses an amount of space that is bounded closely by the theoretical minimum while still supporting queries efficiently (Conway and Bromage, 2011). Conway and Bromage (2011) gave a lower bound for the number of bits required to store a de Bruin graph, and then proposed a succinct de Bruijn data structure using bitmaps with rank and select operations near to this lower bound. Bowe *et al.* (2012) proposed a succinct representation based upon Burrows-Wheeler transforms (Burrows and Wheeler, 1994) where the graph could be represented in $4m + o(m)$ bits, m being the number of edges. Boucher *et al.* (2015) augmented the previous work to build a representation that given a k-mer size K, constructs all de Bruijn graphs with k-mer size $k \leq K$ using only twice the space.

Finally, there is the possibility of not storing the entire de Bruijn graph, but rather a compact approximation. Some of the work mentioned above based upon Bloom filters would fit into this category. In addition, Ye *et al.* (2012) constructed a sparse version of the graph that included only a small portion of the k-mers while having high assembly accuracy.

The data structure proposed in this paper is most similar to the Bloom filter based representations, but is fully dynamic, returns exact answers to membership queries, and gives better theoretical bounds provided that the number of connected components in the graph is small (Belazzougui et al., 2016).

2 Description and implementation of Data Structure

In this section, we describe the data structure for fully dynamic De Bruijn graphs defined in Belazzougui *et al.* (2016). In addition to the description, we discuss how to implement the hash function and the construction of the forest partition for membership queries.

2.1 Overview

Suppose we have De Bruijn graph $G = (V, E)$ with n vertices, where each vertex $v \in V$ is a k-mer; $v = a_0a_1 \dots a_{k-1}$, where each $a_i \in \Sigma$ and $|\Sigma| = \sigma$. The data structure relies on a hash function f to map the k-mers corresponding to each node in the De Bruijn graph into a more succinct form. The details of the hash function depend upon whether the graph is expected to have dynamic or static nodes. The edges of G are stored in two binary matrices, IN and OUT, each of size $n \times \sigma$. Edges in G are stored in these matrices in the following way:

$$(u = ba_1 \dots a_{k-1}, v = a_1a_2 \dots a_{k-1}c) \in E$$
$$\iff OUT(f(u), c) = 1, IN(f(v), b) = 1.$$

For static graphs, the definition of f is described in Section 2.2.1, and the construction of IN, OUT is described in Section 2.2.2.

Membership queries Given a k-mer w, the procedure to test if $w \in G$ is as follows. Let $i = f(w)$. For each index j adjacent to i by IN or OUT, we have a corresponding k-mer w' and $w' \in G \iff w \in G$. Therefore, by following edges in IN, OUT, we can obtain a connected component in the undirected graph G' underlying G, such that either all the corresponding k-mers are in G or none of them are. Therefore, if we can store a set of k-mers R such that any node $v \in G'$ is connected to a node in R, then we may verify if a k-mer w is in the graph by traveling via IN, OUT from $f(w)$ to $f(r)$ for some $r \in R$ and comparing the computed k-mer r' from w to r. Then $w \in G \iff r' = r$. The set R is constructed as the roots of a bounded-height partition of G' into a forest; the process of this construction is detailed in Section 2.2.3.

In Section 2.3, we describe how to efficiently update this data structure in response to edge insertions and deletions for the De Bruijn graph G, and also how the hash function f must be implemented to allow node addition and removal.

2.2 Data structure for static graphs

2.2.1 Hash function *f*

Description The hash function proposed for a De Bruijn graph with static nodes is a combination of Karp-Rabin (Karp and Rabin, 1987) and minimal perfect hashing (Hagerup and Tholey, 2001). Suppose we have a subset S of the universe U of all possible strings of length k over an alphabet. Given a prime P and base $r \in [0, P-1]$, a Rabin-Karp hash function f is a function defined over U such that $f(x_1...x_k) = (\sum_{i=1}^{n} x_i r^i) \text{mod} P$. A minimal perfect hash function f for S is a function defined on the universe such that f is one-to-one on S and the range is $\{0, ..., n-1\}$. A hash function with the properties described in Lemma 1 of the paper may be constructed with high probability in $O(kn)$ time by composing a minimal perfect hash function with a Karp-Rabin hash function, described in more detail below.

