Algorithms and Data Structures

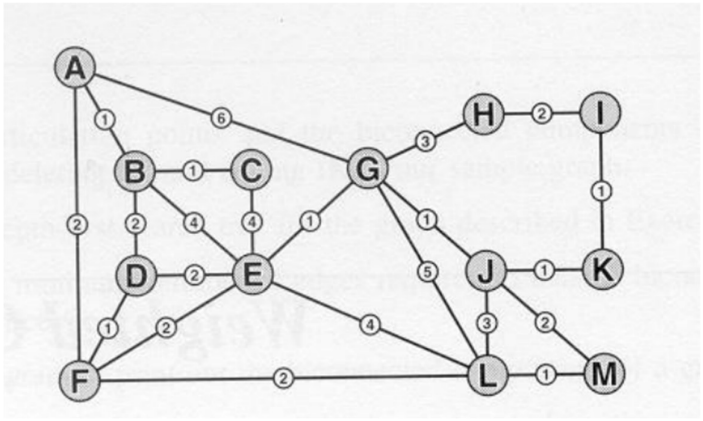
Graph Traversal, MST & SPT Algorithm Assignment Report

GraphAlgs-Hipolito-Ian-C21436494

Introduction:

The goal of this report is to present the implementation and analysis of Prim's and Kruskal's algorithm for finding the minimum spanning tree or MST for a weighted connected graph and Dijkstra's shortest path tree or SPT algorithm. This report will also cover the implementation and analysis of the breath-first traversal or BFS and depth-first traversal or DFS algorithms using adjacency lists data structure. The programs will prompt user input for a text file containing a graph and will also prompt for a starting vertex. The programs will then read the inputted graph and starting vertex and construct the necessary data structures to output the correct minimum spanning tree (MST) and shortest path tree (SPT). As well as that, the algorithms will output the step-by-step workings to the console to display the process by outputting the contents of the "heap", "parent[]" and "dist[]" for each step in Prim's and Dijkstra's algorithms. Furthermore, this report will also include representations of the graph using adjacency lists and a step-by-step construction of the algorithms mentioned, starting at the vertex the user inputs. A step-by-step construction of the minimum spanning tree for Kruskal's algorithm, including the union-find partition and set representations for each step. Diagrams showing the MST superimposed and SST superimposed on the graph will also be included in this report. Finally, screen captures showing all the implemented programs executing and their outputs from the example graph. Overall, this report will provide a detailed and comprehensive explanation and understanding of the algorithms mentioned above, how they are implemented and their process. All code will be written in JAVA code.

The Graph Used in This Assignment



wGraph1.txt

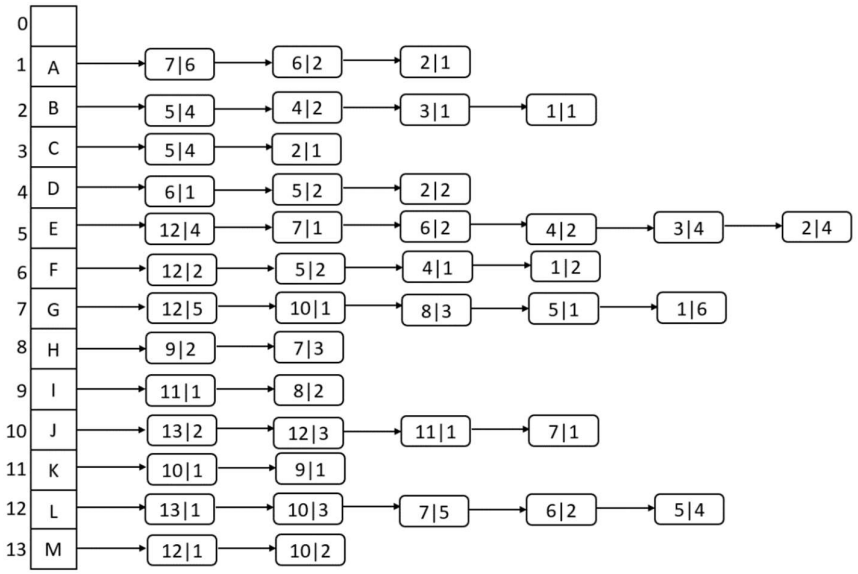
```
13 22
1   2   1
1   6   2
1   7   6
2   3   1
2   4   2
2   5   4
3   5   4
4   5   2
4   6   1
5   6   2
5   7   1
5   12  4
6   12  2
7   8   3
7   10  1
7   12  5
8   9   2
9   11  1
10  11  1
10  12  3
10  13  2
12  13  1
```

