# **Objectives**

The objective of this project was to implement a weighted graph using adjacency lists, as well as Prim’s algorithm to find the minimum search tree of this graph. The nodes of the weighted 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 “tinyDG,” “mediumDG.txt,” “largeDG.txt,” and “XtralargerDG.txt.” Each listed the number of vertices in line one and the number of edges in line two. Each line after the first two contains two whole numbers and one decimal number with spaces in between each of them. The two whole numbers represent the connected nodes of an edge, and the decimal number represents the weight of the edge. Several lines also contain an inconsistent number of spaces.

# **Program Design**

To implement the required functionality for this assignment, three classes were created: a HW7 class containing the driver code, a WeightedGraph class containing the functions necessary to construct the graph, print the adjacency list, and implement Prim’s algorithm, and lastly the Edge class which contains the constructor for each edge. The following functions are contained within these three classes:

**WeightedGraph()**

This constructor initializes a WeightedGraph 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, as well as one float, weight, 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 the first node. A new Edge object is created using the parameters passed into it from main() and added to this adjacency list.

**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. The getWeight() function is also called on each edge within the nested for loop to determine their weight, which is also appended 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.

**findMST()**

The findMST() function implements Prim’s algorithm to find the minimum search tree (MST) of the constructed graph. First a new WeightedGraph object, mst, is initialized, as well as a Boolean array used to determine which vertices have been visited, and a PriorityQueue object that will prioritize the edges with the lowest weights. The MST is started from the first node in the graph at index 0. The node is marked as visited using the Boolean array and then a for loop iterates through all edges that are adjacent to the first node, adding them to the priority queue. A while loop continues until the maximum number of edges in an MST have been added to the graph (number of vertices – 1) and the priority queue is empty. While the loop continues, the edge with the lowest weight is removed from the quest and the attributes are retrieved using the getters in the Edge class. If one of the nodes connected by the edge is marked as visited and the other is not, an edge is added between them using the retrieved attributes in the new mst object and the unvisited node is marked as visited, as it is now included in the MST. A for loop then iterates through the adjacent edges, checking if the endpoints of those edges are visited, adding them to the priority queue if not.

**Edge()**

This constructor initializes the Edge objects when called. The Edge objects contain three attributes: the first node to be connected (u), the second node to be connected (v), and the weight of the edge (weight). There are also a series of getters for these attributes contained within the Edge class. This class also implements the Comparable interface with a function that overrides compareTo() that allows the float values of the weights of the edges to be compared in Prim’s algorithm.

**main()**

This driver function is responsible for reading the text file and calling the functions necessary to build the weighted graph, find the minimum search tree, and printing the results. First the file name is initialized as a string, the number of lines as a long, the weighted graph as a WeightedGraph object, as well as a BufferedReader object and a line counter as an integer to properly iterate through the text file. Inside a while loop that continues if there is a next line for the buffered reader to read, a line counter variable is iterated forward to keep track of what line the reader is currently on, then a series of if statements determine the next operation. If the current line is the first line, this means that the reader has read the number of vertices, so an integer is parsed from the line and the setVertices() function is called to correctly set the graph’s number of vertices. If the reader has read the second line, then the function simply continues, because this line contains the number of edges, and the addEdge() function is responsible for setting the number of edges in the graph. If the current line being read is not the first or second line, then it must represent an edge. So, the line is formatted to remove extra spaces using a regex and split at the space between each number into an array of strings. The integers and floats are parsed out of these strings and assigned to the first node, second node, and weight of the represented edge. The addEdge() function is called on these parameters to add the edge to the weighted graph. This continues until all lines of the text file have been read and all edges have been added to the graph. After the while loop exits and the reader is closed, the printAdjList() function is called to print the adjacency list of the newly created weighted graph. After this, the findMST() function is called to find the minimum search tree of the graph using Prim’s algorithm. The system time is recorded in nanoseconds before and after the execution of this function, and the difference is calculated and printed in different units of time for testing purposes. The adjacency list of the MST is also printed using printAdjList().