Implementation The algorithm to build the hash function described in the paper is outlined in GENERATEHASH. It is our intention to begin with an implementation of a minimal perfect hash function, and to build within/upon this implementation to get the hash function described in the paper. Ideally, the minimal perfect hash function will already be tailored for De Bruijn graphs, as in the software GATB (Drezen *et al.*, 2014). But, it is possible that building into large software like GATB will prove to be too difficult, in which case we will build upon a general purpose implementation of a minimal perfect hash function as in BBHash (Limasset *et al.*, 2017). In addition, if both of the above prove to not be practical, we have the option of not implementing the exact hash function described in the paper but instead using the hash function already provided by an implementation such as GATB, assuming it has desirable enough properties. If all of the above are impossible, then we can implement our own minimal perfect hash function as described in Hagerup and Tholey (2001).

Algorithm 1: GENERATEHASH(S)

```
   Input: $S$, a set of $n$ $k$-tuples over an alphabet $\sum$ of size $\sigma$
1  $R$ = max($\sigma$, $kn^2$);
2  $P$ = getPrime($R$);
3  $r$ = randomNumber(0, $P-1$);
4  $f$ = rabinHash($r$, $P$);
5  while isInjective($f$, $S$) is FALSE do
6      $r$ = randomNumber(0, $P-1$);
7      $f$ = rabinHash($r$, $P$);
8  end
9  $g$ = minimalPerfectHash($f(S)$);
10 return $g \circ f$;
```

The particular functions for the algorithm are as follows

1. **getPrime(R)** returns the smallest prime greater than R. Such a prime may be generated with high probability in $O(n)$ time by the method

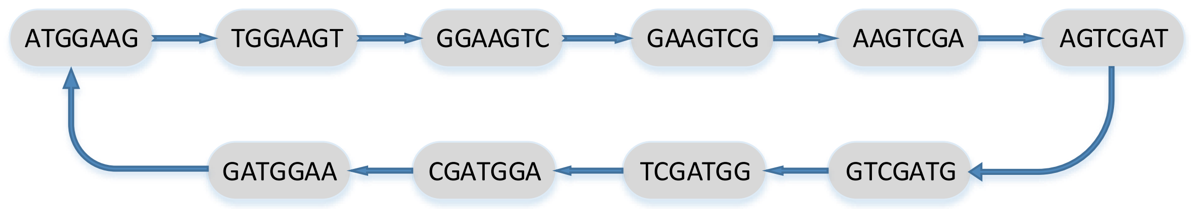

Fig. 1. The de Bruijn graph constructed from string ATGGAAGTCGATGGAAG, with $k = 7$.

described in (Dietzfelbinger *et al.*, 1997). Roughly, some samples in the interval $[R+1, 2R-1]$ are tested until a prime is found.

2. **randomNumber(0,** $P-1$) returns a uniformly distributed random number between 0 and $P-1$.
3. **rabinHash(**r**,** P) returns a Rabin-Karp Hash function with base r and mod P. There is a high probability this is injective.
4. **isInjective(**f**,** S) tests whether f is injective on S.
5. **minimalPerfectHash(**X) returns a minimal perfect hash function on the set X. This may be done by using existing implementations or by the method described in (Hagerup and Tholey, 2001).

2.2.2 IN and OUT

The matrices IN and OUT maintain edge information for each node. Once the hash function f has been constructed, suppose we have k-mers from a single text sorted by their starting position in the text. Then, initializing updating IN and OUT reduces to computing f on each k-mer in the sequence. Since each k-mer in the sequence agrees with the next in $k-1$ positions, once f has been computed for the initial k-mer, it can be updated in constant time to compute each of the rest of the sequence. More generally if the set of k-mers N consists of t reads, the construction step requires time

$$O(kt + n + n\sigma).$$

2.2.3 Forest construction

Given a bound α, we described how to partition an undirected graph H into a forest $\mathcal{F}$ where each $T \in \mathcal{F}$ with $\alpha \le h(T) \le 3\alpha$, where $h(T)$ is the height of tree T, i.e. the longest path from the root to a leaf. The procedure can be summarized as follows: start with a spanning tree of H. For each path down the tree of length exceeding 3α, break off the subtree corresponding to the node at position 2α down the path. It is of height at least α, so we maintain that every tree in the forest is of height at least α. Furthermore, the original tree we started with is now of height at most 3α (and at least 2α). Repeat this process on each tree of height exceeding 3α until all trees meet the condition.