Adjacency Lists – Graph Representation

```
Adjacency List:

adj[A] -> |G | 6| -> |F | 2| -> |B | 1| ->
adj[B] -> |E | 4| -> |D | 2| -> |C | 1| -> |A | 1| ->
adj[C] -> |E | 4| -> |B | 1| ->
adj[D] -> |F | 1| -> |E | 2| -> |B | 2| ->
adj[E] -> |L | 4| -> |G | 1| -> |F | 2| -> |D | 2| -> |C | 4| -> |B | 4| ->
adj[F] -> |L | 2| -> |E | 2| -> |D | 1| -> |A | 2| ->
adj[G] -> |L | 5| -> |J | 1| -> |H | 3| -> |E | 1| -> |A | 6| ->
adj[H] -> |I | 2| -> |G | 3| ->
adj[I] -> |K | 1| -> |H | 2| ->
adj[J] -> |M | 2| -> |L | 3| -> |K | 1| -> |G | 1| ->
adj[K] -> |J | 1| -> |I | 1| ->
adj[L] -> |M | 1| -> |J | 3| -> |G | 5| -> |F | 2| -> |E | 4| ->
adj[M] -> |L | 1| -> |J | 2| ->
```

Adjacency List I created after reading all the edges:



An adjacency list is a data structure commonly used to represent a graph, as illustrated in the screen captures above. It comprises/consists of an array of linked lists, where every vertex in the graph is connected to a corresponding linked list. However, the adjacency list only records and maintains information about the vertices that are connected to a given vertex through an edge. Each edge in the graph is represented by a node within the linked list, containing essential data such as the destination vertex and the weight of the edge. Consequently, every edge is represented by two linked list nodes.

The provided diagram that I have made of the adjacency list depicts a graph, with each vertex represented/denoted by a letter and the adjacent vertices and their corresponding weights indicating the edges. For example, in the first line “adj[A] -> |G|6| -> |F|2| -> |B|1|

->", indicates that the vertex "A" has edges connecting it to vertex "G" with a weight of "6", to vertex "F" with a weight of "2", and to vertex "B" with a weight of "1". This adjacent list implementation is highly efficient/effective as it only stores information about the existing edges, rather than representing/showcasing all possible edge combinations.

Overall, the adjacency list data structure provides a concise and effective way to represent a graph by only representing and focusing on the actual edges and their associated vertex connections, resulting in a more compact and optimized representation.

Prim's Minimum Spanning Tree Algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This is when the algorithm finds a subset of the edges that forms a tree, which includes every vertex and the total weight of all the edges in the tree is minimized. The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all the vertices. Prim's algorithm is vertex based unlike Kruskal's algorithm as it is edge based. Prim's algorithm's Time Complexity For Adjacency List is: O(E + V Log V)12

