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

OXFORD

Dynamic de Brujin Graphs: Adding Dynamic Vertices

Anthony Colas[1], Ayush Khandelwal[1], Harish Balaji[1], and Shubham Shukla[1]

[1]Department of Computer and Information Science, University of Florida, Gainesville, 32611

Abstract

De Bruijn Graphs provide a way to represent extremely large genome sequences in terms of its k-mer components, and recent research has been heavily focused on efficiently implementing this structure. These graphs have many uses in bioinformatics and have been commonly used in de novo assembly. In *Fully Dynamic de Bruijn Graphs*, Belazzougui *et al.* (2016b) has proposed a fully dynamic, space-efficient and time-efficient implementation of de Bruijn graph. Our goal is to implement and evaluate this approach for addition and deletion of vertices dynamically.

1 Introduction

The kth-order de Bruijn graph is defined as the directed graph whose nodes are distinct k-tuples contained within strings and in which there is an edge from u to v if there is a (k+1)-tuple connecting those strings with a prefix of length k is u and whose suffix of length k is v (Belazzougui *et al.*, 2016b). Because genome datasets can have a size of a few gigabytes, de Brujin graphs representing these sequences can get overwhelming and their implementation intractable. Hence, there arises a need to represent these massive structure in the most minimalistic way possible. Fully dynamic de Brujin graph is one such efficient way to represent these graphs with better space and time bounds.

1.1 The Biological Problem

One applications of de Bruijn graph is de novo assembly. The intuition behind de novo assembly is to start assembly with the assumption that we have no prior knowledge of the genome. We need to make sense of the DNA sequences by analyzing sequences of smaller reads forming longer patterns. We can extract the nucleotide sequence from a path once we know the initial k-mer of the first node and the sequences of all the nodes in the path, because adjacent k-mers overlap by k-1 nucleotides. Each node of the de Bruijn graph represents a series of overlapping k-mers (Zerbino and Birney, 2008).

1.2 The Computational Framework

The set of basic DNA fragments obtained after sequencing are called reads. These are used to create the de Bruijn graph which is a directed graph based on how edges are defined from source vertex to destination vertex. A directed graph that visits each edge is an Eulerian walk and an Eulerian walk of de Bruijn graph will help find longer DNA sequences consisting of existing reads present in the node. In the section above, we have highlighted the importance of de Bruijn graphs. Static de Bruijn graphs can be used for smaller applications like analyzing after shotgun sequencing where you have a defined number of reads after sequencing. However, in case of read corrections, these static graphs have no way to update the graph. Therefore, there is need to make insertions and deletions in the de Bruijn to solve this problem. Belazzougui *et al.* (2016b) introduces fully dynamic de Bruijn graphs to update the read by not only inserting, but also deleting edges and vertices from the graph, especially in de novo assembly when we are continuously receiving reads.

Compact representations of de Bruijn graphs have been based on Bloom filters (Pell *et al.*, 2012; Chikhi and Rizk, 2013; Salikhov *et al.*, 2013) and the Burrows-Wheeler Transform (Bowe *et al.*, 2012; Boucher *et al.*, 2015; Belazzougui *et al.*, 2016b). Belazzougui *et al.* (2016b) proposes a new approach that is based on minimal perfect hash functions (Mehlhorn, 1982) which performs better than Bloom filters, in that it's semi-dynamic and supports only insertions. Minimal perfect hashing has better theoretical bounds for a small connected graph, that is fully dynamic: insertion and deletion of nodes and edges can be done efficiently. Belazzougui *et al.* (2016b)'s data structure used a combination of both Karp-Rabin hashing (Karp and Rabin, 1987) and minimal perfect hashing (Mehlhorn, 1982).

Perfect hash functions are used to map distinct elements of a set S to a set of unique integer values. Minimal perfect hashing ensures that the mapping is not only collision-free but also compact without any vacant slots in the hash table (Mehlhorn, 1982). Using this gives us the ability to both insert and delete vertices and edges. In Section 3.3, we expand on both the Karp-Rabin and minimal perfect hashing algorithms utilized to map the k-mers to the hash values. Additionally, we explain how they are integrated into a forest of nodes in order to efficiently represent a de Bruijn graph.

1.3 Problem Statement

In the de Bruijn graph, vertices are representation of k-mers. As and when we are sequencing new reads, we will need to add new vertices dynamically. Our problem lies in creating a data structure to emulate dynamic de Bruijn graphs that allows adding new vertices.

1.4 Implementation

This project implements and evaluates the data structures proposed in *Fully Dynamic de Bruijn Graphs* (Belazzougui *et al.*, 2016b) with respect to the dynamic vertices, that is adding and deleting k-mers. We have evaluated the performance of the data structure while using a modified minimal perfect hashing function to handle changes and collisions in the hash table. We then represent the de Bruijn graph succinctly with an IN and OUT matrix, where IN resembles all the incoming edges and OUT the outgoing edges. Finally, we construct a forest consisting of shallow rooted trees, where each unique k-mer is represented by a vertex. This structure tests for false positives caused by a collision in the minimal perfect hashing function.

Section 2 surveys over the related work, including some current implementations of semi-dynamic de Bruijn graphs. Section 3 will fully explain the data structures. Section 4 details our results and our evaluation, while also providing details about the dataset we test on. Current limitations will also be briefly discussed. Section 5 discusses our results found in each of the result tables. In Section 6 we discuss future work and conclude. Section 7 concludes are implementation and provides a link to the source code.

2 Related Work

There has been a copious amount of work denoting how to efficiently store and represent de Bruijn graphs. De Bruijn graphs in sequencing were first employed in Euler-SR (Chaisson and Pevzner, 2008), Velvet (Zerbino and Birney, 2008), and ABySS (Simpson *et al.*, 2009). Chaisson and Pevzner (2008) store the graph using a hash function and adjacency list, while Simpson *et al.* (2009) represent the de Bruijn graph as a distributed hash table and parallelize the assembly of billions of short reads and were able to store a set of reads from the human genome in 336 GB. Chaisson and Pevzner (2008) construct the de Bruijn graph using O(L) * (k+1) bytes, where L is the length of the genome, k is the amount of memory alloted to each vertex, and l is the number of adjacent pair tuples in a set of reads.

