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

OXFORD

Project Proposal

Implementation and Evaluation of Dynamic de Bruijn Graph by Adding Dynamic Edges

Apurv Amrutkar [1] , Keyur Baldha [1], Sakshi Dubey [1]

[1] Computer & Information Science & Engineering, University of Florida, Gainesville, 32306

* To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

The production of millions of reads from DNA samples for genome sequencing is now possible due to rapid advancement in sequencing technologies. De Bruijn graph is a prominent data structure used for the efficient representation of this genome sequence. The barriers currently faced in de Bruijn implementation is its memory utilization and runtime. A space and time efficient fully dynamic implementation of de Bruijn graphs was proposed in “Fully Dynamic de Bruijn Graphs“ by Belazzougui et. al (2016). We intend to implement the data structure presented in this paper and focus on the approach of addition and removal of edges dynamically, thereby improving the total memory usage and the running time.

1 Introduction

The curiosity to explore more about human genome has lead to the exponential rise in the creation of DNA sequencing technologies. DNA sequencing can be defined as the process of figuring out the accurate order of nucleotides within a DNA molecule. Initially, early DNA sequencing methodologies were used to sequence human genome. The major breakthrough came with the first generation sequencing when Sanger and Illumina sequencing methodologies evoked a huge interest of researchers in the domain of DNA sequencing. Consequently, the amount of sequenced data generated was gradually rising. As soon as the second generation sequencing technologies came into existence, researchers observed a rapid advancement in the methods of gathering sequence data. One of the first major commercial success was ’next-generation sequencing’ technology. The ongoing development in sequencing technologies began to face an efficiency challenge. The current challenge included the efficient processing of high volume of reads and shorter read lengths for de novo assembly. The huge volume of reads, along with short read length, high coverage and sequencing errors, poses a great challenge to genome assembly. The advancements done in space feasibility for de novo assembly didn’t match the requirements of the gathered sequenced data during the second generation sequencing , and in-practice advancements in technology didn’t match the memory requirements of tools as well.

In microbiology, one of the key challenge has been assembling a genome from a huge amount of reads from various DNA sequences. In bioinformatics, de Bruijn graphs were used to cope with the memory complexity of DNA sequencing of human genome. The de Bruijn graph data structure has certainly proven useful in reducing the time and space complexity for construction of DNA sequences. The de Bruijn graph is a data structure which plays an important role in second-generation sequencing applications and tends to yield a large number of short sequence fragments (Conway and Bromage,2011).

Kth order de Bruijn graph for a set of sequences $S = \{S_1, S_2, S_3...S_t\}$ for a given alphabet Σ of size σ, is defined as the directed graph whose nodes are the distinct $k-tuples$ in those strings in which there is an edge from u to v . If there is a $(k+1)-tuple$ in those strings whose prefix of length k is u and whose suffix of length k is v then the generalized construction steps for de Bruijn graph is as follows:

- Choose a value of k.
- For each $k+1-mer$ that exists in any sequence create an edge with one vertex labeled as the prefix and one vertex labeled as suffix.
- Glue all vertices that have the same label.

(Pevzner,Tang&Tesler,2004)

For sequence assembly, the set S is a collection of overlapping short DNA sequences, called reads. The task is to find an ordering of these reads such that a longer DNA sequence containing these reads is revealed. The longer sequence should correspond to a Eulerian walk of the de Bruijn graph. The static de Bruijn graph was widely used in sequencing DNA. Bloom Filter encoding technique that uses static de Bruijn graph and decreases the amount of memory needed by an order of magnitude to store the graph and an additional data structure for better false positive detection was proposed by (Chikhi, R. and Rizk, G.,2013). A Bloom filter based approach proposed by (Salikhiv *et al*,2014), helped in false positive detection as well as further was able to reduce memory by 30-40% using cascading

Bloom Filter. The other data structure is by (Ye *et al*,2011), which stores only a subset of nodes of the de Bruijn graph to save memory. Sparse Assembler 1 also works on the methodology to skip a certain number of immediate k-mers. Further, a succinct de Bruijn graph representation using Burrows Wheeler Transaform by (Bowe *et al*,2012) could represent graph only in $4m + o(m)bits$. Though de Bruijn graph modified the DNA sequencing technologies to a great extent yet De Bruijn graph was yet to achieve some significant benchmarks. One of the major drawback of using de Bruijn graph in practice is the high memory operation for large genome sequences. The human genome encoded in a de Bruijn graph with a k-mer size of 27 requires 15GB to store the node sequences (Chikhi, R. and Rizk, G.,2013). Another drawback of static De Bruijn Graph is that it needs to be entirely reconstructed if there is any change in the structure of the De Bruijn graph.

The motivation for fully dynamic data structure was arrived to avoid the reconstruction of whole De Bruijn graph in order to add or delete any edges/edges in the existing De Bruijn graph. The concept of dynamically addition and removal of edges/vertices has significantly reduced the memory consumption. In this project, we aim to particularly deal with the addition and removal of edges in a dynamic De Bruijn graph. In (Belazzougui *et al*,2016) a fully dynamic data structure of de Bruijn graph has proposed a combination of Rabin-Karp hashing and minimal perfect hashing functions to represent k-mers. The proposed implementation in this paper uses two algorithms. One of the most important used algorithm is the hashing technique. This paper uses a combination of Rabin-Karp and minimal perfect hashing. Another important pillar of this paper is Forest Cover Method. This paper uses forest cover for rooted tree components to store edge information using IN, OUT matrices.