**Code Screenshots:**

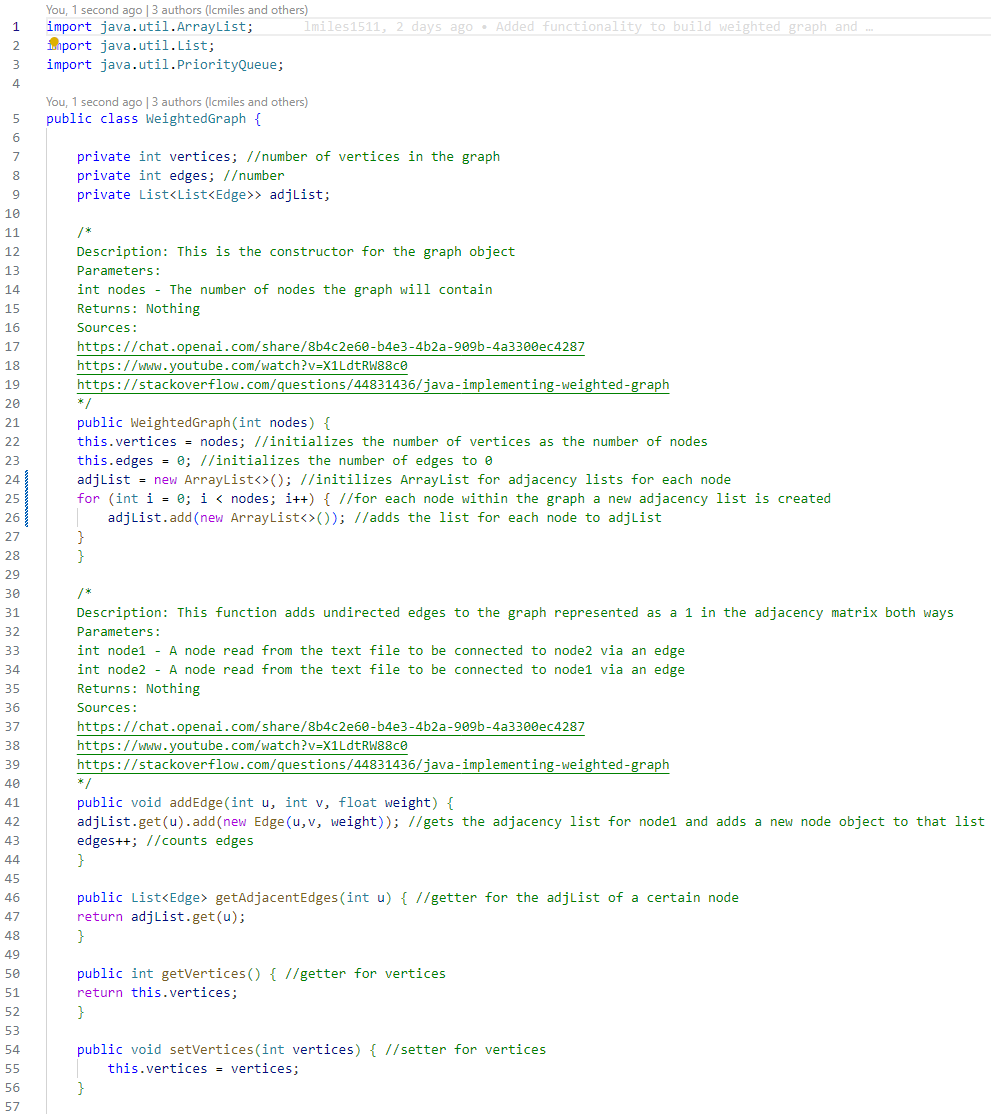


Figure 1: WeightedGraph.java

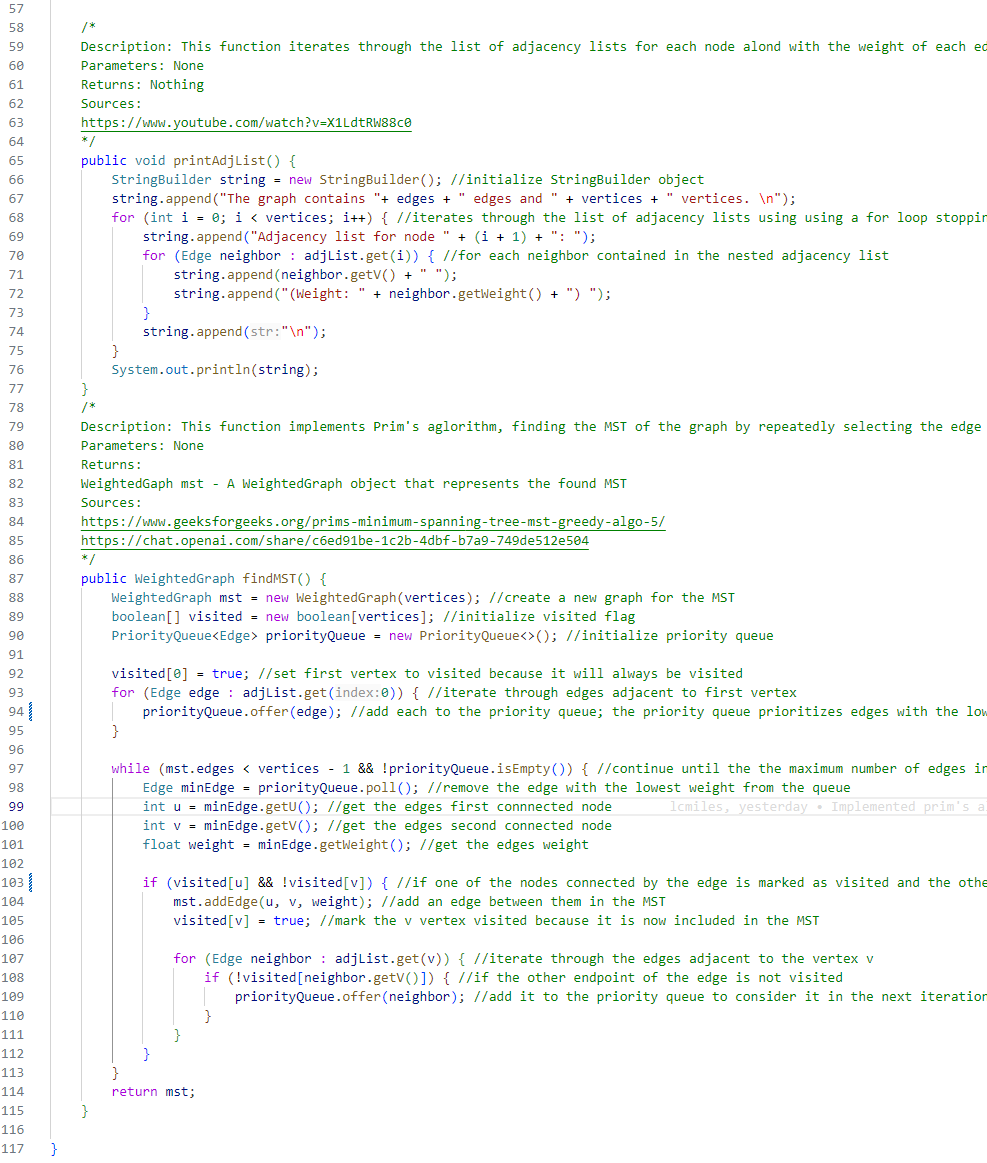


Figure 2: WeightedGraph.java



