De Bruijn Graph Construction For Genome Assembly
HaVec: An Efficient Algorithmic Approach

Amirtha Narayana K
S3 CSE AIE A Batch
AM.EN.U4AIE21012

Nandana B
S3 CSE AIE A Batch
AM.EN.U4AIE21021

Santhosh Mamidisetti
S3 CSE AIE A Batch
AM.EN.U4AIE21042

Akash Ranjan
S3 CSE AIE A Batch
AM.EN.U4AIE21070

Anupa Sajikumar
S3 CSE AIE A Batch
AM.EN.U4AIE21071

Abstract — Today's next-generation sequencing technologies can produce massive amounts of data. The so-called de Bruijn graph is a popular and useful tool for analysing these massive amounts of data. The main barrier in de Bruijn graph-based genome assembly is memory and runtime due to the large number of nodes. Furthermore, this area has received considerable attention in contemporary literature.We present an algorithm that makes a balance between memory and runtime. Our approach stores the de Bruijn graph in a hash table with an auxiliary data structure which improves the total memory usage and runtime with no false positives. In the whole assembly process, generally the graph construction procedure takes the major share of the time.

1 INTRODUCTION

Modern next-generation sequencing technologies can generate massive amounts of reads from species-specific DNA samples. The number of reads could be in the hundreds of millions, and the total volume of data could be in the tens or even hundreds of gigabytes. The de Bruijn graph is a popular and useful representation for efficient data processing, particularly for genome assembly (Compeau et al., 2011).. In a de Bruijn graph, the nodes represent distinct k-mers that occur in the reads, and two k-mers are linked by an arc if there is a suffix-prefix overlap of size $k-1$ between them. Because of the huge number of nodes in a de Bruijn graph, it is important to represent it as compactly as possible.The Bloom filter is a space efficient hash-based data structure that allows us to test whether an element is in a set. It consists of a bit array of m bits, initialized to zero. We also have h hash functions. To insert or test the membership of an element, h hash values are computed, which produce h array positions. To insert, we set all corresponding bits to 1. The membership operation returns yes if and only if each of the bits at these positions is 1. A negative answer means that the element is definitely not in the set. A positive answer indicates that the element may or may not be in the set. This type of error is known as the 'false positive error'. Recent research on de Bruijn graphs has been mostly focused on designing more lightweight data structures. . Conway and Bromage (2011) applied sparse bit array structures to store an implicit, immutable graph representation. These methods compute local assemblies around sequences of interest, using negligible memory, with greedy extensions of portions of the de Bruijn graph. Ye et al. (2012) showed that a graph roughly equivalent to the de Bruijn graph can be obtained by storing only one out of g nodes ($10 \leq g \leq 25$).

The probabilistic de Bruijn graph, defined by Pell et al. (2012), is a de Bruijn graph stored as a Bloom filter. They demonstrated that the graph can be encoded with as few as four bits per node. The Bloom filter introduces false nodes and false branching, which is an unfortunate drawback of this representation. They have discovered, however, that the global structure of the graph is roughly preserved, up to a certain false positive rate. Pell et al. did not directly perform the assembly by traversing the probabilistic graph. They have instead used the graph to divide the set of reads into smaller sets, which are then assembled in turn using a traditional assembler.

Recently, Chikhi and Rizk (2013) have proposed a new encoding of the de Bruijn graph based on a Bloom filter, with an additional structure to remove critical false positives. Notably, their approach is very much dependent on the hard disk free space. When the number of unique k-mers in a file is much higher (e.g., 2 × 109) performance falls rapidly as critical false positive calculation takes more than 10 hours. In this approach, RAM is used, but in parallel to that the total free hard disk space is used over and over again. For this reason, the runtime becomes very high. Another limitation is that it can not use even numbers for k, i.e., the k-mers need be of odd length.

In the present state of the art, the main problem with the memory efficient Bloom filter representation of the de Bruijn graph is the false positive calculation and the runtime. Other traditional approaches, on the other hand, face problems with higher memory consumption.

In this paper, we make an effort to alleviate these problems. In particular, we present a new algorithm based on hashing and auxiliary vector data structures and call this algorithm HaVec. The key features of Havec are as follows, which can be seen as the main contributions in this paper:

1) Havec is a new error free graph construction approach that runs in a very short time and uses sufficiently low memory.

2) It introduces the idea of using hash table with an auxiliary vector data structure to store k-mers and corresponding neighborhood information.

3) It constructs such a graph representation from which no false positive can be generated and only true neighbours can be found for traversing the whole graph.

