GENERATING MAXIMAL NON-BRANCHING PATHS OF A GRAPH

PROJECT REPORT

For the course

19BIO201-Intelligence of Biological Systems 3

Submitted by

BL.EN.U4AIE19007 - APOORVA M

BL.EN.U4AIE19010 - BHUVANASHREE MURUGADOSS

BL.EN.U4AIE19027 - KARNA SAI NIKHILESH REDDY

Guided and Evaluated by

Dr. S. Santhanalakshmi
Computer Science Engineering
Asst. Professor
ASE-Bangalore

Dr. Amrita Thakur
Department of Chemistry
Asst. Professor
ASE-Bangalore

in partial fulfillment for the award of the degree

0f

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE ENGINEERING

AMRITA SCHOOL OF ENGINEERING, BANGALORE

AMRITA VISHWA VIDYAPEETHAM

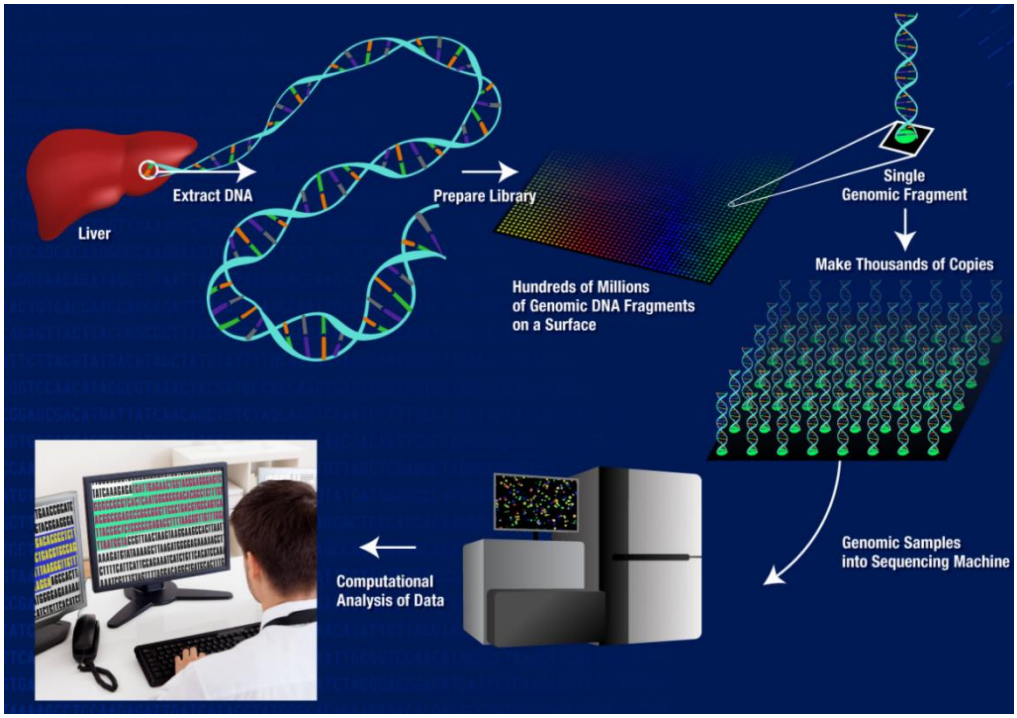
BANGALORE 560 035

DECEMBER-2020

TABLE OF CONTENTS

GENOME SEQUENCING

The task of the genome assembly is to reconstruct the original genome from the fragments of reads. The rapid increase of next generation genome sequencing with short reads has given rise to a lot of opportunities to explore but concurrently has also raised difficult and overwhelming computational challenges in genome assembly. Hence, Genome assembly remains to portray one of the most important algorithmic problems in bio-informatics.



