Bioinformatics
doi.10.1093/bioinformatics/xxxxxx

Deriving DNA Sequence using fully dynamic de Bruijn Graphs

Fully Dynamic de Bruijn Graphs: Adding Dynamic Edges

Akshat Bhardwaj [1], Amar Nath Vaid [1], Sahil Tiwari [1] and Naga Satya Karthik Narikimilli [1],

[1] Computer and Information Science Engineering, University of Florida, Gainesville, FL-32608

Abstract

A genome is an entire DNA contained in a single cell of an organism. In bioinformatics, genome assembly is the process of fragmenting the chromosomes, sequencing them, and then reconstructing the original sequence by putting back the fragments or reads together. This process is crucial because there is no technology developed enough to read the sequence in a single go. We present an efficient approach to implement de Bruijn graphs which are directed graphs responsible for representing a DNA sequence as nodes of (k-1)mers. Each k-mer is a read of length 'k-1'. We implement a de Bruijn graph which supports dynamic insertion and deletion of edges by taking motivation from and following the paper "Deriving DNA sequence using fully dynamic de Bruijn Graphs" by Belazzougui et al.. However, we are just focusing on edges and not concerned about overlapping nodes in our implementation of the data structure as described in the paper. We use a combination of minimal perfect hashing and Rabin-Karp hashing to construct our data structure. Later we implement and test this data structure on an input dataset of an Illumina Sequencing of E-Coli. The evaluation is done based on the average time taken and peak memory usage for various operations such as addition, deletion and search for an edge and thereafter, the results have been recorded.

1 Introduction

de Bruijn Graphs are basically used for representing a sequence as its k-mers and also reconstructing a string from a set of its k-mers. This implies that it is used for genome assembly sequencing (Conway and Bromage, 2011. This makes de Bruijn graphs utmost important in the field of Bioinformatics and DNA sequencing. A string represented as a k-th order de Bruijn graph is a directed graph with nodes equal to the unique (k-1)tuples in the string. These (k-1) tuples are constructed with the first (k-1) nucleotides and moving forward with the next (k-1) overlapping nucleotides excluding the first nucleotide in the present tuple. The edge from u to v is represented as a k tuple somewhere in the string where the k-1 length prefix is u and the k-1 length suffix is v. Later, in the last step, the tuples or nodes that are identical are connected together and a directed graph is formed. This is how a de Bruijn graph deals with overlapping (k-1)mers. The directions of the arrows are such that the arrows originate from the (k-1)mer with the first (k-1) overlapping nucleotides and ending towards the (k-1)mer with the next k-1 overlapping nucleotides [1].

These de Bruijn graphs have a wide spectrum of uses in the field of Bioinformatics, the most important being *de novo* assembly[15], read correction and pan-genomics[13]. The DNA sequences that are the input datasets of such graphs are huge, taking up a lot of space varying from a few megabytes to hundreds of gigabytes and hence powerful machines must be used for the implementation of such graphs in order to represent the sequences as k-mers. However, for graphs to handle such massive datasets, pointer-based implementations are impractical. There could be several approaches for construction of a de Bruijn Graph compactly as proposed by many researchers over the years. The most common ones are Bloom filters and Burrows-Wheeler Transform. We aim at implementing the representation of a de Bruijn graph using minimal perfect hash functions[6] in this paper. The approach is identical to the one used in Bloom Filter representation of de Bruijn Graphs but has significantly better bounds theoretically when the graph size is small and the graph is fully dynamic. By fully dynamic, it is implied that we can perform both insertions and deletions of nodes efficiently. Implementations using bloom fillers are only semi-dynamic, that just support insertions and not deletions. For this project, we have used a data structure based on the combination of Rabin-Karp Hashing[8] and Minimal Perfect Hashing in order to minimize the amount of memory used by the k-mers in the sequence.The advantage of using de Bruijn graphs is to efficiently store the sequences as k-mers along with handling the problem of overlaps in order to carry out DNA Sequence Assembly. de Bruijn graphs were existent for long until they gained light with the advent of new advancement in Bioinformatics that proposed Next Generation Sequencing

(NGS). The NGS technology could readily and rapidly decode DNA sequences into a large number of small fragments, the count usually going up to millions and that too inexpensively. Although the number of fragments was large, the read length was shorter than the previously used technology which is known as Sanger reads. The technique used previously for DNA Sequencing of Sanger reads was Overlap-Layout-Consensus (OLC). However, for NGS library, the OLC method was not enough as it got harder to assemble such a massive number of fragments with repetitive regions. OLC may even yield erroneous results. Hence, OLC completely broke down with those short reads as the extensive amount of overlaps scale quadratically with the massive number of reads. This is how the use of de Bruijn graphs came into existence because of its ability to benefit from both, a large number of fragments and shorter read lengths. DNA Sequence Assembly involves the merging of short fragments of DNA called reads into a longer reconstructed DNA Sequence. The assembly is necessary as reading the whole genome by modern DNA Sequencing technology is not possible. Hence, the sequence must be broken down, read and then sequenced together again forming a reconstructed sequence. This genome assembly still utilizes a large number of resources and is a cumbersome process because of its high computational complexity and memory requirements. de Bruijn Graphs are basically used to improve this memory situation by handling the assembly of reads having repetitive regions. This will highly improve the space efficiency.

