1.1.Graph Implementation

Pseudocode:

Create a Graph object g of size n O(1)
For i in range n: O(n^2)
For j in range n: O(n)
If i is equal to j: O(1)
continue to the next iteration O(1)
If the distance between i-kid and j-kid is less than or equal to strength of i-kid and j-kid: O(1)

add an edge between i and j in g, set adjacency_matrix[i][j]=1, push j back of graph[i] list. O(1)

Time Complexity:

The time complexity of the main function is O(n^2) due to the nested loop iterating through each pair of nodes in the input. The addEdge function of the Graph class has a time complexity of O(1) for adding an edge to the adjacency list and O(1) for updating the adjacency matrix, so the time complexity of adding all edges is O(n^2). Overall, the time complexity of the algorithm is O(n^2).

1.2.Breadth First Search

Pseudocode:

BFS(source, target): O(V+E)
queue is empty queue data structure O(1)
visited is empty set data structure O(1)
enqueue start to queue O(1)
mark start visited O(1)
while queue is not empty: O(V(E/V+1))
node is dequeued element of queue O(1)
for neighbour in range of node.neigbours : O(E/V)
if neighbour not in visited: O(1)
mark neighbour as visited O(1)
enqueue neighbour to queue O(1)
if node is equal to target: O(1)
break O(1)

Complexity:

If I used adjacency matrix instead of its list, Its time complexity was O(V^2), because we had to check every row and column of adjacency matrix. But I used adjacency list to less runtime.

Loop over each adjacent vertex in the adjacency list: This takes time proportional to the number of edges from the current vertex, or O(E/V) on average, since the average vertex degree is E/V. The time complexity of the entire loop is O(E/V) * O(1) = O(E/V).

Time complexity of the algorithm is O(V+E).The time complexity for creating the arrays and the queue is O(V). The time complexity for iterating over the adjacent nodes in each iteration of the while loop is O(E), since each edge is visited once. The time complexity for building the path is O(V), since the path has at most V nodes. Therefore, the overall time complexity of the BFS function is O(V+E).

In best case E=1, time complexity is O(V).In worst case, E= V^2, time complexity is O(V^2).

1.3.Depth First Search

Pseudocode:

String find_cycle(int source): O(V+E)
initialize visited, parent and finished arrays to false and -1 respectively
add source as vertex to path O(1)
    If bool dfs(first neighbor on source list,source,source) is true: O(V+E)
        Save path O(1)
    else no cycle detected:
        save path as -1 O(1)

bool dfs(int node, int source,int p): O(V+E)
//recursive function to perform DFS and detect cycle
mark node as visited O(1)
set p as parent of node O(1)
add node as vertex to path O(1)
if adjacency matrix[node][source]= 1 and nodes number in path >2: O(1) //cycle
    return true. O(1)
for neighbor in range of node.neigbours: O(V(E/V+1))
    if neighbor is not visited: O(1)
        if bool dfs(neighbor,source,node) is true: O(E/V)
            return true O(1)
        //cycle
        if source is parent of node and nodes number in path >2: O(1)
            return true O(1)

Complexity:

If I used adjacency matrix instead of its list, Its time complexity was O(V^2), because we had to check every row and column of adjacency matrix. But I used adjacency list to less runtime.

Loop over each adjacent vertex in the adjacency list: This takes time proportional to the number of edges from the current vertex, or O(E/V) on average, since the average vertex degree is E/V. The time complexity of the entire loop is O(E/V) * O(1) = O(E/V).

The time complexity of the DFS algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph. In the given code, the function find_cycle calls the function dfs for each unvisited neighbor of the source node, which takes O(V+E) time. Therefore, the overall time complexity of the find_cycle function is O(V+E).

In best case E=1, time complexity is O(V).In worst case, E= V^2, time complexity is O(V^2).

2-

Maintaining a list of discovered nodes in BFS and DFS is important for both algorithms to work correctly and efficiently.And it decreases time complexity.Let's assume there is no error in searching, Time complexity will be O(V *(V+E)) instead of O(V+E)

In BFS, maintaining a list of discovered nodes is crucial for avoiding revisiting nodes that have already been visited. This is important because without such a list, BFS can enter an infinite loop if the graph has a cycle. If BFS visits a node that has already been visited, but that node has not been marked as processed, it means that there is a cycle in the graph. Therefore, by maintaining a list of discovered nodes, BFS can ensure that it only visits each node once and terminates correctly even if there are cycles in the graph.

In DFS, maintaining a list of discovered nodes is also important for avoiding revisiting nodes that have already been visited. Without such a list, DFS can get stuck in an infinite loop if the graph has a cycle. However, in addition to this, maintaining a list of discovered nodes is also important for ensuring that the algorithm explores all nodes in the graph. Without this list, DFS may not explore all nodes, leading to incorrect results.

