PROJECT REPORT

Student 1: Preethi Subramanian **UTA ID** - 1002059233

Student 2: Sai Praneetha Katragadda **UTA ID** - 1002082493

Project Description:

Implement two graph search algorithms BFS(Breadth First Search using Queue data structure) and DFS(Depth First Search using Stack data structure) and compare their performances.

Pseudo Code:

A)Breadth First Search:

Input: A Graph G = (V, E) in Adjacency List, Source vertex(Starting vertex)
Output: Breadth First Traversal

Breadth_first_search (graph, source):
1.Initialize a list named visited to return the bfs traversal path from source node
2.Initialize an empty queue data structure to keep track of vertices that are to be explored next
3.Add the source vertex to the queue data structure
4.Remove the vertex from the front of the queue
5.While queue is not empty
6.The graph vertex is dequeued from the queue and stored as vertex variable
7.For each adjacent node/neighbour nodes present in the vertex's adjacency list

1. If the adjacent node is is not visited
2. Append the adjacent node to the visited list
3. Append the adjacent node to the end of the queue

```
def bfs(graph, source):
      visited = []
      queue = []
      queue.append(source)
      while len(queue):
         vertex = queue.pop()
         for v in graph.data[vertex]:
             if v not in visited:
               visited.append(v)
               queue.append(v)
      return visited
```

B)Depth First Search:

Input: A graph G=(V,E) in adjacency list and source vertex
Output: Depth First Traversal

Depth_First_Search(graph,source):
1.Initialize a list named visited to return the dfs traversal path from source node
2.Initialize a stack to keep track of vertices that are to be explored next
3.Since the source node is visited, add it to the stack
4.While stack is not empty
5.The vertex that is pushed last into the stack is popped out of the stack
6.If the vertex is not visited then
7.Add the vertex to the visited list
8.All the adjacent nodes are appended/pushed onto the stack
9.Explore the adjacent nodes of the vertex added into the list
10.Return the visited list which provides us the order of exploration of Depth First Search Traversal

```
def dfs(graph, source):
   visited = []
   stack = [source]
   # while stack:
   while len(stack):
       vertex = stack.pop()
       if vertex not in visited:
           visited.append(vertex)
           for adjacent_nodes in graph.data[vertex]:
               stack.append(adjacent_nodes)
   return visited
```

Runtime Analysis:

A)BFS:

In BFS,each vertex is enqueued and dequeued utmost twice since it is an undirected graph.The enqueue and dequeue operations take constant amount of time i.e O(1).Hence, the total queue operations would take O(V) where V is number of nodes/vertices.When the node is dequeued, the node is scanned across the adjacency list utmost once to get their neighbour nodes.The sum of all the adjacency lists is O(E).The total time taken to scan the lists is O(E).
As initialization takes O(V) and scanning takes O(E).The total time complexity of BFS is O(V+E).

B)DFS:

DFS examines each edge at most twice since it is an Undirected graph, one from each end of the node and stack supports push and pop operation in constant O(1) time.
A constant amount of time is performed per edge which takes O(m).
Initialization takes O(n) time. The running time for DFS would be O(m+2n) or O(V+2E) which is ~ O(V+E) where n =|V| and m =|E|.

Implementation:

A)Dataset Generation:

The dataset for the project is generated using Snap.Stanford module (Random Data Generator) listing the number of nodes and edges.

```
import snap

# Graph random generator
graph = snap.TUNGraph.New()    # TUN - Undirected graph creation
graph = snap.GenRndGnm(snap.TUNGraph,6000,30000) # 6000 edges
FOut = snap.TFOut("test_sample_undirected_6knodes_30k_edges.graph")
graph.Save(FOut)
FOut.Flush()
FIn = snap.TFIn("test_sample_undirected_6knodes_30k_edges.graph")
Undirected_Graph = snap.TUNGraph.Load(FIn)
Undirected_Graph.SaveEdgeList("test_sample_undirected_6knodes_30k_edges.txt",
"Save as tab-separated list of edges")
```

Here we are creating an undirected graph(single edge between the unordered pair of nodes). For instance, we are generating an undirected graph with 6000 nodes and 30000 edges using the random generator (GenRndGnm).
With Snap, we created the graph and saved it in the text format and later it is used to experiment and compare the performance of graph search algorithms.