1.1 Related Work

There has been a lot of research on different methods for implementing de Bruijn graphs with efficient memory usage and also avoid any false positives. We started looking at previously done research on de Bruijn graphs and the first paper we came across was by Compeau *et al.*(2011). It covered the details on how a de Bruijn graph could be applied to genome assembly[5]. It was crucial for us to understand the advantages of using de Bruijn graphs in the first place and the reason for them replacing the Overlap-Layout-Consensus (OLC) method. Zhenyu Li *et al.*(2011) wrote a paper bringing out the comparison between the assembly algorithms using OLC and de Bruijn graphs. It gave us a better understanding of the reason for the creation of de Bruijn graphs after knowing the differences in the computational complexities of both classes of algorithms and the effect of repeats in the sequence on each algorithmic class[9]. However, we wanted an efficient use of de Bruijn graphs to minimize memory utilization. J.T. Simpson *et al.* (2012) was the first one to reduce the space utilization of de Bruijn graphs using distributed hash table[12]. This method required 336 GB to store the graph. Using a similar hash table with the addition of a sparse bit vector to represent the edges, Conway and Bromage were able to bring down the space requirement to 32GB[6]. He mentioned his finding in the paper "ABySS: A parallel assembler for short read sequence data". In 2012, Alexander Bowe, Taku Onodera, Kunihiko Sadakne and Tetsuo Shibuya collaboratively wrote a paper named "Succinct de Bruijn Graphs"[3]. In their paper, using the Burrows-Wheeler Transform, they were able to present a de Bruijn graph representation that took 2.5GB of space for storing a short read of the human genome. In 2013, in the paper "Space-efficient and exact de Bruijn graph representation based on a Bloom filter"[4], Chikhi and Rizk implemented an encoding of de Bruijn graph using Bloom Filter and used an auxiliary structure to handle the false positives. They could achieve a *de novo* assembly of short reads for a human genome consuming 5.7 GB of memory and it took 23 hours to complete the whole process. In the paper "Efficient de Bruijn Graph Construction For Genome Assembly Using a Hash Table and Auxiliary Vector Data Structures"[10] (2014) Mahfuzer Rahman Limon, Ratul Sharker, Sajib Biswas and M. Sohel Rahman used a similar approach to implement de Bruijn graph with the help of hash tables which improved the runtime and memory complexities. They worked to improve on the time complexity taken for the formation of the de Bruijn graph from genome sequence considering all the nodes and edges. Fabiano C. Botelho and Nivio Ziviani proposed an external perfect hashing scheme for very large datasets. They used an external memory based algorithm to construct perfect and minimal perfect hash functions which they used for hashing the large key sets[2]. Their method to deal with large datasets has been one of our motivations to take up minimal perfect hash functions to implement de Bruijn graphs.

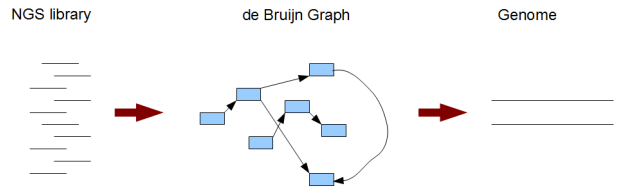


Figure 1. Formation of De Bruijn graphs from short reads and formation of the full genome sequence from the graph

2 Methods

In this section we discuss in detail about the Process Flow in 2.1 of our project starting with the input as a FASTQ file of Illumina sequencing of E.Coli MG1655 till the output that we obtain using a de Bruijn Graph with supported dynamic insertions and deletions. We discuss the hash functions used that encompasses minimal perfect hash function and Rabin-Karp hash function, internal data structures and the operations on the de Bruijn graph data structure that serves purpose of addition and deletion of dynamic edges. About hash functions, we explain in detail the modified Rabin-Karp Hash function that we use and also the requirement of the Minimal Perfect Hash function.[6] We will discuss about the necessity of these hash functions and talk about the assumptions that we made for moving forward with the implementation of Dynamic Edges. In the next section,2.2 we talk about using these hash functions for the construction of the static de Bruijn graph data structure. Once the construction of the data structures used to represent the static de Bruijn graph is explained, we talk about an additional data structure that is required for constructing the dynamic de Bruijn graph. Next, we see the method for constructing the underlying undirected graph from the de Bruijn graph. We call this underlying undirected graph $buildForest$. During this process, we see how we store and update the parent pointers. Once we construct all the underlying data structures using these operations, we talk about the addition, deletion and search operations on the data structure. In order to check the validity and the performance metrics of our algorithm, we talk about how we check for the invariants and the various test cases that are written in order to perform the functionality testing. We compare the analysis of our algorithmic metrics for various runs in Section 3.

2.1 Process Flow

Inputs to our algorithms can be either FASTA or FASTQ files. We preprocess these files into sequence reads and eliminate the rest of the lines that are not required. In FASTQ, for example, we have the Sequence Identifier, + character and the quality scores in first, third and fourth line respectively for each sequence. These 3 lines which are not currently in our interest are skipped and only the second line i.e the original sequence is taken into account. This whole process is known as pre-processing. We take each sequence and construct (k-1)mers from the sequence and

calculate the modified Rabin-Karp Hash. Next, we pass all the hash values to the Minimal Perfect Hash function which generates an integer from 0 to (N-1) where N begin the number of (k-1)mers in the whole file. We also measure the time taken for constructing the (k-1)mers , Rabin-Karp Hash & Minimal Perfect hash function for the whole file in the section 3. Next we construct the underlying data structures that are required for de Bruijn graph which will be discussed in section 2.3. Once we build the underlying data structures that are required, we can perform the addition, deletion and search operations using the underlying data structure. These methodologies are explained in the Section 2.4 while performing each operation we check for the invariant condition[6] before and after the operation for proper functionality in our code.

For the first read, we construct the de Bruijn graph using the process followed in the construction of static de Bruijn graph [6]. For all the reads following the first one, we generate all the $k-mer$s and add the vertex in the DB_g if not present and add the edge to the already constructed DB_g. Our main focus for this paper is to be able to add an edge into a constructed DB_g. For this purpose we add the vertex in the DB_g directly with minimal conditional checks. After processing all the steps, we will be left out with an enormous DB_g. Construction of the DNA from this graph and validation of the functionality are the future proceedings of this work. Assumptions that we took for implementing this project are that we know before hand about the space of the input k-mers (distinct input k-mers for constructing the input matrix). Optimized K value (size of $Kmer$ is already given. σ (size of the Σ space. In this specific use case, size of Σ will be 4 (ATGC)). The pseudo-code for the In Out Flow is as follows.

Algorithm 1 ConstructDeBruijn Graph

procedure dbGraphConstruction(sequenceReads, K) ▷ Takes Input as the list of Sequence read
  $K \leftarrow SizeofKMers$ from the input sequence.
3: For each sequence of size N split the sequence into array of size $(n-k)$ each element with length k.
  Initialize the Graph Db_g with matrix of size N X σ (σ being the size of the Σ).
  for do$read \leftarrow sequenceReads$
6: $kmers \leftarrow constructKMers(read)$
  for do$Kmer_1, Kmer_2 \leftarrow kmers$
    Add the $Kmer_1$ and $Kmer_2$ vertices if not present.
9: Add the Edge into the Db_g using AddDynamicEdge

2.2 Hash Functions

The aim of our project is to support addition and deletion of edges. We take an assumption that we already know all the nodes that we process. Since we know all the nodes in our dataset before even constructing the dynamic de Bruijn graph, we can modify our Rabin-Karp to eliminate the redundancies in our hash values. Since the hash values that are generated by the Rabin-Karp are in a bigger range we couldn't construct matrices with Rabin-Karp hash values alone. For this purpose, we are using the Minimal Perfect Hash functions which maps all the unique Rabin-Karp hash values to an integer value from 0 to (N-1) where N is the number of (k-1)mers.