Maintaining a list of discovered nodes affects the outcome of both BFS and DFS algorithms by ensuring that each node is visited only once. Without this list, the algorithms may get stuck in cycles or loops and never reach the end of the graph, resulting in incorrect output. With the discovered nodes list, the algorithms are guaranteed to visit every node in the graph exactly once, which ensures that the output is correct.

3-A-

The runtime of BFS and DFS are dependent on the number of edges in the graph. In the worst-case scenario, where every node is connected to every other node, the runtime of BFS is O(V^2), where V is the number of nodes in the graph. However, in most real-world cases, the number of edges is much smaller than the number of possible edges, resulting in a much lower runtime.

- But let's double V.

Pseudocode for DFS:

String find_cycle(int source): O(2V+E)
initialize visited, parent and finished arrays to false and -1 respectively
add source as vertex to path O(1)
If bool dfs(first neighbor on source list,source,source) is true: O(2V+E)
Save path O(1)
else no cycle detected:
save path as -1 O(1)

bool dfs(int node, int source,int p): O(2V+E)
//recursive function to perform DFS and detect cycle
mark node as visited O(1)
set p as parent of node O(1)
add node as vertex to path O(1)
if adjacency matrix[node][source]= 1 and nodes number in path >2: O(1) //cycle
return true. O(1)
for neighbor in range of node.neigbours: O(2V(E/2V+1))
if neighbor is not visited: O(1)
if bool dfs(neighbor,source,node) is true: O(E/2V)
return true O(1)
//cycle
if source is parent of node and nodes number in path >2: O(1)
return true O(1)

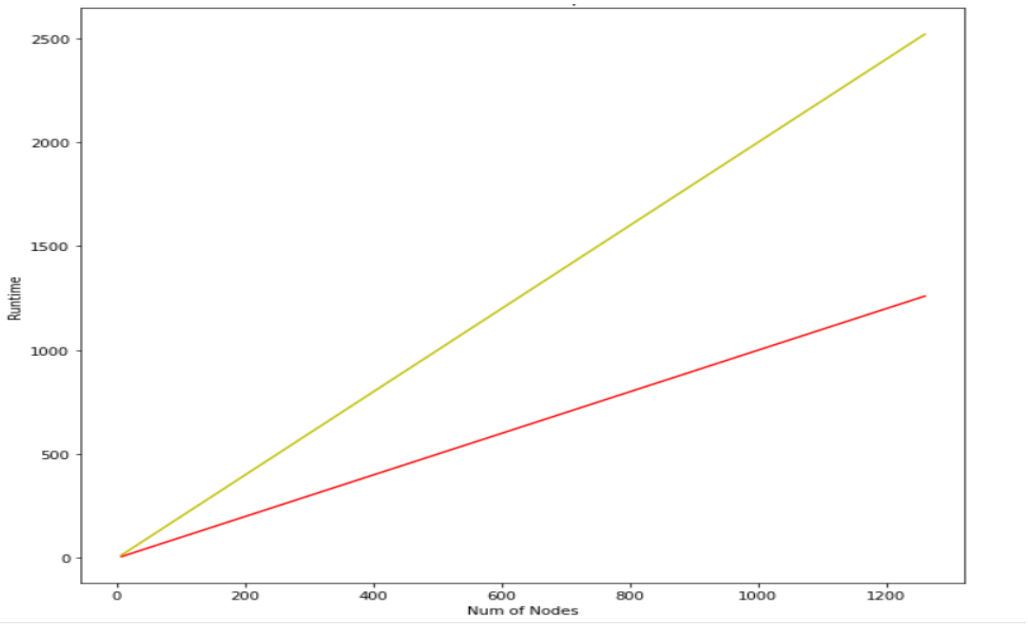
- Lets think E is equal to V^2 in worst case.Time complexity would be O(2V+E) which E is (2V) ^2. And it is O(2V+4V^2) also O(4V^2) instead of O(V^2). Time complexity is quadruple.
- Lets think E is equal to1 in best case.Time complexity would be O(2V) instead of O(V). Time complexity is doubled

Pseudocode for BFS:

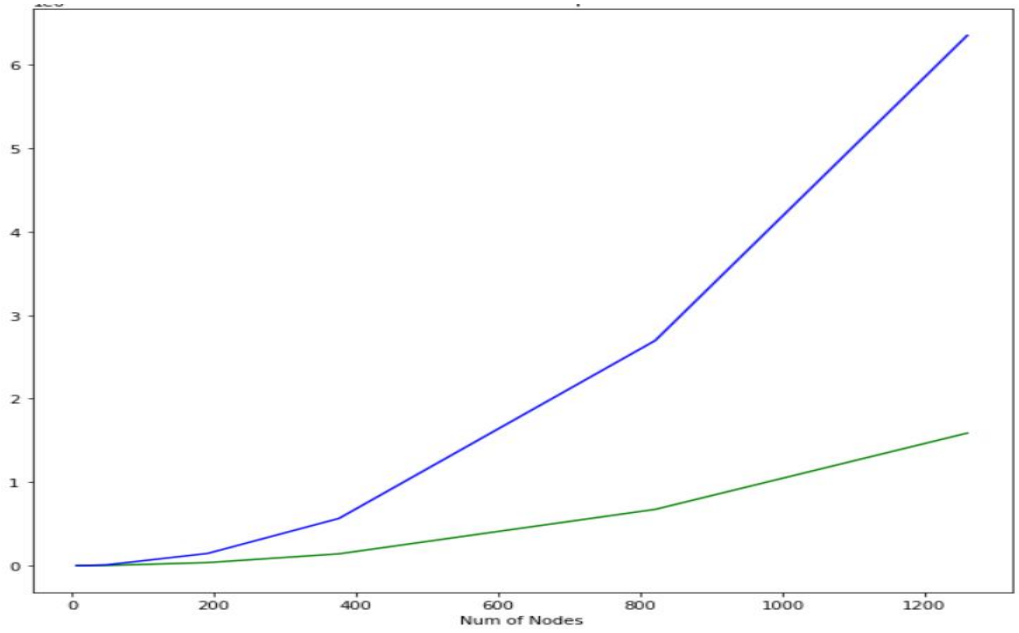
BFS(source, target): O(2V+E)
queue is empty queue data structure O(1)
visited is empty set data structure O(1)
enqueue start to queue O(1)
mark start visited O(1)
while queue is not empty: O(2V(E/2V+1))
node is dequeued element of queue O(1)
for neighbour in range of node.neigbours : O(E/2V)
if neighbour not in visited: O(1)
mark neighbour as visited O(1)
enqueue neighbour to queue O(1)
if node is equal to target: O(1)
break O(1)

- Lets think E is equal to V^2 in worst case.Time complexity would be O(2V+E) which E is (2V) ^2. And it is O(2V+4V^2) also O(4V^2) instead of O(V^2). Time complexity is quadruple.
- Lets think E is equal to1 in best case.Time complexity would be O(2V) instead of O(V). Time complexity is doubled.

Red is Best case original(Time comlexity is O(V)) and Yellow is Best case original with doubled num of nodes (Time comlexity is O(2V))

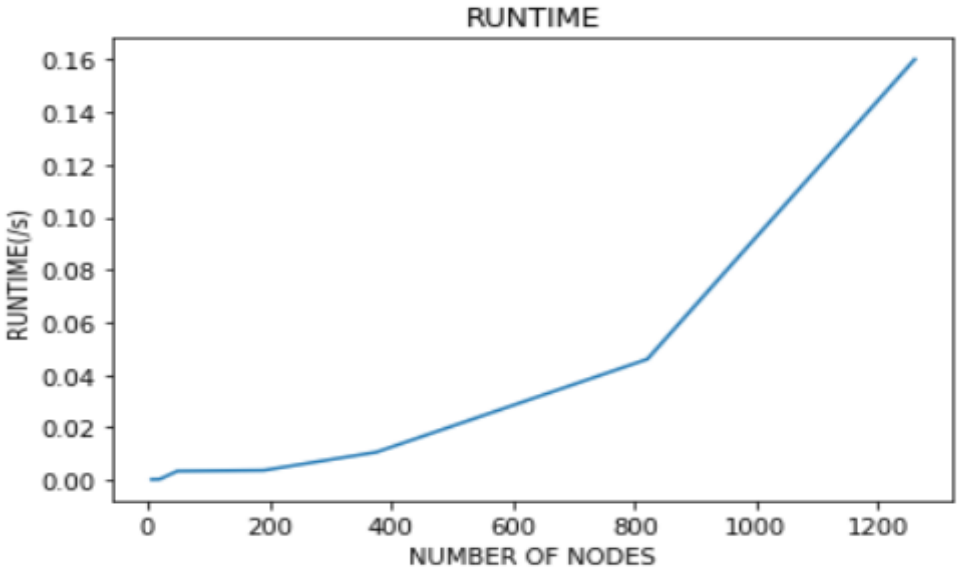


Green is Worst case original(Time comlexity is O(V^2)) and Blue is Worst case original with doubled num of nodes (Time comlexity is O(4V^2))



3-B-

NUMBER OF NODES	RUNTIME (/SECOND)
7	0.000146022
10	0.000167968
20	0.000227307
49	0.00333387
191	0.00363386
376	0.0105426
821	0.0459649
1260	0.160106


RUNTIME
RUNTIME(/s)
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
200
400
600
800
1000
1200
NUMBER OF NODES