# **Objectives**

The objective of this project was to implement an undirected graph using adjacency lists and a breadth-first search algorithm on the graph using a high-level programming language. The nodes of the undirected graph had to be read from a text file, then added into an undirected graph via a linked list of connected nodes to each node. The text files that were given in this assignment were “mediumG.txt” and “largeG.txt.” Each listed the number of vertices in line one and the number of edges in line two. The “mediumG” file contains 1275 lines, representing 250 nodes and 1273 edges. While the “largeG” file contains 7586065 lines, representing 1 million nodes and 7586063 edges. The graph had to then be verified by a print function that displays the adjacency list of each node in the graph. The breadth-first search algorithm must then search for the path that can access all other nodes and then print said path node-by-node.

# **Program Design**

To implement the required functionality of the assignment, two class files were developed: a HW5 class containing the driver code and a Graph class containing the functions necessary to construct the graph, print the adjacency lists, and perform breadth-first search. The following functions are contained within the two classes:

**Graph()**

This constructor initializes the graph object when called. The vertices attribute is assigned the number of nodes that the constructor is passed as a parameter. The edges is initialized as 0 because the number of edges is iterated forward every time the addEdge() function is called. A new array list object that contains array lists is also initialized to contain array lists that represent the adjacency list of each node. A for loop is used to add a new array list for each node.

**addEdge()**

This function is passed two integers, node1 and node2, that are read from the text file in main. It then calls the get() function on adjList to get the current adjacency list for that each node and the add() function on that list to each node to the others adjacency list. This adds an edge in the graph between both nodes in both directions. The nodes are added to each other’s list because the graph is supposed to be unidirectional.

**printAdjList()**

This function is responsible for representing the graph as an adjacency list for each node. The function first prints a statement containing the number of edges and vertices. Next, it iterates through the list of adjacency lists for each node by iterating from 0 to the number of vertices in the graph, which corresponds to the number of lists contained within the adjacency list. A nested for loop then iterates through each neighbor contained in the current adjacency list. The function then appends the neighbor to the string builder object. Once the nested for loop exits, a newline is appended to the string. After the outer for loop exits, the function prints the string.

**getVertices(), getEdges(), getAdjList()**

These getter functions return the vertices integer, edges integer, and adjacency list array list object from the graph object. They are called throughout the other functions in the Graph and HW5 class.

**setVertices() and setEdges()**

These functions set the vertices integer and the edges integer of the graph object initialized in the HW5 class.

**BFS()**

The BFS() function is passed an integer called startNode from which it performs breadth-first search. The function accomplishes this using an array of Boolean values corresponding to each vertex and a queue of integers used to queue and print neighbors. The function first adds the starting node to the queue and assigns it as visited in the Boolean matrix as it is the starting node and must be visited. While the queue is not empty, each node is first dequeued then printed to indicated which node is currently being processed. A for loop then iterate through each neighbor contained in the adjacency list of that node. If the node has not been visited, the node is marked as visited and it is added to the queue. This continues until the while loop exits due to the queue being empty. A flag variable called hasUnvisitedNodes is then initialized as false, and a for loop iterates through the vertices again, this time checking their index in the Boolean matrix to see if they have been visited. If they have not, the flag is set to true, and the loop exits. A final if statement checks if the flag is true, printing a statement that some nodes were inaccessible.

**main()**

The driver code contained in main() is responsible for reading the text file and then calling the above functions from the Graph class to achieve the codes desired functionality. First the file name is initialized as a string. The count() function is then called on the lines in the file to determine how many lines, and therefore how many nodes should exist in the graph when it is constructed. The constructor from the Graph class is then called. It is passed the numberOfLines variable mentioned before, only casted to an integer. A buffered reader object is then initialized and passed the filename variable. A while loop continues as long the line that the reader reads is not null, meaning it stops at the end of the file. An integer lineCount is also initialized and then iterated forward with each iteration of the loop to keep track of the current line. If the current line is the first one, the setVertices() function is called from the graph which sets the number of vertices attribute in the graph object to the number specified at the beginning of the text file. The second if statement checks if the current line is two, continuing if so. This is because the number of edges is determined by the addEdge() function. Finally, the printAdjList() and BFS() functions are called from the Graph class. The nanoTime() function from the System class records the time before and after execution of the BFS() function, then calculates the difference and prints it in different units of time. This was used in testing to compare the time complexity of the algorithm among different text file sizes and starting nodes.

Code Screenshots:

A screen shot of a computer program

Description automatically generated

Figure : HW5.java

A screenshot of a computer program

Description automatically generated

Figure : Graph.java (1)

A screen shot of a computer program

Description automatically generated

Figure : Graph.java (2)

# **Testing**

Testing consisted of executing the code on both files and recording the execution time of the BFS() algorithm on different starting nodes and verifying printAdjList() outputs.