B)BFS function:

BFS uses Queue data structure (First In First Out) as the algorithm explores the neighbour nodes by traversing the graph level by level starting from the source node. Then it will be traversed in the direction of the neighbour nodes on the following level.While exploring the neighbour nodes, BFS goes wider.

Initialize a list named visited to return the BFS traversal path.Then,Initialize an empty queue data structure, to keep track of vertices that are to be explored next.Add the source vertex to the queue data structure,since it is visited.While queue is not empty,the graph vertex is dequeued from the queue and stored as a vertex.For each edges/adjacent node present in the vertex's adjacency list,if the adjacent node is is not visited.Then,append the adjacent node to the visited list and to the end of the queue.

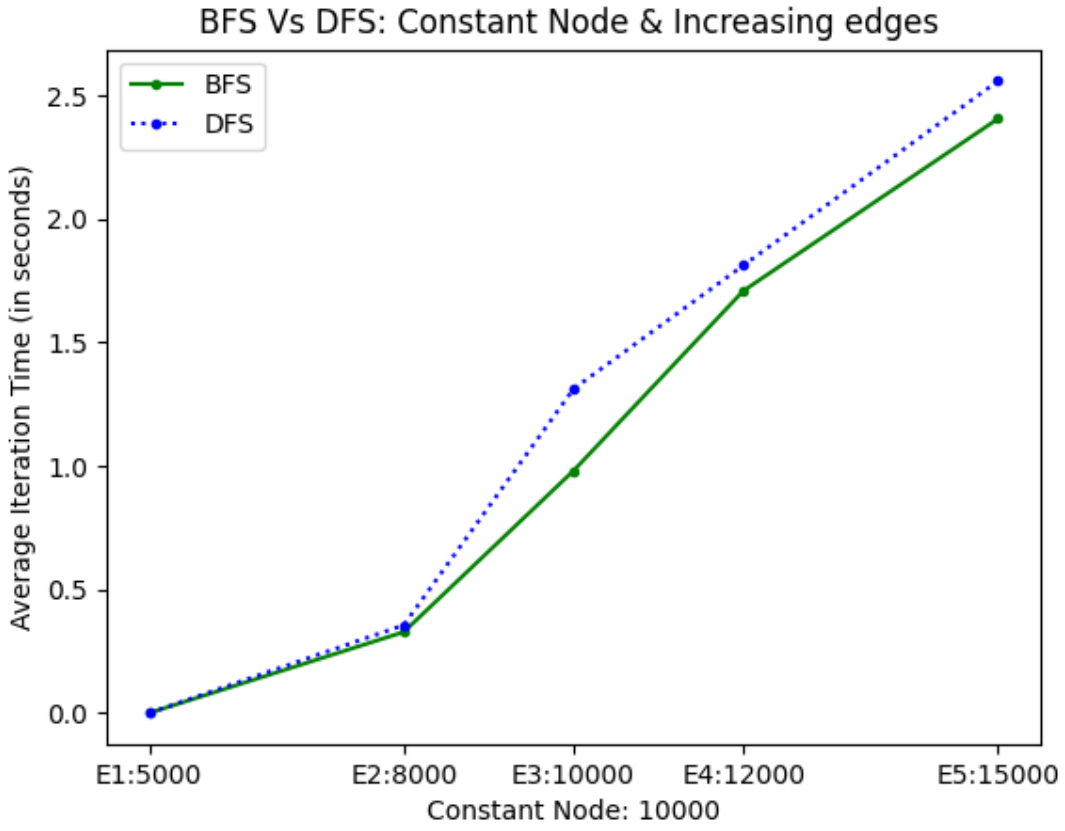
C)DFS function:

DFS uses stack data structure (Last In First Out) as the algorithm explores the neighbour nodes by traversing the graph layer by layer starting from the source node. Then it will be traversed in the direction of the neighbour nodes on the following layer.While exploring the neighbour nodes, DFS goes deeper.

Initialize a list named visited to return DFS traversal path.Then,initialize a stack data structure to keep a track of vertices that are to be explored next.Then add the source vertex to the top of the stack,since it is visited.If the stack is not empty,the vertex that is pushed last into the stack is popped out of the stack.If the vertex is not visited then,add the vertex to the visited list.All the adjacent nodes are appended/pushed onto the stack.Later on,explore the adjacent nodes of the vertex added into the list.We return the visited list which provides us the order of exploration of Depth First Search Traversal.