Figure 3: Edge.java

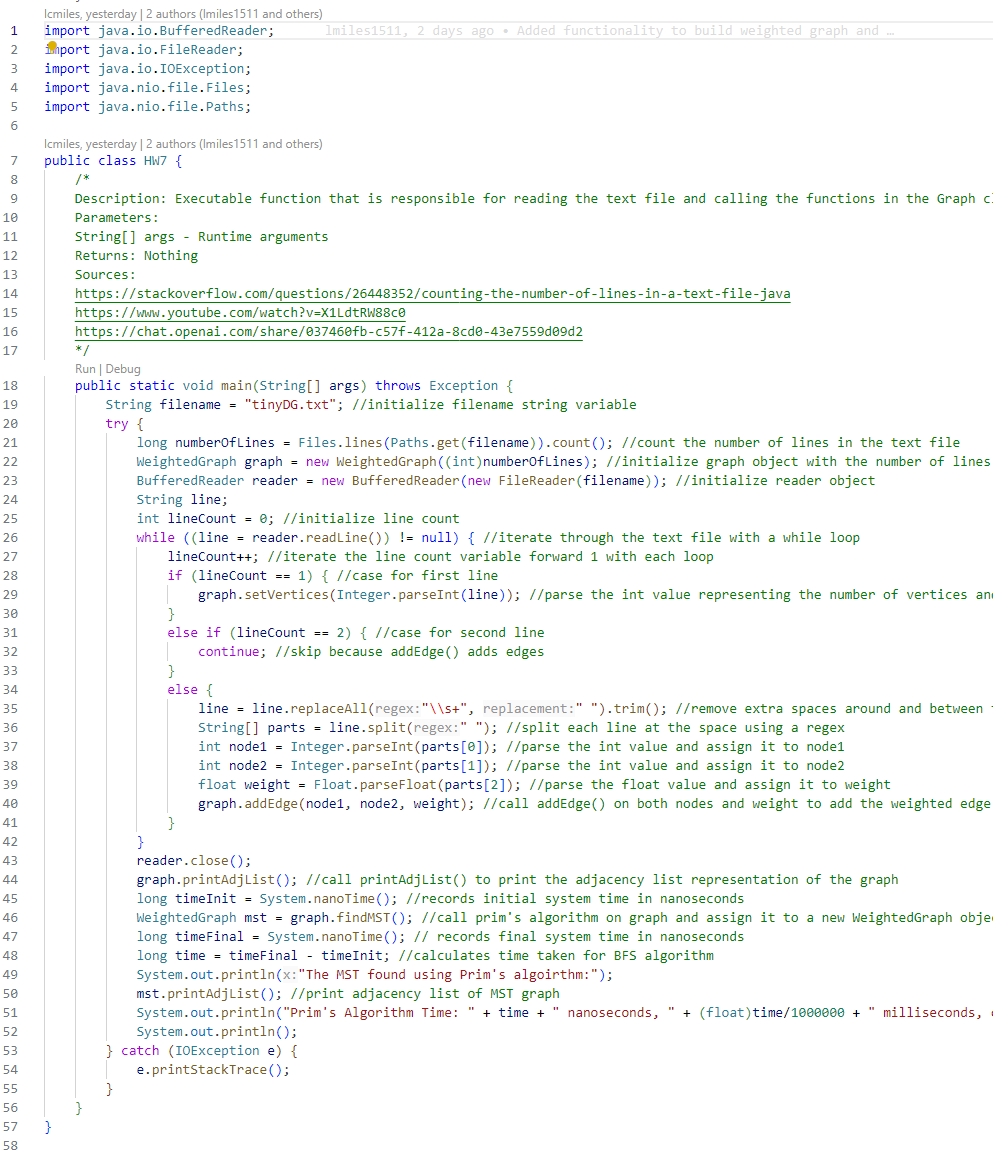


Figure 4: HW7.java

# **Testing**

Testing involved executing the code with on the following text files: “tinyDG,” “mediumDG.txt,” “largeDG.txt,” and “XtralargerDG.txt.” The execution times and resulting graphs of the functions were recorded.

