GENOME ASSEMBLY USING BREADTH FIRST SEARCH

A PROJECT REPORT

Submitted by

BL.EN.U4AIE19041 M. TANUJ

BL.EN.U4AIE19049 P. VENKATA AKHIL

BL.EN.U4AIE19068 AISHWARYA.V

For the course

19BIO201 – Intelligence of Biological Systems - 3

Guided and Evaluated by

S. SANTHANALAKSHMI and AMRITA THAKUR

IN

ARTIFICIAL INTELLIGENCE ENGINEERING

AMRITA SCHOOL OF ENGINEERING, BANGALORE

BANGALORE 560 035

ABSTRACT

The main aim of our project is to study the Breadth First Search Method in graphs and how it is useful in genome assembly. So, in our project, we have shown the traversal of de-Bruijn graphs using Breadth First Search algorithm.

BIOLOGICAL CONCEPTS

Genome assembly

The assembly process consists of (1) building the sparse de Bruijn graph with the n-kmers and (2) graph traversal. Like with the standard de Bruijn graph, we explore the linkage or adjacency relations and branching information of the n-k-mers for reconstructing the genome. The procedure for building contigs is similar to that of the standard de Bruijn graph. A new traverse begins at an n-kmer not visited in previous traverses, and breaks when branching bases are detected. The separate traverses form the contigs.

De-Bruijn Graph

The de Bruijn graph is a directed graph used for representing overlapping strings in a collection of k-mers. It represents every k-mer as an edge between its prefix and suffix. And then all nodes with identical nodes are glued. These sequences can be reconstructed by moving between nodes in graph. Each De Bruijn graph is Eulerian and Hamiltonian. The Euler cycles and Hamiltonian cycles of these graphs (equivalent to each other via the line graph construction) are De Bruijn sequences. Here is an example in which the sequence "ATGCTAGCAC" of the length 10, is assembled from five reads, each of length six.

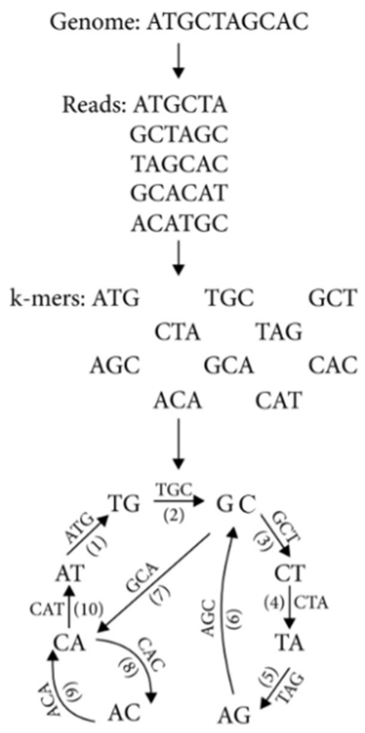


Fig: Construction of De-Bruijn graph from a genome

Graph traversals

Graph theory and in particular the graph ADT (abstract data-type) is widely explored and implemented in the field of Computer Science and Mathematics. Consisting of vertices (nodes) and the edges (optionally directed/weighted) that connect them, the data-structure is effectively able to represent and solve many problem domains. One of the most popular areas of algorithm design within this space is the problem of checking for the existence or (shortest) path between two or more vertices in the graph.

Breadth First Search

Breadth-first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures. The full form of BFS is the Breadth-first search. The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Remember, BFS accesses these nodes one by one.

Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them. Once visited, all nodes are marked. These iterations continue until all the nodes of the graph have been successfully visited and marked.

Steps involved in BFS Traversal:

- A queue (FIFO-First in First Out) data structure is used by BFS.
- You mark any node in the graph as root and start traversing the data from it.
- BFS traverses all the nodes in the graph and keeps dropping them as completed.
- BFS visits an adjacent unvisited node, marks it as done, and inserts it into a queue.
- Removes the previous vertex from the queue in case no adjacent vertex is found.
- BFS algorithm iterates until all the vertices in the graph are successfully traversed and marked as completed.
- There are no loops caused by BFS during the traversing of data from any node.