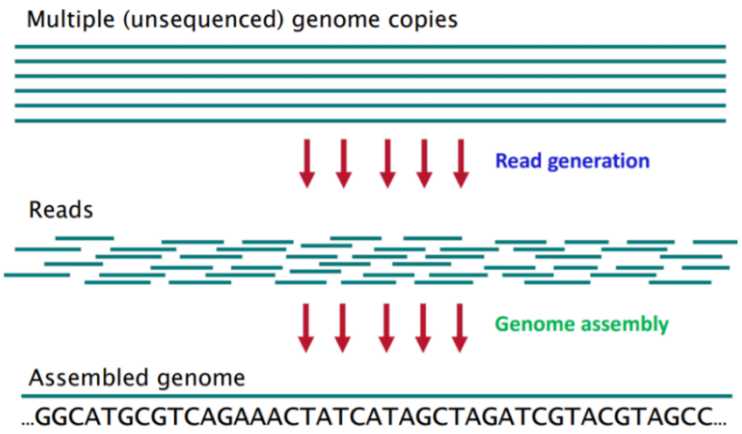
PROBLEM STATEMENT

Generate All Maximal Non-Branching Paths in a Graph

Maximal Non-branching path approach iterates through all nodes of the graph that are not 1-in-1-out nodes and generates all non-branching paths starting at each such node. In the final step, it finds all isolated cycles in the graph.

THEORY

The modern genome sequencing machines can only shred the genomes as small fragments called reads. The locations of the read in the genome are unknown, so the task is to reconstruct the original genome sequence.



The Overall picture of Genome sequencing and assembly can be seen from the given image. Basically, the overlapping portions are found and merged together. The first approach that was proposed was the Naïve approach, where we try to merge the K-mers in lexographical order. The extension of this approach led to forming a genome path with each node comprising of the Kmers.

Here, we simply connect Kmer1 with Kmer2, if suffix (Kmer1) = prefix (Kmer2). If the suffix and prefix of a K-mers matches with more than one K-mers, then that particular node will be connected to multiple nodes. This is how the Genome paths gets converted to Genome graph.

Next, we have the Overlap graph method which is basically a slight variation of the previous approach that we studied. Here, the prefix and suffix will be the first and last (k-1) nucleotides. So, if an arbitrary K-mers pattern is given, we form a graph consisting a node for each K-mers in the pattern. Whenever the prefix(pattern1) matches with the suffix(pattern2), they are connected using a directed edge.

The resulting graph is called an overlap graph. Hamiltonian path is found for the graph is order to reconstruct the genome. Another perspective of seeing the problem is by assigning K-mers as the edges and the Prefixes and suffixes as part of the nodes. Eulerian path is found for this graph in order to reconstruct the genome.

Since each pair of consecutive edges represent consecutive 3-mers that overlap in two nucleotides, we will label each node of this graph with a 2-mer representing the overlapping nucleotides shared by the edges on either side of the node. This approach is called String Reconstruction Problem using Eulerian Path.

Since there are chances of nodes being repeated many numbers of times, this is method is further enhanced my gluing all the identical nodes together. Doing this reduces the complexity of the graph. Eulerian path is found for the respective graph. This approach is called the De Bruijn Approach.

Why Maximal Non-Branching paths?

In the read breaking approach, majority of the assemblies were still found to have gaps in k-mer coverage, leading the de Bruijn graph to consist of missing edges, making the Eulerien path search fail.