As previously mentioned, more recent research has utilized both Bloom filters (Pell *et al.*, 2012; Chikhi and Rizk, 2013; Salikhov *et al.*, 2013) and the Burrows-Wheeler Transform (Bowe *et al.*, 2012; Boucher *et al.*, 2015; Belazzougui *et al.*, 2016a) in order to more efficiently represent de Bruijn graphs in memory. Bloom filters are a space-efficient randomized hash-based data structure, which tests whether an element is in a set when queried. Although Bloom filters can present false positives and support only insertions (Pell *et al.*, 2012), they are typically utilized because they minimize space usage (Kirsch and Mitzenmacher, 2006). Pell *et al.* (2012) were the first to store the assembly de Bruijn graph as Bloom filters. They were able to store k-mers in as little as 4 bits. However, the authors do acknowledge the trade-off between memory and accuracy, claiming a false positive rate of 15% when storing each k-mer in under 4 bits. Chikhi and Rizk (2013) refine Pell *et al.* (2012)'s work by detecting and storing critical false positives (cFP) that cause false branching. Even by adding the cFP structure, Chikhi and Rizk (2013) report that their structure has comparable memory usage to the one without false positive detection, storing the human genome in 5.7 GB. This is a significant improvement from Conway and Bromage (2011) who's structure takes 32 GB to store the human genome. Salikhov *et al.* (2013) improved Chikhi and Rizk's structure by using cascading Bloom Filters in order to consume 30% to 40% less memory. Compared to Chikhi and Rizk's structure which took up about 13 to 15 bits per k-mer, Salikhov *et al.* (2013)'s structure uses 8.5 to 9 bits per k-mer.

The Burrows-Wheeler transform permutes a character string in order to perform compression and indexing (Wheeler *et al.*, 2000). Bowe *et al.* (2012) implements a succinct de Bruijn graph based on the XBW-transform, which is an extension of the Burrows-Wheeler transform used for storing trees. Conway and Bromage (2011) define succinct data structures as those representing a sequence of integers with their space bounded closely by the theoretical minimum. With the succinct de Bruijn graph, Bowe *et al.* (2012) was able to represent a k-mer graph with m edges in 4m + O(m) bits and the same set of reads of the human genome as Simpson *et al.* (2009) in 2.5 GB. Bowe *et al.* (2012)'s development is referred to as BOSS. Additionally, they were able to make the graph in O(Nklogm/loglogm) time. Boucher *et al.* (2015) expand on Bowe *et al.* (2012)'s work by allowing multiple de Bruijn graphs of different orders to be represented in a single succinct data structure. This data structure also allows the order to be altered, while increasing the space usage by a factor of two. Belazzougui *et al.* (2016a) use a bidirectional Burrows-Wheeler transform in order to augment to BOSS and traverse the edges of a de Bruijn graph both forward and backward. In order to represent colored de Bruijn graphs efficiently, Belk *et al.* (2016) use a generalizable succinct data structure (referred to as VARI) and are able to store plant reference genomes in 19 GB.

Cazaux *et al.* (2016) proposed a method to update the order of the de Bruijn graph order by utilizing a Generalized Suffix Tree or a Generalized Suffix Array. Although this allows an update of the graph without reconstructing the whole graph, Cazaux *et al.* (2016) focus on updating the whole k-mer graph order and not on adding or removing vertices. Our focus is on dynamically performing updates on the current graph vertices, not the graph order.

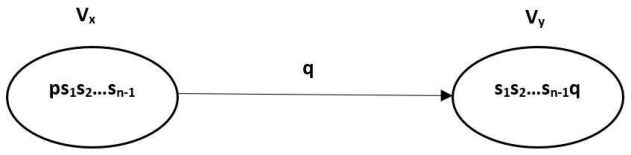
The fully dynamic Bruijn graph will closely resemble the Bloom filter approach and be based on minimal perfect hashing (Mehlhorn, 1982) and Karp-Rabin hashing (Karp and Rabin, 1987). Contrary to Bloom filter implementations, our approach will be able to delete vertices and have a better theoretical bounds when the number of connected components in the de Bruijn graph is comparably small (Belazzougui *et al.*, 2016b).

3 Approach

The method used towards addressing the problem of dynamic de-brujin graph can be addressed by dividing the problem into two:

1. Efficient storage of de-brujin graph
2. Handling the addition and deletion of nodes in the graph

We use a matrix representation to store the de-brujin graph and support addition and deletion of nodes, while using additional data structure of forest to handle the false positives that might arise in case of a query or deletion of a string. The matrix is represented using 2D BitArray of the size nXσ where n represnts the number of nodes in the de-brujin graph and σ represents the size of the alphabets('A', 'C', 'G', 'T', 'N'). The main challenge lies in detection of false positives during vertex deletion and search. The presence of these false positives arise the need for usage of an additional data structure, which is forest in this case.Both the data structures, the IN/OUT matrix and Forest are explained in greater detail in the sections below.



(a)

	a_0	a_1	.	p	.	a_{e-1}
V_0						
V_0						
.						
$f(v_y)$				1		
.						
V_{n-1}						

IN Matrix

	a_0	a_1	.	q	.	a_{e-1}
V_0						
V_0						
.						
$f(v_x)$				1		
.						
V_{n-1}						

OUT Matrix

(b)

Figure 1: (a)An example de Bruijn Graph where an edge goes from vertex v. (b) Representation of the graph in part (a) through two binary matrices IN and OUT.