2 METHODOLOGY

2.1 Overview

The de Bruijn graph must first be constructed in de Bruijn graph-based genome assembly. The assembly operation is then carried out using this de Bruijn graph. We must build it and store it in memory for later use. Traditional graph representation approaches will require massive amounts of memory if the graph has millions of nodes and edges. A Bloom filter is a less memory-intensive option that stores a graph by using a present bit for each node rather than explicitly storing any edges. A hash value is generated from a graph node. An index in the table is calculated using this hash value and the table size.

Generally, the hash value is divided by table size and the remainder is taken as the index of the hash table. And if a node is present in the graph that particular bit is set. Later, we can check whether a node is present or absent by checking this bit value. If the bit is set then the node is possibly present and absent otherwise. From that particular node, all possible neighbours are generated and subsequently, it is checked whether they are present or not in the Bloom filter. And if a particular neighbour of that node is present in the Bloom filter, an edge may exist between these two nodes. Now, the problem is that if the neighbour node is a falsely generated one, i.e., the corresponding bit is set by another node, then a false edge will be created in the graph. Another node may have set this bit because it’s hash value may have produced the same remainder when divided by the table size. Since, the number of unique hash values is much greater than the table size, more than one hash value will point to the same index due to the same remainder. This is the reason for getting false positives in a Bloom filter.

Our approach capitalizes on this different quotient values when we get the same remainder. So, if a node points to an index in the hash table, we can verify whether it is indeed a real node or a falsely generated one by examining its quotient value. However, to achieve this, we have to store a mapping between hash values and quotients for all indices in the table.

For each index, if there are two or more competing nodes, we only keep the presence information of the first one at the hash table. The latter ones are kept in an auxiliary vector data structure. We make use of h different hash functions. Each hash function produces a different hash value for a single node. Using these different values we can point to different indices at the hash table. At first, we generate the hash value of a node using the first hash function and try to store the node into the corresponding index. If however, that index is already occupied, we switch to the next hash function. This procedure is repeated for a maximum of h (number of hash functions) times. If we still fail to find a free index (i.e., even after h attempts), we then move to our auxiliary vector data structure to store the node’s information. Using the index value resulting from the h-th hash function, we find a particular position in the vector data structure. Since, multiple index values can point to the same position at the auxiliary vector data structure, we have to maintain a list of indices in that position. For a particular index of that list, we maintain another vector structure. All the collided nodes on that index after using the h-th hash function are stored.

2.2 de-Bruijn Graphs using Hash Tables and Auxiliary Vector Structures

We don't need to explicitly save the graph structure in our approach. The de Bruijn graph can be built using the k-mer information stored in the hash table and
The auxiliary vector data structures. We can generate the correct neighbouring kmers by inspecting each k-neighbor mer's bits. The same process can be used to generate all of their neighbouring k-mers from these newly created k-mers. The entire procedure will be covered in the sections that follow. We specifically discuss the hash table structure first, followed by a description of the auxiliary vector data structures.
Notably, in our approach, both the hash table and the auxiliary vector data structures reside in main memory (RAM), allowing for fast computation.
However, due to three levels of indirection, the latter's update and access times are much slower than the former's.

2.2.1 Hash Table Data Structure

Each k-mer can have at most four neighbouring k-mers. We get a neighbour by removing the first symbol of it and then appending one of the four nucleotides at the end. Now one k-mer can either have no neighbours at all or only one neighbour or only two or only three or, finally, all four neighbours. In other words, we have a total of 16 possibilities. Hence, we need 4 bits for this purpose. Each of the 4 bits is dedicated for one of the 4 nucleotides

.

2) Next 3 bits of information gives us which hash function is used to produce the hash value. Clearly a maximum of 8 hash functions can be used in this setting.

3) The remaining 33 bits is used to store the quotient we get after dividing the hash value by the table size.

2.2.2 Auxiliary Vectors

In our approach, we used three levels of vectors (Figure 1). The first level vector's entries each point to a list of one or more hash table indices. Each entry in the second level specifies a hash table index map to a reference to all collided k-mer information for that specific hash table index. The third level vector is a vector containing all of the collided k-mers information for a specific hash table index, as indicated by a second level vector entry.

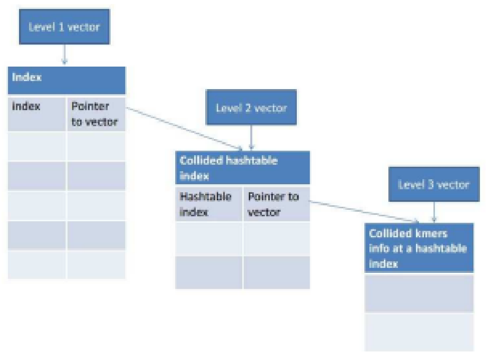


Fig. 1. Three levels of vectors are used in our approach.