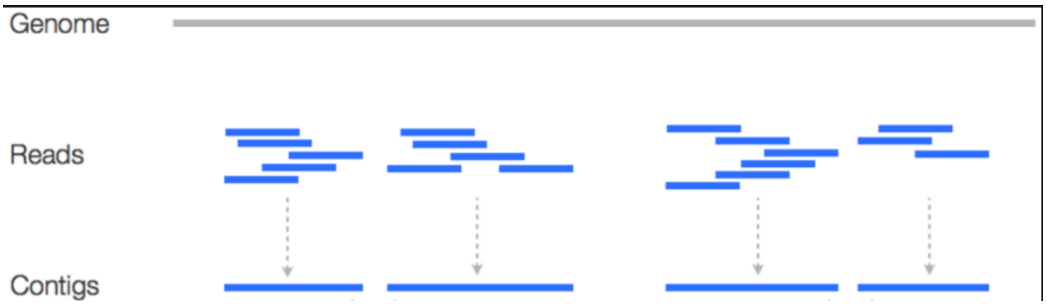


Fig: Contigs

Due to this, Biologists mostly try to settle on assembling of contigs instead of the complete chromosomes. Contigs are nothing but long, contiguous segments of the genome. A typical bacterial sequencing project may consist of about hundred contigs, which has length ranging from a few thousand to a few hundred thousand nucleotides. Mostly for genomes, the order in which the contigs are along the genome, remains to be unknown. Needless to say, biologists would prefer to have the entire genomic sequence, but the cost of ordering the contigs into a final assembly and closing the gaps using more expensive experimental methods is often prohibitive. Fortunately, we can derive contigs from the de Bruijn graph. Maximal non-branching path is a path whose internal nodes are 1-in-1-out node and whose initial and final nodes are not 1-in-1-out nodes. There can also be a case where the graph has a connected component that is an isolated cycle, in which all nodes are 1-in-1-out nodes.

We are interested in these paths because the strings of nucleotides that they spell out must be present in any assembly with a given k-mer composition. For this reason, contigs correspond to strings spelled by maximal non-branching paths in the de Bruijn graph.

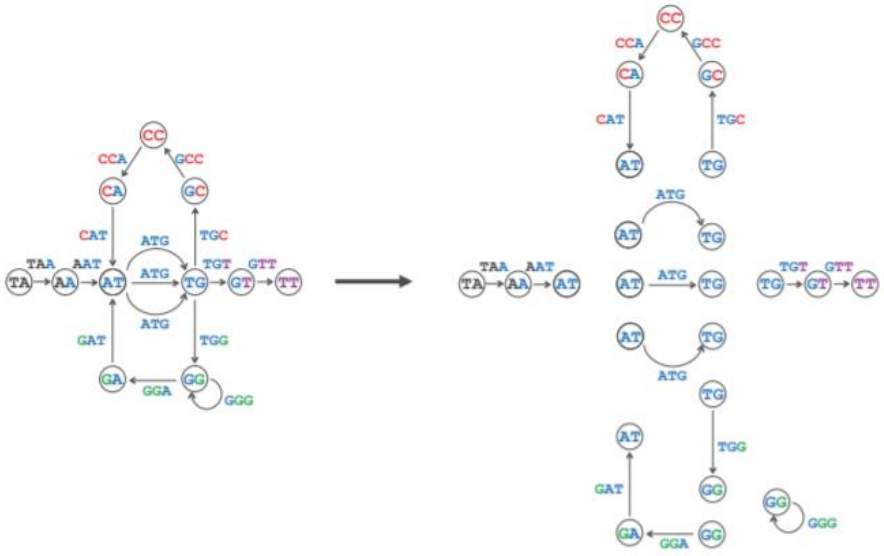


Fig: Breaking the graph DEBRUIJN3(TAATGCCATGGGATGTT) into nine maximal non-branching paths representing contigs TAAT, TGTT, TGCCAT, ATG, ATG, ATG, TGG, GGG, and GGAT.