2.2.1 Modified Rabin-Karp Hash

In order to generate unique values for our k-mers, we have used Rabin-Karp hashing like algorithm. This algorithm keeps sliding the pattern one by one and compares it with the substring of the read using a hashing technique which results in an integer value. The hash of the pattern is matched with the substring of the sequence read of the same length as the pattern where we get next successive substring in $O(1)$ time. Consider an M-character sequence as an M-digit number in base b, where b in our implementation is the number of letters in the alphabet. The text subsequence t[i .. i+M-1] is mapped to the number x(i) = t[i]*b^{M-1}+ t[i+1]*b^{M-2}+...+ t[i+M-1]. Furthermore, given x(i) we can compute x(i+1) for the next subsequence t[i+1 .. i+M] in constant time, as follows: x(i+1) = t[i+1]*b^{M-1} + t[i+2]*b^{M-2} +...+ t[i+M]. So x(i+1) = x(i)*b shifts left one digit and (-t[i]*b^{M}) subtracts leftmost digit from the sum and add the rightmost digit (t[i+M]) to the old value. In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character. The Rabin-Karp Hash Function is mathematically defined as f(x_1, x_2, ...,x_k) = (x_1 + x_2^m+ $x_3^{m^2}$ +$x_k^{m^{k-1}}$) mod P.

Algorithm 2 modified Rabin Karp Hash algorithm

procedure modifiedRKHash(kmer)
  $hash \leftarrow 0$
  for do$i \leftarrow 0 : kmer.length$ ▷ for each character
    $hash \leftarrow res * 4 + characterToI(kmer[i])$ ▷ Update the value of res to res*4 shifting left side, and add the integer value of the character at the position at i.
  return $hash$

The pseudo code for the Rabin-Karp Algorithm 2.

In our case, we modified the algorithm slightly. To avoid generating large hash values, we didn't divide the sequence with a large prime number, P. If a sufficiently large prime number is used for the hash function, the hashed values of two different patterns will usually be distinct. If this is the case, searching takes O(N) time, where N is the number of characters in the larger body of text. Taking the modulus with P in our function causes collision of hash values in the mapping of nodes as row indices from (0 to N-1). Hence, in our modified Rabin-Karp algorithm2 we remove the mod P and retain the previously computed value. This may serve as a limitation based on our assumption about unique hash values that modified Rabin Karp hash couldn't work on all values of k. Each value generated from the Rabin Karp is mapped as key to the Minimal Perfect hash function which returns values as the index in the range of (0 to N-1). In this way, values generated from the Minimal Perfect Hash function is used to map the rows in the matrix. A detailed discussion about the matrix has been done in 2.3. Therefore, it becomes imperative to keep the values in a fixed range from 0 to N-1, where N is the number of unique (k-1)mers in the dataset. After generating the large hash values from our modified version of Rabin-Karp algorithm,we use the minimal perfect hash library to generate hash values in the range (0 to N-1). We used an open source library, BBHash for doing the same.

Algorithm 3 construct Rabin Karp Hash Values for a sequence

procedure getRabinKarpValues(sequence)
  kmers $\leftarrow getKmers(sequence)$ ▷ getKmers takes the sequence as input and gets all the splits of k-1 mers
  first $\leftarrow modifiedRKHash(kmers[0:k-1])$ ▷ modifiedRKHash is RabinKarp Hash with no mod P in it
  $res \leftarrow first$ ▷ Store the computed hash in the result
  for do$i \leftarrow [k-1 : kmers.size()]$ ▷ for each kmer value in starting from position k-1.
    $hash \leftarrow computeNextHash(first, i)$ ▷ computeNextHash computes the has value from the previous value as we discussed in the previous section.
  $res \leftarrow hash$ ▷ Store the hash values in res.
  return res

K-1Mer	RabinKarp*	Minimal Perfect H
ATGGAAGT	6665	6
TGGAAGTC	26663	1
GGAAGTCG	41118	8
GAAGTCGA	33400	0
AAGTCGAT	2529	2
AGTCGATG	10118	7
GTCGATGG	40474	3
TCGATGGA	30824	9
CGATGGAA	57760	5
GATGGAAG	34434	4

Figure 2. Generated hash values from Rabin-Karp Hash function and Minimal Perfect Hash function for the sequence: ATGGAAGTCGATGGAAG

Lemma 1. *Given a static set N of n k-tuples over an alphabet Σ of size σ, with high probability in $O(kn)$ expected time we can build a function $f : \Sigma^k \to \{0, \ldots, n-1\}$ with the following properties:*

- *when its domain is restricted to N, f is bijective;*
- *we can store f in $O(n + \log k + \log \sigma)$ bits;*
- *given a k-tuple 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.*

Figure 3. Lemma: Stating about the minimal Perfect Hash functions

Lemma 2. *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).*

Figure 4. Lemma2 : Dynamic Hash functions

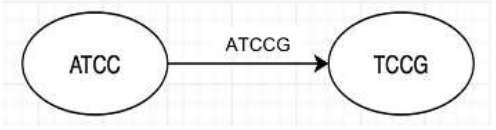
In the above Fig 2 we have shown generated hash values using the modified Rabin-Karp hash function and minimal perfect hash function for a sequence $ATGGAAGTCGATGGAAG$.

2.2.2 Minimal Perfect Hash Function

A perfect Hash function is a function that maps input values to a range of values with no collision in the keys. Minimal Perfect Hash function is a Perfect Hash function with no space / slots in between the keys. Hence, Space that is required by the minimal perfect hash functions is less when compared with the perfect hash function alone. We use minimal perfect hash function for that purpose in our project in order to reduce the space complexity of mapping the modified Rabin Karp hash values to an integer range from (0 to N-1). Minimal Perfect Hash functions follows the following Lemma in fig 3[8].

In our project we are using BBHash[11] which is an implementation of minimal Perfect Hash function. It is available open Source under the MIT License.

