Bioinformatics
doi.10.1093/bioinformatics/xxxxxx

Deriving DNA Sequence using fully dynamic De Bruijn Graphs

Project Proposal

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

De Bruijn Graphs, in Bio Informatics, are used for assembly of genome sequences obtained from a Next Generation Sequence (NGS) library. These De Bruijn graphs must be implemented using data structures that are space and time efficient. Djamal Bellazzougui laid out an approach to implement De Bruijn graphs efficiently which supported dynamic insertion and deletion at the same time in his paper “Fully Dynamic De Bruijn Graphs”. The aim of our project is to implement Dynamic addition of edges into De Bruijn graph which is constructed using a similar compact data structures.

1 Introduction

Using an NGS library, a De Bruijn graph is constructed for genome assembly(Conway and Bromage, 2011). This makes De Bruijn graphs imperative in the field of Bio-Informatics and DNA sequencing. De Bruijn graphs enable us to represent a sequence in terms of k-mers. The graph is basically a directed graph with unique nodes representing k-tuples. These k-tuples are decided by starting with the first k nucleotides and then moving forward with the next k overlapping nucleotides excluding the first nucleotide in the present tuple. Later, the tuples that are identical are joined and a directed graph is formed. The direction of the arrows are such that the arrow originate from the k-mer with the last k-1 overlapping nucleotides and ending towards the k-mer with the first k-1 overlapping nucleotides.[1]
There can be several ways in which a De Bruijn graph can be constructed. Research has been done in order to implement De Bruijn graphs using Burrows Wheeler transform or Bloom filters[4]. There are more approaches to implement De Bruijn graphs and using minimal perfect hash functions is one of them[1]. We use the same to implement De Bruijn graphs, the components of which are dynamic implying that nodes and edges can be both inserted and deleted dynamically. This implementation is highly efficient for a smaller graph size and yields a better result than using bloom filter, as the graph they generate are only semi-dynamic and hence, only support dynamic insertions and not deletions[4]. We use a data structure based on Karp-Rabin Hashing[6] and Minimal Perfect Hashing in order to minimize the amount of memory used by the k-mers in the sequence.
DNA Sequence Assembly basically implies merging aligned short fragments known as reads into a longer reconstructed DNA Sequence. This assembly is necessary as reading the whole genome by modern DNA Sequencing Technology is not possible. Despite having such an advancement in the modern era, the genome assembly is a cumbersome process because of the 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 improve the space efficiency. The main motivation for implementing this project of Dynamic De Bruijn graphs is trying to construct the whole genome sequencing out of the De Bruijn Graph. Challenges that we might encounter are stated in the section 4.
Proposed Work We aim at implementing and then carrying out an evaluation of the data structure used in the research carried out by Belazzougui et al. (2016). We will be using the mentioned data structures in section 3 to represent and store dynamic De Bruijn graphs.

1.1 Related Work

There has already been a lot of research on De Bruijn graphs and ways to implement them with efficient memory usage and also avoiding any false positives at the same time. J.T. Simpson *et al.* (2012) was the first one to reduce the space utilization of De Bruijn graphs using distributed hash table[5]. This method, that he devised, 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 could bring down the space requirement to 32GB[8]. 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 “Succint De Bruijn Graphs”[3]. In their paper, they were able to give a De Bruijn graph representation that took 2.5GB of space for storing a short read of the human genome. They took their inspiration from the Burrows Wheeler Transform to carry out their implementation. 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 take care of false positives encountered. They could achieve a De Novo assembly of short reads for a human genome consuming 5.7 GB of memory and completed the whole process in 23 hours. In the paper “Efficient De Bruijn Graph Construction For Genome Assembly Using a Hash Table and Auxiliary Vector Data Structures”[7] (2014) Mahfuzer Rahman Limon, Ratul Sharker, Sajib Biswas and M. Sohel Rahman carried out a similar approach to implement De Bruijn graph with the use of hash tables for runtime and memory improvement. They worked to improvise on the time that is taken in the formation of the De Bruijn graph from genome sequence considering all the nodes and edges. Fabiano C. Botelho and Nivio Ziviani gave and explained an external perfect hashing scheme for very large data sets. They used an external memory based algorithm to construct perfect hash functions and minimal perfect hash functions which they used for hashing the large key sets[2]. Their method to deal with large data sets has been one of our motivations to take up minimal perfect hash functions to implement De Bruijn graphs.