D)Plotting the Average time taken for the BFS and DFS against constant number of nodes and increasing edges

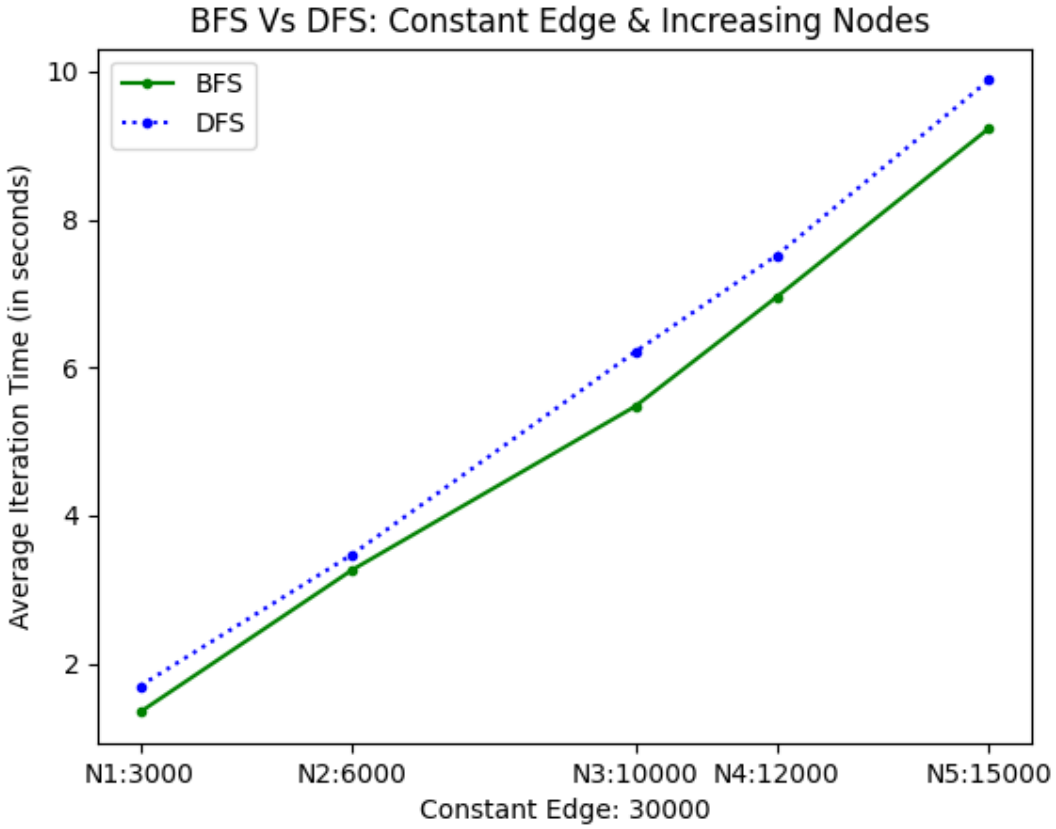
The BFS and DFS functions are called at least 5 times and the average time is taken to plot the graph with a constant number of nodes and increasing edges.



The X- axis has a constant number of nodes and increasing edges. Whereas, Y-axis takes the average iteration time.
Since, the time complexity of BFS and DFS are the same, which is O(V+E), the same is depicted in the graph. The plots of DFS and BFS are slightly increasing with the increase in the number of edges.

E)Plotting the Average time taken for the Bfs and Dfs against constant number of edges and increasing nodes

The BFS and DFS functions are called at least 5 times and the average time is taken to plot the graph with a constant number of edges and increasing number of nodes.



The X- axis has a constant number of edges and increasing nodes. Whereas, Y-axis takes the average iteration time.
Since, the time complexity of BFS and DFS are the same, which is O(V+E), the same is depicted in the graph. The plots of DFS and BFS are slightly increasing with the increase in the number of nodes.