2 Related Work

In order to deal with the space complexity in processing DNA sequences, many different methods have been proposed. One of the initial strategies to process DNA sequences was to use the De Bruijn graph. One of the possibilities of decreasing memory usage is by using succinct data structures. A succinct data structure is one that uses an amount of space that is bounded closely by the theoretical minimum while still supporting queries efficiently (Conway and Bromage,2011). The representations in (Conway and Bromage,2011) were typical of sets of integers that use the theoretical minimum space bound. The authors further proposed a succinct de Bruijn graph data structure using bitmaps with rank and select operations performed near this minimum bound. (Bowe *et al*,2012) proposed a succinct representation based upon Burrows-Wheeler transforms (Burrows and Wheeler, 1994) where the graph could be represented in $4m + o(m)$ bits,m being the number of edges. (Boucher *et al*,2015) augmented the previous work to support new operations that let them change the order (value of$K$) and thus effectively represented all the the de Bruijn graphs of the order up to the certain value of K in a single data structure. (Pandey *et al*,2017) have discussed in detail about the compact designing of the weighted De Bruijn graph. Their outcome resulted in zero errors and reduced the space requirements by less than 18-28 % than the original requirement. One of the drawbacks of this approach is that this method trades off with accuracy to increase the space efficiency. (Armas *et al*,2016) has also done work in the domain of DNA sequencing using De Bruijn graph and k-mers. It described how k-mer mapping which is an integral process for many assembly methods counts towards increasing the computational challenge which in turn increases its main memory consumption. (Armas *et al*,2016) discussed about the indexing methods to deal with the k-mers. They proposed a model to make a comparison in performances of hashing index structures in an ad-hoc cost model. These indexes have been used for the detection of duplicate k-mers and thus it improves the execution time. They have done a significant contribution in the domain of bioinformatics by achieving significant performance gains with the help of reducing RAM requirements. We observe that most of the works which have been done in the field of succinct de Bruijn graphs are trading off with its accuracy. Thus this drawback limits the use of succinct de Bruijn graphs in many practical implementations of DNA sequencing.

Another efficient use of de Bruijn graph has been proposed using Bloom filters in various works. A Bloom filter is a space-efficient, probabilistic data structure built upon multiple hash functions that are used to test whether an element is in a set, with the possibility of false positives. The use of hash functions to represent k-mers which correspond to nodes in the graph is one of the approaches to reduce the memory usage. There are various papers that decrease the memory usage of de Bruijn Graphs. The de Bruijn graph which is discussed in (Chikhi, R. and Rizk, G.,2013) throws light on the use of bloom filters. It describes a technique that requires less space in comparison to the current representations. The paper proposed by (Chikhi, R. and Rizk, G.,2013) describes about the encoding technique based on a Bloom filter. One of the major concepts which have been highlighted in this paper is the retrieval method of distinguished k-mers. The paper also discusses the method to cope with sequencing errors. It favors in keeping only the solid k-mers to reduce the possible sequencing errors. It also discusses a way to cope with reverse implementation. Apart from this approach, it also discusses a traversal algorithm to make a record of previously visited nodes. This recording structure also serves as space-efficient algorithm since it only accounts for k-mer subsets information. Their algorithm uses an additional structure to remove critical false positives. An important drawback of this representation is that the Bloom filter introduces false nodes and false branching. Though it focuses on maintaining a separate module for false positives which may become a source of false branching, yet there is still a scope for a more efficient method to deal with the false positives. Another work related to Bloom filter was presented by (Salikhiv *et al*,2014) by changing the representation of false positives. They applied a cascade of Bloom filters to represent a set of false positives. This new method required 30% to 40% less memory than (Chikhi, R. and Rizk, G.,2013). They added a new data structure to detect false positives and were able to perform a complete de novo assembly of the human genome using 5.7 GB of memory with the help of a software 'Minia'. The algorithm suggests that the input set of k-mers will be written on the disk. Afterwards, we choose either the k-mer or its reverse complement for computing the calculations. It will be selected in a lexicographically smaller order. The disadvantage of using Bloom filter based approaches is that the de Bruijn graph creation is semi-dynamic, that is it only supports dynamic insertions.

There are various other works which came up with an alternative to reduce the space complexity. Not storing the entire de Bruijn graph is also one of the possibilities to decrease the memory usage by a huge factor. This is a sparse and compact representation of de Bruijn graph. An approach proposed by (Ye *et al*,2011) involves skipping some fraction of the K-mers or reads, thus reducing the size of the overall assembly graph necessary to capture the information. In their approach the K-mers stored in the graph attempts to approximate a uniform K-mer sampling of the genome, rather than based on lexicographic ordering. Another technique which can be utilized from the succinct de Bruijn graph is data compression. In this approach, a statistical method is used to compress a character. In an extension to this approach (Ye *et al* SparseAssembler2,2011) exploited the idea of finding the links between K-mers and then traverse the graph. Thus to realize this new idea they restricted themselves to only some k-mers which might not even be overlapping ones, and build the links between these k-mers indicated by the reads. Their approach improves the space usage by almost 90%. (Md Mahfuzer Rahman *et al*,2017) gave a hashing based approach to decrease the memory usage with 0 false positive rates. The algorithm proposed by (Md Mahfuzer Rahman *et al*,2017)uses auxiliary

data structures where 6 bytes are used to store the $k-mer$ information. HaVec proposed by (Md Mahfuzer Rahman *et al*,2017) is used to achieve a balance between the running time and memory consumption. It also uses a hash table method along with an auxiliary vector data structure to compute De Bruijn graph. The most noteworthy benefit of using HaVec is that it exhibits no false positive error. A lot of work has been done using parallel algorithms to reduce the space complexity. (Chikhi *et al*,2016) proposed a parallel sequencing algorithm that distributes the input based on a hashing technique. They also proposed a tool bcalm2 in order to achieve a compact de Bruijn graph. It has been successful in reducing the computational load of the de Bruijn graph to roughly an hour with the help of 3 GB of memory only. They also used bcalm2 in generating the compacted graphs from raw data in less than 2 days and 40 GB of memory on a single machine. Bcalm2 has been quite successful to reduce the memory consumption with the help of parallel algorithms. In bioinformatics, reducing the load on external memory can also be an alternative in reducing the memory load. One of the similar work has been done by (Bonizzoni *et al*,2016). They have proposed a disk-based algorithm in order to compute the string graphs in external memory: the light string graph(LSG). This technique is dependent on the representation of the FM-index. FM index has been used to save a partial amount of main memory requirement that is independent of the size of the data set. The whole analysis contributed towards a significant decrease in the memory usage though it has caused a moderate increase in the running time. There is another proposed work on the parallel algorithm by (Flick *et al*,2014). They have worked on the weakly connected components in the de Bruijn graph. They have presented a distributed memory algorithm to locate the connected subgraphs to minimize the communication volume. Their method achieved a runtime of 22 min with the help of 421 GB uncompressed FASTQ dataset. Their solution can be applied as a generalized method to find the related components in undirected De Bruijn graph. Most of the discussed related work use the static de Bruijn graph which requires the reconstruction of whole de Bruijn graph in order to accomodate any change in the graph structure. This ignited a motivation for Dynamic De Bruijn graph.