The code for Prim's MST Algorithm in my program:

```
//method to find the minimum spanning tree using Prim's algorithm, starting at vertex s
    public void MST_Prim(int s) {
        int v, u; //vertices
        int wgt, wgt_sum = 0; //wgt = weight, wgt_sum = total weight of MST
        int[] dist, parent, hPos; //dist = distance, parent = parent, hPos = heap position
        dist = new int[V + 1];
        parent = new int[V + 1];
        hPos = new int[V + 1];
        Heap h = new Heap(V, dist, hPos); //h is a heap that stores the vertices not yet in the MST
        Node t; //node in the adjacency list of a vertex

        //initialise dist, parent, and hPos arrays
        for (v = 0; v <= V; v++) {
            dist[v] = Integer.MAX_VALUE;
            parent[v] = 0;
            hPos[v] = 0;
        }

        h.insert(s); //insert the starting vertex into the heap
        dist[s] = 0; //set the distance from the starting vertex to itself to 0

        System.out.print("\n--------------------------------------------------------------------------------
--------------------------------------------------------------------------------\n\n");
        System.out.print("\n\nPrim's Minimum Spanning Tree Algorithm\n");
        System.out.print("\n\nStarting from vertex " + s + "\n");

        //while there are no vertices in the MST yet
        while (!h.isEmpty()) {
            v = h.remove(); //remove the vertex with the smallest distance from the heap
            dist[v] = -dist[v]; //set the distance to the vertex to negative to indicate that it is
now in the MST
```

```
        System.out.print("\nVertex " + v + " removed from heap.\n");

        //show contents of heap, parent[], and dist[] arrays
        System.out.print("Heap: ");
        h.showHeap();
        System.out.print("Parent: ");
        showArray(parent);
        System.out.print("Distance: ");
        showArray(dist);

        //for loop to iterate through the vertices
        for (t = adj[v]; t != z; t = t.next) {
            u = t.vert;
            wgt = t.wgt;

            System.out.print("Examining vertex " + u + " with the weight " + wgt + ".\n");

            //if weight if edge is less than the distance of the vertex
            if (wgt < dist[u]) {
                //if the vertex is already in the heap
                if (dist[u] != Integer.MAX_VALUE) {
                    wgt_sum -= dist[u];
                }

                dist[u] = wgt; //update minimum distance of vertex "u" to the MST
                parent[u] = v; //update parent vertex of "u" in the MST to "v"
                wgt_sum += wgt; //update total weight of the MST

                if (hPos[u] == 0) {
                    h.insert(u);
                    System.out.print("Neighbour " + u + " inserted into heap.\n");
                }
                else {
                    h.siftUp(hPos[u]);
                    System.out.print("Neighbour " + u + " updated in heap.\n");
                }
            }
        }

    }

    System.out.print("\nWeight of MST = " + wgt_sum + "\n");

    mst = parent;
    showMST();
```

```
        System.out.print("\n-----------------------------------------------------------------------------------
-------------------------------------------------------------------------------\n\n");
    }
```

Step-By-Step Explanation of Code Provided Above:

1. The method “MST_Prim(int s)” takes an integer value “s” as input. This integer “s” represents the starting vertex the user inputs for Prim’s algorithm to find the minimum spanning tree, in case the starting vertex is “12” / ”L”.
2. Variables are then initialized these being, “v” and “u”, which represent the vertices. The variable “wgt” represents the weight of the edge and “wgt_sum” represents the total weight of the minimum spanning tree. Arrays are then initialized as “dist”, “parent” and “hPos”, they are used to keep track of the distance, parent and heap position of each vertex.
3. Heap “h” is then created, and it will store the vertices that are not yet in the minimum spanning tree, using the “Heap” class.
4. The “dist”, “parent” and “hPos” arrays are then set to have values of “Integer.MAX_VALUE”, “0” and “0”. This represents that all the vertices are not yet in the minimum spanning tree and have infinite distance from the starting vertex.
5. The starting vertex “s” is then inserted into the heap “h” and its distance from itself is set to 0.
6. Then the program outputs print messages to output and displays them in the terminal.
7. The while loop is used to make sure that the program continues to execute as long as there are vertices not in the minimum spanning tree yet.
8. The vertex with the smallest distance from the source vertex (“12” or “L”) is removed from the heap “h” during each iteration of the while loop, and its distance is set to negative to indicate and show that the vertex is now in the minimum spanning tree.
9. The next print statement displays the vertex that has been removed as shown in the screen captures above. The contents of the heap “h”, “parent[]” and “dist[]” arrays are also printed out and can be seen in the screen captures above.
10. After that the for loop then iterates through all the vertices in the adjacency list of the vertex that has just been removed from the heap. Additionally, it examines and looks at each vertex and its corresponding weights, determining whether the weight of the edge connecting the vertex to its parent is less than the vertex’s distance.
11. If the weight of the edge is less than the vertex’s distance, then the distance of the vertex is updated to the weight of the edge and the parent is updated to the vertex that has just been removed from the heap. While doing so the total weight of the minimum spanning tree is also updated.

12. The vertex is inserted into the heap if it is not already in it. Otherwise, the vertex is updated in the heap. The while loop stops executing when all the vertices have been added into the minimum spanning tree.
13. The program will then print out the total weight of the minimum spanning tree, in this example the MST weight is 16.
14. The minimum spanning tree is then printed out and displayed using the "showMST" method.

Step-By-Step Construction of The Minimum Spanning Tree

MV = Max Value

Starts at Vertex L or 12

1st Traverse:

Vertices E, F, G, J, M are inserted into the heap.