**Testing Screenshots:**

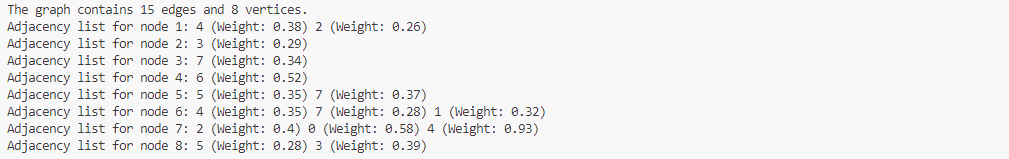
****

Figure 5: Resulting weighted graph from tinyDG.txt

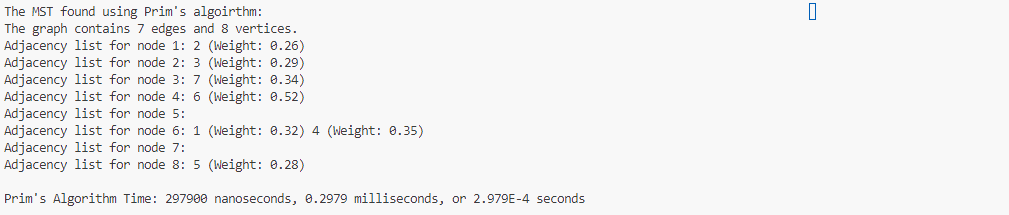
****

Figure 6: Resulting MST from tinyDG.txt

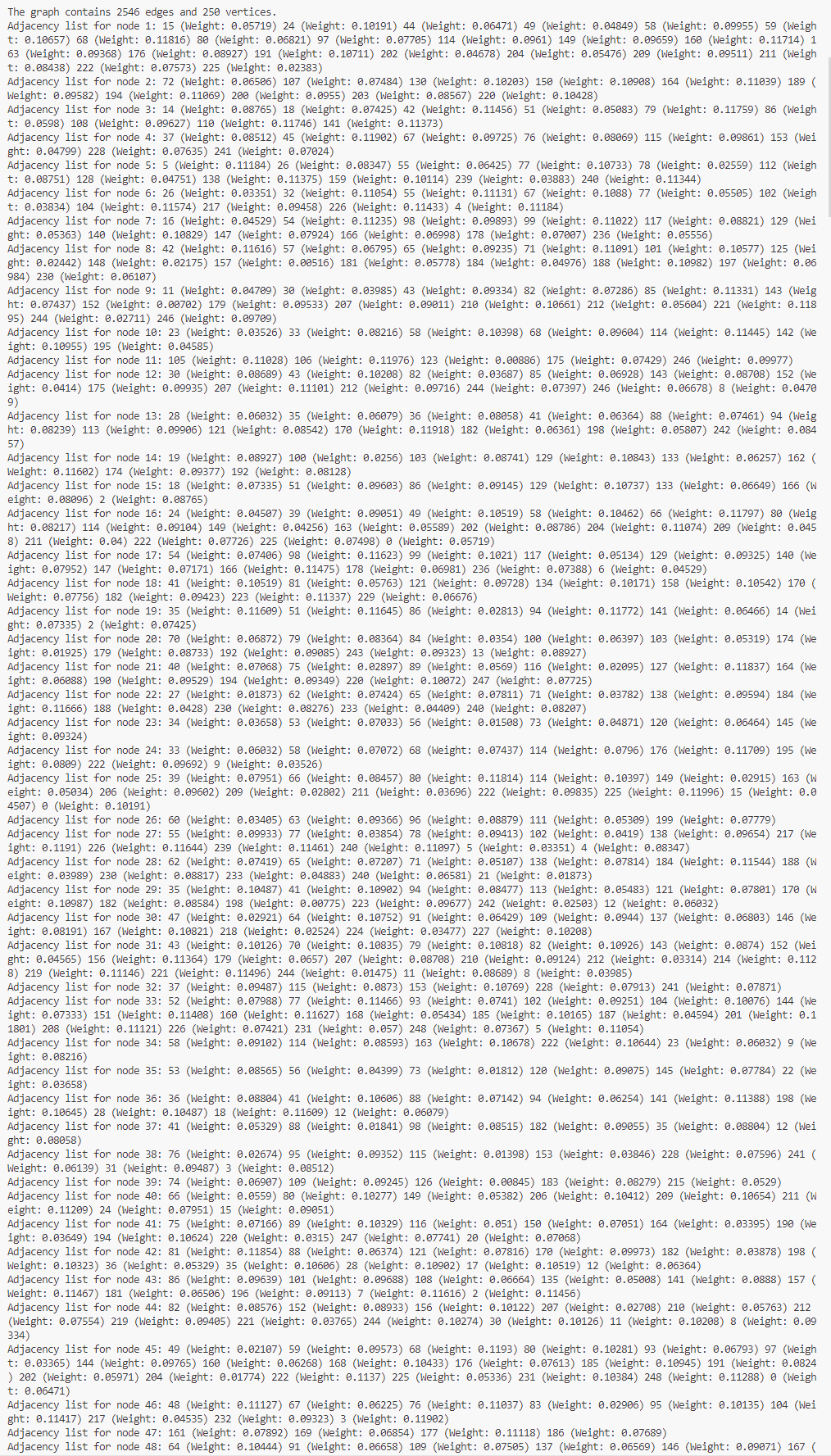
****

Figure 7: Resulting weighted graph from mediumDG.txt

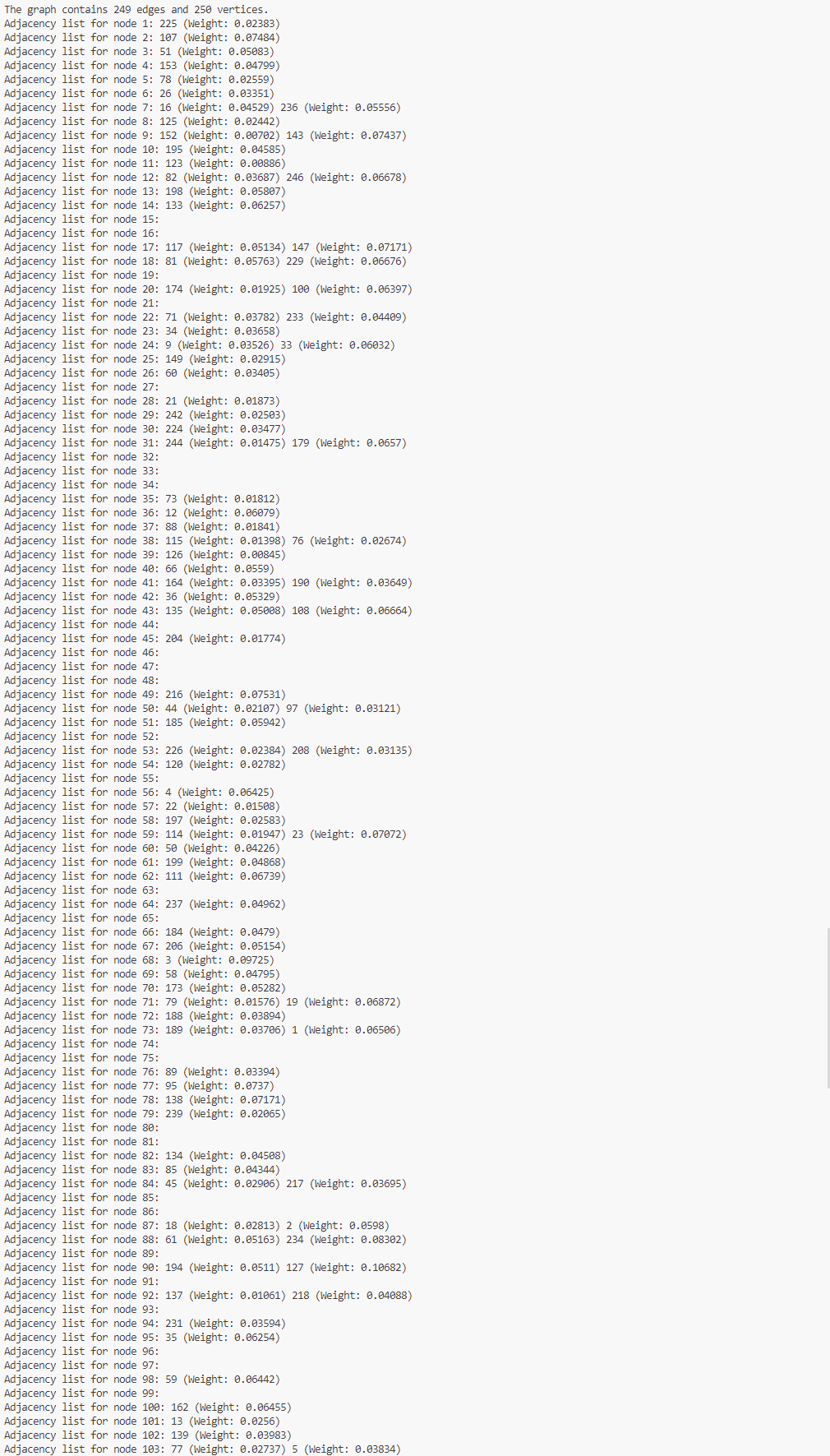