As per the author if the N is dynamic in nature we are supposed to use the function *f* where function *f* is similar to 3 but invalidates the following constraints. This is presented in the following Lemma in fig 4[6].



IN	A	T	C	G
ATCC(0)	0	0	0	0
TCCG(1)	1	0	0	0

OUT	A	T	C	G
ATCC(0)	0	0	0	1
TCCG(1)	0	0	0	0

Figure 5. fig:The IN and OUT matrices for the outgoing and incoming edges from V_x to $V_y 2$

2.3 Description of Data Structure

For storing the de Bruijn Graph we are using matrix structure to store the Incoming and outgoing edges between k-1 mers. For the dynamic de Bruijn graph nature we store the parent pointer information in a separate vector with all the parent modified Rabin Karp Hash value stored. We can optimize it to store the character instead of the hash value. Time taken in the case when storing the character instead of hash value is $o(1)$ as we can compute the previous and next hash values easily given the current hash value and the value of k.

First we see about the Matrix decomposition for storing the de Bruijn graph and then the Storing the forest. Next we see about the construction of forest. We talk about operations on the data structure in the section 2.4.

2.3.1 Matrix decomposition

The structure of the de Bruijn Graph G is stored in two binary matrices, IN and OUT where size of each matrix is n * σ . At this point we have Minimal Perfect Hash values of all the k-1 mers in the sequence. All the rows are the represented as k-1 mers and the columns of the matrix is equal to the σ of the characters in the sequence. In our case it will be 4 characters A, T, G, C. IN matrix is used to represent the incoming edges to a k-1 mer and OUT matrix is used to represent outgoing edges to a k-1 mer. Population of the matrix is done by taking 2 k-1 mers which are having k-2 characters in common. The match in the k-2 characters should be a substring match and only last k-2 characters of a k-1mer should match with first k-2 characters in k-1 mer. If such 2 nodes are found then we add an edge in between such 2 k-1 mers by updating the matrices as follows. For each k-tuple v_x = $c_1\ c_2\ \ldots\ c_{k-1}$ *a, the former stores a row of length σ such that, if there exists another k-tuple v_y = b*$c_1\ c_2\ \ldots\ c_{k-1}$ and an edge from v_y to v_x, then the position indexed by b of such row is set to 1. Similarly, OUT contains a row for v_y and the position indexed by a is set to 1. After testing our code with different values for k, we chose the value of k to be 15 which is used to split the input sequence. To check whether an edge exists between two k-mers, we check in the IN and OUT matrices where rows are represented by nodes and columns by a character from the sequence. If there is an outgoing edge from V_x(with sequence a$c_1 c_2 .. c_{k-1}$) to V_y(with sequence $c_1 c_2 .. c_{k-1}$b), the corresponding OUT matrix for V_x marks 1 in its OUT[f(V_x)][char]) while the corresponding IN matrix for V_y sets to 1 in its IN[f(V_y)][char]. This phenomenon is shown in Fig 5. If any of the 2 indexes are set to 0 that means that there is no edge in between these two k-1 mers.

2.3.2 Additional Components for dynamic de Bruijn Graph
As seen in the section 2.3.1, we use here the IN and OUT matrix to store the de Bruijn graph. For the de Bruijn graph to support dynamic operations such as addition and deletion of nodes and edges, we need to construct an underlying undirected graph for the graph to make minimal changes in the cases of the addition and deletion of nodes and edges. For the construction of underlying undirected graph we store the parent pointers of each k-1 mer with a hash value of its parent k-1 mer. Here we store the modified Rabin Karp hash value.

The underlying undirected graph should be a set of trees, which we call forest. Constructing the forest out of the de Bruijn graph is also similar to the problem of finding the number of connected components in a graph. We run Depth First Search like algorithm for finding the connected components in the graph. While running the depth first search we also update the parent pointer information for each k-1 mer by the way that depth first search is traversed. In Depth First Search at first we mark all the nodes as unvisited. We choose an unvisited node (mark is as visiting) and visit all the neighbors of this node in a depth wise fashion instead of breadth wise recursively as long as we are visiting an unvisited node. We stop at either all the nodes in the graph are visited or the height of the each component is $\geq 3 * k \log \sigma$. We repeat the same process for all the unvisited nodes in the graph to form the forest out of the de Bruijn graph. At the same time while running the forest we are also updating the parent pointer information (the way that forest information is stored) and the root of each component is stored in a separate vector along with the size of each component. We use the stored component size and index of the root information later while doing operations on this graph.

2.4 Operations

In section 2 we have discussed about the hash functions, data structures that are used for static de Bruijn graphs and dynamic de Bruijn graphs also discussed about the construction of the underlying forest. Now using these data structures we will see how we can perform the operations like add, delete and searching an edge in the de Bruijn and also maintain the invariants after each condition.

Before proceeding forward we would like to discuss the invariants that author has mentioned[6] while performing each operation. We call the invariants as Size invariant and Height invariant. Size Invariant discusses about the size of each component and the Height invariant discusses about the height of each component in the forest (underlying undirected graph).

Definition 2.1. Size Invariant Size of each component in the underlying forest should be greater than $k * \log \sigma$

Definition 2.2. Height Invariant Height of each component in the underlying forest should be at most of $3 * k * \log \sigma$

2.4.1 Adding Dynamic Edges
Adding Dynamic Edges into the De Bruijn graph[6] is the core operation of this project. Given a Graph G and 2 K-mers, we compute the hash of incoming vertex U and outgoing vertex V for finding the indices of the IN and OUT matrices as we discussed in the section 2.3.

Since we want to ensure desired size of each component in the de Bruijn graph should be at least k $\log_2 \sigma$, we initialize the constants $desiredsz$, $minht$, $maxht$, $midht$ as stated below in the pseudo-code. IN and OUT matrices are set to 1 for adding an edge between 2 K-mers. To calculate the height and size of these disjoint components, we ensure that we are at the topmost position that is the root. And then check for each component whether the Height and Size Invariant is satisfied or not. Next, we need to see if both U and V are both in the same or different components. For this we calculate the corresponding roots for the incoming vertex U as C_i and outgoing vertex V as C_j.

