IT251 – DATA STRUCTURES & ALGORITHMS II

ASSIGNMENT 2

Name: **Sachin Prasanna**
Roll No.: **211IT058**

Question Statements:

1. Submit your analysis of Dijkstra’s algorithm in a one-pager
Hint: Pay attention to the ADT Q

Ans: Refer Dijkstra’s Algorithm Analysis PDF.

2. Implement Dijkstra algorithm in a language of your preference with all the General structure (Init, Relax) and core algorithm steps as explained in the class
Extend the program to

- Do analysis of the Dijkstra algorithm as per your implementation
- Support the theoretical analysis you submitted and
- Give proof through the program o/p that you have the fastest running time

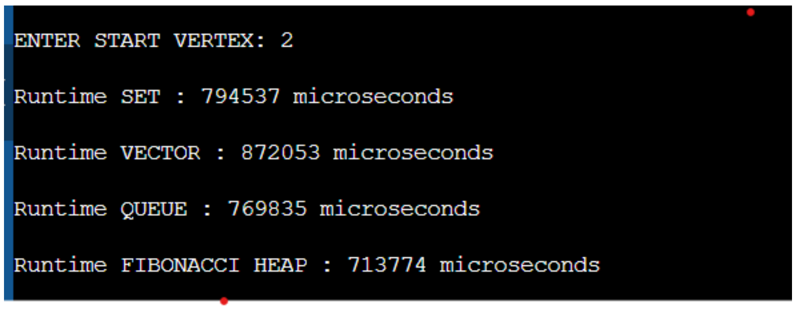
Ans: Refer Dijkstra’s Algorithm Analysis PDF for general outline.

Graph inputted:

1)

A graph of 100 nodes, where all nodes were connected to each other was first inputted. This is a very densely connected graph. The outputs were as follows for the time taken to complete Dijkstra's algorithm:

Output:



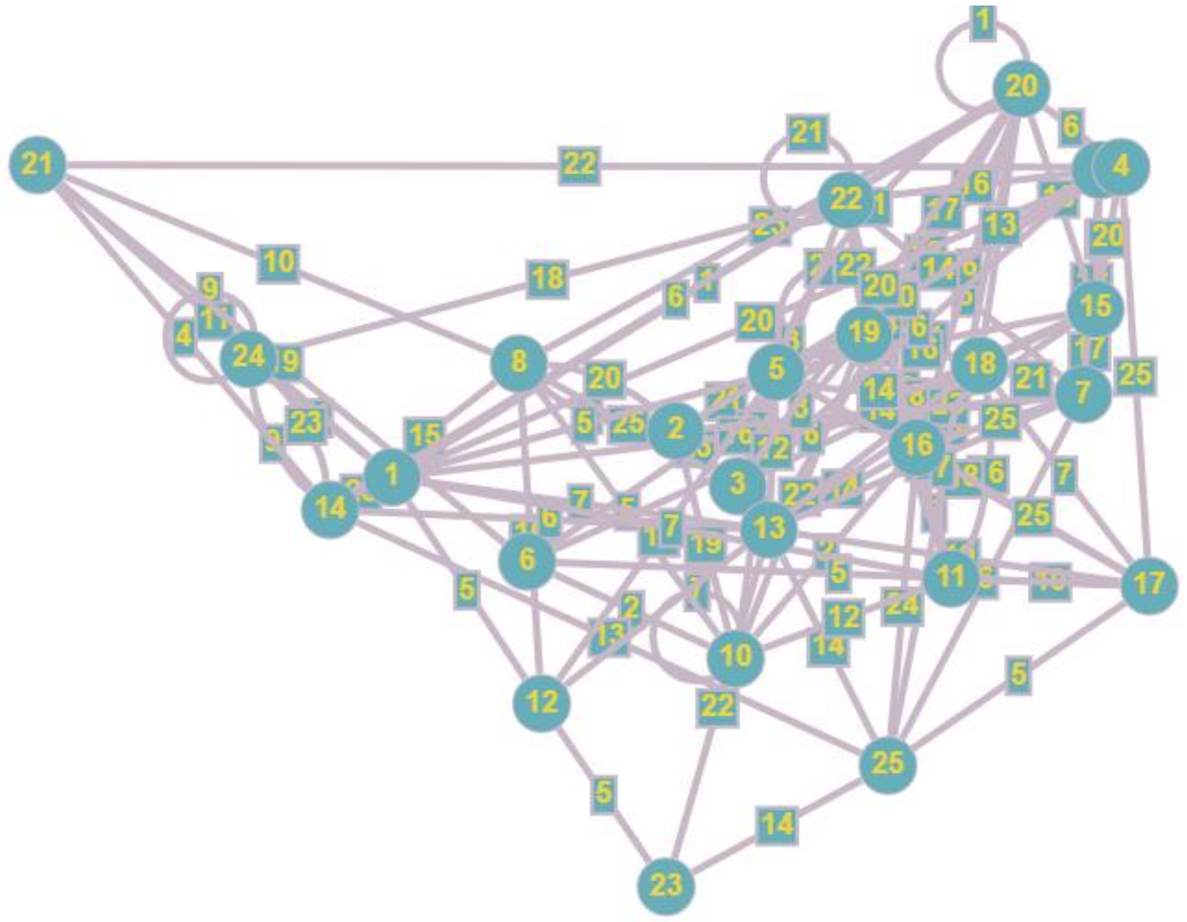
```
ENTER START VERTEX: 2

Runtime SET : 794537 microseconds

Runtime VECTOR : 872053 microseconds

Runtime QUEUE : 769835 microseconds

Runtime FIBONACCI HEAP : 713774 microseconds
```