Finally, various researches got inclined towards making a dynamic version of the de Bruijn graph to avoid the reconstruction of graphs in order to introduce any change in the existing graph. (Belazzougui *et al*,2016) have proposed an approach to achieve the dynamicity by building a De Bruijn graph structure from nodes of unique length-k fragments, or k-mers, of a genome. Their approach proposes a combination of Karp-Rabin hashing and minimal perfect hashing functions to represent k-mers. In their approach they store the de bruijn in 2 binary matrices IN and OUT. They also propose creation of an undirected graph called as forest cover for achieving the 0 false positive rate.

In our paper, the proposed data structure implements only a partial dynamic de Bruijn graph in reference to (Belazzougui *et al*,2016) and returns exact answers to membership queries and gives better theoretical bounds provided that the number of connected components in the graph is small. The sole motivation of this paper is to implement one of the solutions suggested by the (Belazzougui *et al*,2016) to reduce the memory consumption. This paper focuses on the implementation of the dynamic addition and removal of edges in the de Bruijn graph. This is achieved by implementing Hashing technique and Forest cover method. The hashing technique includes the combination of SHA1 and Minimal Perfect Hashing to represent k-mers while the Forest cover method is used to store edge information using IN OUT matrices. This paper aims to significantly contribute in the domain of bioinformatics to cope with the space complexities issues for sequencing DNA with the help of Dynamic De Bruijn graph.

3 Description and Implementation of Data Structure

3.1 Overview

In this section we discuss about the technical formulas we are going to use in this project. The Lemmas are as follows: **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 : $\Sigma^k \rightarrow$ 0, ..., n-1 with the following propoerties*

- *when its domain is restricted to N, f is injective;*
- *we can store f in O(n+log k + log σ) bits*
- *given a k-tuple v, we can compute f(v) in 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(v) then we can compute f(v) in O(1) time;*
- *deletions take O(k) amortized expected time.*

Let us consider N as the node-set of a de Bruijn graph. A node in our de Bruijn graph can be stored using the function as described in Lemma 1. In the subsection 3.2 we describe the construction of the de Bruijn graph data structure in a way that, given a pair of $k-tuples$ u and v of which at least one is in N, we can check whether the edge (u,v) is in the graph. This means that, if we start with a $k-tuple$ in N, then we can explore the entire connected component containing that $k-tuple$ in the underlying undirected graph which we call the $forestcover$. On the other side, if we start searching for a $k-tuple$ not in N, then we will learn this fact quickly as soon as we try to cross from one $k-tuple$ in N to another. Thus to handle the possibility that we never try to cross such an edge we cover the vertices with a forst of shallow rooted trees. In the forest we store the graph such that the root of each component in the forest is stored as $k-tuple$, and other nodes in this component are stored in 1+log σ bits indicating which of its which of its edges leads to its parent. The response to a membership query to check whether a $k-tuple$ we are considering is indeed in the graph, we ascend to the root of the tree that contains it and check that $k-tuple$ is what we expect. In the subsection ahead we have explained the implementation steps for creating the de Bruijn graph data structure. In addition to this, we discuss about the combination of Rabin-Karp and minimal hashing functions, algorithm on forest cover for the tree and algorithm for dynamically updating the graph by additions or deletions of edges.

3.2 Construction of de Bruijn Graph

Let $G = (N, E)$ be a de Bruijn Graph of the order k with n vertices. Let $E = a_0, a_1, ..., a_e1$ be the set of edges and $N = v_0, v_1, ..., v_n1$ be the set of vertices. In this paper we call each v_i as a vertex or a $k-tuple$, using these terms interchangeably as the vertex and the labels have a one-to-one correspondance.

Graph G data structures is stored and maintained in two binary matrices, IN and OUT, of size $n \times \sigma$. IN is a binary matrix which represents the incoming edges towards a $k-tuple$ whereas OUT is a binary matrix which represents outgoing edges from the $k-tuple$. For example, for each $k-tuple$ $v_x = c_1c_2...c_{k-1}a$, the IN matrix stores a row of length σ such that, if there exists another $k-tuple$ $v_y = bc_1c_2...c_{k-1}$ and an edge from v_y to v_x, then the position indexed by b of such row is set to 1. Similarly OUT binary matrix stores a row of length σ where for row v_y the position indexed by a is set to 1. In the data structure for the graph proposed in this paper, each $k-tuple$ is uniquely mapped to a value between 1 to n using a hash function f as described in $Lemma1$ and explained in section 3.3. Thus, as we are represeting values between 1 and n we can use these in our binary matrices directly for the row label of each $k-tuple$. Thus in our previous example, for the IN matrix the value

of $IN[f(v_x)][b]$ would be set to 1 and similarly for the OUT matrix the value of $OUT[f(v_y)][a]$ will be set to 1.

With the above mentioned data structure to store the graph, we can now check whether there exists an edge from bX to Xa. Let $f(bX) = i$ and $f(Xa) = j$, now let us assume bX is in G and thus we check the values for the edge between bX and Xa in our IN and OUT matrices, using the previous representation of i and j. Thus, if values $IN[j][b]$ and $OUT[i][a]$ are set to 1, we can confirm that the edge is present in G. Otherwise, if any of the two values is 0, we would the edge to be absent. Moreover, we can note that if bX is in G and $OUT[i][a] = 1$, then we can conclude that Xa is also in G. Similarly, if Xa is in G and $IN[j][b] = 1$, then we can conclude that bX is also in G. Therefore, if $OUT[i][a] = IN[j][b] = 1$, then bX is in G if and only if Xa is in G. Thus we can note from this property of our data structure of two binary matrices that, if we have a path Z and if we confirm all edges in path Z using the IN and OUT matrix, then either all the vertices are present in G or none of them is.