Testing Screenshots:

**A screenshot of a computer screen

Description automatically generated**

Figure : BFS() on 10 in mediumG.txt

A screenshot of a computer screen

Description automatically generated

Figure : BFS() on 100 in mediumG.txt

A black background with white numbers

Description automatically generated

Figure : BFS() on 10 in largeG.txt

A black background with white numbers

Description automatically generated

Figure : BFS() on 100 in largeG.txt

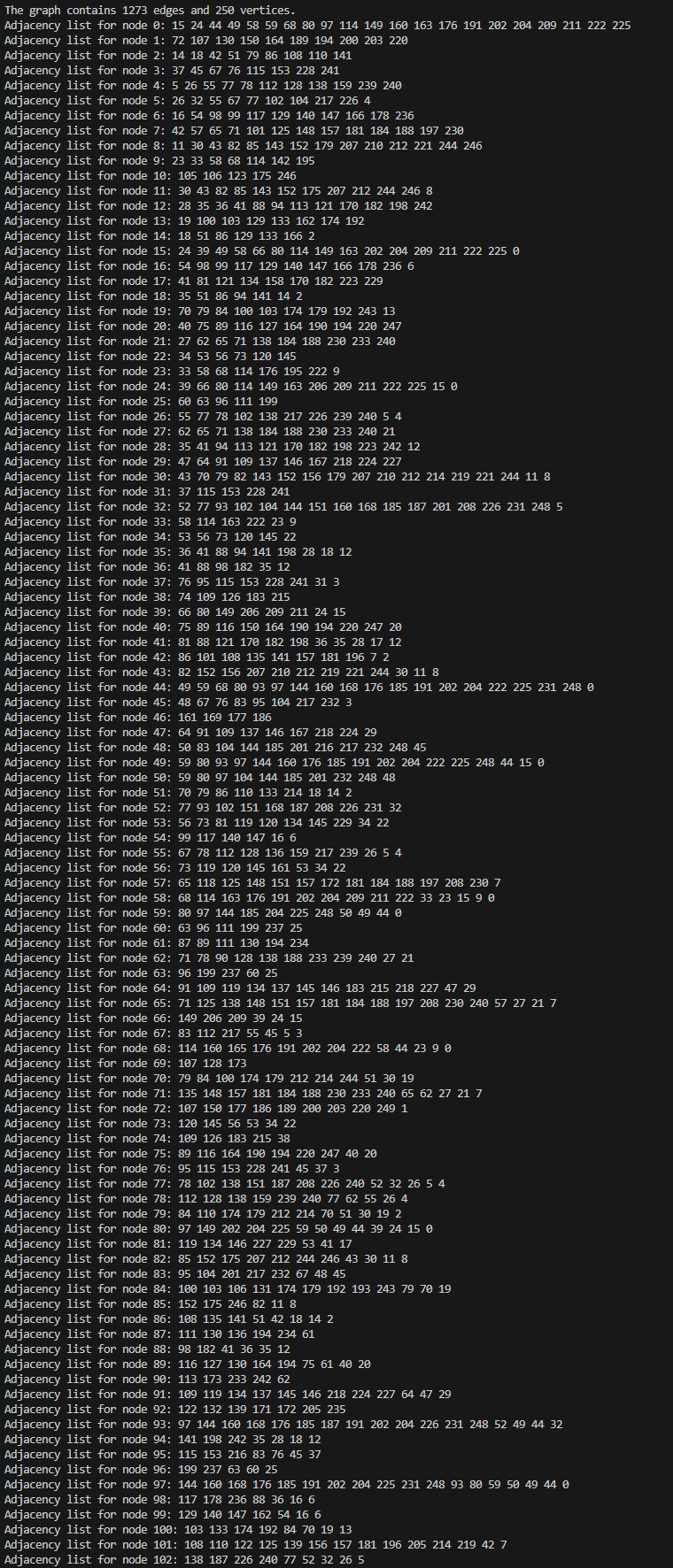
****

Figure : printAdjList() output from mediumG.txt

A screen shot of a computer screen

Description automatically generated

Figure : printAdjList() output from largeG.txt

The graph above displays the difference in execution time of the breadth-first search algorithm when it is called on different starting nodes within the medium and large text files. The largeG text file represents four thousand times as many nodes within the constructed graph. Therefore, the algorithm must traverse many more edges between these nodes to find the desired path. The time complexity of this algorithm is represented by O(V+E) where V is the number of vertexes (or nodes) and E is the number of edges. This is supported by the above graph, as the number of represented vertices and edges in largeG is many more than that of medium, causing the time complexity of the algorithm to increase dramatically. There is also a very slight difference in the execution time between the two different starting nodes in each text file. This could be due to the resulting differences in the path taken by the algorithm which results in a minor change in the execution time for the algorithm in mediumG and a slightly larger but still ultimately negligible change in execution time in largeG.

# **Sources:**

<https://chat.openai.com/share/8b4c2e60-b4e3-4b2a-909b-4a3300ec4287>

<https://chat.openai.com/share/3615b67f-45b3-411b-80d5-4ac6984224e1>

<https://www.geeksforgeeks.org/convert-adjacency-matrix-to-adjacency-list-representation-of-graph/>

<https://www.youtube.com/watch?v=X1LdtRW88c0>

<https://stackoverflow.com/questions/26448352/counting-the-number-of-lines-in-a-text-file-java>

<https://stackoverflow.com/questions/5262308/how-do-implement-a-breadth-first-traversal>