Design and Analysis Algorithms

Assignment 3

Vertex k -Labeling of Non - Homogeneous Caterpillar using Algorithmic Approach

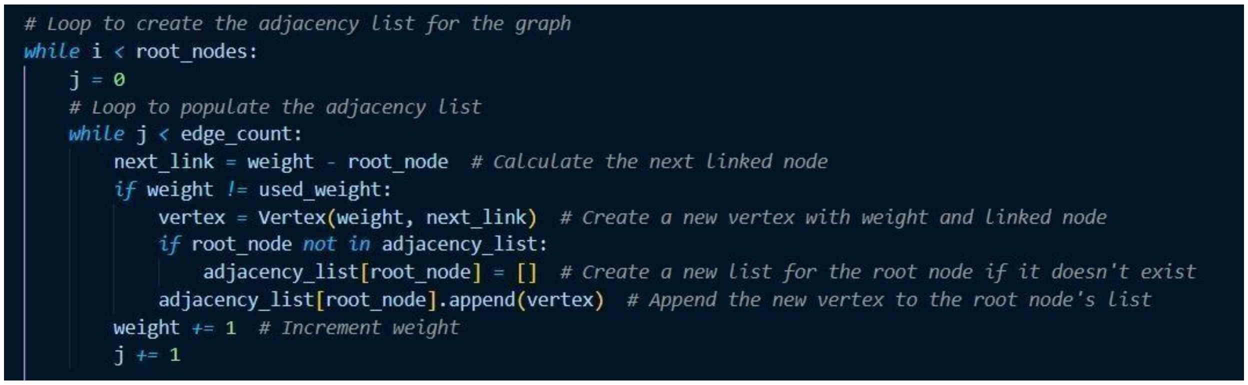
Charita Tummala - 16338814

Irregular Labeling of Graph Models Using Algorithmic Approach

1. Find out the best data structure to represent/store the graph in memory.

The graph is represented using an adjacency list data structure.

Adjacency List Representation: The graph structure is stored and manipulated as an adjacency list. This data structure efficiently represents connections between nodes by mapping each node to a list of its neighboring nodes and their corresponding weights. This strategy facilitates easy access to specific connections and simplifies adding or removing edges later.



```
# Loop to create the adjacency list for the graph
while i < root_nodes:
    j = 0
    # Loop to populate the adjacency list
    while j < edge_count:
        next_link = weight - root_node  # Calculate the next linked node
        if weight != used_weight:
            vertex = Vertex(weight, next_link)  # Create a new vertex with weight and linked node
            if root_node not in adjacency_list:
                adjacency_list[root_node] = []  # Create a new list for the root node if it doesn't exist
            adjacency_list[root_node].append(vertex)  # Append the new vertex to the root node's list
        weight += 1  # Increment weight
        j += 1
```

2. Devise an algorithm to assign the labels to the vertices using vertex k-labeling definition.

1. Initialization:
- Create an empty adjacency_list dictionary.
- Set weight = 3, edge_count = 3, root_node = 2, i = 1, and used_weight = -1.

2. Loop for Root Nodes:
While i < root_nodes:
Inner Loop for Edges:
While j < edge_count:
Calculate next_link = weight - root_node.
If weight != used_weight:
Create a Vertex(weight, next_link).
Add the vertex to the adjacency_list[root_node] list (creating it if necessary).
Increment weight.
Increment edge_count.

3. Special Case for Second-to-Last Root Node (Optional):

If i == root_nodes - 2:
Update used_weight = max_label + root_node.
Create a vertex Vertex(max_label + root_node, root_node).
Add the vertex to adjacency_list[max_label].
Update root_node = max_label.

4. Update Root Node (Except for Last):

If not in the special case above (i != root_nodes - 2):
Update root_node = weight - root_node.