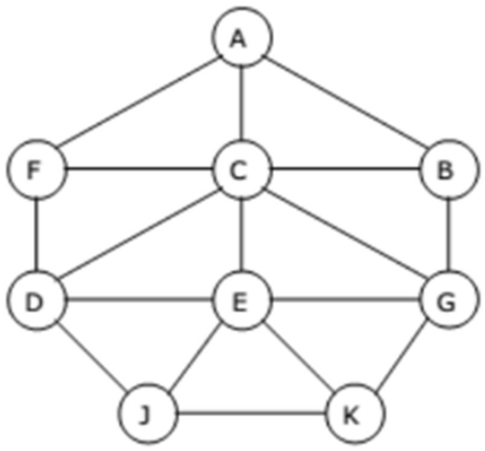
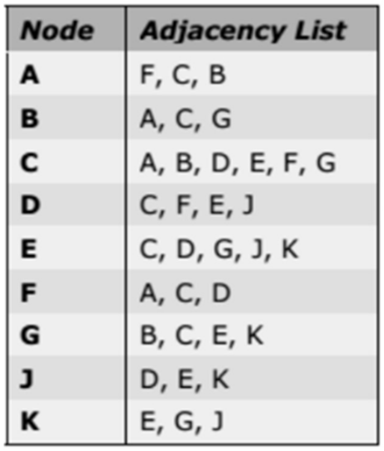
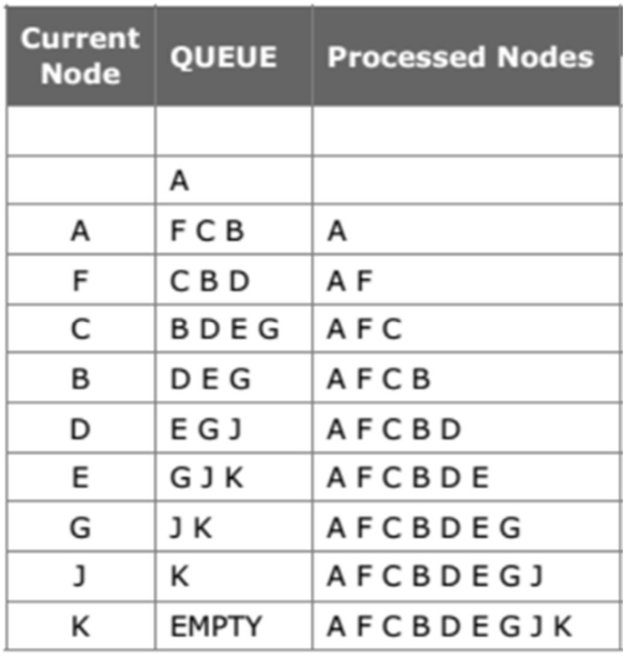


Fig: Graph considered for Breadth First Search



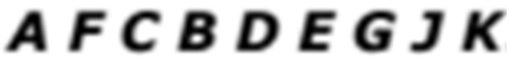
Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Fig: Adjacency list of all nodes in the graph



Current Node	QUEUE	Processed Nodes
	A	
A	F C B	A
F	C B D	A F
C	B D E G	A F C
B	D E G	A F C B
D	E G J	A F C B D
E	G J K	A F C B D E
G	J K	A F C B D E G
J	K	A F C B D E G J
K	EMPTY	A F C B D E G J K