3.1 Static de Brujin Graphs

Let G denote a de Brujin Graph of order k, such that each vertex of the graph represents a tuple of length k and is distinct, while the edge represents a tuple of length (k+1) such that its prefix of length k is the tuple corresponding to the source vertex while the its suffix of length k is the tuple corresponding to the destination vertex. Now, let N be the set of graph nodes and E the set of graph edges, hence what we have is:

$$G = (N, E) \tag{1}$$

$$N = v_0, v_1, ...v_{n-1} \tag{2}$$

$$E = a_0, a_1, ...v_{e-1} \tag{3}$$

Here, v_i, where $v_i \in$ N,represents a node or a k-tuple and is referred with either, interchangeably. While a_i is a single element belonging to the set of alphabet(Σ), hence $a_i \in \Sigma$

The de Bruijn Graph G is represented by two binary matrices, referred to as an IN and OUT matrix. For each node, the IN matrix stores the value for corresponding to the incoming edge while the OUT matrix stores the value corresponding to the outgoing edge. Both matrices are of size (nxσ), where n is the number of vertices and σ is the size of the set alphabet (σ = Σ). For example, let there be a vertex v_x = p$s_1 s_2 ....s_{k-1}$ and another vertex v_y = $s_1 s_2 ....s_{k-1}$q such that there exists an edge from v_x to v_y. As a result, the IN matrix for the row v_y and position indexed by a is set to 1 while in the OUT matrix for the row v_x and position indexed by b is set to 1. As defined in Lemma 1 in Belazzougui *et al.* (2016b),in case through hashing we derive a function f such that all the n distinct k-tuples are uniquely mapped to a range from 0 to n-1. These hashed values are used as indices for the rows of the IN and OUT matrix. Thus, for the above example we would have

$$IN[f(v_y)][p] = 1 \tag{4}$$

$$OUT[f(v_x)][q] = 1 \tag{5}$$

This is diagrammatically demonstrated in Fig.1(b).

In order to check for the presence of an edge between two vertices or k-tuples, we can perform a lookup in the IN and OUT matrix in O(1) time. Let's suppose we want to verify the presence of an edge from vertex with substring pX to vertex with substring Xq. We perform a look up in the IN and OUT matrix for the values IN[f(Xq)][p] and OUT[f(pX)][q]. If both the values equals 1, then we can confirm the presence of an edge, while in all other case the edge is absent. As a corollary, we can say:

$$pX \in GandOUT[f(px)][q] = 1 \implies Xq \in G \tag{6}$$

and

$$Xq \in GandOUT[f(Xq)][p] = 1 \implies pX \in G \tag{7}$$

As an inference, we can conclude:

$$if(OUT[f(pX)][q] = 1 and IN[f(Xq)][p] = 1) \tag{8}$$

then,

$$pX \in G \iff Xq \in G \tag{9}$$

As an extension to the above stated inference, we can deduce that if we have a path in the graph such that all the edges that are a part of it is verified using the IN and OUT matrices, then either all that nodes that are a touched by the path are in the graph or none of them are. (Belazzougui *et al.*, 2016b).

3.2 Dynamic de Bruijn Graph

A dynamic de Bruijn Graph is an able approach to storing the graph in a memory efficient manner. Similar to the implementation using a Bloom filter, this approach uses minimal perfect hashing to reduce the storage space and provides better theoretical bound when the number of connected components in the graph are small. It is an efficient algorithm for handling insertion and deletion of nodes and edges. Furthermore, we allow for the deletion of k-mers which cannot be handled by using Bloom filters. In this paper, we focus on implementing an efficient algorithm for handling dynamic vertices and compute its effectiveness in terms of memory usage and time complexity. We discuss methods for handling dynamic edges in Future Work.

3.3 Hashing

The memory efficiency of the IN/OUT matrix is largely due to hashing. Hashing reduces the storage space by replacing the k-mer strings by numerical values. These numerical values are then used as the indexes of the IN/OUT matrix, thus removing the need for storage of strings as the row headers. In order to use the hashed values as the row headers of the matrix, the values must satisfy 2 major conditions:

1. Values must be distinct
2. Values must be contiguous
3. Values mus be in the range of 0 to n-1, where n is the number of strings to be hashed.

In order to find such a hash function that outputs the values satisfying the above properties, a combination of two hashing algorithms has been used. First Rabin-Karp hashing is used to map the k-mers into distinct values, satisfying the above mentioned first condition. Secondly, a minimal perfect hashing is used to map these distinct values to a contiguous range starting with 0. Thus, Rabin-Karp maps the k-mer strings injectiveley to the integer range, while minimal perfect hashing makes the mapping bijective.

3.3.1 Rabin-Karp Hashing

Rabin-Karp hashing is derived from the Rabin-Karp string matching algorithm that is used to efficiently search a string from a larger string or text. Unlike the brute force approach of string matching that takes an order or O(nm) time, where n is the size of the pattern while m is the size of the text,

the Rabin-Karp is an efficient algorithm that performs the string search in O(n+m) time. The basis of the Rabin-Karp algorithm lies in its usage of hashing that maps the string to values, that are then used for comparison of the strings. In order to improve over the standard string hashing algorithm, Rabin-Karp uses a rolling hash to compute the numerical equivalents of the input strings. The rolling hash algorithm uses the following formula:

$$H = c_1 a^{k-1} + c_2 a^{k-2} + ... + c_k a^0 \quad (10)$$

Where a is a constant equal to the size of the alphabet, $c_1 .....c_k$ are the characters of the input string and k is the length of a string.

To better understand the implementation of the rolling hash for mapping a string to a value, consider the following example of finding a genome string in the entire genome read:

$$H(\text{"}ACT\text{"}) = 1 * 5^2 + 2 * 5^1 + 4 * 5^0 \quad (11)$$

Here, the alphabet set(σ) is ('A', 'C', 'G', 'T', 'N') and thus its length 5. Making the value of a as 5, while 'c' is derived using from a map storing the mapping between the character and its equivalent numerical value.

$$char\_to\_value\_map = \{A:1, C:2, G:3, T:4, N:5\} \quad (12)$$

Since the length of the query string in this case 3, we have k as 3. Thus, for the following example as well as our implementation the values of the above constants are as follows:

- $a = sizeofalphabet \Rightarrow 5$
- $c = numericalequivalenceofcharacter \Rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5$
- $k = lengthofstring \Rightarrow kmerlength$

Now, in order to compute the hash of the next string, such that first n characters of the new string are the last n characters of the previous string, we can use the mathematical advantage of the hashing formula. Let us consider one such string as "CTG". Now, in order to compute the hash of the new string, we perform the following computations:

$$H(\text{"}CTG\text{"}) = (H(\text{"}ACT\text{"}) - H(\text{"}A\text{"}))x5 + H(\text{"}G\text{"}) \quad (13)$$