2.2.3 Size of the Quotient Value

To represent the hash value of a k-mer, we need 2K bits. Now, recall that we get the quotient by dividing the hash value by the table size. So, as long as the hash table size (total hash table index) is at least 2^ |2k-33|, we can store any quotient value in 33 bits.

To illustrate the relation of the quotient size with the memory requirement, we discuss another example. Suppose we want to store quotient value in 20 bits. Then for k = 32, the minimum hash table size will be 2^ 64−20 or 2^44, which will increase the total memory requirement.
As the number of bits used to save quotient value is reduced by one, the minimum table size is doubled. Also, allocating more bits for quotient would result in fewer entries in the hash table, which in turn would require higher use of the auxiliary vector structures, resulting in increased running time. As it turns out, keeping 33 bits for the quotient value makes the memory requirement and the running time at an acceptable level, ensuring that we can handle up to 32 as the value of k.

2.2.4 Hash Function Considerations

the hash value indices will never collide if the hash table consists of 2^64 entries. But it is not practical to have a hash table of that size. So we consider a much smaller hash table and then use multiple hash functions in order to reduce the probability of collision, filling as much space in the hash table as possible. So, it is mandatory to keep track of which hash function has been used for which k-mer. Now, it seems that if we increase the number of hash functions, we can populate the hash table more efficiently. But there is a cost for storing the index of the hash function used for a particular k-mer. Clearly, if we allocate n bits for storing the hash function's index then 2^n number of hash functions can be used. In our implementation using 5 bytes, we allocate 33 bits to store the quotient and 4 bits for the next nucleotide(s). So, we can allocate the remaining 3 bits for the index value of the hash function. So we can use upto 2^3 = 8 different hash functions.

2.3 Procedure

The process begins by reading a file in FASTA or FASTQ format, which contains a sequence of nucleotides. From this sequence, the process generates k-mers, which are substrings of the original sequence of a fixed length (in this case, k = 5). These k-mers are then used to generate hash values, which are used to index them in a hash table. In this example, the hash table has a size of 11 and two hash functions, hash1 and hash2, are used.

Figure 2 depicts the corresponding de Bruijn graph.

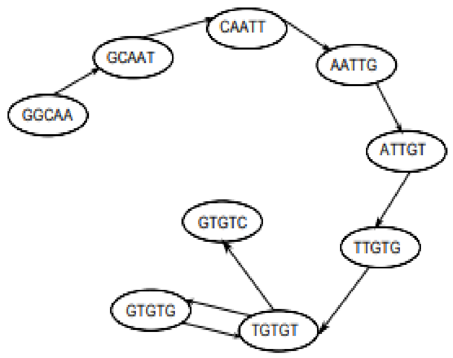


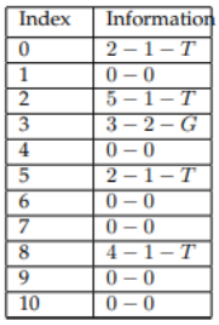
Fig. 2. The de Bruijn graph for the k-mers. Nodes are {GGCAA, GCAAT, CAATT, AATTG, ATTGT, TTGTG, TGTGT, GTGTG, TGTGT, GTGTC}

The k-mers are placed in the hash table. If the index calculated from a k-mer's hash value is already occupied, the process uses the second hash function to calculate a new index. If that index is also occupied, the process resorts to using an auxiliary vector data structure, which is used to handle collisions in the hash table. The process for handling collisions involves taking the hash table index and dividing it by the size of the first level vector, and taking the remainder. This remainder is used as the index in the first level vector, and a new entry is added to that index for the hash table index that generated the collision.

TABLE 1
Reads are Broken into k-mers. All k-mer's hash value are shown here

k-mers	Hash Values	Comments
GGCAA	57	
GCAAT	27	
CAATT	24, 36	
AATTG	52	
ATTGT	36, 27	Put in Vector
TTGTG	22	
TGTGT	34, 30	Put in Vector
GTGTG	49, 47	Put in Vector
TGTGT	34, 30	Found in Vector; Update
GTGTC	38, 25	Put in Vector

TABLE 2
Hash table information. We have a total of 11 indices. At each index, the corresponding k-mer's quotient, hash function and it's neighbour information is saved.