The Fibonacci Heap and Priority Queue were relatively quicker in this case, than the other two data structures.

2)

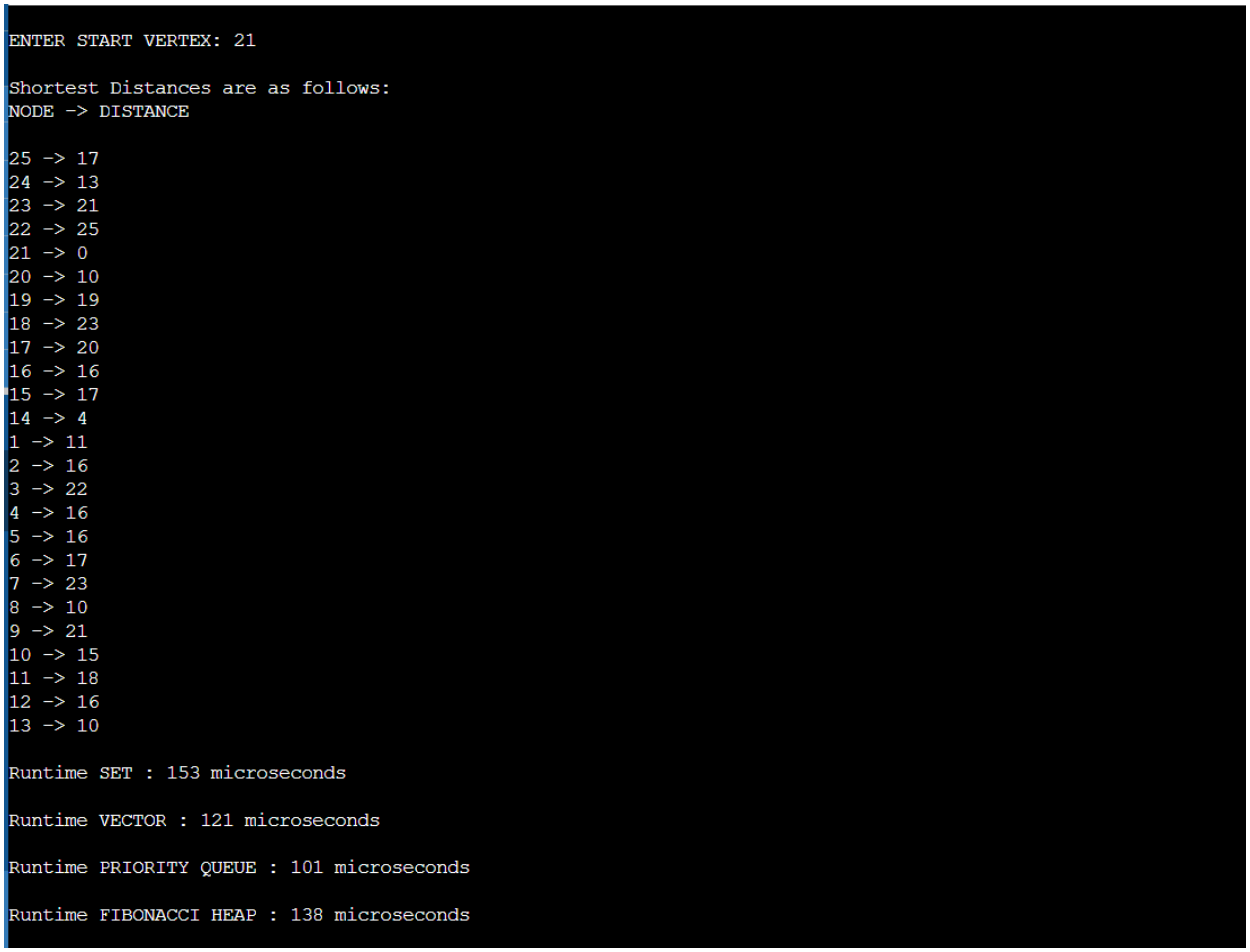
Secondly, a randomly generated graph with 100 edges and maximum of 25 edges was inputted.