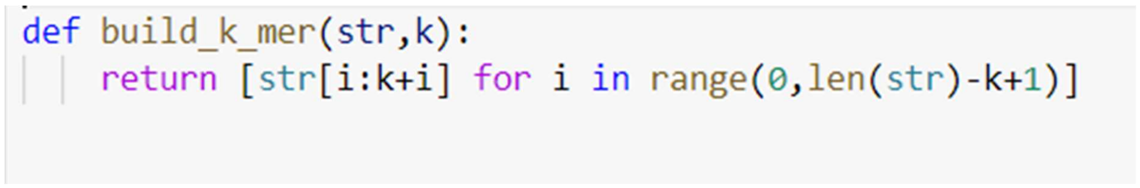
Fig: Table showing the status of nodes



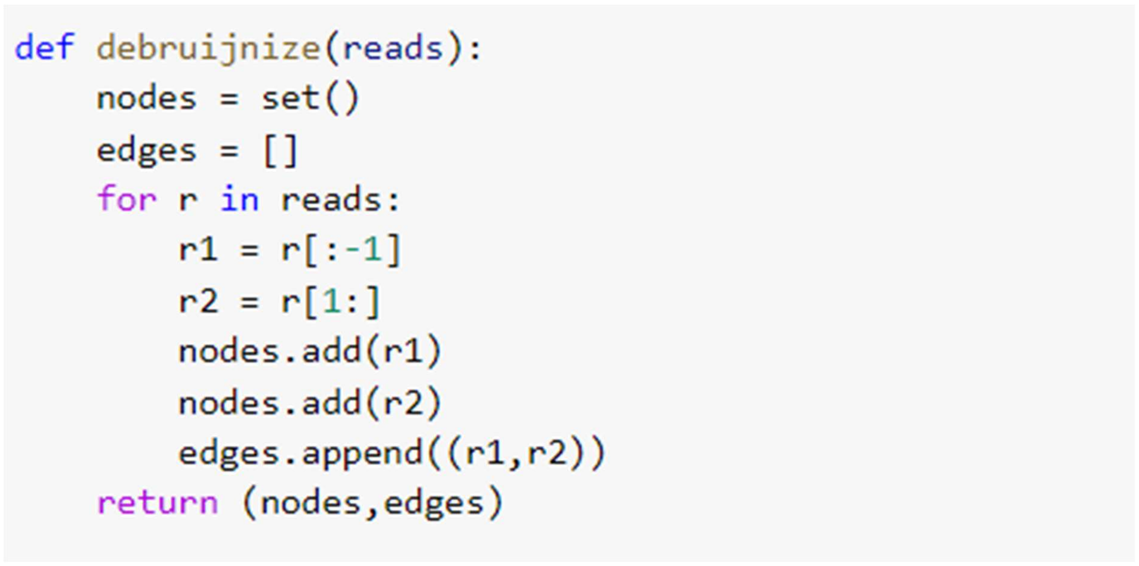
A F C B D E G J K

Fig: Output Sequence

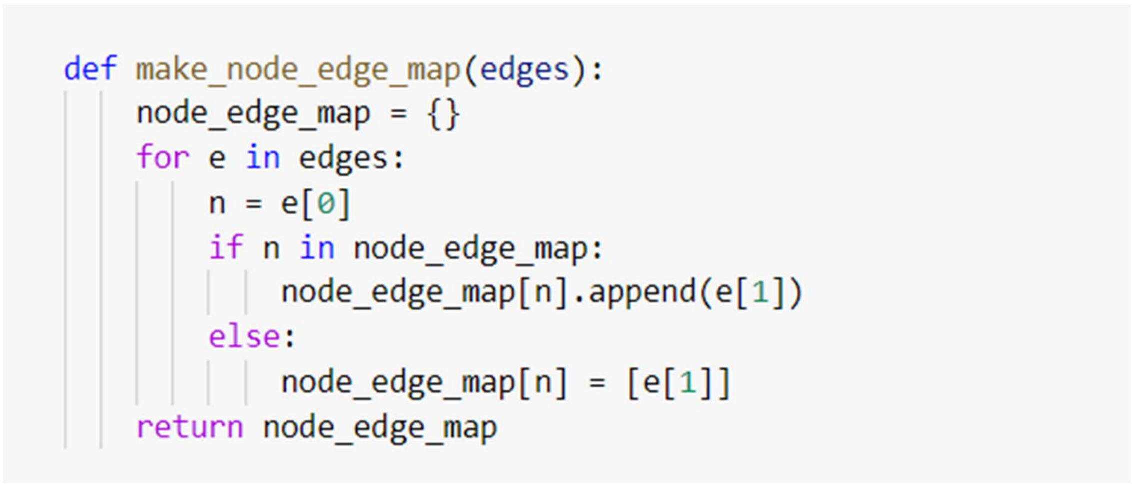
CODE



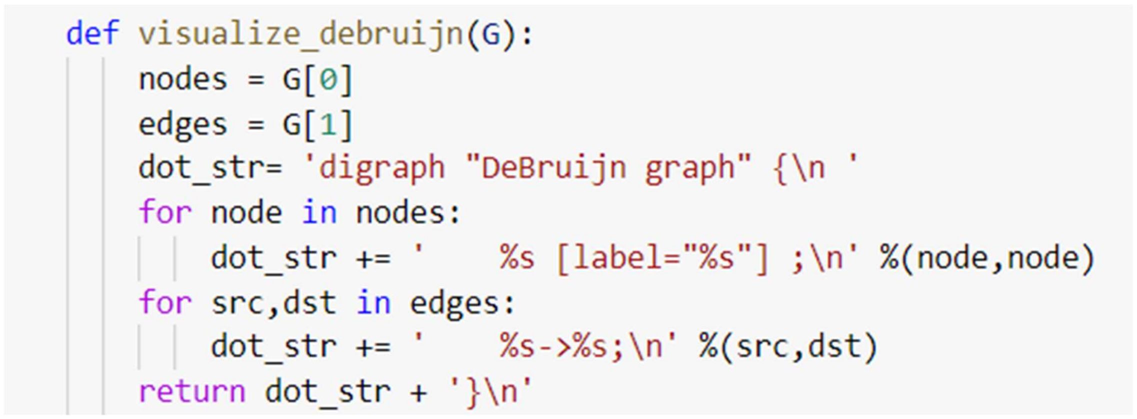
```python
def build_k_mer(str,k):
    return [str[i:k+i] for i in range(0,len(str)-k+1)]
```



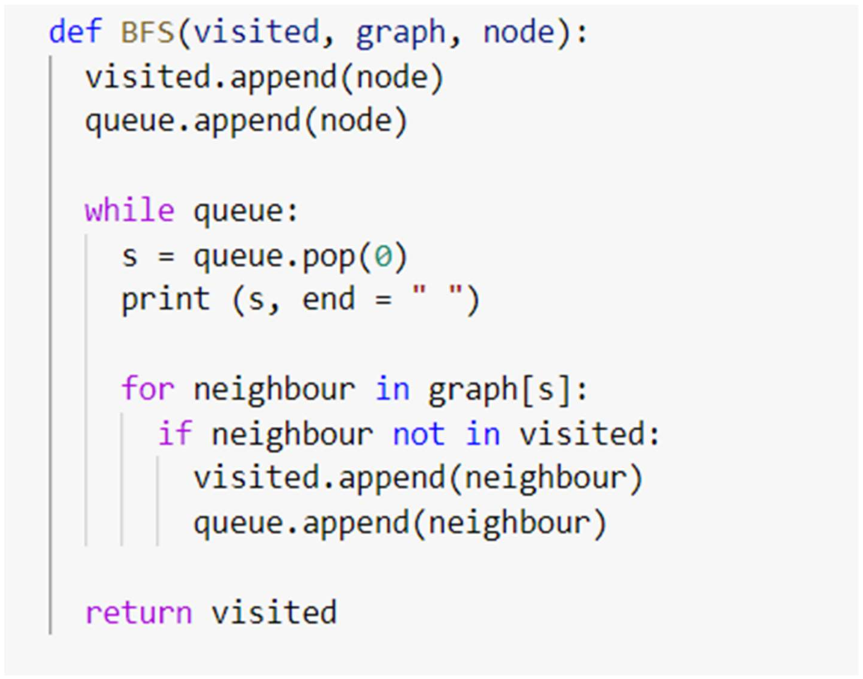
```python
def debruijnize(reads):
    nodes = set()
    edges = []
    for r in reads:
        r1 = r[:-1]
        r2 = r[1:]
        nodes.add(r1)
        nodes.add(r2)
        edges.append((r1,r2))
    return (nodes,edges)
```



```python
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
```



```python
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'
```