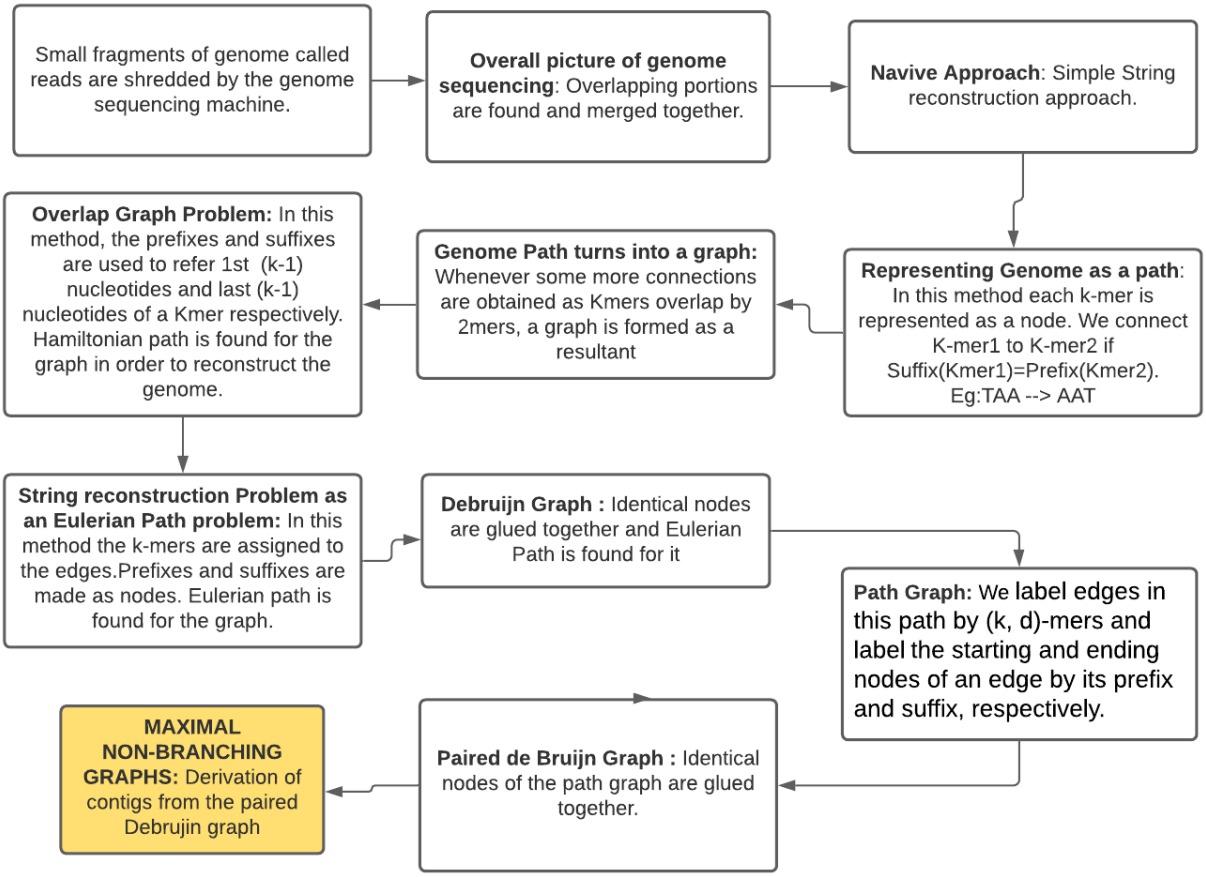


Fig: Flowchart on the sequence of different algorithms for genome sequencing

ALGORITHM

```
MaximalNonBranchingPaths(Graph)
    Paths ← empty list
    for each node v in Graph
        if v is not a 1-in-1-out node
            if out(v) > 0
                for each outgoing edge (v, w) from v
                    NonBranchingPath ← the path consisting of the single edge (v, w)
                    while w is a 1-in-1-out node
                        extend NonBranchingPath by the outgoing edge (w, u) from w
                        w ← u
                    add NonBranchingPath to the set Paths
    for each isolated cycle Cycle in Graph
        add Cycle to Paths
    return Paths
```

Fig: Maximal Non-Branching Algorithm

IMPLEMENTATION

Input : The adjacency list of a graph whose nodes are integers.

