De Bruijn Graph Construction in Genome Assembly

Abhijith.A.Thampi
Roll No: AM.EN.U4AIE20102
Computer Science(AI)
Amrita School of Engineering, Amritapuri, Kollam, Kerala

Ajay G Nair
Roll No: AM.EN.U4AIE20108
Computer Science(AI)
Amrita School of Engineering, Amritapuri, Kollam, Kerala

Govind Nandakumar
Roll No: AM.EN.U4AIE20129
Computer Science(AI)
Amrita School of Engineering, Amritapuri, Kollam, Kerala

Vivin
Roll No: AM.EN.U4AIE20173
Computer Science(AI)
Amrita School of Engineering, Amritapuri, Kollam, Kerala

***Abstract*— The rapid advancement of sequencing technologies has made it possible to regularly produce millions of high-quality reads from the DNA samples in the sequencing laboratories. To this end, the de Bruijn graph is a popular data structure in the genome assembly literature for efficient representation and processing of data. The use of next generation sequencing (NGS) technologies, which is due to their reduced cost, has revitalized the algorithmic research focused on de novo assembly, since the huge amount of available data poses new computational challenges. A fundamental tool used for de novo assembly is a graph representation of the relationships between the portions of the genome, called reads, sharing a common prefix and suffix. A commonly used representation is the de Bruijn graph, which is based on the notion of k-mers —a length k substring of a read.**

Keywords—de-Bruijn, Eulerian path, k-mer, Genome Assembly

I. INTRODUCTION

Genome assembly can be described as a computational process of drawing together numerous short sequences called *reads* derived from different portions of the target DNA within the cell of an organism. These reads are generated by sequencing machines through randomly sampling the original sequence. The De Bruijn graph based genome assembly algorithms have been shown effective for assembling a large number of short reads and have been adopted in state-of-the-art assemblers. This is an algorithm-driven automated process. DNA-sequence-assembly programs have utilized sequence overlaps for sequence assembly in the correct order.

A. DE BRUIJN GRAPH

They are directed graphs representing overlaps between sequences of symbols. Vertices/nodes in the graph are k-mers. Edges represent consecutive k-mers (which overlap by k-1 symbols). If we have a set of m symbols $S = \{s_1, s_2 \ldots S_m\}$, then the set of vertices $V = |S|^n$, all possible length-n sequences of the given symbols. If one of the vertices can be expressed as another vertex by shifting all its symbols by one place to the left and adding a new symbol at the end of this vertex, then the latter has a directed edge to the former vertex

B. EULER PATH

An Euler path, in a graph or multigraph, is a walk through the graph which uses every edge exactly once. A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The Eulerian Path problem is Polynomial time. String can be reconstructed by finding an Eulerian path in the de Bruijn graph

II. METHODOLOGY

The solution involved constructing a graph with all possible (k − 1)-mers as nodes. Each k-mer was an edge directed from node A to node B if the (k − 1)-mer in node A is a prefix, and that in node B, a suffix of the k-mer. de Bruijn graphs become less and less tangled when read length increases. As soon as read length exceeds the length of all repeats in a genome, the de Bruijn graph turns into a path.

III. DE BRUIJN GRAPH ASSEMBLERS

The overlap–layout–consensus technique, which involves comparing all pairs of reads to identify overlaps, was formerly an effective assembly method for first-generation sequencing data. However, doing the computations required to combine these reads using first-generation approaches was neither practical nor daunting.

In the 1940s, a Dutch mathematician called Nicolaas de Bruijn became interested in finding the shortest circular string of characters that contains all possible substrings, each of same length, in a given alphabet. He came up with a solution that included creating a network with all feasible (k 1)-mers as nodes. If the (k 1)-mer in node A is a prefix and that in node B is a suffix of the k-mer, each k-mer was an edge going from node A to node B.

The answer to the stated problem now was to find a path through the graph that traverses each edge exactly once, or in other words Eulerian trail**.** Here is an example in which the sequence “ATGCTAGCAC” of the length 10, is assembled from five reads, each of length 6.