The algorithm for construction of the above matrices does simple iteration on the set of $k - tuples$ which are created from the evaluation genome sequence. The IN and OUT matrices are filled for a particular $k - tuple$ and the next $k - tuple$ by calculating the prefix and suffix of the $k - tuples$, calculating the hash values of the $k - tuples$ as described in the overview and accordingly updating the particular row of IN and OUT matrices as we have previously explained.

If any one of the above matrix element is 0 then the edge does not exist. This algorithm, depending on the hash function may also sometimes generate false positives in the binary matrices. Hence, if the corresponding index is 0, then the edge is definitely absent; otherwise, the edge is possibly present. We then focus on maintaining proper memory usage and detecting false positives. More accurately, we partition nodes in a graph G' underlying G into a forest of rooted trees of height at least $klog\sigma$ and at most $3klog\sigma$. We will describe more about the partition creation using Forest tree cover in section 3.4. We will also describe the hash functions used for this data structure in Section 3.3. For dynamic updating of de Bruijn graph, we describe the algorithm for additions and deletions of edges in Section 3.6 and 3.7. We also discuss algorithm about querying the data structure for a membership query of an edge in Section 3.5.

3.3 Hash Function

A hash function is used to convert a string into a numeric hash value. For example for a hash function we might have hash('abcd')=5. We use hash function to exploit the fact that if two strings, in our case $k - tuples$, are equal, their hash values are also equal. This would help us in reducing the space required to store the $k - tuple$ as string, instead now would be stored as hash values. Another advantage of this is, string matching gets reduced to computing the hash value of the search query. Hash function determined by the Rabin-Karp algorithm seeks to speed up the comparison of equality of the pattern to the substrings in the text.

Let us consider a subset S of all possible strings of length k over an alphabet of the universe U. Given a prime P and base $r[0, P1]$, a Rabin-Karp hash function f is a function defined over U such that $f(x_1....x_k) = (x_i r_i n_i = 1) mod P$. f is a one-to-one minimal perfect hash function on S defined on the universe and the range for the same is $0, ..., n1$.

However, there are certain problems with this approach. Firstly, as there are a lot of strings and very few hash values, there are possibilities of different strings having the same hash value. We can have multiple strings in the subset S which evaluate to same hash value but the strings may not match. Thus, there can be collision which may occur for our Rabin-Karp hash function generated. The solution we have implemented for this situation is to have a Rabin-Karp hash function which is injective, i.e, every string in the subset S would have a unique hash value. We achieve this by using a Las Vegas algorithm. In this algorithm we construct a bitvector B of size n initialized with all its elements set to 0. For each element v_x of S we compute $f(v_x) = i$ and check the value of $B[i]$. If we find the value to be 0 we set it to 1 and proceed to the next element in S, if the value is already set to 1, then we have a collision for the Rabin-Karp hash function which we are currently considering; thus we select a different f and restart the procedure from the first element in S. We now know that once we finish this procedure we would have a f which is injective.

However, Rabin-Karp with Las Vegas algorithm for finding an injective function(collision free) is good for small dataset and small value of k. After experimenting with huge dataset, finding an injective function consumes a lot of time, as it becomes difficult to find a perfect combination of r and P for Rabin-Karp. Also, for higher values of k the computation of the huge number takes a lot of memory space and computing time. The solution applied for this is using SHA1 hash on the string and then only considering last specific number of bits according to the value of k. Using the last few bits of SHA1 tackles the issue of huge number computation by Rabin-Karp. We also check injective behavior of SHA1 hash function for only considering the last few bits, using Las Vegas algorithm, and increment the number of SHA1 bits to be considered in the hash. Experimentally we get very less or no collision with SHA1 for huge sequence length.

As described in the overview of the algorithm description, and also according to the $Lemma$ 1 we want the hash value to be between 1 and n, where n is the number of $k - tuples$ in N. This is actually required by our IN and OUT. To achieve this we have proposed a combination of Rabin-Karp (Karp and Rabin, 1987) and minimal perfect hashing (Hagerup and Tholey, 2001) function.

A hash table with no collisions can be built with a technique called as Perfect Hashing. This is only possible when we know all the keys in advance. A minimal perfect hash function is actually a perfect hash function which maps n keys to n consecutive integers which in our case are the numbers from 0 to $n1$. To express this in more formal way: Suppose j and k are elements of a set S. We consider F as a minimal perfect function if an only if $F(j) = F(k)$ implies $j = k$, i.e., injectivity. We use two levels of hash functions. The first one, H(key), gets a position in an intermediate array, G. The second function, F(d, key), uses the extra information from G to find the unique position for the key. The scheme will always returns a value, so it works as long as we know for sure that what we are searching for is in the table. Otherwise, it will return bad information. Fortunately, our forest creation as discussed in the overview would prove an advantage over here as we are using it to detect false positives, which can occur if we are not able to find the exact hash value for some query string. We construct the intermediate table G for our minimum perfect hashing function by incrementing d in F(d, key). We follow the following steps while constructing our minimal perfect hash fucntion:

1. we place the keys into buckets according to the first hash function, H.
2. we process the buckets largest first and try to place all the keys it contains in an empty slot of the value table using F(d=1, key). If that is unsuccessful, we keep trying with successively larger values of d. Since we try to find the d value for the buckets with the most items early, they are likely to find empty spots. When we get to buckets with just one item, we can simply place them into the next unoccupied spot.

A hash function described in the paper is outlined as GENERATEHASH below.The particular functions for the algorithm are as follows

- **getPrime(R)** - smallest prime is returned. Such a prime may be generated with high probability in O(n) time by the method described in (Dietzfelbinger et al., 1997).
- **randomNumber(0, P - 1)** - returns a uniformly distributed random number between 0 and P - 1.
- **rabinHash(r, P)** - returns a Rabin-Karp Hash function with base r and mod P. There is a high probability this is injective.

- **isInjective(f, S)** - tests whether f is injective on S.
- **minimalPerfectHash(X)** - returns a minimal perfect hash function on the set X. This may be done by using the above explained method.