Lemma 1. *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 $\alpha \le T(u) \le 3\alpha$*

Proof. 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: FOREST(G, r)

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

2.3 Updating for dynamic graphs

For edge addition and removal, IN, OUT and updating the covering forest can be updated as described in Belazzougui *et al.* (2016); we briefly summarize the procedure here:

Edge addition When an edge (u, v) is added between two k-mers, IN and OUT may be updated in constant time. The procedure for updating the covering forest depends on the size of the trees $T(u), T(v)$ containing u, v respectively. If both trees are of size not less than α, then there is no change. If both are less than α, they are merged. For the last case, suppose without loss of generality that $T(u) > \alpha$ and $T(v) < \alpha$. Then if the depth of u is less than 2α, we simply merge the trees. Otherwise $T(u)$ is broken into two subtrees preserving the condition.

Edge removal Again, IN and OUT may be updated in constant time. If the removed edge is not in any tree of the covering forest, no update is necessary. Otherwise, a tree division similar to the one in the edge addition case may be required.

Node addition / removal 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. Time permitting, we will implement the dynamic version of the hash function, which has the properties outlined in Lemma 2. The dynamic version resembles the static version, however the prime for the Rabin-Karp will be chosen according to an upper bound placed on n, and a dynamic perfect hash table rather than a minimal perfect hash table must be implemented. Details of how to implement such a hash table can be found in Mortensen *et al.* (2005).

3 Experimental evaluation

In this project, we will use C++ to implement the data structure as described in Section 2. The evaluation will be performed on a server with Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz (18 cores) with 396 GB RAM. Our source-code will be shared in a github repository under an open-source license.

3.1 Evaluation

To evaluate our implementaion of the data structure, we will test the *construction time*, *memory usage*, and *membership query time* metrics when employed on static De Bruijn graphs created from real genome read data, described in Section 3.2, with $k = 20$. We will compare the performance on these metrics to the following compact data structures for De Bruijn graphs:

- Bloom filter implementation in the de Bruijn package of GATB (Drezen *et al.*, 2014)
- Implementations of variants of bloom filters as described in (Pellow *et al.*, 2016). These implementations are designed to reduce the false positive rate while still providing fast set containment queries; also, they are designed to improve space requirements by leveraging k-mer overlap information. Our implementation guarantees a false positive rate of 0; still, we can experimentally test this and compare to the false positive rates in these implementations.

In addition, we will evaluate the efficiency of adding and removing edges from the data structure. Time permitting, we will evaluate running time for the addition and removal of nodes as well, but this requires updating the hash function as described in Section 2.3.

3.2 Data description

We will test the data structures on the datasets summarized in Table 1, which provide different sizes on which to test the data structures. To create k-mer membership queries, we will select 1 million k-mers in the input and mutate one base, as in Pellow *et al.* (2016); these queried k-mers will resemble the sequence data but still provide negatives on which to test the false positive rates of the bloom filter implementations and the performance of the forest partition.

Table 1. Datasets

Accession number	Type	Read count	Read length
SRX182671	human RNA-seq	66,396,200	49
chr15	human chromosome	1	81 Mbp
ERA000206	*E. coli*	$27 \cdot 10^6$	100

References

Belazzougui, D., Gagie, T., Maekinen, V., and Previtali, M. (2016). 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.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm.

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.

Conway, T. C. and Bromage, A. J. (2011). Succinct data structures for assembling large genomes. *Bioinformatics*, **27**(4), 479–486.

Dietzfelbinger, M., Hagerup, T., Katajainen, J., and Penttonen, M. (1997). A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, **25**(1), 19–51.

Drezen, E., Rizk, G., Chikhi, R., Deltel, C., Lemaitre, C., Peterlongo, P., and Lavenier, D. (2014). Gatb: Genome assembly and analysis tool box. *Bioinformatics*, **30**, 2959–2961.

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. Springer.

Idury, R. M. and Waterman, M. S. (1995). A new algorithm for dna sequence assembly. *Journal of computational biology*, **2**(2), 291–306.

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 e-prints*.

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.

Pellow, D., Filippova, D., and Kingsford, C. (2016). Improving bloom filter performance on sequence data using k-mer bloom filters. In *International Conference on Research in Computational Molecular Biology*, pages 137–151. Springer.

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.

Ye, C., Ma, Z. S., Cannon, C. H., Pop, M., and Douglas, W. Y. (2012). Exploiting sparseness in de novo genome assembly. *BMC bioinformatics*, **13**(6), S1.