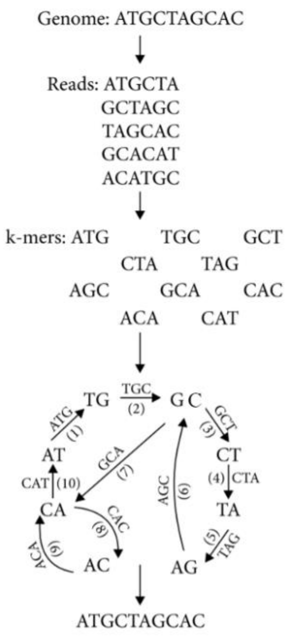


Figure 1

Reads are broken into smaller fragments of a specified size k. In the above example, k corresponds to 3. k-mers are identified and a de Bruijn graph with (k–1)-mers as nodes and k-mers as edges drawn as described in the text. A Eulerian path is traced through this network resulting in the reconstruction of the original genome sequence.

The following steps can be specified in the process of reconstructing the original sequence from the given short reads.

STEPS:

1. Take all (k-1)-mers from the set of k-mers, Eg. ATG, TGC-> AT, TG, GC. We should have gone past the size of k-mer reads.

2. Construct a multi-graph with nodes being k-1-mers; draw an edge between two k-1 mers only if the two k-1 mers are taken from the same read. Eg. GCT & CTA (Figure 2)

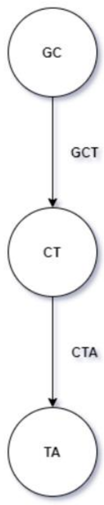


Figure 2

3. Graph constructed this way is guaranteed to have a Eulerian trail, follow the trail and connect the nodes to form our original sequence. The graph similar to this will appear. (Figure 3)

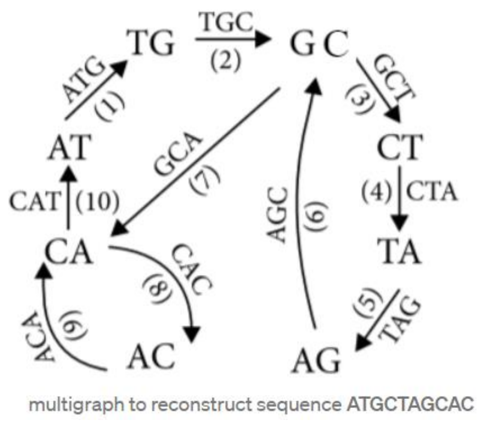


Figure 3

Now this algorithm can be used to assemble k-mer reads. Further, it is convenient to reconstruct the parts that are easier to assemble (contigs) and leave out parts that are ambiguous.

IV. Drawbacks and Challenges

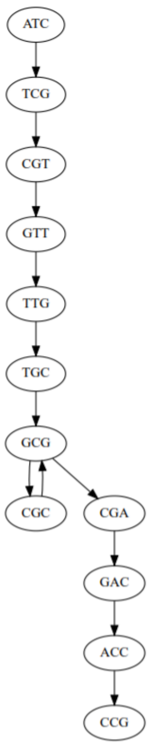
Although de bruijn assembler is a popular means to implement assembling, there still exist some challenges for de bruijn genome assembly. Sequence error, uneven sequencing depth, repetitive sections and computational cost are few of the leading challenges.

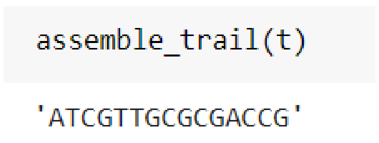
Another major disadvantage of de Bruijn graphs is the fact that valuable context information stored within the reads is lost for assembly, because the k-mers have to be shorter than the actual read length.Loss of positional information is

the biggest drawback of going from sequence space to de Bruijn graph space. Longer the read size, more one has to lose.
If the Sanger read has repetitive structure, de Bruijn graph for the read itself will contain loop. That means we cannot go back from the de Bruijn graph to the read. However, the read itself is a true representation of a part of the genome. By converting it into de Bruijn graph, we lose what is already known about that part of the genome.