The algorithm for GENERATEHASH has two levels, first level finds an injective Rabin-Karp hash function f for the set of $k-tuples$. This hash function uses the $getPrime()$ and $randomNumber()$ functions as described above to generate a hash value as described previously. To obtain an injective Rabin-Karp function for our set of strings we keep on trying different random numbers using the $randomNumber()$ function as described above. Second level of hash function is the minimal perfect hashing as described above. The Rabin-Karp hash function is given as input to our minimal perfect hashing inorder to generate a map of n keys which are numbers from 0 to n as described above, where n is the size of the set of $k-tuples$.

3.4 Forest Construction

According to the overview of our data structure, we can run into situations where we can have some membership queries to the graph G for vertices which have the same hash value according to our hash function described in 3.1, thus causing a false positive to occur. A false positive is basically a result which incorrectly indicates that a particular condition or attribute is present. So we focus on detecting these false positives in our forest data structure maintaining a reasonable memory usage. We basically sample a susbset of nodes for which we store the plain text $k-tuple$ and connect all the other vertices which are unsampled to the ones which are sampled. We basically partition the vertices in the graph G' underlying G into a forest of trees which are rooted components, of height at least $k\ lg\ \sigma$ and at the most $3\ k\ lg\ \sigma$, where σ is the size of our alphabet Σ size. In each forest component after the root vertex for all other vertices we store a pointer to its parent in the tree, this pointer takes upto $1+\ lg\ \sigma$ bits per vertex, and we sample the $k-mer$ at the root of such forest component. In our implementation we do not have a hard constraint on the lower bound of the height of the forest component to be $k\ lg\ \sigma$, e.g when it covers a connected component. Thus the invariant maintained throughout forest construction is : any forest component or a rooted tree must have height at least $k\ lg\ \sigma$ and atmost $3k\ lg\ \sigma$ except when the tree covers (all vertices in) a connected component of size at most $k\ lg\ \sigma$.

Now returning back to no false positive generation in our forest data structure implementation. We can now thus check whether a given node v_x is in G by first computing $f(v_x)$, where f is the same hash function as described in section 3.1, and then checking and ascending at most $3\ k\ lg\ \sigma$ edges, thus updating v_x and $f(v_x)$ as we move ahead. Once we have reached the root of our connected component in the forest, we can now compare the resulting $k-tuple$ with the one we sampled at the root to check if v_x is in the graph. The current procedure for verification of whether a vertex is present in our graph requires $O(klg\sigma)$ time since computing the outer first hash value of the query, $f(v_x)$, requires $O(k)$, ascending the tree upwards to the root requires constant time per edge, and thus comparing $k-tuples$ requires $O(k)$.

We create the forest components as described above with only root containing the $k-tuple$ and rest of the connected vertices as $f(v_x)$ value with only an additional parent pointer, this is because we don't want to store the whole $k-tuple$ for every vertex, as this would take too much space. If the trees are big enough, then the sample is sparse enough that it doesn't add much to the overall space. On the other hand, if the trees are shallow enough, then we can get from any vertex to the root of its tree quickly following the procedure as described above. Thus in our undirected graph G' we want to choose a subgraph that is a forest of fairly big but fairly shallow trees that together contain all the vertices in the graph.

The procedure to create the Forest F for de Bruijn graph G is as follows:

1. Start with a random $k-tuple$ string and make it a root node.
2. Create the hash value of $k-tuple$ string
3. Find neighbors using OUT matrix. For every neighbor create a new node and set the parent pointer. Continue this procedure till we maintain the height invariant of the forest.
4. Break off the subtree corresponding to node at position $3klog\sigma$.
5. Continue the above process until we have traversed all the $k-tuples$.

The algorithm for forest creation uses Breadth First Search to traverse through the IN and OUT matrices starting with a random $k-tuple$. We use Breadth First Search as our goal of creating the forest is shallow and which would contain maximum number of nodes. Nodes in the forest are basically objects of class $TreeNode$ which stores a parent pointer(hash value of the parent node), height of the tree from this node, if this is a root node then we also store the plain text $k-tuple$. We follow the algorithm as described previously by maintaining the forest invariant during the traversal through the matrices.

3.5 Membership Query for an Edge

We describe a method to query an edge in our de Bruijn data structure, which is a combination of IN, OUT matrices, $Hash-function$ and the $forest$. Suppose the membership query is to check whether there is an edge from bX to Xa. Letting $f(bX)=i$ and $f(Xa)=j$. We now have to first check whether node j and node i in forest actually correspond to the string bX and Xa respectively. This is achieved by ascending k parent edges in the forest and thus checking the corresponding string generated against the query prefix i and suffix j nodes. If either of them is not present we conclude that the edge is not present. Thus forest construction is finally used to prove the 0 false positive rate, as we are confirming our hash values generation with the string comparison. The case, if both the nodes are present in the forest, then to check whether the edge is present, we check the values of $OUT[i][a]$ and $IN[j][b]$. If both values are 1, we report that the edge is present and we say that the edge is confi

rmed by IN and OUT; otherwise, if any of the two values is 0, we report that the edge is absent. Moreover, note that if bX is in G and $OUT[i][a]=1$, then Xa is in G as well. Symmetrically, if Xa is in G and $IN[j][b]=1$, then bX is in G as well. Therefore, if $OUT[i][a]=IN[j][b]=1$, then bX is in G if and only if Xa is. This means that, if we have a path P and if all the edges in P are confi

rmed by IN and OUT, then either all the nodes touched by P are in G or none of them is.

3.6 Removing Dynamic Edges

In this section we describe the case of removing an edge to the graph in our data structure. Now suppose we want to remove an edge $e=(v_x,v_y)$ where v_x and v_y are the vertices in G. Similarly as described in previous section, we first update the IN and OUT matrices, but now we set the values to 0 of $OUT[f(v_x)][a]$ and $IN[f(v_y)][b]$ in constant time.