Experiments:

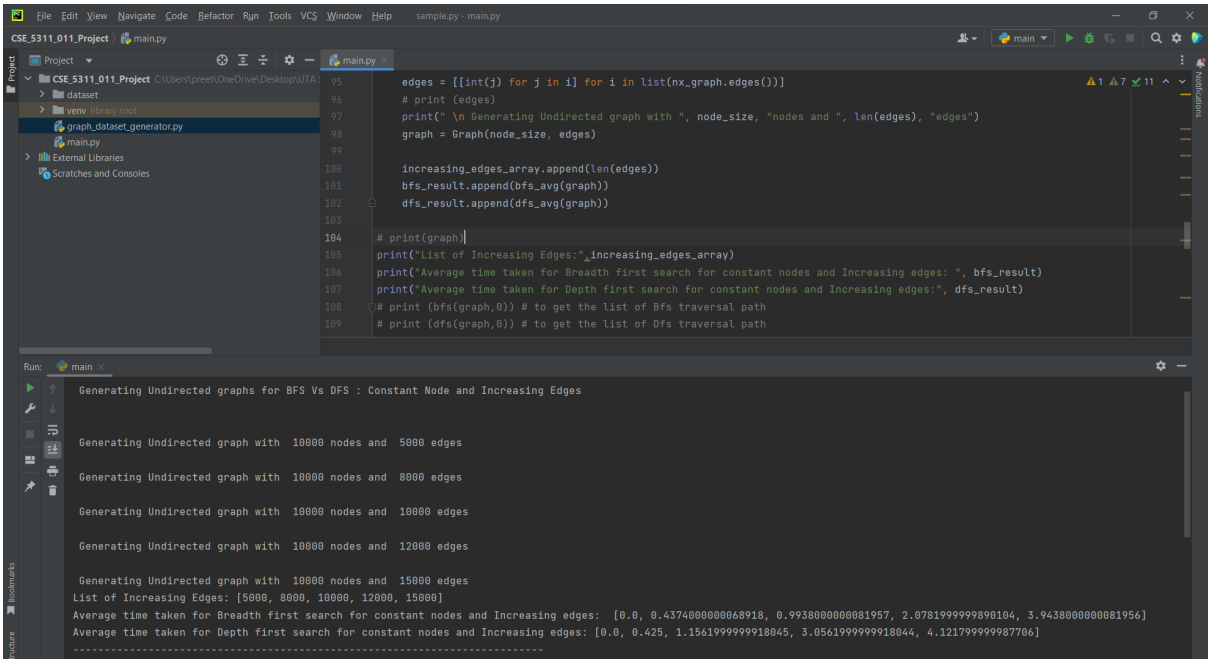
The experimentation is done in two phases.

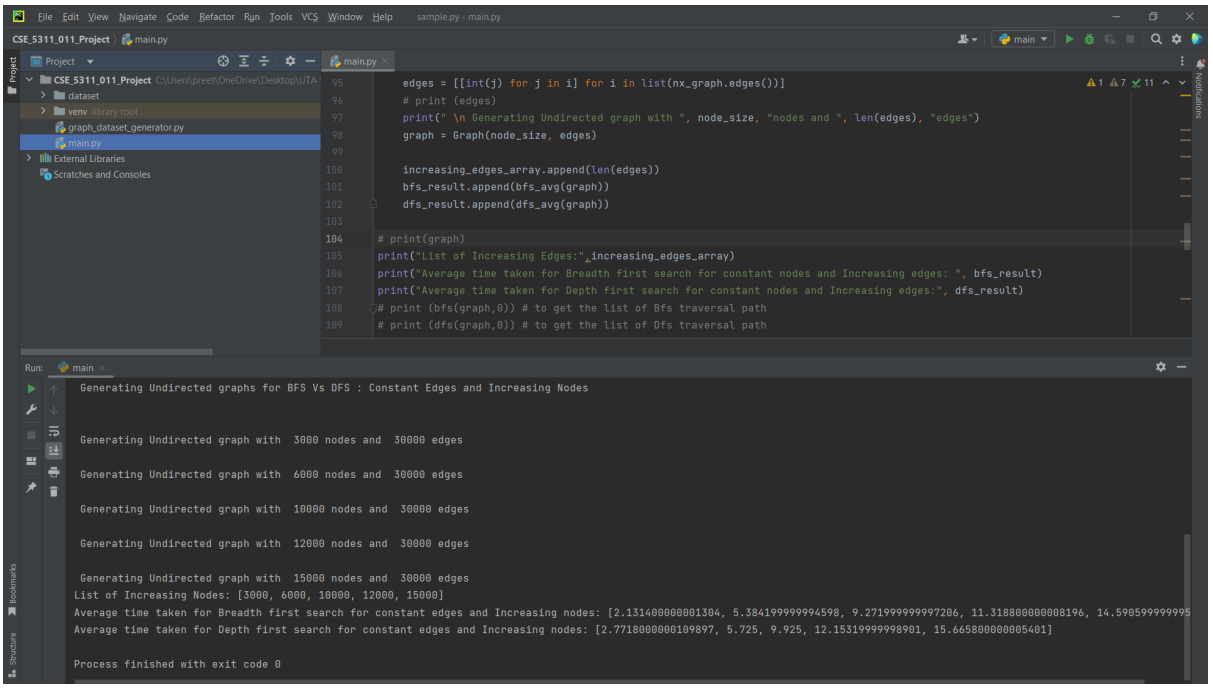
1. Keeping constant number of nodes and increase the number of edges gradually
   - ☐ We have chosen the number of nodes to be constant is 10000 and the edges starting from 5000, 8000, 10000, 12000, 15000.
   - ☐ We tried to experiment the behaviour of graphs with different sets of datasets and we were able to identify that the performance of both the plots i.e BFS and DFS are similar with constant nodes and increasing number of edges.