****

Figure 8: Resulting MST from mediumDG.txt



Figure 9: Resulting weighted graph from largeDG.txt

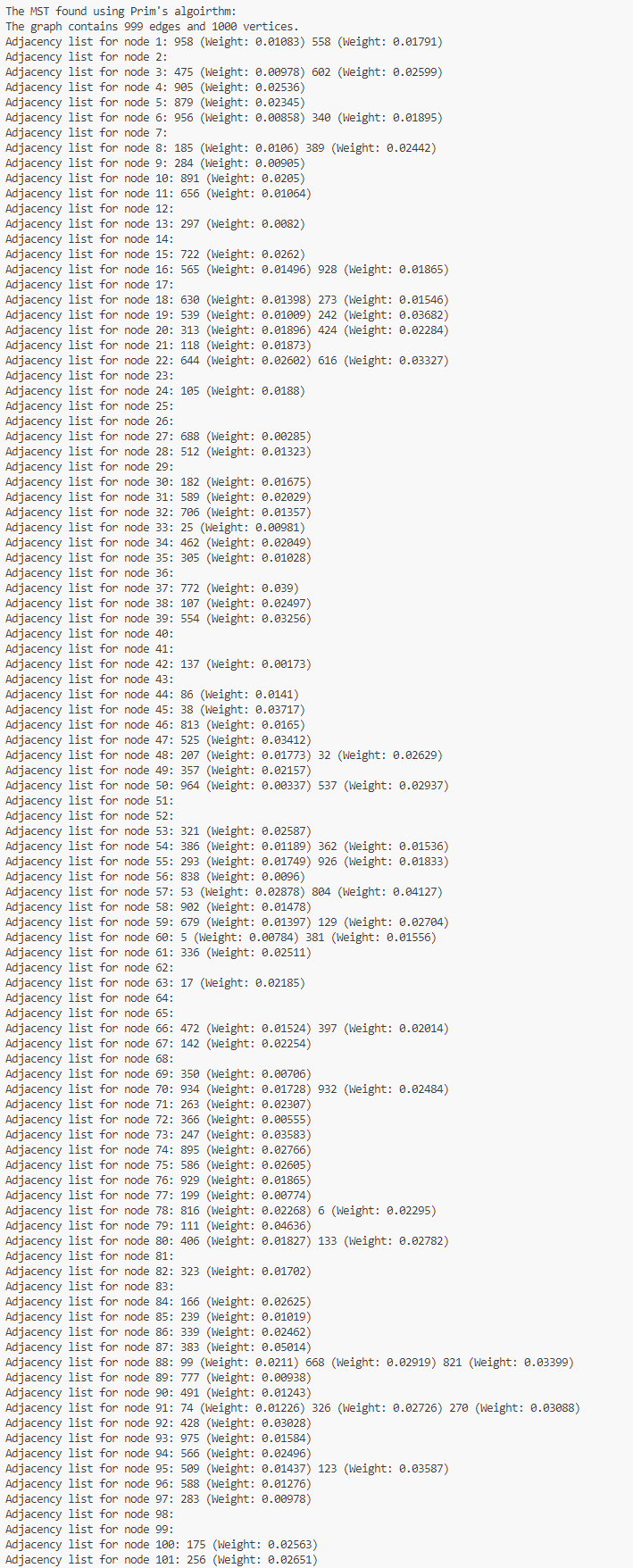


Figure 10: Resulting MST from largeDG.txt



Figure 11: Resulting weighted graph from XtralargeDG.txt

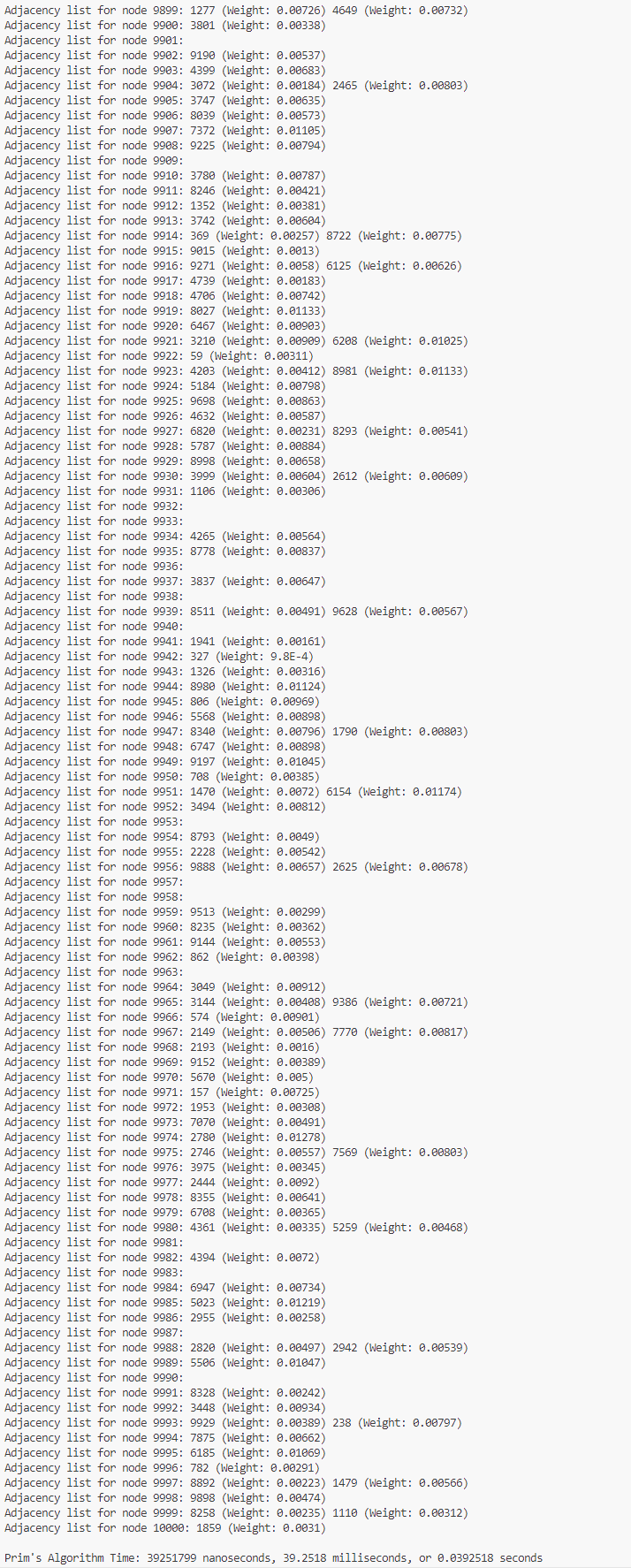
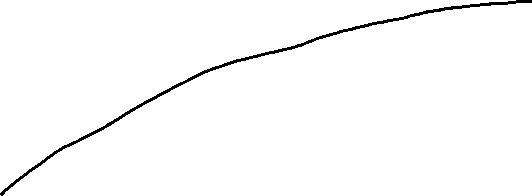


Figure 12: Resulting MST from XtralargeDG.txt



The graph above shows the execution time of the findMST() function that implements Prim’s algorithm. This algorithm is supposed to have a time complexity of O(E\*log(V)) where E is the number of edges and V is the number of vertices, or nodes. This is supported by the above graph as it has a vaguely logarithmic trendline.

# **Sources**

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

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

<https://chat.openai.com/share/037460fb-c57f-412a-8cd0-43e7559d09d2>

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

<https://stackoverflow.com/questions/44831436/java-implementing-weighted-graph>

<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

<https://chat.openai.com/share/c6ed91be-1c2b-4dbf-b7a9-749de512e504>