Graph Inputted:



Output:

PTO

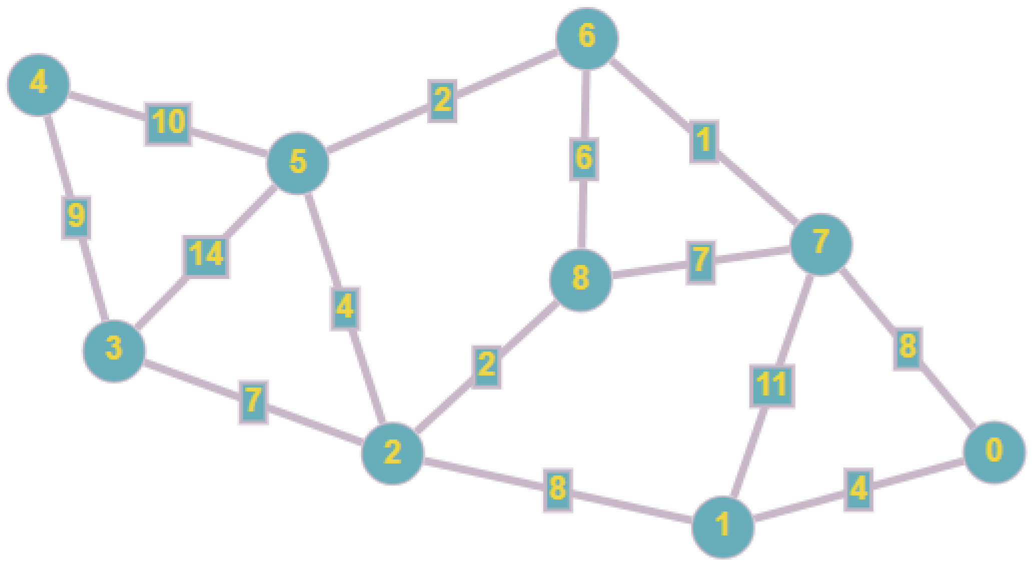


```
ENTER START VERTEX: 21

Shortest Distances are as follows:
NODE -> DISTANCE

25 -> 17
24 -> 13
23 -> 21
22 -> 25
21 -> 0
20 -> 10
19 -> 19
18 -> 23
17 -> 20
16 -> 16
15 -> 17
14 -> 4
1 -> 11
2 -> 16
3 -> 22
4 -> 16
5 -> 16
6 -> 17
7 -> 23
8 -> 10
9 -> 21
10 -> 15
11 -> 18
12 -> 16
13 -> 10

Runtime SET : 153 microseconds

Runtime VECTOR : 121 microseconds

Runtime PRIORITY QUEUE : 101 microseconds

Runtime FIBONACCI HEAP : 138 microseconds
```