Thus, by making use of the previously computed hash value , we have reduced the computation for hashing of the next string to just one multiplication, addition and subtraction which are all done in O(1) time. Calculation of hash this way ensures zero collision as it also accounts for the order of the occurrence of characters. Since, the hash function involves exponential calculation, for a large string or K-mer the hash value could become really large and thus result in overflow of the value. Thus, we require modulo operation to contain the value of the hash in operable domain. The hash function is modified to the following:

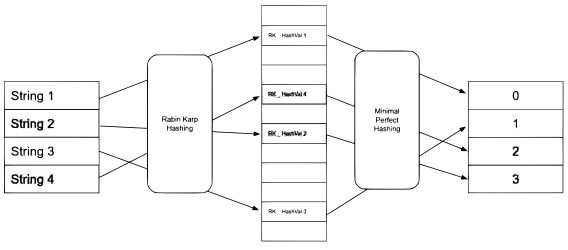


Figure 2: This figure represents the entire Hashing work flow for mapping strings to there numerical equivalent values

$$H(\text{"}CTG\text{"}) = (H(\text{"}ACT\text{"}) - H(\text{"}A\text{"}))x5 + H(\text{"}G\text{"})\%P \quad (14)$$

Where P is sufficiently large prime number. For our case, P is statically set to a 20 digit long prime number.

So, the algorithm used in stated as Algorithm 1:

Algorithm 1 Rabin Karp Hash

Input: List of nodes in de-brujin graph
Output: Map of node strings to hashed value

procedure DynamicVerterx($nodeList, , char_to_value_map, k$)
  if first node of the list **then**
    $H = c_1.5^{k-1} + c_2.5^{k-2} + ... + c_k^0 \% P$
    $c_k = ValueOfChar$
  for each of the n-1 node in nodeList **do**
    $H_new = H_{Old} -$
    $(H(\text{"}FirstCharOfStrOld\text{"})) * 5$
    $+H(\text{"}LastCharOfStrNew\text{"})$
  return $INandOUT$

3.3.2 Minimal Perfect Hashing

Perfect hashing is a technique used for building a hash table with no collisions. The concern with using this technique is that all of the keys have to be known in advance. A perfect hash function for a set A is a hash function which maps distinct elements in A to a set of integers, with no collisions.

A minimal perfect hash function is a form of perfect hashing that maps n keys to n consecutive integers. In our implementation the minimal perfect hash function's range is from 0 to n-1.

We have adapted an implementation from Hanov (2011)).The method utilizes a two-step approach by which in the first step, the row position in an intermediate table is found for a particular key. The next step involves finding the exact position for a key which would also be unique, with the help of the intermediate table. For the construction of the intermediate table, the algorithm starts with a low value of d (d=1) and keeps increasing so that unique slots can be found in each row of the intermediate table.

At first, the keys are placed into buckets according to the first hash function. Then the buckets are processed in descending order such that the ones with the largest number of slots are processed first. After this, it attempts to place all the keys in the bucket which contains an empty slot using the minimum value for d.

Therefore, the hashing of strings to values involves combination Rabin-Karp and minimal perfect hashing as shown in Fig2. These hashed values are used as the indices of the IN/OUT matrix. A more optimal and efficient

way to hashing could be implementing the dynamic hashing, which maps the n strings to 3n-1 values. This way the algorithm accounts for additional insertions that might occur. Hence, providing a better time bound for insertions, though compromising over space to an extent.

3.4 Constructing the Covering Forest

We now focus on detecting false positives in our data structure while maintaining a reasonable memory usage. The strategy described by (Belazzougui *et al.*, 2016b) is to sample a subset of nodes for which we store the plain-text k-tuple and connect all the unsampled nodes to the sampled ones. More precisely, we partition nodes in the undirected graph *G0* underlying *G* into a forest of rooted trees of height at least *k lg* σ and at most *3k lg* σ. For each node we store a pointer to its parent in the tree, which takes *1 + lg* σ bits per node, and we sample the k-mer at the root of such tree. We allow a tree to have height smaller than *k lg* σ when necessary, e.g., if it covers a connected component. Our covering forest implementation is a modified version of what is described in *Fully Dynamic de Bruijn Graphs*, Belazzougui *et al.* (2016b). Our main difference lies in sampling. Belazzougui *et al.* (2016b) samples the roots first when detecting false positives for a membership query in the forest. This method in which the forest is constructed enables them to check for the roots after the search method converges. In this way, there is no need to later check whether the final node reached is present in the sample list of roots. A membership query is defined as a search for a specific k-mer in the represented de Bruijn graph, returning whether or not the queried k-mer is contained.

Our covering forest is initialized by first sampling a k-mer, which could be a random node. In our test cases, we have simply chosen the first k-mer set in the OUT matrix to initialize our first tree in the forest, with this k-mer set to first root node of the forest. Next, we continue adding nodes, traversing depth first, to the currently rooted tree by performing a check on the outgoing edges from the OUT matrix. The column headers which have the bit set to 1 are then stored to a an *outList* and each is appended to the [1:k-1] characters of our current k-mer. Since it is not a root, the hash value for the k-mer is calculated by inserting the k-mer value *v* to the hash function *f(v)*, storing the hash value to the newly created node. The new node's parent pointer is then updated and it is attached to our tree. This process continues until the tree has reached the specified max height of *3k lg* σ. Once the the max height is reached, the next node is added to a root list. More specifically, the nodes following the root node are new roots for the trees in the forest. The root list is our list of nodes to sample. Since this approach is depth first, we return and the process recursively continues for the other values contained in each outList. Once all the values each outList are exhausted, the new root in the root list is sampled. This new root is the root of our newly created tree. The list of roots represents the full cover forest.

Another edge case our forest must consider is duplicate k-mer values. Duplicate k-mer values occur when the string contains a repetition of *k* characters. Graphically, duplicates contain multiple nodes coming in or out. To handle this case, our forest structure contains a bit array *visitedList* of size *N*, where *N* is the number of distinct k-mers in the graph. Because minimal perfect hashing outputs values from [0:n-1], each hash value is represented by the corresponding index in *visitedList*. Once a new Node is created, its hash value is obtained and the bit is set: *visitedList*[*f(v)*] = 1. If we have already visited a node, we do not re-initialize the node.