Output : The collection of all maximal non-branching paths in the graph.

```
import pandas as pd
import numpy as np
import os
```

Loading the Input file

```
dataset = 'sample_maximal_nonbranching_paths'
dataset_folder = os.getcwd() + "\\TestData\\"
with open(dataset_folder+dataset+'.txt') as file:
    data = [line.rstrip() for line in file]
# print(*data, sep="\n")
inp = data[1:data.index('Output')]
out = data[data.index('Output')+1:]
# print(*out, sep="\n")
data = inp
```

Generating the adjacency list corresponding to the input

```
def convert_to_graph_edges(raw_data):
    graph = {}
    for line in raw_data:
        x = line.split(" -> ")
        a,b = x[0],x[1].split(',')
        subset = graph.get(x[0],[])
        graph[x[0]] = subset+b

    return graph
```

convert_to_graph_edges: this function converts the raw data into graph edges by storing each node and its respective outgoing nodes in a dictionary. Here each node will be represented as the key and its outgoing nodes will be its value.

Generating the adjacency matrix from the adjacency list

```
def load_adj_mat(graph_edges): # this method creates adjacent matrix
    st = set()
    for key in graph_edges.keys():
        st.add(key)
        for value in graph_edges[key]:
            st.add(value)

    st = list(np.sort([int(x) for x in list(st)]))
    st = [str(x) for x in st]# lexicographical order
    adj_mat = pd.DataFrame(np.zeros((len(st),len(st)),dtype=int),index=list(st),columns=list(st))

    for key in graph_edges.keys():
        for value in graph_edges[key]:
            adj_mat.loc[key,value] = 1

    return adj_mat
```

Load_adj_mat: This function creates an adjacency matrix by taking the edges of the graph as the parameter. Firstly, the graph edges that were stored in the form of dictionary is stored in a set. Then it is sorted (ascending, lexographically). Later a dataframe is created by using the node values as rows and columns. The cells that are a part of the key index and come under value columns, are marked one. This is how the adjacency matrix is created.

Performing DFS for linear and cyclic paths

```
def do_DFS(matrix,start,List):
    List.append(start)
    outgoing_nodes = matrix.index[matrix.loc[start].isin([1])].tolist()

    if (outgoing_nodes == []):
        return List

    for outnode in outgoing_nodes:
        if outnode not in List:
            List = do_DFS(matrix,outnode,List) # Recursive Call
    return List
```

do_DFS: Using the Adjacency matrix, we first detect the edges, find the outgoing nodes and append it to a list. If there are no outgoing nodes, we return the list.
Every time we come across a new outgoing node, we check whether that node is present in the list or not. If it is not present in the list, we perform DFS for that node. This happens recursively.

```
#For isolated cycle detection
def i_dfs(matrix,u,ipaths,status,parent):
    status[u] = 'G'
    for v in matrix.columns[matrix.loc[u].isin([1])].tolist():
        if status[v] == 'G':
            parent[v] = u
            add_cycle(u,parent,ipaths)
            break
        if status[v] == 'W':
            parent[v] = u
            i_dfs(matrix,v,ipaths,status,parent)
    status[u] = 'B'
```

i_dfs: This function is used to detect the isolated cycles. Initially the path will only consist of the starting node which is u. Since we are traversing through as isolated cycle, the degree of all the node will be 1-in-1-out. Primarily the status of all the nodes will be white. we change the status of the starting node to Grey and as we traverse the cycle, we'll be changing the status of the visited nodes to grey and the visited node will become the new 'u'. Till the travel encounters a node with status as grey, it will go on. Once it encounters a grey node, the cycle(path) will be returned.