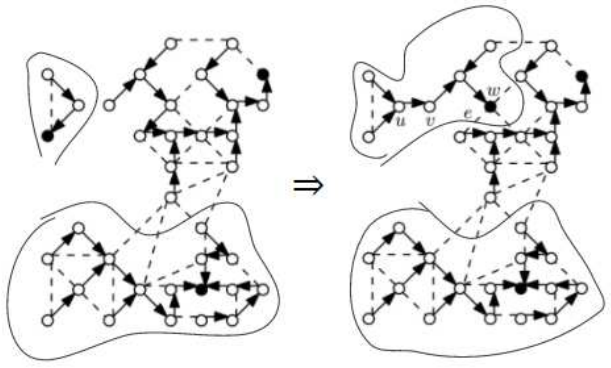


Figure 6. Adding an Edge and then merging components

If the roots are dissimilar then that means that U and V are in 2 different components. If both vertex U and V are having the same root then both the vertex are in the same component. If they are in same component then adding an edge in the underlying Undirected graph G'will form a cycle that needs to be prevented. Hence, we would like to add edges between the 2 nodes only if both the vertex are in different components. Also, calculate the height and size of both components C_i and C_j. We check if both the k-1 mers are in different components in the G in $O(k \log_2 \sigma)$ time Using Depth First Traversal. This part is discussed in the the section 2.3. As stated in paper [6], if the Sz_{C_i} and Sz_{C_j} components are both < $desiredsz$ then $Merge$ both the components with no break required. Break here means that we need not divide the bigger components into 2 halves. If only one of the 2 connected components are bigger than k $\log_2 \sigma$ then we merge the smaller component with the bigger component. While doing this merge we check the depth of the bigger component from V and see if it crosses 2*k $\log_2 \sigma$, if so break the component into 2 halves else just merge.

If sizes of C_i and C_j is greater than $desiredsz$ then we check out the bigger and smaller of those two component sizes and obtain the $depth$ assuming V is in the bigger component.If the $depth$ of the bigger Component is < $midht$ then we might need to just $Merge$ with no break needed in the bigger component as maximum height of smaller component is lesser than $desiredsz$ which doesn't affect any violation to our conditions. We repeat this process till all the Edges are added into the G. If it is the case that the smaller component height is greater than $midht$ then break the bigger component as its height would be greater than $maxht$ as described in the [1] after connecting components. Then merge(We will discuss Merge in more detail in the pseudo code below.) G, U, V with break.

2.4.2 Merging 2 Connected Components
Given a graph G and 2 k-1 mers U_i and V_i (minimal perfect hash (modified Rabin Karp hash(k-1 mer))), we need to merge U to V. This merge function's 2nd parameter is always the smaller component and the 3rd. Within the $Merge$ method, the first step is to mark U as the root of the smaller component. This is done by checking all the outgoing pointers to U and marking their parent pointer information to U. Repeat this process for all the nodes in the neighbor of U until there is no outgoing edge from U. Once all the nodes in the smaller component are pointing to U we can now add and edge in the underlying directed graph (i.e marking the parent pointer of U to V). This takes the time $O(k \log_2 \sigma)$. Once all the parent pointers are changed to U, mark U as the new root of U's Component. Now, the bigger component might be needed to break into two parts in order to satisfy the height conditions. If the size of the bigger component is > $midht$, then the condition(< $maxht$) on height may be violated when we merge the smaller component into the bigger one[1]. We traverse the

Algorithm 4 Adding DynamicEdge into DB Graph

1: **procedure** AddDynamicEdge(G, U, V)
2: $i \leftarrow$ *ComputeHash(U)*
3: $j \leftarrow$ *ComputeHash(V)*
4: $desiredsz \leftarrow k \log_2 \sigma$
5: $minht \leftarrow desiredsz$
6: $maxht \leftarrow 3 * desiredsz$
7: $midht \leftarrow 2 * desiredsz$
8: IN[j][b] $= 1$ ▷ b as described above in Matrix decomposition
9: OUT[i][a] $= 1$ ▷ a as described above in Matrix decomposition
10: $C_i =$ *getRoot(i)*
11: $C_j =$ *getRoot(j)*
12: **if** C_i != C_j **then** ▷ If both vertices are in the same component then adding an edge in the underlying Undirected Graph G'results in a Cycle. So we only consider adding an edge in different components.
13: $Ht_{C_i}, Sz_{C_i} \leftarrow$ *HeightSize(C_i)*
14: $Ht_{C_j}, Sz_{C_j} \leftarrow$ *HeightSize(C_j)*
15: **if** ($Sz_{C_i} < desiredsz$) $\wedge$ $Sz_{C_j} < desiredsz$ **then** ▷ if both of the Sz_{C_i} and Sz_{C_j} is < $desiredsz$.
16: $Merge$(G,U,V, $withoutBreak$) ▷ 1. Update parent pointers to U 2. Add parent pointer of U as V (adding edeg from U to V) 3. Update C_j Component size and height after combining both components.
17: **else if** ($Sz_{C_i} > desiredsz$) $\wedge$ $Sz_{C_j} > desiredsz$ **then** ▷ if any one of the Sz_{C_i} and Sz_{C_j} is > $desiredsz$.
18: $Big_c \leftarrow$ *Max(Sz_{C_i},Sz_{C_j})*
19: $Small_c \leftarrow$ *Min(Sz_{C_i},Sz_{C_j})*
20: *depth* $\leftarrow$ *getDepth(V)* ▷ Assuming V is in Big_c
21: **if** $depth < midht$ **then** ▷ No need to break the bigger one, as the max height of the smaller one is less than k $\log_2 \sigma$
22: $Merge$(G,V,U, $withoutBreak$)
23: **else if** $Ht_U > midht$ **then** ▷ break the bigger one as ht might be > $maxht$ after connecting
24: $Merge$(G,V,U, $withBreak$)

Algorithm 5 Merging 2 Connected Components into One in a DB Graph

procedure Merge(G, U_i, V_i, withBreak) ▷ merge U and V
2: $desiredsz \leftarrow k \log_2 \sigma$
Ht_U, $Sz_U \leftarrow$ *HeightSize(U)*
4: Ht_V, $Sz_V \leftarrow$ *HeightSize(V)*
Change all the parent Pointers in $U's$ Component to point to U. This takes O(k $\log_2 \sigma$).
6: **if** $withBreak$ **then**
Traverse k $\log_2 \sigma$ from v in height and break the V's Component into 2 parts one with height k
8: Keep all the nodes in the boundary (at height k $\log_2 \sigma$) in a temporary array and remove all the edge links for these nodes if they are crossing the $minht$.
if If root(V) is beyond the $minht$ **then** ▷ Old root will be in the broken segment
10: Retain the root of the previous V's component as it is if it is in the broken segment
Choose one among the nodes around the boundary as the parent of the whole U's and V's component.
12: **else**
Sample the nodes in the broken segment for new root.