2 Description and Implementation of Data Structure

In this section we describe the data structure that we will use for implementation of De Bruijn graphs. First, we explain about the description of IN and OUT Matrices that are used for storing the edge information. Next, we will go about detailing the Minimal Perfect/Rabin Karp Hashing algorithms. Lastly, we describe in detail, the design of the project by explaining the pseudo code.

2.1 IN and OUT Matrices(Matrix decomposition):

These matrices are the representation of the De Bruijn graph. Call G as a graph with set of nodes N=v_0,v_1,...,v_n. Then the structure of this graph is maintained in binary matrices IN and OUT of size $n * S$, where S is the cardinal number of distinct bases in sequence and σ is size of Σ space. To check whether an edge exists between two k-mers, we check out 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$c_1c_2..c_k - 1$) to V_y(with sequence $c_1c_2..c_k - 1$b), the corresponding OUT matrix for V_x marks 1 in its index(hash(V_x)) while the corresponding IN matrix for V_y sets to 1 in its index as shown in Fig 1. 1 denotes whether edge is present while 0 signifies that edge is absent.

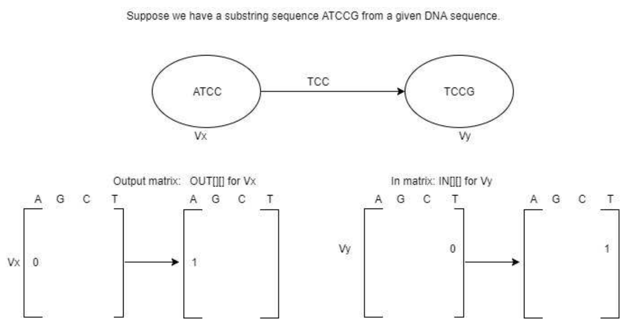
Suppose we have a substring sequence ATCCG from a given DNA sequence.
ATCC
TCC
TCCG
Vx
Vy
Output matrix: OUT[][] for Vx
In matrix: IN[][] for Vy
A G C T
A G C T
A G C T
A G C T
Vx
0
1
Vy
0
1

Fig. 1. fig:The IN and OUT matrices for the outgoing and incoming edges from V_x to V_y

2.2 Hashing algorithm:

Here we plan to generate a minimal perfect hashing function by a method proposed by Botelho and Ziviani (2007) to generate a hash function for a large number of keys[1]. If there are 100 billion keys in a data set, it can be split into a billion buckets with about 100 keys each. A hash function can then be created that maps the n keys uniquely from 0 to n-1 where n is 100.

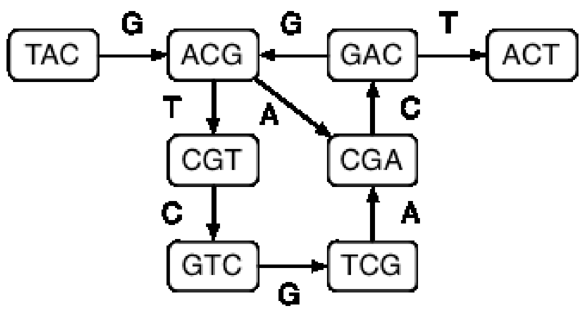

Fig. 2. De Bruijn graph showing common nodes and how edges are embed into graph

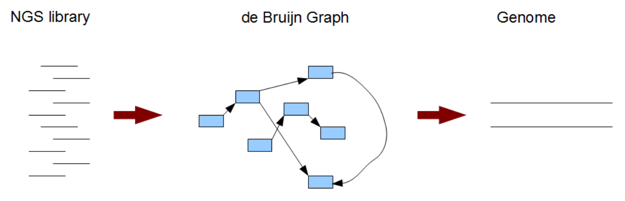

Fig. 3. Formation of De Bruijn graphs from short reads and formation of the full genome sequence from the graph