5. Increase Counter:
Increment i.

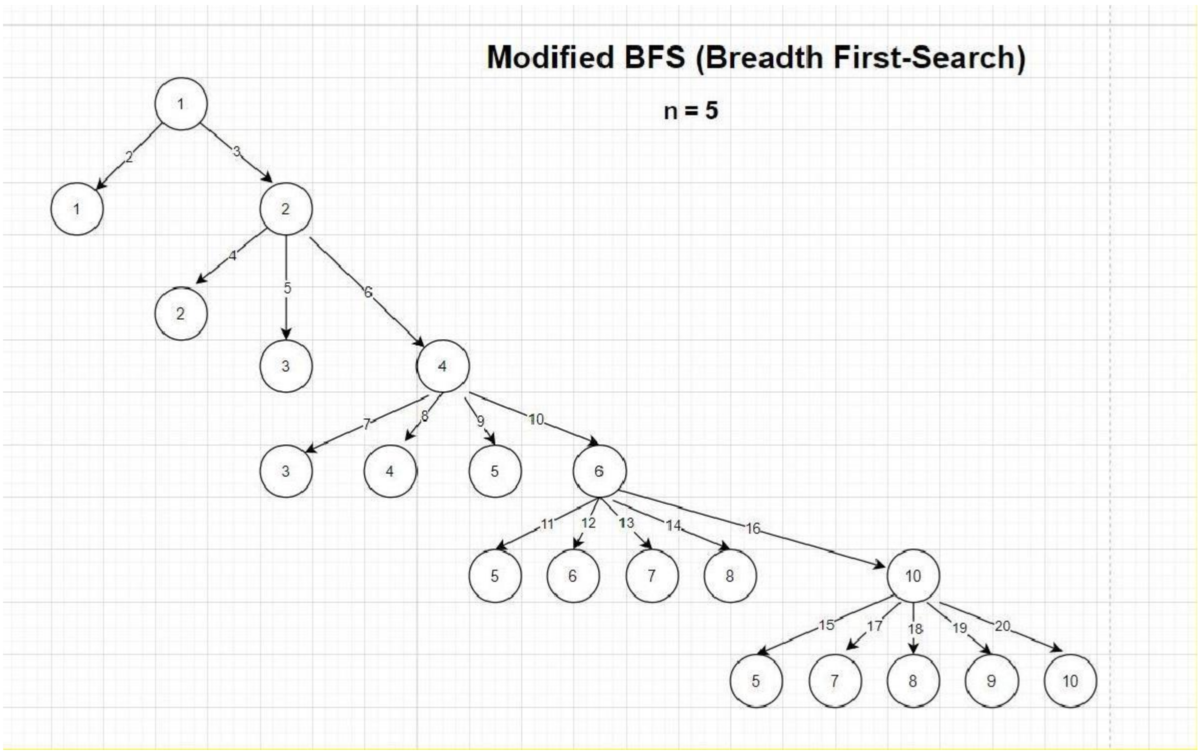
6. Return:
Return the final adjacency_list.

3. What design strategy you adopted? And how you deduced that applied strategy is most appropriate

We used Greedy algorithm as a design strategy in this algorithm. At each level we made a locally optimize choice for labelling the vertex with the hope of finding a globally optimum solution. We assigned the next edge by incrementing the previous edge by 1 based on edge weight we are labeling the next least vertex label which satisfies the condition.

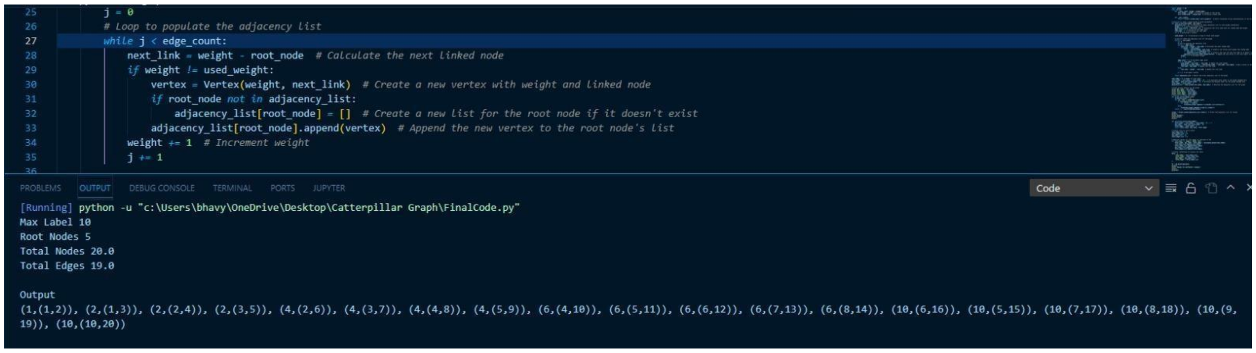
4. How traversing will be applied?

BFS is applied for the graph by modifying it according to this algorithm. The Breadth-First Search (BFS) is a graph traversal algorithm that explores a graph level by level.
Usually in the conventional BFS algorithm once the root is visited we have the flexibility to visit either of the left and right nodes of the root. However, we modified our BFS algorithm in such a way that after visiting the root, the next visit should be for the node which has no branches, once this is done we proceed to the other node of the root.