Now to update the forest we need to take multiple cases into consideration. First, we check in $O(k)$ time whether e is an edge in some forest component by computing $f(v_x)$ and $f(v_y)$, checking for each vertex if the current edge in consideration is the one that points to their parent. If e is not in any tree we do not move ahead with our procedure, as no update is required in the constructed forest, this case is represented in $Figure1$. If the edge is present in some component of the graph, then we check the size of each tree in which v_x and v_y reside in. We call the component which had a parent pointer in the edge as the child component while the other one is referred as parent component. Multiple cases can occur if the edge

is present in the forest as it would lead to update of the forest to be carried on in a particular way, by taking the forest invariant into consideration. We have discussed the cases as follows:

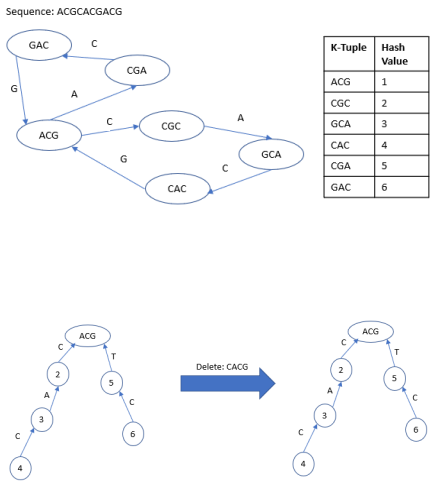
1. If any of the trees is small, i.e., size less than $klg\sigma$, we then search any outgoing edge from the tree that connects it to some other tree.

   a. If we are unable to find the edge in our current traversal we conclude that we are in a small connected component that is covered by the current tree. We then sample a vertex,i.e., add a plain text k-tuple to that node and make it a different rooted tree component, this is as shown in $Figure2$
   b. If the outgoing edge is found, then we merge the smaller component with the outgoing edge node. We then check for the invariant of the forest, this can generate the following cases:

   (1) If new updated tree component follows the forest invariant then we do not need to move ahead with the procedure, this is as shown in $Figure3$
   (2) If new updated tree component does not follow forest invariant then we traverse the tree upwards from the deepest vertex in the tree(in this case from our child component) for $2klg\sigma$ steps,delete the edge pointing to the parent of the reached vertex, and we create a new tree.

   After performing the forest update, we also need to update the height of every vertex till the root in the parent component.

The algorithm used for achieving the above goal of removal of the edge and updating the forest, uses the array of the TreeNode(a node in the forest) to find a node in the forest, then check for the presence of the node, similar to the procedure followed in membership query. We make use of the IN matrix to find the node which can be reached by an outgoing edge from the small component. If the node is not found we make the small component node as the root, and thus update the TreeNode value with the plain text k-tuple value. If the outgoing edge node is found, as discussed in the cases we attach this small component to that rooted tree component. The update of height of every node till the root node in the parent component is achieved by finding heights of all the children of every node and then making an update with the maximum value. We find all the children using the OUT matrix, then calculating hash value of every child found and finally checking the parent of those children nodes in the forest against the current parent component.



K-Tuple	Hash Value
ACG	1
CGC	2
GCA	3
CAC	4
CGA	5
GAC	6

Fig. 1: Case - no update in forest is required

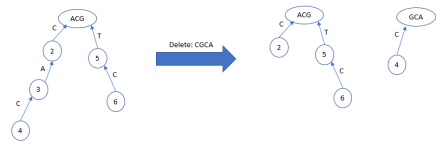


Fig. 2: Case - 1.a

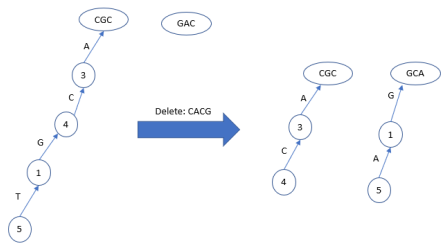


Fig. 3: Case - 1.b.1

3.7 Adding Dynamic Edges

In the previous sections we described about construction fo the data structure for de Bruijn graph, we would now present how we make this data structure dynamically. In this section we will basically focus on adding or inserting an edge and thus updating the graph. For our data structure update for the graph basically boils down to update of the forest. While updating the forest we maintain the invariant we considered while constructing the forest.

Let v_x and v_y be two vertices in G, $e = (v_x, v_y)$ be an edge in G, and let $f(v_x) = i$ and $f(v_y) = j$.

Now, considering we want to add e to G. First, we need to update the IN and OUT matrices we described in section 3.1. So we set to 1 the values of $IN[j][b]$ and $OUT[i][a]$ in constant time. Now coming to update of the forest, we first check whether v_x or v_y are in different forest components of size less than $k\ lg\ \sigma$ in $O(klg\sigma)$ time for each vertex. If both components have size greater than $klg\sigma$ we do not have the need to update the components since the roots tree will not change. If both connected components have size less than $klg\sigma$ we merge their trees in time $O(klg\sigma)$, we also discard the samples at the roots of both tress and sample a new root in $O(k)$ time.

However, if anyone of the connected component we need to merge has a size greater than $klg\sigma$ we select it and then we traverse that forest component to check whether the depth of the vertex is less than $2klg\sigma$. If it turns to be true, then we just connect the two trees as in the previous case. But if they are not, we will have to traverse the component which is bigger upwards for $klg\sigma$ steps, we now delete the edge pointing to the parent of the vertex we reached, thus creating a new tree which we then merge with the smaller one. This is done in order to maintain our forest invariant. We also update the height of the every node till the root node, in the new tree where the smaller component is attached.

The algorithm used for achieving the above goal of adding of the edge and updating the forest, uses the array of the TreeNode(a node in the forest) to find a node in the forest, then check for the presence of the node, similar to the procedure followed in membership query. The case of nodes present in different forest components having a height less than what is required by the invariant, is handled by updating the parent pointer of the node. The update of height of every node till the root node in the parent component is achieved by finding heights of all the children of every node and then

making an update with the maximum value. We find all the children using the OUT matrix, then calculating hash value of every child found and finally checking the parent of those children nodes in the forest against the current parent component.

4 Experimental Evaluation

We have used python to build the data structure and the entire source code is shared on github in a repository. To test the performance of the code, we have used a system with the configuration, Intel Xeon E5-2676v3 with 2.4 GHz and 32 GB RAM.

4.1 Data Description

We will test the data structures on one dataset Escherichia coli str. MG1655 but with different sizes of sequence length considered for evaluation is summarized in Table 1. To create edge membership queries, we select multiple $k+1$ length strings from the datasets. We test the membership queries with both before and after deletion of the edge, thus to test our feature of 0 false positive rate. We have performed evaluation on one dataset, by considering different sequence lengths of the same dataset and for different values of k.