Parent[]	Dist[]	Heap
A -> @	A -> MV	E -> 4
B -> @	B -> MV	F -> 2
C -> @	C -> MV	G -> 5
D -> @	D -> MV	J -> 2
E -> 12	E -> 4	M -> 1
F -> 12	F -> 2	
G -> 12	G -> 5	
H -> @	H -> MV	
I -> @	I -> MV	
J -> 12	J -> 2	
K -> @	K -> MV	
L -> 0	L -> 0	
M -> 12	M -> 1	

2nd Traverse:

Vertex M is removed from the heap.

Parent[]	Dist[]	Heap
A -> @	A -> MV	E -> 4
B -> @	B -> MV	F -> 2
C -> @	C -> MV	G -> 5
D -> @	D -> MV	J -> 2
E -> 12	E -> 4	
F -> 12	F -> 2	

G -> 12	**G -> 5**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	
J -> 12	**J -> 2**	
K -> 12	**K -> 1**	
L -> 0	**L -> 0**	
M -> 12	**M -> 1**	

3rd Traverse:

Vertex F is removed from the heap. Vertices A and D are inserted into the heap.

Parent[]	**Dist[]**	**Heap**
A -> @	**A -> MV**	**E -> 4**
B -> @	**B -> MV**	**G -> 5**
C -> @	**C -> MV**	**J -> 2**
D -> @	**D -> MV**	**A -> 2**
E -> 12	**E -> 4**	**D -> 1**
F -> 12	**F -> -2**	
G -> 12	**G -> 5**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	
J -> 13	**J -> 2**	
K -> @	**K -> MV**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

4th Traverse:

Vertex D is removed from the heap. Vertex B is inserted into the heap. Vertex E is updated in the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> 2**	**E -> 2**
B -> @	**B -> MV**	**G -> 5**
C -> @	**C -> MV**	**J -> 2**
D -> 6	**D -> -1**	**A -> 2**
E -> 6	**E -> 4**	**B -> 2**
F -> 12	**F -> -2**	
G -> 12	**G -> 5**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	
J -> 13	**J -> 2**	
K -> @	**K -> MV**	

L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

5th Traverse:

Vertex A is removed from the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**E -> 4**
B -> 4	**B -> 2**	**G -> 5**
C -> @	**C -> MV**	**J -> 2**
D -> 6	**D -> -1**	**B -> 2**
E -> 6	**E -> 2**	
F -> 12	**F -> -2**	
G -> 12	**G -> 5**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	
J -> 13	**J -> 2**	
K -> @	**K -> MV**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

6th Traverse:

Vertex B is removed from the heap. Vertex C is inserted into the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**E -> 4**
B -> 1	**B -> -1**	**G -> 5**
C -> @	**C -> MV**	**J -> 2**
D -> 6	**D -> -1**	**C -> 1**
E -> 6	**E -> 2**	
F -> 12	**F -> -2**	
G -> 12	**G -> 5**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	
J -> 13	**J -> 2**	
K -> @	**K -> MV**	
L -> 0	**L -> 0**	

M -> 12	**M -> -1**	

7th Traverse:

Vertex C is removed from the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**E -> 4**
B -> 1	**B -> -1**	**G -> 5**
C -> 2	**C -> -1**	**J -> 2**
D -> 6	**D -> -1**	
E -> 6	**E -> 2**	
F -> 12	**F -> -2**	
G -> 12	**G -> 5**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	
J -> 13	**J -> 2**	
K -> @	**K -> MV**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

8th Traverse:

Vertex J is removed from the heap. Vertex K inserted into the heap. Vertex G is updated in the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**E -> 4**
B -> 1	**B -> -1**	**G -> 1**
C -> 2	**C -> -1**	**K -> 1**
D -> 6	**D -> -1**	
E -> 6	**E -> 2**	
F -> 12	**F -> -2**	
G -> 12	**G -> 5**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	

J -> 13	**J -> -2**	
K -> @	**K -> MV**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

9th Traverse:

Vertex K is removed from the heap. Vertex I inserted into the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**E -> 4**
B -> 1	**B -> -1**	**G -> 1**
C -> 2	**C -> -1**	**I -> 1**
D -> 6	**D -> -1**	
E -> 6	**E -> 2**	
F -> 12	**F -> -2**	
G -> 10	**G -> 1**	
H -> @	**H -> MV**	
I -> @	**I -> MV**	
J -> 13	**J -> -2**	
K -> 10	**K -> -1**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