5. Store the labels of vertices and weights of the edges to print as separate.

In the below output we can see the edge weights and vertices are stored



```
        j = 0
        # Loop to populate the adjacency list
        while j < edge_count:
            next_link = weight - root_node  # Calculate the next linked node
            if weight != used_weight:
                vertex = Vertex(weight, next_link)  # Create a new vertex with weight and linked node
                if root_node not in adjacency_list:
                    adjacency_list[root_node] = []  # Create a new list for the root node if it doesn't exist
                adjacency_list[root_node].append(vertex)  # Append the new vertex to the root node's list
            weight += 1  # Increment weight
            j += 1
```

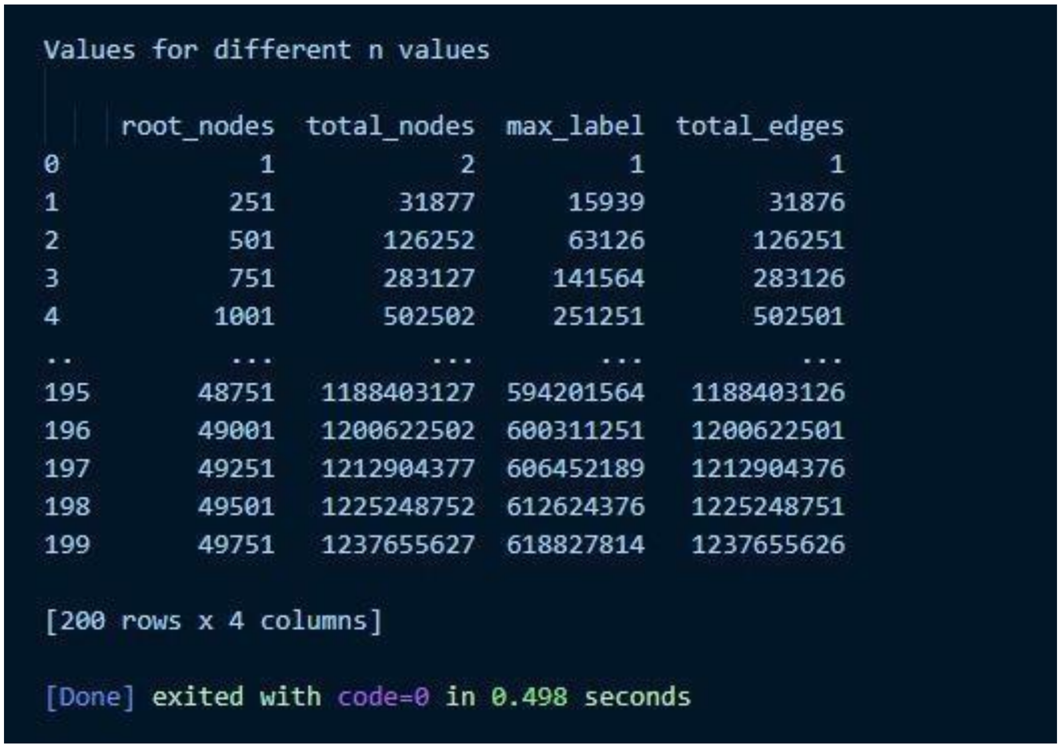
```
[Running] python -u "c:\Users\bhavy\OneDrive\Desktop\Catterpillar Graph\FinalCode.py"
Max Label 10
Root Nodes 5
Total Nodes 20.0
Total Edges 19.0

Output
(1,(1,2)), (2,(1,3)), (2,(2,4)), (2,(3,5)), (4,(2,6)), (4,(3,7)), (4,(4,8)), (4,(5,9)), (6,(4,10)), (6,(5,11)), (6,(6,12)), (6,(7,13)), (6,(8,14)), (10,(6,16)), (10,(5,15)), (10,(7,17)), (10,(8,18)), (10,(9,
19)), (10,(10,20))
```

6. Weights must be unique, so devise a subroutine to maintain distinctive property of edge weights.

In order to guarantee that every vertex and edge in the Non-Homogenous Caterpillar graph is appropriately weighted, a careful method is used by the subroutine responsible for edge and vertex labeling maintenance. Label-1 is the first vertex that we have assigned, and the edge weight begins at 2. Edge weights can be adjusted by incrementing the previous edge weight and storing the current weight, starting with the first edge weight value. The process is repeated for each subgraph, guaranteeing that the sum of any two connected vertex weights equals the corresponding edge weight and maintaining the mathematical integrity of the graph. For vertices, the label is subtracted from the edge weight. The result is then stored and assigned to the next vertex.