Sequence Name	Read Count
K12 substr. MG1655	1088314
K12 substr. MG1655	2142853
K12 substr. MG1655	3393761

4.2 Results

We have proposed an experiment with a goal to analyze the behavior of Fully Dynamic de Bruijn Graph by just developing a partial implementaition on the basis of the research proposed by (**?**) in the paper "Fully Dynamic De Bruijn Graph".This experiment is done on the fastq input file as described in Section 4.1 with various k values and varied sequence lengths. We have performed experiments for comparing time required for each section of the algorithm described in Section 3.x.

Comparison of first level hash function is displayed in $Figure4$. According to the results the computation of Rabin-Karp and making it injective takes a lot of time as compared to our new method proposed later which uses SHA1 as the first level hash function. As explained earlier Rabin-Karp utilizes more memory and time while calculating hashes for higher values of k, the results in $Figure4$ clearly compare the overall time taken by Rabin-Karp and SHA1 for $k = [10, 32]$ and for a sequence length of $1.02million$. Thus switching to SHA1 was proven to be a better alternative. In SHA1 we are only considering small number of digits from the LSB of the hash value generated, in the future work for a huge sequence length we need to find a better relation between k and the injective hash function value LSB length.

Further, the comparison of time required to generate an overall hash function generation with the minimal perferect hash against the various sequence lengths and values of k is as shown in $Figure5$. We can conclude that the value of k does not play a bigger role than that of sequence length. The hash function generation time as seen increases linearly. The future work would include comparing the time for hash function generation for bigger sequence lengths and values of k, accordingly find a relation between the value $LSB\ length$ of SHA1, sequence length and k.

Construction of de Bruijn Graph and represnting them in IN and OUT binary matrices also becomes an important factor in the time required for generation of the data structure proposed. $Figure6$ compares the construction time of binary matrices with the sequence length taken into consideration, according to the experiment we can conclude there is linear increase in construction time with the increase in sequence length.

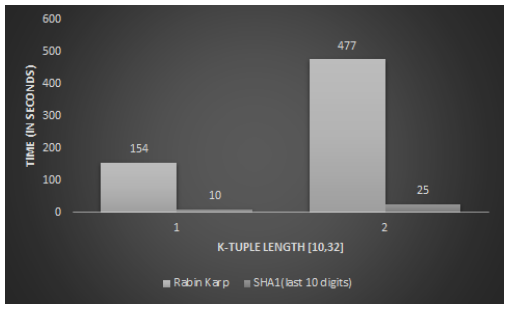


Fig. 4: Comparison of Rabin-Karp with SHA1

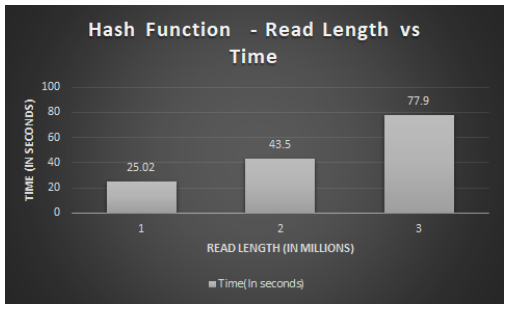


Fig. 5: Comparison of Time against Read Length for Hash function construction

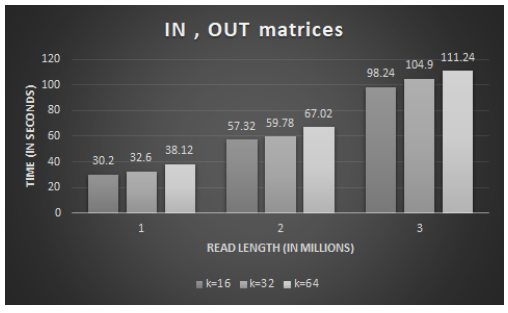


Fig. 6: Comparison of Time against Read Length for IN, OUT matrices construction

Forest construction in implementation is done using pointer based implementation of the TreeNode class. $Figure7$ compares the construction time of the forest with various sequence lengths. Conclusion from the figure turns out to be a linear relation between the length and time requireed for construction. Finally, the overall construction time of the data structure for different sequence lengths and different values of k appears to be linear as seen from $Figure8$.

One of the experiment with 1.02 million sequence length also captured the memory required for storing a python Object consisting of the whole data structure to represent our proposed de Bruijn Graph, i.e, IN,OUT matrices, $Hashfunction$, and $Forest\ TreeNode$ in a map, takes in total of 311 MB of disk space.

In the data structure implemented we have also compared the time required by membership queries for edges of $k+1$ length. $Figure9$ shows a comparison between sequence length, k and time required to perform a membership query when the edge is present and not present in the dataset.

The data structure proposed is a part of Fully Dynamic De Bruijn Graph, that includes addition and deletion of edges. $Figure10$ shows a comparison of deletion time against sequence length and k. Addition and Deletion of an edge is dependent on the procedure of finding the edge first which is the membership query, and then performing the procedure to maintain the Forest invariant. The overall time consumed for this theorotically should be $O(klog\sigma)$ as complexity and practically we get values in microseconds.

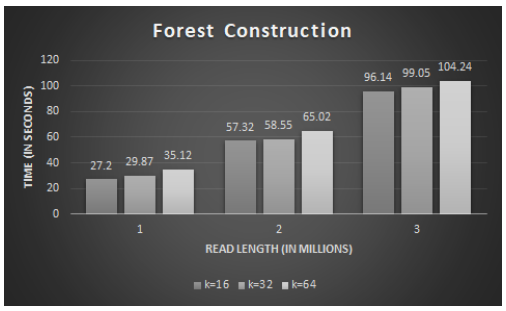


Fig. 7: Comparison of Time against Read Length for Forest construction

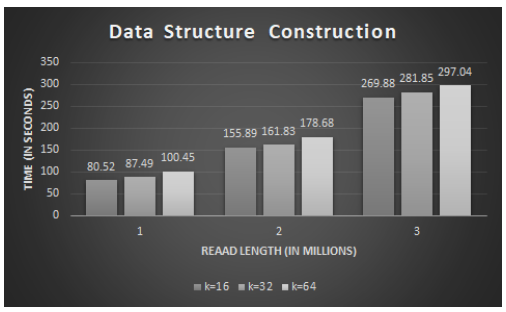


Fig. 8: Overall Data Structure construction

4.3 Conclusion

We can conclude that in the proposed algorithm the practical time for construction of the graph increases linearly with the increase in the sequence length of the dataset. We have only compared the performance of the algorithm for construction and dynamic update of the de Bruijn Graph in the data structure proposed.