To completely initialize the forest, the above approach is repeated until all the nodes in our OUT matrix are initialized in the covering forest. All nodes our finally added to a dictionary, where key is the address and value is the hashValue or string. This dictionary will index our Node when checking for false positive values. The created covering forest tells us whether a given k-mer (or node) is present in our de Bruijn graph representation. It gives one details about the parents connected to a particular node, such that if any node has an incoming edge from another node, the previous node is a parent to the latter. Algorithm 2 presents the above in psuedocode.

Algorithm 2 Forest Construction

Input: matrix InOut, dictionary MPH
Output: Forest

```
1: procedure constructForest(InOut, MPH)
2:     MaxHeight ← (3 ∗ lg(sigma)
3:     initialNode = Node(value : k − mer, level : 0, parent :
   None, isRoot : True)
4:     Dict.update(Node)
5:     while rootIndex < len(rootList) do
6:         currentNode = rootList[nextNode]
7:     insertNode(node, kmer_value)
```

```
1: procedure insertNode(node, kmerValue)
2:     if node.currLevel > maxTreeheight then
3:         return node
4:     Marknodeasvisited
5:     outList = getOutColumnHeaders(kmerHash)
6:     while index in outList do
7:         newNode = getOutgoingNodes()
8:         if !currentNode then
9:             loop
10:        insertNode(node, kmerValue)
11:    returnnode
```

3.5 Forest Operations

The forest allows one to test if the dynamic hash functions that we have come up with using the Las Vegas algorithm are giving us legitimate indices for all our nodes in the de-Bruijn graph. In this section, we fully describe the different operations our implementation can perform on the represented de Bruijn graph, namely: *membership query, insertion, and deletion*. Both the In and Out Matrix and Forest will need to be constructed prior to running any operations. We begin by describing the static operation search, which insertion and deletion fully utilize. After doing so, we expand on the insertion and deletion algorithms.

3.5.1 Search

The first and most prominent operation is search. Search represents the membership query function, taking in a potential k-mer contained in the graph. If found, search returns the k-mer. If not, search returns NULL and indicates that the k-mer is not found. Search utilizes the In and Out Matrix to obtain a hash value, while climbing up a tree in the forest to detect false positives. Suppose we want to search for a k-mer *v*. We first calculate its hash value *f(v)*. If no hash value is found, we say the k-mer is not contained within the de Bruijn graph. Next, we search the dictionary *nodeDictionary* using key *f(v)* in order to get the address of the potential node. If the found node is a root or its parent is a root, the node is found. If the node is not a root, we utilize the IN matrix to obtain the column headers of all the edges incoming to the current node - similary to how the outList was initilized in the forest construction. A list of *potentialParentHashValues* is then set by hashing string consisting of the column headers obtained

and the [1:k-1] characters of the current k-mer. If the current node's parent hash value is in not the potentialParentHashValues list, we claim the node is not found. If found in the list, the *search* function is ran once more on the parent string until a root node is reached or the *max tree height* is exceeded. Once the *max tree height* is exceeded, the k-mer does not exist. Thus there exists two exit conditions for k-mers not contained in the graph: if the max height is exceeded or if the true hash value does not equal one of the potential hash values. *Algorithm 4* formally defines search.

Algorithm 3 Search Node

Input: kmervalue
Output: K-mer value FOUND/NULL (NOT FOUND)

procedure searchNode($kmerVal$)
  $hash = getHash(kmerVal)$
  if $!hash$ **then**
    $returnNOTFOUND$
  if $node.isRoot||node.parent.isRoot$ **then**
    $returnFOUND$
  $node = getNodeDict(hash)$
  $inList = getColumnHeaders(kmerVal)$
  $potentialParentHashValues.set()$
  while $index < length(inList)$ **do**
    $potential = getPotentialKmerParent()$
    $updatePotentialParentHashValue(potential)$
  if $trueParentHashnotinpotentialParentHashValues$ **then**
    $returnNOTFOUND$
  if then$count > maxTreeHeight$
    $returnNOTFOUND$
  $returnsearch(parentKmer)$

3.5.2 Handling Dynamic Vertices

Building on the data structures used for representing the static de Bruijn graph, we will now look at an algorithm that would be able to handle the insertion and deletion of vertices in the graph by making suitable changes to both the In and Out Matrix representing our de Bruijn graph.

Whenever we perform any dynamic operations over our original graph or any succint data structure which is a representation of our input reads, the changes need to be reflected in our covering forest as well. If we work with a rigid implementation of our forest, we risk giving out false positive results of our operations on the de-Bruijn graph. Furthermore, there are scenarios where the hash functions will map different nodes to the same indices, and our forest should identify them as different nodes or else if it fails, it will make changes to the nodes that haven't been modified. The dynamic operations that can be performed include insertion and deletion. Our case only handles dynamic operations on the vertices and not the edges. However, we need to update edges when deleting, else we risk an incorrect graph.

3.5.3 Insertion

Insertion of a node in the graph is trivial to handle, since the newly added node would not have any edges associated with it and hence would be an isolated node. As a result the only task would be to recompute the hash function such that we still have distinct hashed values for each k-tuple in the graph. The re-computation would require O(k) amount of amortized time, since k is the length of each vertex of the graph. To maintain a correct implementation of the graph, the In and Out Matrix must be updated by adding a new row. The column headers are then initialized to zero, because it is a root node. Because no edges need to be connected to the new node, after initializing this new node into our forest, it is declared a root node and therefore must be added to the *rootList*.Whenever one makes an insertion in such a way, the algorithm checks to ensure that no same node is inserted twice in any of our data structures. When commencing, we need to first hash the input k-mer and check if the hashed value is already present in our forest. To ensure the hash gives the correct value, the *search* query operation is called, taking care of the false positives as well. Once it returns that it is not already present, the In and Out Matrix and forest are updated If successfully inserted, *Insert* return a boolean value of true, otherwise it will return false. The following algorithm formally defines insert.