2.Keeping constant number of edges and increase the number of nodes gradually

We have chosen the number of edges to be constant as 30000 and the nodes starting from 3000, 6000, 10000, 12000, 15000.

Output Snapshots:





Learnings Outcomes:

1.Time Complexity:

From the plotting and the flow of BFS and DFS, we understood that the performance of BFS and DFS are similar as their time complexity is O(V+E).

- **BFS:**

In BFS,each vertex is enqueued and dequeued utmost twice since it is an undirected graph.The enqueue and dequeue operations take constant amount of time i.e O(1).Hence, the total queue operations would take O(V) where V is number of nodes/vertices.When the node is dequeued, the node is scanned across the adjacency list utmost once to get their neighbour nodes.The sum of all the adjacency lists is O(E).The total time taken to scan the lists is O(E).
As initialization takes O(V) and scanning takes O(E).The total time complexity of BFS is O(V+E).

- **DFS:**

DFS examines each edge at most twice since it is an Undirected graph, one from each end of the node and stack supports push and pop operation in O(1) time.
A constant amount of time is performed per edge which takes O(m).
Initialization takes O(n) time.
The running time for DFS would be O(m+2n) or O(V+2E) which is ~ O(V+E) where n=|V| and m =|E|.

2. Space Complexity:

- **BFS:**

The space complexity of BFS is O(V) as in the worst case it corresponds to the maximum number of vertices present in the graph that may be stored in the queue.

- **DFS:**

The space complexity of DFS is O(V) as in the worst case it corresponds to the maximum number of vertices present in the graph that may be stored in the stack.

3.Plotting Of Graphs:

The matplotlib library is used to plot the graphs.In plotting, X-axis would be the nodes and edges consisting of two different scenarios

(1) With increasing number of edges and constant number of nodes

(2) With increasing number of nodes and constant number of edges

And Y-axis would depict the average iteration time for the datasets.
The iteration time is captured using the monotonic() function.

Challenges faced:

- Initially we faced some challenges in generating the dataset using a random generator. Later, we studied the Snap.Stanford module for generating a large dataset for the graph.

- While experimenting with the wide range of data (in millions), the loading of a huge dataset was taking a little longer time, thus the nodes and edges were decreased to thousands and later we generated the plotting for the above scenarios.

Results :

Comparing the performances of the Breadth first search and Depth first search graph search algorithms, they are not much different as the time complexity discussed in run time analysis is O(V+E) where V is the vertices and E is the edges of the graph.
Although they seem to perform very much similar from the plots, it is evident that BFS performs slightly better than DFS. The reason is Breadth first search tries to reach the neighbour nodes with minimum number of edges from the source vertex whereas Depth first search might be a little slower as it traverses more edges to reach the vertex from the source node as it traverses deeper . In general, when the source node is closer to the target node, then Breadth first search performs better when compared to depth first search as it tries to reach the destination/target node with minimal edges.

References:

SNAP: SNAP Tutorial (stanford.edu)
GenRMat — Snap.py 6.0 documentation (stanford.edu)
Difference between BFS and DFS - GeeksforGeeks

DONE BY:

Preethi Subramanian (1002059233)
Sai Praneetha Katragadda(1002082493)