However, we need to be taken into consideration that the relation between k, sequence length and calculation time of the Hash function is still a black box for us, as it totally depends on the dataset as input. Theorotical time complexity for the same cannot be defined based on the data at our disposal.

Moreover, the memory disk space required to store the python object is a practical space we have calculated in the experiment. Theorotically the space required to store our proposed data structure should only take $O(\sigma n)$ bits for the IN and OUT matrices and $O(klog\sigma)$ bits for each connected component in the forest constructed. Thus from implementation point of view forest and matrix construction can still be improved in the future work.

Finally, the partial dynamic behavior of data structure is proposed by addition and deletion of the edge in the graph and accordingly updating

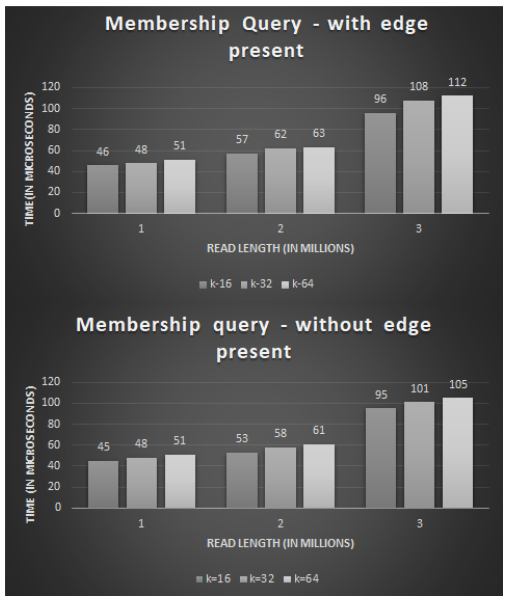


Fig. 9: Comparison of Time against Read Length for membership query

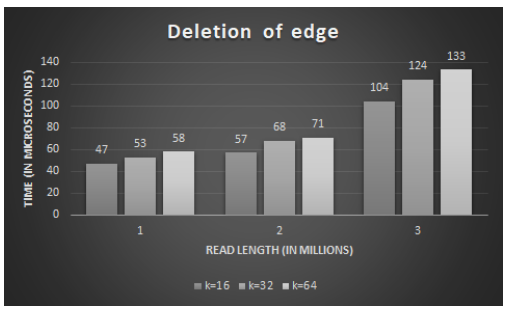


Fig. 10: Comparison of Time against Read Length for deletion of an edge

the Forest and IN and OUT matrices. The time consumed for doing so is negligible as compared to the time required for construction of the graph. Thus we can conculde that "Fully Dynamic De Bruijn Graph" proposed and implemented in this paper which is in reference to the paper by (Belazzougui *et al*,2016), is definately a great step in comparison to 'Static de Bruijn Graph' when only edge addition and removal is taken into consideration.

References

[Armas *et al*,2016]Armas, E. M., Haeusler, E. M., Lifschitz S., Holanda, M. T., Silva, W.M.C., and Ferreira, P.C.G. (2016). K-mer maping and de Bruijn Graphs: The case for velvet fragment assembly .*Bioinformatics and Biomedicine (BIBM)* ,, 16604259.

[Belazzougui *et al*,2016]Belazzougui, D., Gagie, T., Maekinen, V., and Previtali, M. (2016). Fully Dynamic de Bruijn Graphs.*In String Processing and Information Retrieval*,, pages 145âŁ"152.

[Bonizzoni *et al*,2016]Bonizzoni, P., Vedova, G. D., Pirola, Y., Previtali M. and Rizzi, R. (2016). LSG: An External-Memory Tool to compute String Graphs for Next-Generation Sequencing Data Assembly.*J Comput Biol*,,137-49.

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

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

[Chikhi, R. and Rizk, G.,2013]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.

[Chikhi *et al*,2016]Chikhi, R., Limasset, A., and Medvedev, P. (2016). Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, ,i201-1208.

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

[Flick *et al*,2014]Flick, P., Jain, C., Pan, T. and Aluru, S. (2014). Reprint of "A parallel connectivity algorithm for de Bruijn graphs in metagenomic applications". *Parallel Computing*, ,pages 54-65.

[Md Mahfuzer Rahman *et al*,2017]Md Mahfuzer Rahman, R. S. (2017). HaVec: An Efficient de Bruijn Graph Construction Algorithm for Genome Assembly. *International Journal of Genomics*, ,22.

[Pandey *et al*,2017]Pandey, P., Bender, M. A., Johnson, R., and Patro, R. (2017). deBGR: an efficient and near-exact representation of the weighted de Bruijn graph . *Bioinformatics , 2017*, ,pages i133-i141.

[Pell *et al*,2011]Pell J, Hintze A, Canino-Koning R, Howe A, Tiedje JM, Brown(2011). Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Arxiv preprint arXiv:1112.4193. 2011*, ,11.

[Pevzner,Tang&Tesler,2004]Pevzner,Tang&Tesler,2004. De novo repeat classification and fragment assembly. *Genome Res. 2004 Dec;*, **14(12)**,2510.

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

[Ye *et al*,2011]Ye, C., Ma, Z. S., Cannon, C. H., Pop, M., and Douglas, W. Y. (2012). Exploiting sparseness in de novo genome assembly. *BMC bioinformatics*, **13(6)**, S1.

[Ye *et al* SparseAssembler2,2011]Ye, C., Ma, Z. S., Cannon, C. H., Pop, M., and Douglas, W. Y. (2012). SparseAssembler2: Sparse k-mer Graph for Memory Efficient Genome Assembly. *BMC bioinformatics*, **13(6)**, S1.