$desiredsz$ height in bigger component and break all the parent points at the level of height $desiredsz$-1. And we mark the point before break as the new root of the bigger component and change all the meta information that we stored about the smaller and the bigger components size and their Component Root index. We can also choose any of the sample in the bigger component as the new root which might optimize the number of components in the forest later on. Currently, We took the last node in the broken component and change the directions of parent pointers for finding the new root of the broken component if necessary.

Before merging the two components we check for the height and the size invariants and see if both of them are satisfied. During the merge, at first we check if the height invariant condition is not satisfied, if so then we call the merge with the break so that at the end of the merge height Invariant is satisfied. Size Invariant will be satisfied because after adding 2 components into 1 the size of the components will be greater than $k \log \sigma$. Hence size Invariant at the end of the merging 2 components will be satisfied.

2.4.3 Deleting an Edge

Deleting Dynamic Edges into the De Bruijn graph[6] is the another important operation of this project. Given a Graph G and 2 K-mers, we compute the hash of incoming vertex U and outgoing vertex V for finding the indices of the IN and OUT matrices as we discussed in the section 2.3. We want to ensure the Size Invariant2.1 and the Height invariant 2.2 of the components involved are satisfied before and after the delete operation. For following the invariants, as we did earlier we initialize the constants $desiredsz, minht, maxht, midht$ to $k * \log \sigma$, $2 * k * \log \sigma$ and $3 * k * \log \sigma$ respectively. After computing the minimal perfect hash values of the U and V we see if the IN and OUT matrices are set to 1. If both the values are set to 1 then there is an edge and we set that to 0. This is achieved in constant time. In general we need to see if there is an edge in any tree, but based on our assumption described earlier we assume that edge is always present in the de Bruijn graph. In that case , if there is no edge in any tree we do not need to proceed forward. If such edge is present in some tree, then we see for different cases. Case 1, if both U and V are in the same component there are 2 different cases to check here. If both U and V are directly connected or indirectly connected.

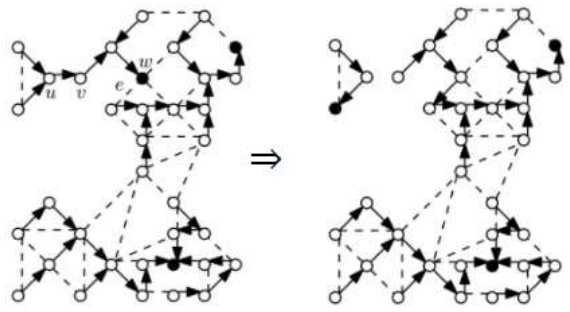


Figure 7. Deletion of an edge and separating components

Case 1.1 If directly connected then we remove the edge dividing the component into 2 halves. Then we check the size of each divided component and see if it doesn't satisfy the Size Invariant2.1. If both the disconnected components satisfy the size invariant then nothing needs to be done. If the size Invariant is not satisfied then we take such component and see if there is any out going edge from that component to any other

component from U and if such an out going edge is found then merge with the smaller component with the other component found. To calculate the size of the both the components in which U and V are we start with U and calculate the height of all the neighbors whose parent is U and then by calculating recursive the size of all such neighbors till no such node is found. The time taken for calculating the size is $o(sizeofcomponent)$ as we touch each and every vertex in that component.

Case 1.2 If both U and V are in the same component but not directly connected then we are not concerned at all and just returns.

Case 2 If both U and V are in different component then it is same as Case 1.2. We need not concern about this as they are not connected at all and just by updating the IN and OUT matrix to 0 will do.

Merging the smaller disconnected component to a bigger component is quite different as we see in the section 2.4.2. Here we calculate the size of the 2 components instead of comparing the height of the combined document first. Based on this we decide whether we have to add the smaller component to a bigger one or mark it as a new component and update all the meta information about the parent pointers and the component information (root index and the size of each component). Once we choose which other component that the smaller component should be added to we can use the same to merge two Components by checking the depth of the other component and decide if break is required or not. If the size invariant is not satisfied then we need not call the merge at all. So to conclude at first we calculate if the size Invariant is satisfied or not based on it we choose any other component and then check for the satisfaction of height Invariant. If the height invariant is not satisfied then the small and other component is broken into two different components. So at the end of the delete operation both Height and Size invariants are checked.

For deleting an edge between 2 vertex U and V

2.4.4 Searching for an edge

Based on our assumptions that we know before hand about the input data, and there are no collisions in our node values and that a node is always present in the de Bruijn graph. For searching whether there is a k mer present in the graph or not we can see the 2 k-1 mers of the k mer and calculate the corresponding minimal perfect hash (modified Rabin Karp hash(k-1 mers)) and check the IN and OUT matrices whether there is an edge from the first k-1 mer to the second k-1 mer.

This searching implicit in nature, but when we integrate this code with the addition of dynamic vertex we might include addition $o(k)$ traversals for checking whether each node is present or not in the graph. This includes additional overhead in the time complexities.

2.5 Testing

For the purpose of testing our code we wrote different test files. These test files performed both Unit Testing as well as Integration Testing. The integration testing is a necessity because after having made sure the different methods work, it is imperative to make sure that these methods interact with each other the way they are supposed to. In order to generate the required sequence from our FASTQ file which is basically the second line for each sequence, we wrote a *PreProcessTest* method. The purpose of this method was to extract the sequence from the input file. So we input the large file and iterated over it to get the second line for each sequence. We also checked the time that it was taking for our input file to be read and the sequence to be generated. The next test method that we used for our purpose is the *ConstructKMersTest*. In this test method, we tested that the (K-1)mers that are being formed are the ones that we are expecting. The (K-1)mers generated are stored in a vector and the vector size was tested to make sure that the size is what we expect. Then we iterated through the vector to make sure that the sequences stored in it are what we expect. Since we didn't want to have any collisions it was also important to check that the vector contains only the unique (k-1) mers. Also the time taken to convert the input sequence to the (k-1) mers is generated and evaluated. We tested this with different values of (k-1) mers to make sure that the time being consumed was optimal for most of the cases.