Also, an offset table is built which has one value for each bucket, where the value is the sum of the number of keys in all previous buckets. So, b = h(key), hashValue = offset[b] + perfectHashFunction[b](key) is a minimal perfect hash for the 100 billion keys. All the billion buckets can construct their perfect hash functions in parallel. This ensures that all the 100 billion keys have a unique mapping such that there are no collisions. The main advantage of this method is that it scales way better but may not be as fast. In order to get fast results Rabin-Karp[6] algorithm can be used to search for given contiguous pattern sequence in a continuous read. 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 substring of the sequence read of the same length as the pattern where we get next successive substring in $O(1)$ time. The calculated hash function should have the following property:
Hash for next shift should be easily computable from the current hash value and next character in sequence read. Mathematically this can be represented as:
hash(str[start+1…start+m]) = hash(hash(str[start…start+m-1]), str[start+m])

Algorithm 1 Rabin Karp algorithm

```
1:  procedure CalculateHash
2:      $H_s \leftarrow$ calculate hash(pattern sequence)
3:      $H_s \leftarrow$ calculate hash(substring of str(say s) with length=pattern.length)
4:      $l_s \leftarrow$ substring length orlength of s
5:      d=256 $\leftarrow$ (Number of characters in input alphabet)
6:  For i=1 to str.length()-$l_s$:
7:      if $H_p = H_s$ then
8:          Match from s[i to i+$l_s$] with pattern
9:      else if
10:         then$H_s \leftarrow (d * (H_s - (s[i+1] * (d^{l_s\text{-}1})) + (s[l_s + i + 1]) \bmod prime$
    ▷ where prime is a prime number
```

2.3 Pseudo Code

2.3.1 In Out Flow

Input is a FastQ file which contains a list of sequence reads. Each read is passed sequentially into the the DBGraph Construction which constructs a De Bruijn graph dynamically adding new vertices and edges into the graph DB_g. For the first read, we construct the De Bruijn graph using the process followed in the construction of static De Bruijn graph [1]. For all the reads following the first one, we generate all the $k-mer$s and add the vertex in to the DB_g if not present and add the edge to the already constructed DB_g. Our main focus in this paper is to add a dynamic edge into the constructed DB_g. For this purpose we add the vertex in to 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 errors (as discussed in section 4) are the future proceedings of this work. Assumptions that we took for the project are that For this project, we take an assumption that we know 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 2 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 $N\ X\ \sigma$ ($\sigma$ being the size of the $\Sigma$).
       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
```

2.3.2 Adding Dynamic Edges

Adding Dynamic Edges into the De Bruijn graph[1] is the main purpose of this project. Given a Graph G and 2 K-mers, we compute the hash of U and V for finding the indices of the IN and OUT matrices as we discussed in the section 2.1. Initialize all 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. Next, we check if both the K-mers are in different components in the G in $O(k\log_2\sigma)$ time Using Depth First Traversal. As stated in [8] if the Sz_U and Sz_V components are both < $desiredsz$ then $Merge$ both the components. If only one of the U or V component's size is lesser than $desiredsz$ then we check for height of the bigger component. If the height of the bigger component is greater than the $midht$, then we need to break the bigger component into two components as its height might cross $maxht$ as described in the [8]. We will discuss Merge in more detail in the pseudo code below. If the height of the bigger Component is < $midht$ then we might need to just $Merge$ with no break needed in the bigger component as addition of new component of height < $desiredsz$ doesn't affect any violation to our conditions. We repeat this process till all the Edges are added into the G.

2.3.3 Merging 2 Connected Components

Given a graph G and 2 K-mers U_i and V_i (Indices of the K-mers), 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, it changes

Algorithm 3 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:     $diff = DFS$(G,i,j)       ▷ Check whether both $U$ and $V$ are in different connected components in the graph $G$. This takes O(k $\log_2\sigma$
11:     if $diff$ then                                    ▷ If different components
12:         $Ht_U$, $Sz_U \leftarrow HeightSize(U)$
13:         $Ht_V$, $Sz_V \leftarrow HeightSize(V)$
14:         if ($Sz_U$ < $desiredsz$) $\wedge$ $Sz_V$ < $desiredsz$ then   ▷ if both of the $Sz_U$ and $Sz_V$ is < $desiredsz$.
15:             $Merge$(G,U,V, $withoutBreak$)
16:         else if !($Sz_U$ > $desiredsz$) $\wedge$ $Sz_V$ > $desiredsz$ then   ▷ if any one of the $Sz_U$ and $Sz_V$ is < $desiredsz$.
17:             Let's Assume that U be the bigger component in height
18:             if $Ht_U$ < $midht$ then      ▷ No need to break the bigger one
19:                 $Merge$(G,V,U, $withoutBreak$)
20:             else if $Ht_U$ > $midht$ then   ▷ break the bigger one as ht might be > $maxht$ after connecting
21:                 $Merge$(G,V,U, $withBreak$)
```