Here, since the graph is not very densely connected, the results obtained were slightly different. Priority Queue was the fastest, followed by vector. The set data structure performed the slowest.

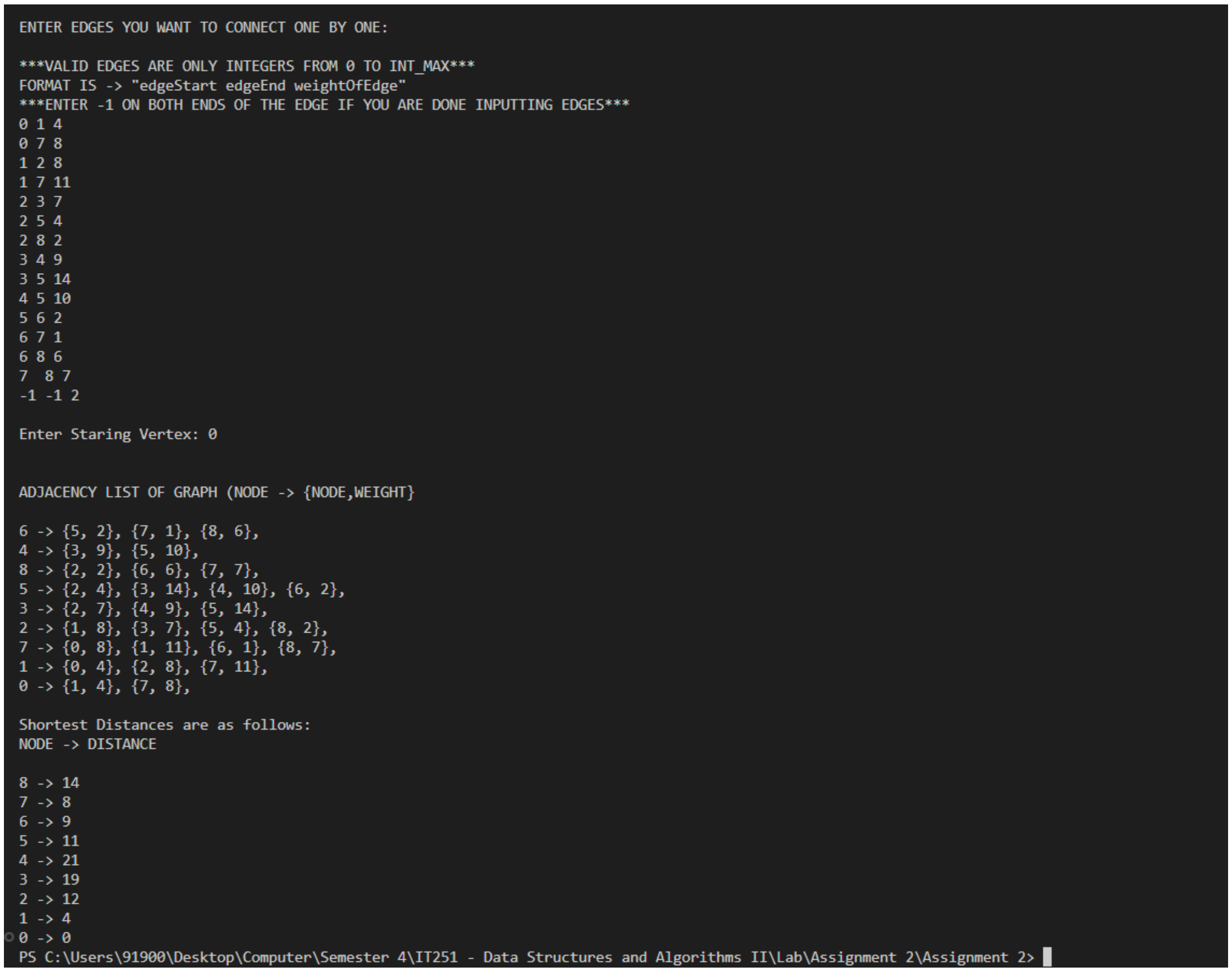
3)
Sample of how to input into the graph and get an output:

Say one wants to input the following graph:



PTO

Then, the input format and output are as follows:



```
ENTER EDGES YOU WANT TO CONNECT ONE BY ONE:

***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***
FORMAT IS -> "edgeStart edgeEnd weightOfEdge"
***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING EDGES***
0 1 4
0 7 8
1 2 8
1 7 11
2 3 7
2 5 4
2 8 2
3 4 9
3 5 14
4 5 10
5 6 2
6 7 1
6 8 6
7  8 7
-1 -1 2

Enter Staring Vertex: 0


ADJACENCY LIST OF GRAPH (NODE -> {NODE,WEIGHT}

6 -> {5, 2}, {7, 1}, {8, 6},
4 -> {3, 9}, {5, 10},
8 -> {2, 2}, {6, 6}, {7, 7},
5 -> {2, 4}, {3, 14}, {4, 10}, {6, 2},
3 -> {2, 7}, {4, 9}, {5, 14},
2 -> {1, 8}, {3, 7}, {5, 4}, {8, 2},
7 -> {0, 8}, {1, 11}, {6, 1}, {8, 7},
1 -> {0, 4}, {2, 8}, {7, 11},
0 -> {1, 4}, {7, 8},

Shortest Distances are as follows:
NODE -> DISTANCE

8 -> 14
7 -> 8
6 -> 9
5 -> 11
4 -> 21
3 -> 19
2 -> 12
1 -> 4
0 -> 0
PS C:\Users\91900\Desktop\Computer\Semester 4\IT251 - Data Structures and Algorithms II\Lab\Assignment 2\Assignment 2>
```

Code:

IMPORTANT: Implementation of the Fibonacci Heap was done through the #include <boost/heap/fibonacci_heap.hpp> command. This has to be downloaded to be able to run on a local computer. Hence, please run the code on **onlinegdb compiler** to see results.

There are 3 code files included with this folder. The first one named **DijkstraAlgorithmTrial** has 4 functions of Dijkstra Algorithm, each using different data structures. The **DijkstraOptimal** code file has the algorithm implemented with a priority queue, which was the most optimal based on the runtimes obtained. The **DijkstraRandom** code file generates 100000 random edges between nodes from 1 to 2000. Thereafter, it runs the algorithm and outputs the run time of the algorithm for each data structure. This can be altered to suit the user's needs, as in if the user wants more nodes, or more edges, etc.

Conclusion:

Theoretically, the Fibonacci heap is supposed to be the fastest data structure for the Dijkstra's algorithm because of its amortized O(1) time for the decrease key operation, but after running the algorithm for all 4 data structures, for various different variations of graphs, the priority queue was the fastest in terms of run time obtained.

Hence, I have picked priority queue as my data structure for doing the Dijkstra's algorithm and the results can be verified by the running the code files on the same.

(Please refer to the code file for the implementations)

****