This was followed by a method for evaluating whether our modified Rabin-Karp algorithm was working the way it was supposed to. We assigned the values of 0, 1, 2 and 3 to the nucleotide bases 'A', 'T', 'G', 'C'. The algorithm that we used was slightly modified for our purpose. The reason for doing this was to avoid collisions since we wanted to do it for a small value of k. We made sure the that the hash values that were being generated are unique and conform to our formula. For its evaluation we calculated our own Rabin-Karp Hash values and asserted that they were the same as those being generated from the formula.

After having generated the hash values which were relatively larger it is difficult to map them to the rows of the matrices. This is where our Minimal Perfect Hash Function was used. Its open source implementation BBHash was used for the same. The values that are generated are in a fixed range. This makes the mapping easier and ensures that there are no collisions. We also evaluated the time that it took for it to generate the minimal perfect hash values for the different kinds of input. This was followed by the evaluation of the underlying forest cover. Firstly, we made sure that the forest is being built with the connected nodes and the structure which we get has all the nodes in proper order. The time taken to build the forest was also calculated. After this, when we have to add a component to the already constructed component, we need to make sure that it satisfies all the constraints. We made sure that when an edge is getting added between two nodes, the parent pointers are updated in the right way. We wrote a method which checked the same as well as calculated the time taken to do the same. Updating parent pointers is necessary because otherwise the location of the nodes wouldn't be lost. During the update of parent pointers it is important to make sure that the two components that are being connected through the addition of edges point to the correct roots in each of the component. The roots for the components should also be updated after every time an edge is added. This is done because the addition of an edge can lead to destruction of the graph into smaller components. When the graph breaks into smaller components each component has its own root. So the time taken to calculate the root for each component is also of concern. We did calculate the time that is taken for the same. The other part that we evaluated was to get the neighbors for the nodes. These neighbors are basically the (k-1) mers that have the k-2 number of characters common among themselves. This was done for all the nodes with the help of their IDs. So for each node the list of neighbors was generated and was compared with our expected results. The time taken to get the neighbors for each node was evaluated. Also deleting an edge between the components was evaluated. The deletion basically depends on the size of the components.

For deleting an edge between two nodes, the location of the node has to be considered. If the nodes are not connected then the edge can't be deleted. If the nodes are in the same component and are connected then the edge can be deleted. If the nodes are not in the same component then the edge can't be deleted.For the purpose of evaluating whether an edge was deleted or not from the right component we evaluated the procedure with sample data. This was done to make sure that if given a larger data, the algorithm works the way it is supposed to. The time taken to perform the operation of deletion on the sample data was also generated.

3 Experimental Evaluation

3.1 Dataset and its characterstics:

We have used a dataset named Escherichia_coli_K_12_MG1655 Illumina Sequencing [7] with the following specifications:

This dataset was contained in a 5.6 GB FASTQ file which consisted of 5718139100 lines. We successfully extracted the sequence reads by dividing the lines of the file by four as well as eliminated the sequence identifiers, break identifier(+) and the quality score through our code. [14]

Chromosomes	Read Count	Read Length	File Size
1	1429534775	200bp	5.6GB

3.2 System Requirements:

Since the dataset is extremely huge that is of the order of gigabytes, a powerful machine must be used for the evaluation so as to deal with the heavy computation that will take place for the construction of the graph and the genome from a very large number of short reads. In order to get precise results we are using a Google Cloud platform to run the code for this dataset.

Platform Specifications:n1-highmem-2 machine with 2 virtual CPUs and 13 GB of memory and 20 GB of standard persistent disk

Operating System: Ubuntu 16.04 LTS amd64 xenial image (Google Cloud platform)

3.3 Test and Performance metrics:

In order to ensure that the performance of our algorithm in terms of space and time complexity, we tested it with different size of sequences that are read from the bigger file. We took samples of 1000, 2000, 3000 sequences from the FASTQ file on our local system. We also divided our read count into chunks of (mention size) we passed the sequence read along with the required k-mer as the parameter to the code that outputs the following observations on given criteria:

k value	k-mers	Static k-mers added (in%)	Trees formed	Avg. tree height	Graph Size	Total Time in sec.	Peak Memory (in MB)
15	5332958	50	81131	40.49	98.4	31	1.9
15	5332958	10	109263	36.9	77.8	34	1.9
20	5562718	10	112634	39.6	88.6	42	2.1
25	5673456	50	136548	42.2	96.8	55	2.2
30	5620464	10	162893	41.8	110.56	61	2.3

Table 1. Testing different metrics on given values of k

Hence the total execution time is actually the sum total of the time taken to construct k-mers, the time taken for the calculation of Rabin Karp Hash and BBHash values, the time taken to store nodes in the IN and OUT matrices, the time to build the forest cover and the time taken to merge these components. Mathematical formulation of the total time would be:
Total time = Constructing k-mers + Hashing + IN and OUT matrices + Forest Build and Merge

A fraction of static k-mers are those that contribute towards the construction of staic de Bruijn graphs as the nodes in IN and OUT matrices. Here we do not deal with the underlying undirected de Bruijn graph. We then use the de Bruijn graph constructed by the static fraction of data for the forest construction. This forest is later used for all the dynamic addition of the edges which is (1-static fraction) percent of the k-mers that the dataset have. Hence, static fraction of the data don't contribute to the forest construction and merge.

Though there are only a few experimental readings mentioned, we find that the count of k-mers increase till a certain value of k and then start decreasing. This is because of the non-unique k-1 mers getting created which lead to the formation of identical nodes, thereby reducing the count of k-mers. The average height of the forest cover is directly dependent on the value of k. With the help of our calculations, we can infer that the time consumed is directly proportional to the value of k.