Index	Information
0	$2-1-T$
1	$0-0$
2	$5-1-T$
3	$3-2-G$
4	$0-0$
5	$2-1-T$
6	$0-0$
7	$0-0$
8	$4-1-T$
9	$0-0$
10	$0-0$

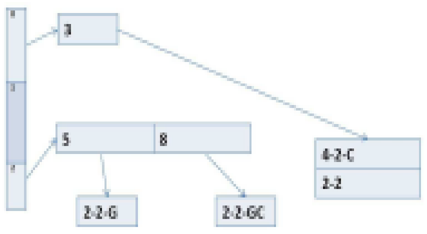


Fig. 3. Auxiliary Vector Data Structures. All collided k-mers' information of hash table index 3 can be found at vector index 0 and all collided k-mers' information of hash table index 5 and 8 can be found at vector index 2.

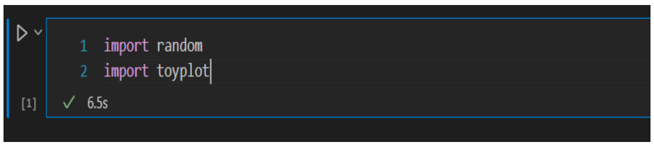
The process updates the neighbour information of each k-mer. Each k-mer is stored in the hash table or auxiliary vector data structure with information on which hash function was used, its quotient, and its neighbour information (the next k-mer in the sequence). The passage also notes that when a k-mer is encountered again, its neighbour information is simply updated. The process continues until all k-mers have been processed, and the final result is a de Bruijn graph that represents the overlap between the k-mers in the original sequence.

2.4 Algorithm

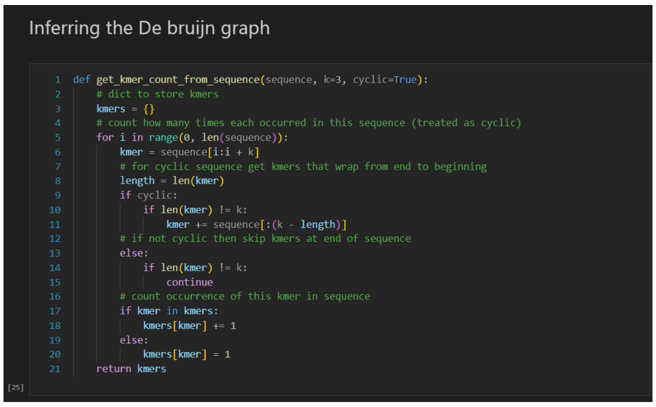
The generalised algorithm is as follows

```
for $whichHashFunc \leftarrow 0$ to LastHashFunc do
    $hashedKhmer \leftarrow$ Hasher(whichHashFunc, rawKmerString)
    $index \leftarrow hashedKhmer\%blockSize$
    $quotient \leftarrow khmerhhashedKhmer \div blockSize$
    if no neighbor is found in memBlock[index] then
        put the quotient into memBlock[index]
        put whichHashFunc into memBlock[index]
        put nextNeucleotide into memBlock[index]
    else if neighbor(s) found in memBlock[index] and hashvalue matched then
        put nextNeucleotide into memBlock[index]
    else if $whichHashFunc = LastHashFunc$ and neighbor(s) found in $memBlock[index]$ and hashvalue does not match then
        $firstLevelVectorIndex \leftarrow index\%mapPointer5Byte.size$
        create tempVect with $tempVect.indexVal \leftarrow index$
        add tempVect to mapPointer5Byte[firstLevelVectorIndex]
        create a tempkmerInfo and put nextNeucleotide in it
        $isFound \leftarrow false$
        if mapPointer5Byte[firstLevelVectorIndex][secondLevelVectorIndex].vect has already this kmer then
            update nextNeucleotide
            $isFound \leftarrow true$
        end if
        if isFound = false then
            put quotient, whichHashFunc in tempkmerInfo
            add the newly updated tempkmerInfo to mapPointer5Byte[firstLevelVectorIndex][secondLevelVectorIndex].vect
        end if
    else
        continue
    end if
```

Code:

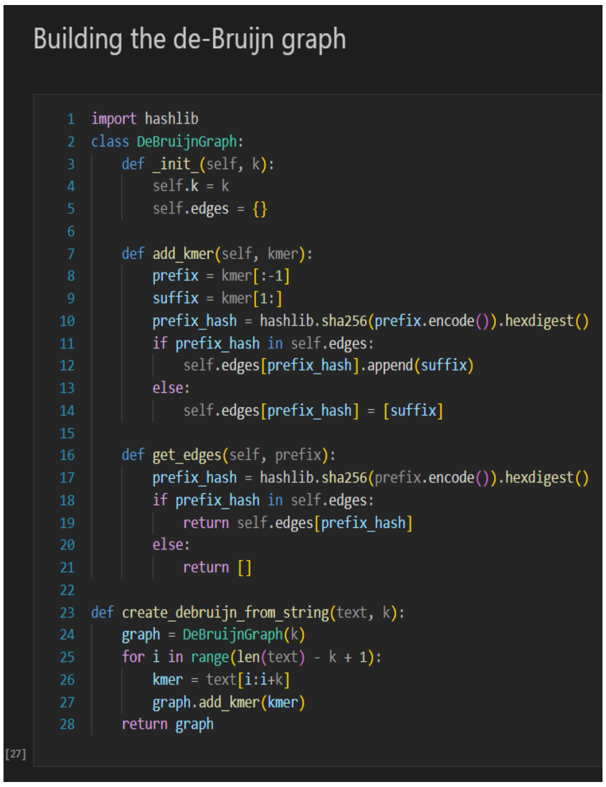


```python
import random
import toyplot
```



Inferring the De bruijn graph

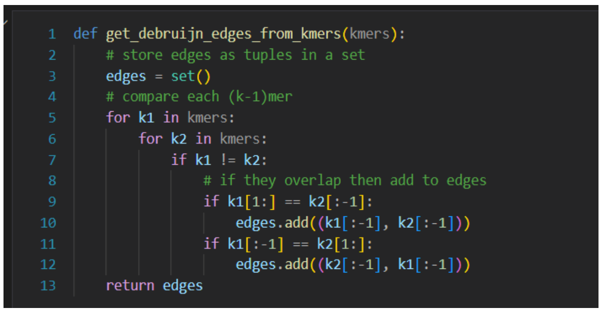
```python
def get_kmer_count_from_sequence(sequence, k=3, cyclic=True):
    # dict to store kmers
    kmers = {}
    # count how many times each occurred in this sequence (treated as cyclic)
    for i in range(0, len(sequence)):
        kmer = sequence[i:i + k]
        # for cyclic sequence get kmers that wrap from end to beginning
        length = len(kmer)
        if cyclic:
            if len(kmer) != k:
                kmer += sequence[:(k - length)]
        # if not cyclic then skip kmers at end of sequence
        else:
            if len(kmer) != k:
                continue
        # count occurrence of this kmer in sequence
        if kmer in kmers:
            kmers[kmer] += 1
        else:
            kmers[kmer] = 1
    return kmers
```



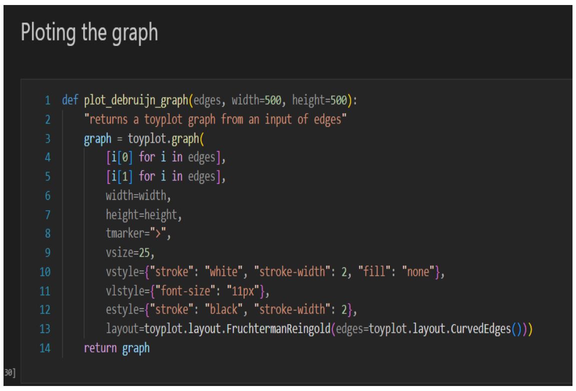
Building the de-Bruijn graph

```python
import hashlib
class DeBruijnGraph:
    def _init_(self, k):
        self.k = k
        self.edges = {}

    def add_kmer(self, kmer):
        prefix = kmer[:-1]
        suffix = kmer[1:]
        prefix_hash = hashlib.sha256(prefix.encode()).hexdigest()
        if prefix_hash in self.edges:
            self.edges[prefix_hash].append(suffix)
        else:
            self.edges[prefix_hash] = [suffix]

    def get_edges(self, prefix):
        prefix_hash = hashlib.sha256(prefix.encode()).hexdigest()
        if prefix_hash in self.edges:
            return self.edges[prefix_hash]
        else:
            return []

def create_debruijn_from_string(text, k):
    graph = DeBruijnGraph(k)
    for i in range(len(text) - k + 1):
        kmer = text[i:i+k]
        graph.add_kmer(kmer)
    return graph
```

[27]