V. Results and Discussions

Here we have taken a string "ATCGTTGCGCGACCG" and the kmer value as 4 and have obtained the results. The reads are generated using this read, and from those reads, we have reconstructed the original string using de bruijn graph. The original and the reconstructed strings can then be compared. The de bruijn graph constructed from the reads which were de bruijnised is shown below




```
assemble_trail(t)
```

```
'ATCGTTGCGCGACCG'
```

VI. Conclusion

In this project we were able to understand and implement the concepts of de-bruijn graphs in genome assembly. We were able to reconstruct the genome from the given input string and k-mer composition which when compared to the initial input string is found to be the same, hence concluding with successful implementation.
It is quite true that assembly methods based on de Bruijn graphs begin, somewhat counter-intuitively, by replacing each read with the set of all-overlapping sequences of a shorter, fixed-length, but it is a popular method for genome assembly.

References

[1] E. Drezen, G. Rizk, R. Chikhi et al., “Gatb: genome assembly & analysis tool box,” Bioinformatics, vol. 30, no. 20, pp. 2959–2961.

[2] R. Li, H. Zhu, J. Ruan et al., “De novo assembly of human genomes with massively parallel short read sequencing,” Genome Research, vol. 20, no. 2, pp. 265–272, 2010..

[3] T. C. Conway and A. J. Bromage, “Succinct data structures for assembling large genomes,” Bioinformatics, vol. 27, no. 4, pp. 479–486, 2011).

APPENDIX

```
import gvmagic

def debruijnize(reads):
    nodes = set()
    not_starts = set()
    edges = []
    for r in reads:
        r1 = r[:-1]
        r2 = r[1:]
        nodes.add(r1)
        nodes.add(r2)
        edges.append((r1,r2))
        not_starts.add(r2)
    return (nodes,edges,list(nodes-
not_starts))


def build_k_mer(str,k):
    return [str[i:k+i] for i in
range(0,len(str)-k+1)]


def make_node_edge_map(edges):
    node_edge_map = {}
    for e in edges:
        n = e[0]
        if n in node_edge_map:
            node_edge_map[n].append(e[1])
        else:
            node_edge_map[n] = [e[1]]
    return node_edge_map

def eulerian_trail(m,v):
    nemap = m
    result_trail = []
    start = v
    result_trail.append(start)
    while(True):
        trail = []
        previous = start
        while(True):

            if(previous not in nemap):
                break
            next = nemap[previous].pop()
            if(len(nemap[previous]) == 0):
                nemap.pop(previous,None)
            trail.append(next)
            if(next == start):
                break;
            previous = next
        # completed one trail
        print(trail)
        index = result_trail.index(start)
        result_trail =
result_trail[0:index+1] + trail +
result_trail[index+1:len(result_trail)]
        # choose new start
        if(len(nemap)==0):
          break
        found_new_start = False
        for n in result_trail:
            if n in nemap:
                start = n
                found_new_start = True
                break # from for loop
        if not found_new_start:
            print("error")
            print("result_trail",result_tra
il)
            print(nemap)
            break
return result_trail
def visualize_debruijn(G):
    nodes = G[0]
    edges = G[1]
    dot_str= 'digraph "DeBruijn graph" {\n
```

```
for node in nodes:
        dot_str += '    %s [label="%s"] ;\n'
%(node,node)
    for src,dst in edges:
        dot_str += '    %s->%s;\n' %(src,dst)
    return dot_str + '}\n'


def assemble_trail(trail):
    if len(trail) == 0:
        return ""
    result = trail[0][:-1]
    for node in trail:
        result += node[-1]
    return result
def test_assembly_debruijn(t,k):
    reads = build_k_mer(t,k)
    G = debruijnize(reads)
    v = visualize_debruijn(G)
    nemap = make_node_edge_map(G[1])
    print(G)
    print(v)
    start = next(iter(G[2])) if (len(G[2]) >
0) else next(iter(G[0]))
    trail = eulerian_trail(nemap,start)
    return assemble_trail(trail)

reads = build_k_mer("ATCGTTGCGCGACCG",4)
print(reads)

G = debruijnize(reads)
print(G)

m = make_node_edge_map(G[1])
print(m)

start = G[2][0] if (len(G[2]) > 0) else
G[0][0]
print (m)

t = eulerian_trail(m,start)
print(t)

get_ipython().magic(u'load_ext gvmagic')

get_ipython().magic(u'dotstr
visualize_debruijn(G)')

assemble_trail(t)
```