7. For each value of n (length of path), compute the values of V(G) & E(G).

For different n values, these are Total nodes V(G), edges E(G) in



```
Values for different n values

     root_nodes  total_nodes  max_label  total_edges
0             1            2          1            1
1           251        31877      15939        31876
2           501       126252      63126       126251
3           751       283127     141564       283126
4          1001       502502     251251       502501
..          ...          ...        ...          ...
195       48751   1188403127  594201564   1188403126
196       49001   1200622502  600311251   1200622501
197       49251   1212904377  606452189   1212904376
198       49501   1225248752  612624376   1225248751
199       49751   1237655627  618827814   1237655626

[200 rows x 4 columns]

[Done] exited with code=0 in 0.498 seconds
```

8. Compare your results with mathematical property and tabulate the outcomes for comparison.

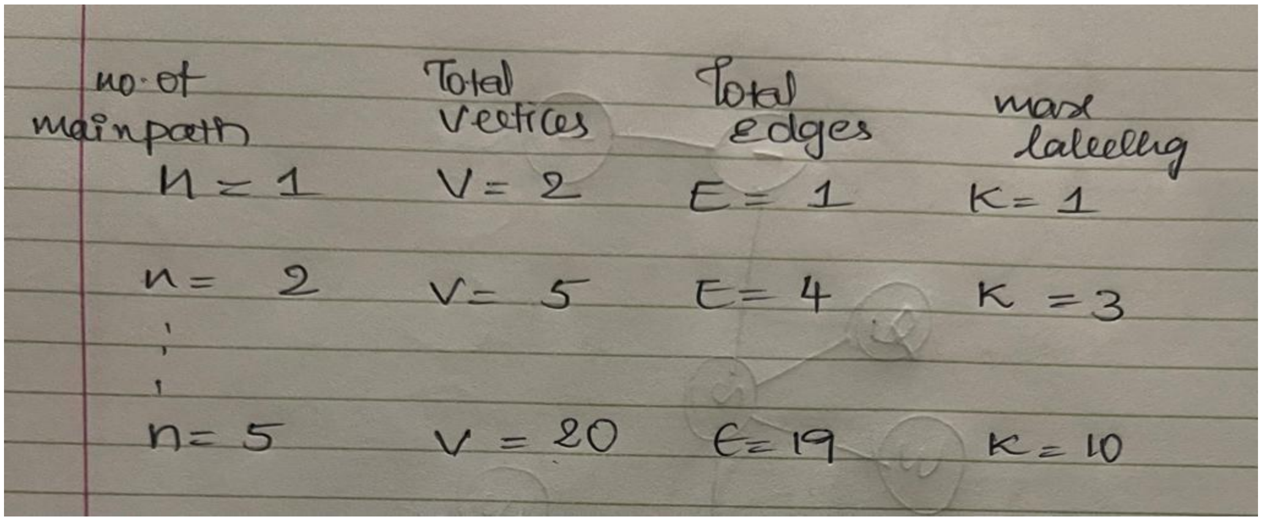
Total No. of Vertices V= n(n+3)/2

Total No. of Edges E = V-1

Max Vertex label k= Ceil(V/2)

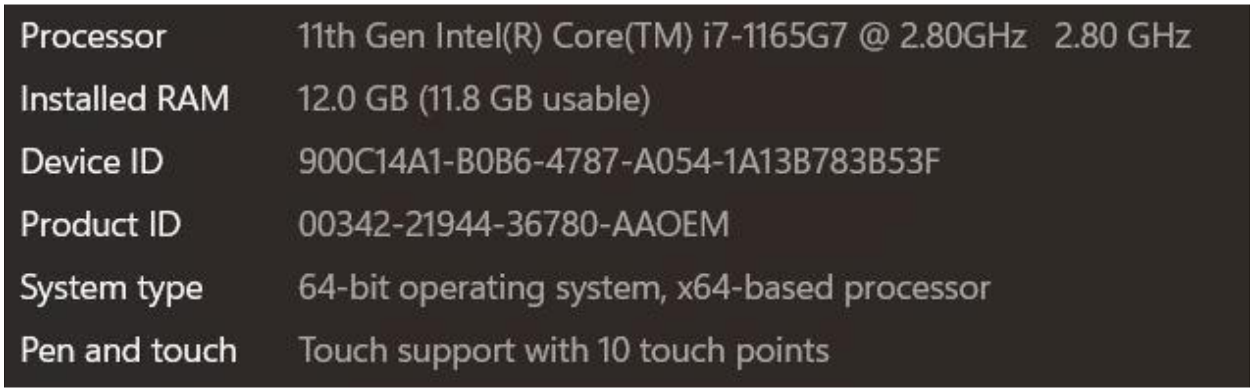
Created the below table using above formulas and compared with the results generated by algorithm:

In the result output we can see the values match with the computed values.



no. of mainpath	Total vertices	Total edges	max labelling
n = 1	V = 2	E = 1	K = 1
n = 2	V = 5	E = 4	K = 3
⋮			
n = 5	V = 20	E = 19	K = 10

9. Hardware resources supported until what maximum value of n and p.



Processor	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
Installed RAM	12.0 GB (11.8 GB usable)
Device ID	900C14A1-B0B6-4787-A054-1A13B783B53F
Product ID	00342-21944-36780-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	Touch support with 10 touch points

For root_nodes(n)	Time(in Sec)
5	0.469
100	1.529
500	4.716
2000	26.467
5000	79.529
10000	143.446
13000	-

10. Compute the Time Complexity of your algorithm T(V,E) or T(n,p).

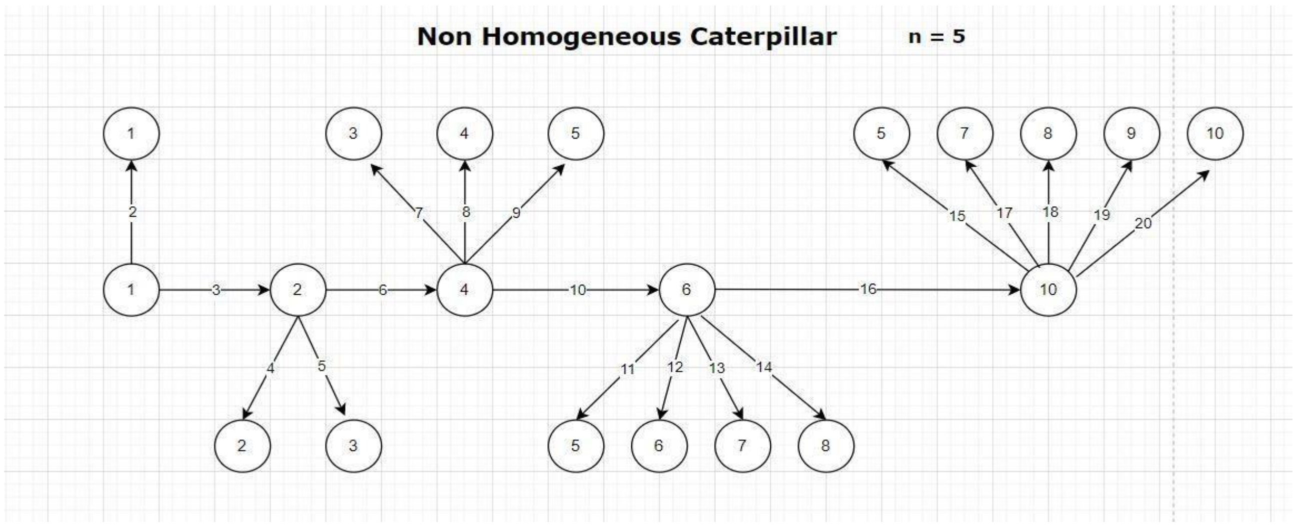
The execution time of the valid algorithm is finite. The algorithm's time complexity is the amount of time it takes to solve a certain task. A highly helpful metric in algorithm study is time complexity.

It is the amount of time required for an algorithm to finish. We must take into account both the cost and the number of executions of each basic instruction in order to determine the temporal complexity.

Considering the dominant factor, the overall time complexity of this algorithm will be O(root_nodes * edge_count)

Results(For n = 5) :

n = 5,


Non Homogeneous Caterpillar
n = 5
1
3
4
5
5
7
8
9
10
2
7
8
9
15
17
18
19
20
1
3
2
6
4
10
6
16
10
4
5
11
12
13
14
2
3
5
6
7
8