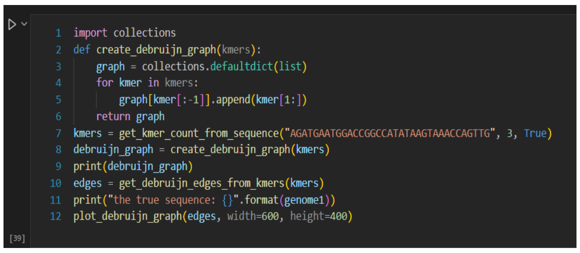
```python
def get_debruijn_edges_from_kmers(kmers):
    # store edges as tuples in a set
    edges = set()
    # compare each (k-1)mer
    for k1 in kmers:
        for k2 in kmers:
            if k1 != k2:
                # if they overlap then add to edges
                if k1[1:] == k2[:-1]:
                    edges.add((k1[:-1], k2[:-1]))
                if k1[:-1] == k2[1:]:
                    edges.add((k2[:-1], k1[:-1]))
    return edges
```



Ploting the graph

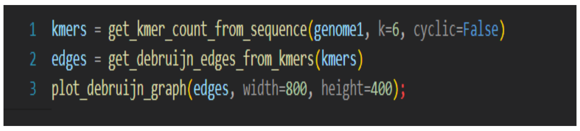
```python
def plot_debruijn_graph(edges, width=500, height=500):
    "returns a toyplot graph from an input of edges"
    graph = toyplot.graph(
        [i[0] for i in edges],
        [i[1] for i in edges],
        width=width,
        height=height,
        tmarker=">",
        vsize=25,
        vstyle={"stroke": "white", "stroke-width": 2, "fill": "none"},
        vlstyle={"font-size": "11px"},
        estyle={"stroke": "black", "stroke-width": 2},
        layout=toyplot.layout.FruchtermanReingold(edges=toyplot.layout.CurvedEdges()))
    return graph
```

30]



```python
import collections
def create_debruijn_graph(kmers):
    graph = collections.defaultdict(list)
    for kmer in kmers:
        graph[kmer[:-1]].append(kmer[1:])
    return graph
kmers = get_kmer_count_from_sequence("AGATGAATGGACCGGCCATATAAGTAAACCAGTTG", 3, True)
debruijn_graph = create_debruijn_graph(kmers)
print(debruijn_graph)
edges = get_debruijn_edges_from_kmers(kmers)
print("the true sequence: {}".format(genome1))
plot_debruijn_graph(edges, width=600, height=400)
```

[39]



```python
kmers = get_kmer_count_from_sequence(genome1, k=6, cyclic=False)
edges = get_debruijn_edges_from_kmers(kmers)
plot_debruijn_graph(edges, width=800, height=400);
```

2.5 Cut-off Value and a 6 Bytes Structure

In genome assembly, the cutoff value is generally defined as the threshold by which we determine the validity of a k-mer. If a k-mer occurs less frequently than the preset cutoff value, the genome assemblers disregard it.

The state of the art has advanced to a level where the sequencing laboratories can produce reads with very high quality and accuracy, the significance of cutoff values is much less than it was few years ago. This is why in our main design we have ignored it. However, we can still keep provisions for cutoff values as follows. To support cutoff values, we need to use additional one byte in the k-mer information structure. This additional bye is used to store the count of the k-mer. In this way, we can support cutoff values between 1 to 255. While processing the input file, we update the neighbour information. At the same time, its count information is also updated. So, when we get a kmer's information, we have both it's neighbour information and its count value. During the assembly process, comparing the count with the cutoff, we can decide whether we should take the k-mer under consideration further or not. In our implementation, cutoff calculation is parameterized. If no cutoff value is selected, then we need only 5 bytes to store the k-mer information resulting in lower memory consumption and lower running time. In what follows, we will refer to these two different implementations as 5 Bytes and 6 bytes implementations.

3 EXPERIMENTAL RESULTS

For our experiment we used a simple dataset about *E.coli* downloaded from NCIB. BamTools was used to convert these files to FASTA format. This tool can be downloaded from here https://github.com/pezmaster31/bamtools

Currently, we have used a demonstration on small-scale sequences, which is a generalized algorithm used for this approach in Python.
https://github.com/Anupa-Sajikumar/De-Bruijn-Graphs-using-Ha-Vec-Approach/blob/main/Creation_de_bruijn.ipynb