Algorithm 4 Insert Node

Input: kmervalue
Output: K-mer value INSERTED/NULL (DUPLICATE)

procedure insertNode($kmerVal$)
  $alreadyPresent = searchNode(kmer)$
  if $alreadyPresent$ **then**
    $returnDUPLICATE$
  $newNode = InitializeNewNode()$
  $hash = newNode.getHash()$
  $rootList.append(newNode)$
  $returnFOUND$

3.5.4 Deletion

The other dynamic operation is the deletion of k-mers. Because the matrices represent the graph, the correct entries from these matrices need to be deleted. Once the entries are deleted from the matrices, the node in question must also be removed from our covering forest. The delete operation in that sense is similar to the insert operation. Because a node will be deleted from the forest, the nodes connected to the deleted node will need to updated as well. The connected nodes are therefore changed to root nodes and their matrices are updated as well. We begin by first making sure the the requested node is already present in our graph, before we proceed further in our deletion strategy. This requires us to first hash the input k-mer and check if the mapped node is valid and present in our initialization. To make sure that the deleted node is correctly identified, its hash needs to first be computed. Once that is finished, the *search* function is used to find its position in the forest and check for the false positives. If it passes all these steps, the k-mer has a high probability of being present in the graph and can be deleted. We begin by getting the appropriate column headers or both the In and Out matrices. However, to correctly get these column headers for the required nodes, these nodes to be identify these nodes from our forest as well. This is to identify all the references to the deleted node and make sure those references are removed as well from the true representation of the de Bruijn graph. The In and Out matrices can give us information about all the nodes that are coming in to this deleted node as well as the ones that are leaving from this node. First *delete* gets all the nodes whose outgoing edges need to be updated after deletion as they won't be pointing towards the deleted node anymore. Then *delete* performs an update on those identified column headers in the Out matrix. Similarly, we do the same for the nodes that have an incoming edge from this now deleted node. Then identify these column headers identified and update the node in In Matrix for the correct node.

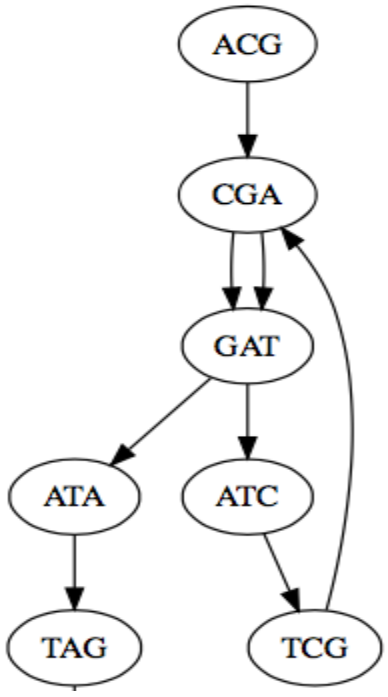


Figure 3: Example de Bruijn graph generated for ACGATCGATAG

Algorithm 5 Dynamic vertices in de Bruijn Graph

Input: graph G, represented through IN and OUT matrix
Output: Updated IN and OUT matrix

procedure DynamicVerterx(IN, OUT)
  $STATE \leftarrow INSERT \mid DELETE$
  if $STATE$ is INSERT **then**
    $ReComputeHashFunction()$
  else if $STATE$ is DELETE **then**
    while $EdgeCheck(DyVertex)$ **do**
      $edge = EdgeCheck(DyVertex)$
      $RemoveEdge(DyVertex, edge)$
    $ReComputeHashFunction()$
  return $INandOUT$

procedure $EdgeCheck(DyVertex)$
  if $edge$ exists for $DyVertex$ **then**
    return $edge$
  else return null

procedure $RemoveEdge(DynamicVertex, edge)$
  if $edge$ is $Incoming$ **then**
    $IN[f(DynamicVertex)][edge] = 0$
  else $OUT[f(DynamicVertex)][edge] = 0$

procedure $ReComputeHashFunction()$
  The paper (Czech *et al.*, 1992) suggests an optimal algorithm for generating minimal hash function in time proportional to O(k) and in space proportional to O(klogk) where k is the size of the input string.

To illustrate the above, we look at the following example in de Bruijn image (Fig 3). Suppose k-mer 'GAT' has an incoming edge from another k-mer 'CGA' and has outgoing edges towards k-mers 'ATC' and 'ATA' in our true representation of the de Bruijn graph. If we delete k-mer 'GAT', correctly identify the affected nodes by identifying the appropriate column headers with our delete query k-mer 'GAT'. The nodes that need to be updated In the Out matrix include setting the column header 'T' to 0 or null for the hash value of node 'CGA'. Similarly, the nodes that need to be updated in the In matrix include updating column header 'G' for hashes of the nodes 'ATC' and 'ATA'.

After updating these matrices, the deletion is reflected in the forest as well. First identify the node that has to be deleted and then check all its outgoing node and compute their hash values. Next, check if any of the nodes of these outgoing hashes to the parent set as the node that needs to be deleted. This confirms that the correct node is deleted and creates a legit new root, which follows this deleted node. This node needs to be converted by making it a root and initializing a new tree for it (adding it to the forest). Lastly, add this to the *rootList* which represents the forest. Verify that the node has been deleted performing a membership query of the query k-mer and it should return a value of false, denoting that the node has been permanently deleted from the data structures.

4 Results & Evaluation

The above mentioned approaches and data structures were all implemented in python 2.7. The evaluation is performed on a sever with Intel Iris Graphics 6100 1536 MB @ 2.7 GHz Intel Core i5 (4 Cores) with 8 GB RAM.