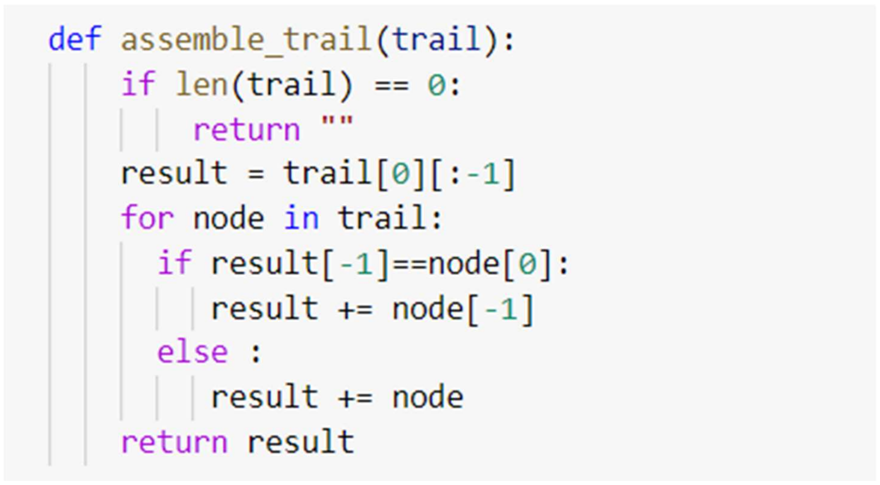


```python
def BFS(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    s = queue.pop(0)
    print (s, end = " ")

    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

  return visited
```



```python
def assemble_trail(trail):
    if len(trail) == 0:
        return ""
    result = trail[0][:-1]
    for node in trail:
      if result[-1]==node[0]:
         result += node[-1]
      else :
         result += node
    return result
```

RESULTS

```
reads = build_k_mer("ACGCGTCG",3)
print(reads)
```

```
['ACG', 'CGC', 'GCG', 'CGT', 'GTC', 'TCG']
```

```
G = debruijnize(reads)
print(G)
```

```
({'GT', 'CG', 'GC', 'AC', 'TC'}, [('AC', 'CG'), ('CG', 'GC'), ('GC', 'CG'), ('CG', 'GT'), ('GT', 'TC'), ('TC', 'CG')]
```

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

```
{'AC': ['CG'], 'CG': ['GC', 'GT'], 'GC': ['CG'], 'GT': ['TC'], 'TC': ['CG']}
```

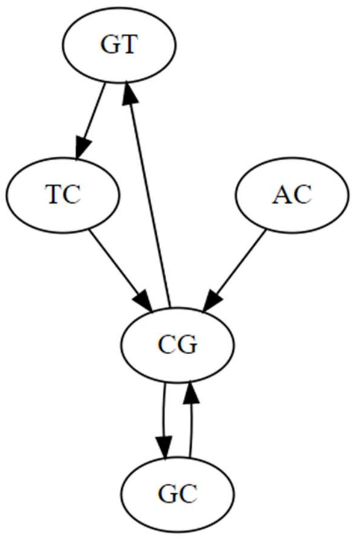
```
b=BFS(visited,m,'AC')
```

```
AC CG GC GT TC
```

```
List = ['AC', 'CG', 'GC', 'GT', 'TC']
```

```
print('De-Bruijn Graph:')
get_ipython().magic(u'dotstr visualize_debruijn(G)')
```

```
De-Bruijn Graph:
```



```
assemble_trail(List)
```

```
'ACGCGTC'
```

CONCLUSION

Thus, by using breadth first search method on a De-Bruijn graph, we were able to assemble the genome. By developing python programs, our skills have improved. Having analyzed the problem, we will now be able to extend this knowledge to perform the same for any other problem regarding genome sequencing.