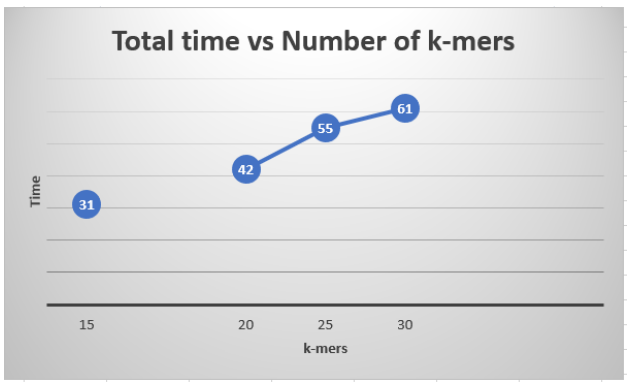


Figure 8. Graph Plot of total execution time vs k

Since the average height of trees directly depends on the value of k as well as the root that is sampled while adding and deleting the edge, using a small value for k makes the process of creating the forest cover relatively easier. All the results generated are completely dependent on k and the static fraction chosen. Reduction in height leads to a reduced query time which

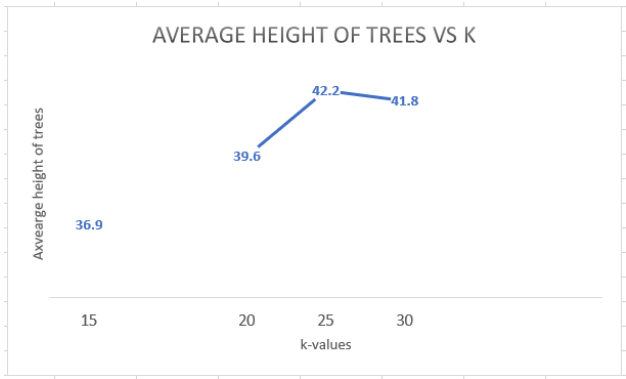


Figure 9. Graph Plot of Average Tree Height vs k

Also, total peak memory consumed by entire algorithm is the sum of RAM required for hashing de Bruijn nodes, RAM utilized for creating and storing IN and OUT matrices, and RAM consumed for building forest and merging different components

Total RAM consumed= RAM consumed for k-mer construction + RAM consumed for construction of IN and OUT matrices + RAM for Forest
We also calculated the insertion, deletion and pattern search average timings and got these observations:

On the basis of these observations we can see that the values increase when there is an increase in the count of unique k-mers. As described earlier, the average tree height plays a very important role in these operations. A smaller tree height leads to a better query search performance.

k-mer/ read length/ read count	Insertion	Deletion	Pattern Search
15/ 2672972/ 200	30	28	11
20/ 4799663/ 200	51	39	14
25/ 5006446/ 200	57	42	15
30/ 5106110/ 200	62	49	14
32/ 5058418/ 200	58	42	13

Table 2. Insertion, Deletion and Search Read timings on given k-mers and Read Length

Therefore, there are better bounds for insertion and deletion when there is an increase in the number of trees and a reduction in the tree height.

4 Challenges

The challenges which we faced during our implementation are as the following:

- Since it was a dynamic De Bruijn graph, we had to consider the possibility of getting a cycle on the addition of a new edge. This in turn lead to collisions while reading sequence and thus, there were multiple permutations of reading the genome sequence since it is a directed graph. Since there was an uncertainty as when to come out of that cycle, it resulted in erroneous reads.

- We had to deal with a case of two parents while we were breaking the graph into two disjoint components or sets after traversing k($\log_2 \sigma$)from the parent as described in the algorithm. This interfered while performing the merge operation and choosing the parent affecting the heights.

- Construction of whole Genome from this compact Data structure lead to a few False Positives.

- Changing the direction of the parent pointers affected the consistency of Edge mappings.

5 Limitations

We have discussed in detail about de Bruijn graphs, advantages of using de Bruijn graphs and its implementation with the capability of adding and deleting dynamic edges. But there are certain limitations of using the de Bruijn graph data structure:

- We have made an assumption that there will be no duplicate hash values from Rabin-Karp Hashing. This assumption will restrict and limit the values of k used.
- For deletion, we assume that edge is already present in the graph. This is because we are assuming that both the nodes are already present and hence this would imply that the edge between the two nodes is also present.

- For certain organisms, using de Bruijn graph consumes a lot of memory. For example, the human genome encoded in de Bruijn graph with each k-mer size as 27 will consume 15 gigabytes of space to store the sequence as k-mers in the nodes of the de Bruijn graph.

- While adding a dynamic edge, if both the connected components are greater than $k \log \sigma$, there is a chance that the height invariant might not be satisfied.

- For the project, we have used a Minimal Perfect Hash function instead of a dynamic hash function. We have used it because of the fact that we are not dealing with node insertions and deletions. Basically, we consider N to be static instead of dynamic in nature where n is the number of (k-1)mers in the whole sequence. If we had the dynamic hash function f, then we could have made N dynamic.

6 Insights

- While adding a dynamic edge, the optimal height and size of the tree could be reached by choosing an optimal route instead of taking the last value before breaking the bigger component into two halves.

- We assume that the value of k given as an input is optimized. However, there may exist values of k that may produce large components in the de Bruijn graph with only a few number a nodes or a small number of components with larger number of nodes.

7 Conclusion

We have successfully implemented the construction of de Bruijn graph using the Illumina sequencing of E.Coli dataset. Given the constraints and invariants, we successfully implemented the search, addition, and deletion of an edge on the de Bruijn graph. Since, we assumed the value of k given as input to be optimum, we had to deal with a trade-off between the number of components and the nodes. We have been able to test our algorithm and have created various test files to perform Unit testing as well as Integration testing. We have created a web page which contains the source code of our algorithm and the instructions on how to run it.

References

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

[2]Botelho, F. *et al.* (2007). Simple and space-efficient minimal perfect hash functions. *Algorithms and Data Structures*, pages 139–150.

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

[4]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.

[5]Compeau, P. E. *et al.* (2011). How to apply de bruijn graphs to genome assembly. *Nature biotechnology*, **29**(11), 987–991.

[6]Conway, T. *et al.* (2012). Gossamer—a resource-efficient de novo assembler. *Bioinformatics*, **28**(14), 1937–1938.

[7]EMBL-EBI (2017). The european bioinformatics institute < embl-ebi. [Online; accessed 8-December-2017].

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

[9]Li, Z. *et al.* (2012). Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in functional genomics*, **11**(1), 25–37.

[10]Limon, M. R. *et al.* (2014). Efficient de bruijn graph construction for genome assembly using a hash table and auxiliary vector data structures. In *Computer and Information Technology (ICCIT), 2014 17th International Conference on*, pages 121–126. IEEE.

[11]rizkg (2017 (accessed 8-December-2017)). Bbhash — github, mit license.

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

[13]Sirén, J. *et al.* (2014). Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, **11**(2), 375–388.

[14]Wikipedia (2017). Fastq format — wikipedia, the free encyclopedia. [Online; accessed 8-December-2017].

[15]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.