4.1 Structure Evaluation

When we work on a problem which deals with storing and processing strings, especially on such a large scale, performance is a major concern. This entire project is evaluated on *memory usage* and *time complexity* metrics. The evaluation is focused on the following:

- The project is an implementation of the dynamic de Bruijn graph as described in (Belazzougui *et al.*, 2016b) and the structure needs a creation of a graph that reflects all the connections between the different reads of the genome. Representation of this tree takes an enormous amount of memory and it is important to reclaim memory of the vertices that are dynamically created. The implementation would ensure that memory is reclaimed with every dynamic removal of the nodes.
- The implementation requires the construction and management of two binary matrices which stores information about the relationship of the overlapping reads. As previously stated, each matrix symbolizes the incoming and outgoing edges of the vertices and stores this information separately. The binary matrices IN/OUT are defined in a manner that reflects the information of the de Bruijn graph and allows reconstruction of the de Bruijn graph from the matrices and vice-versa.

4.2 Dataset

The dataset that is used in the implementation of this paper is summarized in Table 1, which provides brief information about the genome that we plan to test it on and the data format that is expected. The file that we have tested our data structure on contains Illumina Sequence reads of a Escherichia coli strain generated by Illumina Cambridge Limited. Illumina Cambridge used the same file for their study on paired-end sequencing of the genome of Escherichia coli K-12

strain MG1655, using the Illumina Genome Analyzer II (dat, 2013).

A FASTQ file is of the following format:

- Line 1 contains sequence identifier and an optional description preceded by '@' character.
- Line 2 is the raw sequence letters denoting the genome sequence.
- Line 3 contains the '+' character followed by same sequence identifier and optional description.
- Line 4 has the quality values for sequence contained in Line 2 and is of the same length.

The focus is on extraction of Line 2 as this is the raw sequence reads needed to generate the de Bruijn graph. This is required to create the de Bruijn graph and the in and out matrices. Because of our testing machine (which contains only 8 GB of ram) and for testing purposes, smaller size instances of the original FASTQ file was used (5 MB, 10 MB, 20 MB approx.)

4.3 Memory

The storage is evaluated in terms how much memory is used for initialization of the covering forest and the space that is occupied by the In and Out Matrices. If the true representation of the graph is presented in terms of normal de-bruijn graphs, we have to store each of the kmers which are in itself string representations of each node, and so it takes and a large amount of memory space, just to construct the de bruijn graphs. Over that an additional space is required to perform all the dynamic operations such as insert, delete and search operations. So representing a structure in such a way will require a storage of kn number of characters, where each character takes one byte of space. Even after accounting for the framework dependencies, this structure is definitely not an efficient approach specially when fastq files of the input data increases. The approach that has been taken in the project is an attempt to store the same information in a much more compact and storage efficient manner. The solution doesn't store the de-bruijn graph in the memory, instead it is represented in terms of the In and Out matrices, which are in turn a list of bit array vectors, and each bit array vector only holds value for five characters denoting the nucleotides,that are, 'A', 'C'. 'G', 'T', 'N'.

Given a static list of n k-mers with an alphabet size of σ, we can store f in O(n + log k + log σ) bits, where f is the hash function which gives out the minimal perfect hashing functions. The complete storage complexity upper bound for f is O(n(log log n + log log σ)) bits with high probability. The covering forest that is constructed also needs to be initialized memory in the heap. But instead of representing the k-mers with strings, we are storing them in terms of their hash indexes which are just short integers. So the memory representation would be very compact in terms of the storing these k-mers compared to normal de-bruijn graphs. The covering forest for initialize a memory in heap for every single node or k-mer from our input data, and these nodes would only hold memory for the hash indices.

But for the roots of the forest it also stores the k-mer string, instead of the hashes since the k-mer string is required for search of the k-mers and filtering out all the false positives that can be introduced by our hashing functions. And so the entire data structure can be represented by succinctly constructing these bit matrices and building a map of these computed hashes, and linking them to each other by adding parent pointers to each of the node, except for the roots, which don't have parents.

4.4 Time

The time is evaluated for all three operations (search, insert, and delete) along with graph construction. Because of the large size of the file, graph construction takes the most time, that is *O(n)*, where n is the total number k-mers. If we increase the size of k, we will also decrease the number of k-mers.

The table below (Table 1) shows the file sizes and graph construction times for each of the files of the files. As the results show a 5 MB, 10 MB, and 20 MB file took 22.37, 56.78, and 126.99 seconds respectively. Due to the nature of our bit matrix we tested for small k values (4,10, and 15 in this scenario).

Table 1

File size	Chosen K	Number of nodes generated	Initialization time
5MB	4	125	22.377s
10MB	4	125	56.784s
20MB	4	125	126.990s
5MB	10	256528	482.32s
20MB	15	6836157	3389.14s

The table below (Table 2) shows the file sizes and search time for each of the files of the files. As the results show a 5 MB, 10 MB, and 20 MB file took 22.37, 56.78, and 126.99 seconds respectively.

Table 2

File size	Chosen K	Number of nodes generated	Search time
5MB	4	125	9.272s
10MB	4	125	9.656s
20MB	4	125	9.990s
5MB	10	256528	963.12s
20MB	15	6836157	2041.17s

s

The table below (Table 3) shows the file sizes and graph insertion times for each of the files. As the results show a 5 MB, 10 MB, and 20 MB file took 17.21s, 22.34s, and 34.11 seconds respectively.

Table 3

File size	Chosen K	Number of nodes generated	Insertion time
5MB	4	125	10.341s
10MB	4	125	10.102s
20MB	4	125	10.112s
5MB	10	256528	964.16s
20MB	15	6836157	2045.01s

The table below (Table 4) shows the file sizes and graph delete times for each of the files. As the results show a 5 MB, 10 MB, and 20 MB file took 18.41s, 24.17s, and 46.81 seconds respectively.

Table 4

File size	Chosen K	Number of nodes generated	Deletion time
5MB	4	125	10.134s
10MB	4	125	10.512
20MB	4	125	10.313s
5MB	10	256528	964.65s
20MB	15	6836157	2045.05

5 Discussion

As shown in the Approach section, we have successfully implemented the fully dynamic de Bruijn graph proposed by (Belazzougui *et al.*,

2016b). Modifications were needed because we simply take care of the vertices and not the dynamic edges. Because one needs to first check for false positives, the insertion and deletetion algorithms are slower than if only an In and Out Matrix was utilized. However, if that were the case, many false positives would be present, and the algorithm would not efficiently report back what is contained in the de Bruijn graph. As previously mentioned, though Bloom Filters handle insert cases very efficiently they do not handle delete cases. Our implementation may be slower than using Bloom Filters, but is more flexible.