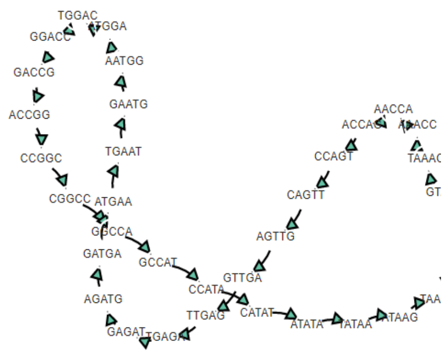
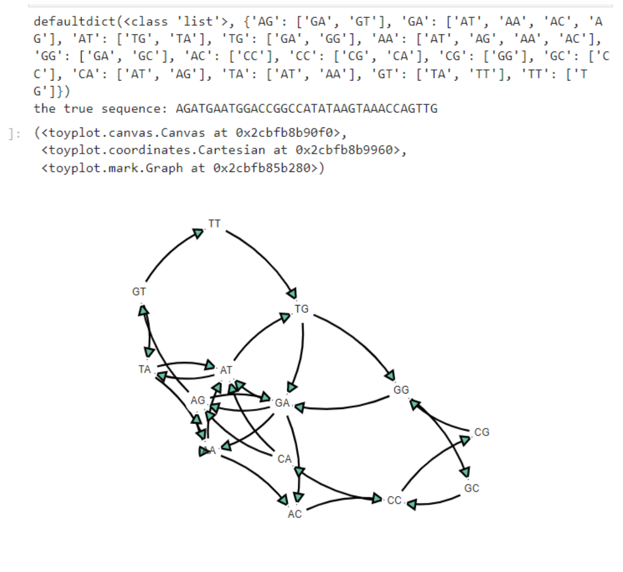


Fig.4. This the graph obtained from the generalised HaVec approach



```
defaultdict(<class 'list'>, {'AG': ['GA', 'GT'], 'GA': ['AT', 'AA', 'AC', 'A
G'], 'AT': ['TG', 'TA'], 'TG': ['GA', 'GG'], 'AA': ['AT', 'AG', 'AA', 'AC'],
'GG': ['GA', 'GC'], 'AC': ['CC'], 'CC': ['CG', 'CA'], 'CG': ['GG'], 'GC': ['C
C'], 'CA': ['AT', 'AG'], 'TA': ['AT', 'AA'], 'GT': ['TA', 'TT'], 'TT': ['T
G']})
the true sequence: AGATGAATGGACCGGCCATATAAGTAAACCAGTTG
```

```
]: (<toyplot.canvas.Canvas at 0x2cbfb8b90f0>,
    <toyplot.coordinates.Cartesian at 0x2cbfb8b9960>,
    <toyplot.mark.Graph at 0x2cbfb85b280>)
```

Fig. 5 shows the output of the haploid method with the HaVec approach..

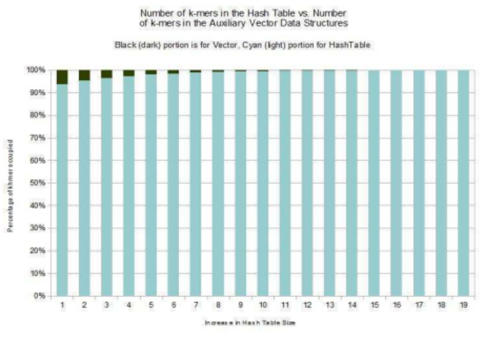


Fig. 6. The relation between the number of k-mers in hash table and the number of k-mers in the vector structure on increasing hash table size. A total of 19 cases are reported in the x-axis, where hash table size of case i+1 is 5% higher than case i (1 ≤ i ≤ 18). As the hash table size increases, the number of k-mers in the hash table increases and the number of k-mers in the vector structure decreases. The same relation with the exact same values holds for both of the 5 bytes and 6 bytes implementations.

In Figure 6, we have illustrated the relation between the number of k-mers in the hash table and the number of kmers in the vector data structure. Here we have reported a total of 19 cases where the hash table size of case i+1 is made 5% higher than case i (1 ≤ i ≤ 18). Here, we can see that as the hash table size increases, the number of k-mers in the hash table increases and the number of k-mers in the vector structure decreases which is certainly desirable. The same relation with the exact same values holds for both 5 bytes and 6 bytes implementations.

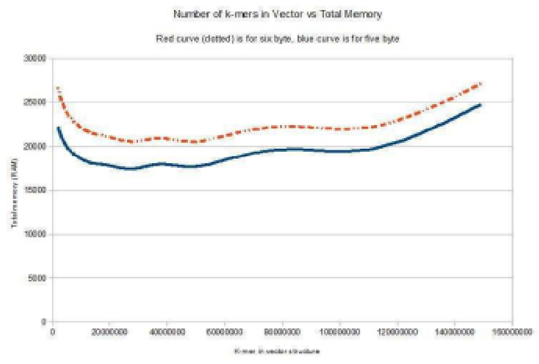


Fig. 7. The graph shows the relation between the number of k-mers in the vector and the total memory. As the number of k-mers in the vector structure increases, the total memory also increases.
In Figure 7, the graph shows the relation between the number of k-mers in the vector and the total memory. We can see that as the number of k-mers in the vector structure increases, the total memory also increases. The difference in memory requirements

between the curves for the 5 bytes and 6 bytes implementations always remains constant.