Finding the Maximal Non Branching Paths

```
def MaximalNonBranchingPaths(graph):
    matrix = load_adj_mat(graph)
    Paths = []
    Set = set()

    #ICycle inits
    status = {}
    parent = {}
    icycle_paths = []

    for key in graph.keys():
        if (not(matrix.loc[key].sum()==1 and matrix[key].sum()==1)):
            if (matrix.loc[key].sum()> 0):
                outgoing_nodes = matrix.index[matrix.loc[key].isin([1])].tolist()
                Set.add(key)
                for outnode in outgoing_nodes:
                    path = key+""
                    while(matrix.loc[outnode].sum()==1 and matrix[outnode].sum()==1):
                        Set.add(outnode)
                        path = path + " -> " + outnode
                        outnode = matrix.index[matrix.loc[outnode].isin([1])].tolist()[0]

                    Set.add(outnode)
                    Paths.append(path + " -> " + outnode)
```

MaximunNonBrancingPaths: This is the main function to find the maximum non-branching paths. Graph is taken as the parameter for this function. Firstly, the adjacency matrix of the graph is loaded and two lists called paths and icycle_paths, a dictionary called status and a set are initialized. In the function, the keys i.e. the nodes that are not one-in-one-out are found. If the outgoing nodes of the node(key) is greater than 0, then that node(key) is added to the set. All the outgoing nodes of the respective node is added to the list call paths (along with the node itself). While the out_going_node is a one-in-one-out node, we extend the NonBranchingPath and add that outnode to the set.

Next, we'll be checking for the isolated cycles. For this we find the nodes(keys) that are part of the matrix but not the set. These nodes are defined under the variable name icycle_nodes. All the icycle_nodes are initialized with status as white.
Now these nodes are traversed one by one. Whenever a node with status white is encountered, i_dfs function is called and the respective node is appended to the icycle_paths list.

```
icycle_nodes = sorted([key for key in graph.keys() if key not in Set])
for node in icycle_nodes:
    status[node] = 'W'
    parent[node] = None

for node in icycle_nodes:
    i_dfs(matrix,node,icycle_paths,status,parent)

newlis = list()
for i in icycle_paths:
    newx = [x for x in i.split(' -> ')[::-1]]
    newlis.append(' -> '.join(newx))

Paths += newlis
return (Paths,matrix,icycle_paths)
```

```
graph = convert_to_graph_edges(data)
Paths,Matrix,ic = MaximalNonBranchingPaths(graph)

with open('output.txt','w') as f:
    for path in Paths:
        f.write(path + '\n')

# print('Raw Data    :',data)
# print('Graph Edges :',graph)
print('\n'+'*'*30+'Generated Maximal Non-Branching Paths'+'*'*30)
print(*Paths,sep='\n')
# print('Adjacent Matrix :')
# Matrix
```

RESULTS

```
Input Graph Edges : {'1': ['2'], '2': ['3'], '3': ['4', '5'],
'6': ['7'], '7': ['8'], '8': ['9'], '10': ['11'], '9': ['6'], '1
1': ['10']}

Adjacent Matrix :
```

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1
11	0	0	0	0	0	0	0	0	0	1	0

```
Generated Maximal Non-Branching Paths
1 -> 2 -> 3
3 -> 4
3 -> 5
11 -> 10 -> 11
9 -> 6 -> 7 -> 8 -> 9
```

CONCLUSION

Through this project we've navigated through the different methods for sequencing a genome and implemented the algorithm to find the Maximal Non-Branching Paths in a given De-Bruijn Graph. We have observed that even after read breaking, most assemblies still have gaps in k-mer coverage, causing the de Bruijn graph to have missing edges, and so the search for an Eulerian path fails.

Hence, we find long continuous segments of the genome called contigs rather than the entire genome. The Maximal Non-Branching Path helps us identify the contig regions and thus enabling us to find the strings of nucleotides that must be present in any assembly with a given k-mer composition.

REFERENCE

1) “Bioinformatics Algorithms: An Active Learning Approach” by Pavel A. Pevzner and Phillip Compeau

2) Rosalind: http://rosalind.info/problems/ba3m/ for algorithm explanation and extra dataset