Our current implementation is limited to the size of k that we set as the k-mer size. In the dataset, when K was chosen to be 4, the number of possible permutations of the string was small i.e 125, thus we observe a constant of 125 nodes irrespective of the size of the file. The number of nodes generated in the graph are both dependent on the size of the file and the k value chosen. In this case, the small value of K limits the number of nodes.

We now discuss the results found in the above section. Table 1 shows that for a k-mer of size 4 and 125 nodes, our algorithm is extremely fast for the construction time. By doubling the file size we can see that the construction time itself has also doubled. However, the number of nodes remains the same because the number of distinct nodes is the same for all three size files caused by the small number assigned to k. When trying different size k-mers, we were limited by both the runtime and construction of the In and Out Matrix. When changing the K value 10 and 15 our search method took a extraordinary amount of time. This may have been due to number of nodes in the forest

Table 2 gives the results for search. Table 2's run time may be similar to Table 3 and 4 due to the fact that insert and delete utilize the search function.

Table 3 gives the results for insertion. Because the number of nodes is the same for each of the results, the insertion size is similar in all cases. If not looking for false positives, the insert function completes almost instantaneously. However, our implementation focuses on an accurate representation and thus we need to search the graph. Similarly, Table 4 also gives similar results for all cases because of the small k-mer size. Because the forest is traversed for false positives, when changing the k value to 10 and 15, the insertion, delete, and search time take an immense amount of time. Thus, for large size k values we do not suggest utilizing our implementation.

6 Future Work

The implementation we have proposed and demonstrated in this paper only dynamically changes the vertices in a graph. (Belazzougui *et al.*, 2016b)'s proposal is both for the adding and removal of vertices and edges. Therefore, in the future we can adapt our implementation to add and delete edges in the graph. This will give better time bounds for looking for false positives, because an inserted vertexs is will be connected to other vertices. Furthermore, it will give a better representation of the de Bruijn graph. In the current implementation, because we simply make the new nodes a root, we are not accurately representing the de Bruijn graph.

Another limitation which exists in our implementation is the way we currently are managing the dynamic hash. Our dynamic hash adds the new nodes to a new hash function, with the starting position one greater than the initialized forest size. We then update the forest size and continue with further insertions. One way which can improve this is to truly dynamically hash the vertices. Though we are able to find the inserted nodes with our current implementation, we do not correctly perform a dynamic hash which can result in errors.

One final limitation is the accuracy of our search function. Since the false positive algorithm relies on previously hashed nodes, if our first hash is false and the k-mer size is small enough, the algorithm will report a true vertex where one does not exist. Though this is a limitation, this method was proposed by (Belazzougui *et al.*, 2016b) and therefore we correctly implemented their method. To accommodate for this, a list of all the parent indices can be stored directly from the de Bruijn graph. However, this will vastly increase our memory size and not represent an efficient de Bruijn using the forest and In and Out Matrix.

Our current implementation is coded in Python, and therefore may not give the most efficient runtime. In the future, we plan to implement our program in C++ to give better results. This way we can also run on large files so that we may more accurately reconstruct a genome. Furthermore, we may use existing C++ hashing algorithms to accommodate for more K value sizes.

7 Conclusion

The implementation proposed in this paper can be successfully utilized for making read corrections when generating a genome. We can do so by inserting and deleting a k-mer in the In and Out Matrix and updating the forest. However, our fully dynamic de Bruijn graph with respect to vertices is solely efficient for short reads. To view our implementation, visit our Github page on https://github.com/thegreedychoice/FullyDynamicDeBruijn.

8 Acknowledgements

The authors of this paper would like to give a special thanks to Dr. Christina Boucher. We would also like to give a special thanks to Dr. Travis Gagie for assisting us in understanding the constructing of the forest and searching for false positive values.

References

(2013). Ecoli mc.

Belazzougui, D. *et al.* (2016a). Bidirectional variable-order de bruijn graphs. In *Latin American Symposium on Theoretical Informatics*, pages 164–178. Springer.

Belazzougui, D. *et al.* (2016b). Fully dynamic de bruijn graphs. In *International Symposium on String Processing and Information Retrieval*, pages 145–152. Springer.

Belk, K. *et al.* (2016). Succinct colored de bruijn graphs. *bioRxiv*.

Boucher, C. *et al.* (2015). Variable-order de bruijn graphs. In *Data Compression Conference (DCC), 2015*, pages 383–392. IEEE.

Bowe, A. *et al.* (2012). Succinct de bruijn graphs. In *International Workshop on Algorithms in Bioinformatics*, pages 225–235. Springer.

Cazaux, B. *et al.* (2016). Linking indexing data structures to de bruijn graphs: Construction and update. *Journal of Computer and System Sciences*.

Chaisson, M. J. and Pevzner, P. A. (2008). Short read fragment assembly of bacterial genomes. *Genome research*, **18**(2), 324–330.

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.

Czech, Z. J. *et al.* (1992). An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, **43**(5), 257–264.

Hanov, S. (2011). Throw away the keys: Easy, minimal perfect hashing.

Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, **31**(2), 249–260.

Kirsch, A. and Mitzenmacher, M. (2006). Less hashing, same performance: building a better bloom filter. In *ESA*, volume 6, pages 456–467. Springer.

Mehlhorn, K. (1982). On the program size of perfect and universal hash functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 170–175. IEEE.

Pell, J. *et al.* (2012). Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, **109**(33), 13272–13277.

Salikhov, K. *et al.* (2013). Using cascading bloom filters to improve the memory usage for de brujin graphs. In *International Workshop on Algorithms in Bioinformatics*, pages 364–376. Springer.

Simpson, J. T. *et al.* (2009). Abyss: a parallel assembler for short read sequence data. *Genome research*, **19**(6), 1117–1123.

Wheeler, D. L. *et al.* (2000). Database resources of the National Center for Biotechnology Information. *Nucleic Acids Res*, **28**(1), 10–14.

Zerbino, D. R. and Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, **18**(5), 821–829.