the direction of all the parent pointers in U's component to point to the vertex U. 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[8]. We traverse the $desiredsz$ height in bigger component and break all the parent points at the level of height $desiredsz$-1. We choose one of the boundary as the new root of the U component. We sample the nodes in the broken component and change the directions of parent pointers for finding the new root of the broken component.

3 Experimental Evaluation

For evaluation phase of the project, we will be using a system with specifications: 16 vCPUs, 2.4 GHz, Intel Xeon E5-2676v3, 64 GB memory. Running the code and implementing the data structure described in previous sections on a powerful machine is crucial because of the data sets being very large.

3.1 Evaluation

Our evaluation will be based on parameters such as the time taken for building the De Bruijn graph and the memory requirements by the graph. These metrics will later be compared with the space and time requirements when using different data structures. We will not be implementing the graphs using a different data structure other than mentioned in the project. We will be using the results from previous research work to make comparisons and come to conclusions. We will be testing the data structure that we will use to implement the graph, against the bloom filter implementation and the BWT implementation of De Bruijn graphs.

Algorithm 4 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.
```

We will be following the proposed time line to achieve the completion of the project. The Gantt Chart represent our course of action for the upcoming months.

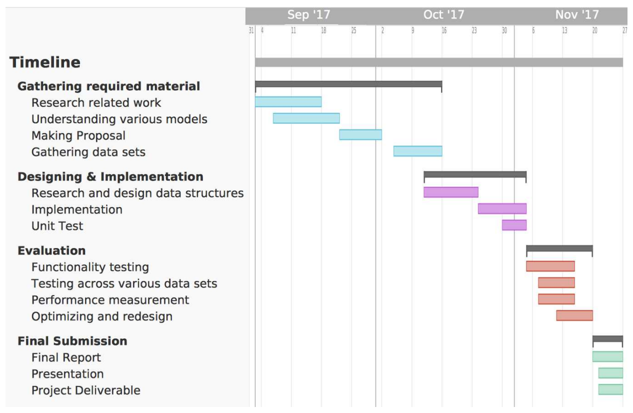

Fig. 4. Proposed Time line

4 Challenges

So far, We have the following challenges lined up for our proposed solution:
1.Since we have a dynamic De Bruijn graph, there is a possibility of getting a cycle on the addition of a new edge. This will in turn, cause collisions while reading sequence and thus, there would be multiple permutations of reading the genome sequence since it is directed graph. Since there would be an uncertainty as when to come out of that cycle, this would also result in erroneous reads.
2.There is a possibility of getting two parents while we are breaking the graph into two disjoint components or sets after traversing k($\log_2 \sigma$)from the parent as described in the algorithm. This would interfere during the merge and choosing the parent would determine the balancing of the heights.
3. Construction of whole Genome from this compact Data structure might lead to a False Positive.
4. Changing the direction of the parent pointers might affect the consistency of Edge mappings.

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]Conway, T. *et al.* (2012). Gossamer—a resource-efficient de novo assembler. *Bioinformatics*, **28**(14), 1937–1938.
[6]Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, **31**(2), 249–260.
[7]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.
[8]Simpson, J. T. *et al.* (2009). Abyss: a parallel assembler for short read sequence data. *Genome research*, **19**(6), 1117–1123.