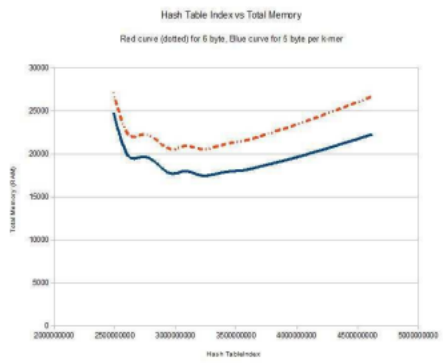


Fig. 8. The relation for hash table index and total memory. Optimum memory use can be achieved when hash table size is 1.25 to 1.5 times the number of unique k-mers**.**

Figure 8 shows the curve for hash table size vs. total memory, from which we can see how total memory changes with the increasing hash table size. As can be seen, larger hash table size doesn't always guarantee lower total memory consumption. Optimum memory use can be achieved when hash table size is 1.25 to 1.5 times the number of unique k-mers.

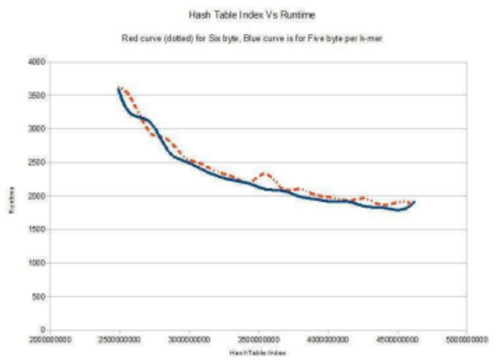


Fig. 9. Relation between the hash table size and the runtime. Generally, runtime for the 6 bytes implementation is slightly higher than that of the 5 bytes implementation.

Finally, the curve in Figure 9 is for hash table size vs. runtime. As can be seen from the figure that the runtime decreases with the increase in the hash table size. Generally, runtime for the 6 bytes implementation is slightly higher than that of the 5 bytes implementation. We have further conducted extensive experiments on all the files listed in Table 3 and the results are reported in Table 4 for k = 27 and k = 32.

6 CONCLUSION

In general, the graph construction procedure consumes the majority of the time during the assembly process. We accomplished this in a very short period of time. Furthermore, our graph construction produces no false positives, making it completely error free. Our approach, we believe, will be extremely useful in de Bruijn graph-based genome assembly. Because de Bruijn approach parallelization has already been attempted in the literature (e.g., Georganas et al. (2014)), an immediate avenue for future research would be to see if we can parallelize our approach by using multi-threading concepts.

REFERENCES

Bankevich, A., Nurk, S., Antipov, D., Gurevich, A. A., Dvorkin, M., Kulikov, A. S., Lesin, V. M., Nikolenko, S. I., Pham, S. K., Prjibelski, A. D., Pyshkin, A., Sirotkin, A., Vyahhi, N., Tesler, G.,

Alekseyev, M. A., and Pevzner, P. A. (2012). Spades: A new genome assembly algorithm and its applications to single-cell sequencing. Journal of Computational Biology , 19(5), 455–477.
EfficientdeBruijngraphconstructionforgenomeassembly
https://eaton-lab.org/slides/genomics/answers/nb-10.2-de-Bruijn.html
Drezen, E., Rizk, G., Chikhi, R., Deltel, C., Lemaitre, C., Peterlongo, P., and Lavenier, D. (2014). Gatb: genome assembly & analysis tool box. Bioinformatics, 30(20), 2959–2961. Georganas, E., Buluc,, A., Chapman, J., Oliker, L., Rokhsar, D., and Yelick, K. (2014). Parallel de bruijn graph construction and traversal for de novo genome assembly. pages 437–448. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., et al. (2010). De novo assembly of human genomes with massively parallel short read sequencing. Genome research , 20(2), 265–272.
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.
Drezen, E., Rizk, G., Chikhi, R., Deltel, C., Lemaitre, C., Peterlongo, P., and Lavenier, D. (2014). Gatb: genome assembly & analysis tool box. Bioinformatics, 30(20), 2959–2961.