10th Traverse:

Vertex G is removed from the heap. Vertex H inserted into the heap. Vertex E is updated in the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**E -> 1**
B -> 1	**B -> -1**	**I -> 1**
C -> 2	**C -> -1**	**H -> 3**
D -> 6	**D -> -1**	
E -> 6	**E -> 2**	
F -> 12	**F -> -2**	
G -> 10	**G -> -1**	
H -> @	**H -> MV**	

I -> 11	**I -> 1**	
J -> 13	**J -> -2**	
K -> 10	**K -> -1**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

11th Traverse:

Vertex I is removed from the heap. Vertex H is updated in the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**E -> 1**
B -> 1	**B -> -1**	**H -> 2**
C -> 2	**C -> -1**	
D -> 6	**D -> -1**	
E -> 7	**E -> 1**	
F -> 12	**F -> -2**	
G -> 10	**G -> -1**	
H -> 7	**H -> 3**	
I -> 11	**I -> -1**	
J -> 13	**J -> -2**	
K -> 10	**K -> -1**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

12th Traverse:

Vertex E is removed from the heap.

Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	**H -> 2**
B -> 1	**B -> -1**	
C -> 2	**C -> -1**	
D -> 6	**D -> -1**	
E -> 7	**E -> -1**	
F -> 12	**F -> -2**	
G -> 10	**G -> -1**	
H -> 9	**H -> 2**	
I -> 11	**I -> -1**	

J -> 13	**J -> -2**	
K -> 10	**K -> -1**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

13th Traverse:

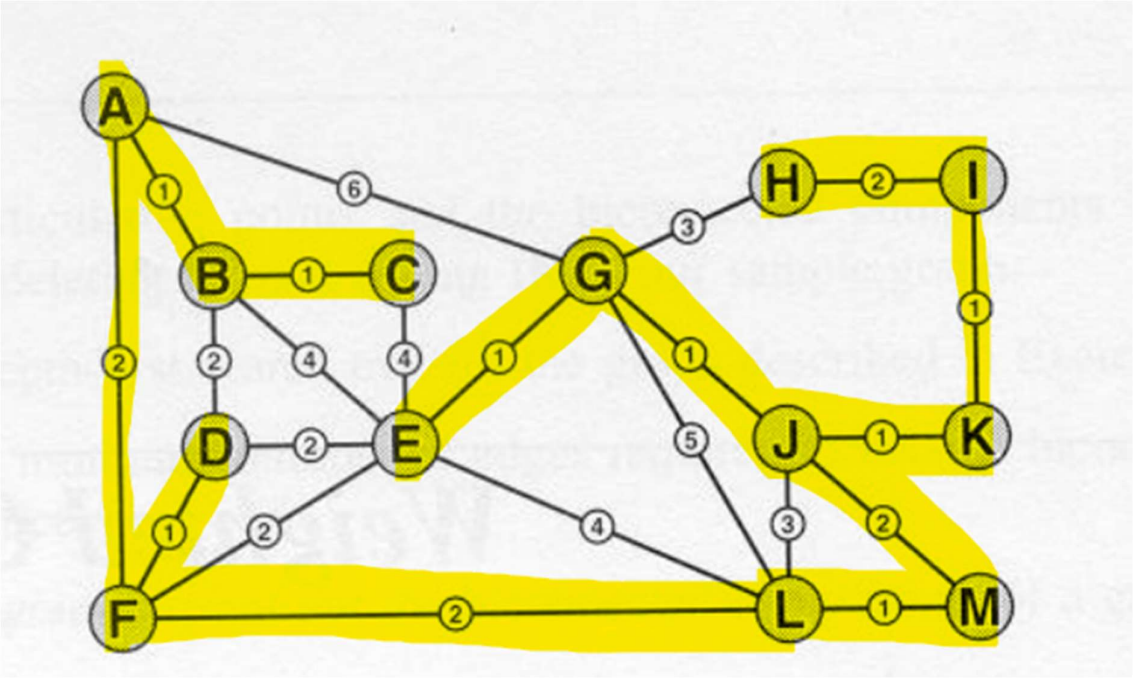
Vertex H is removed from the heap.

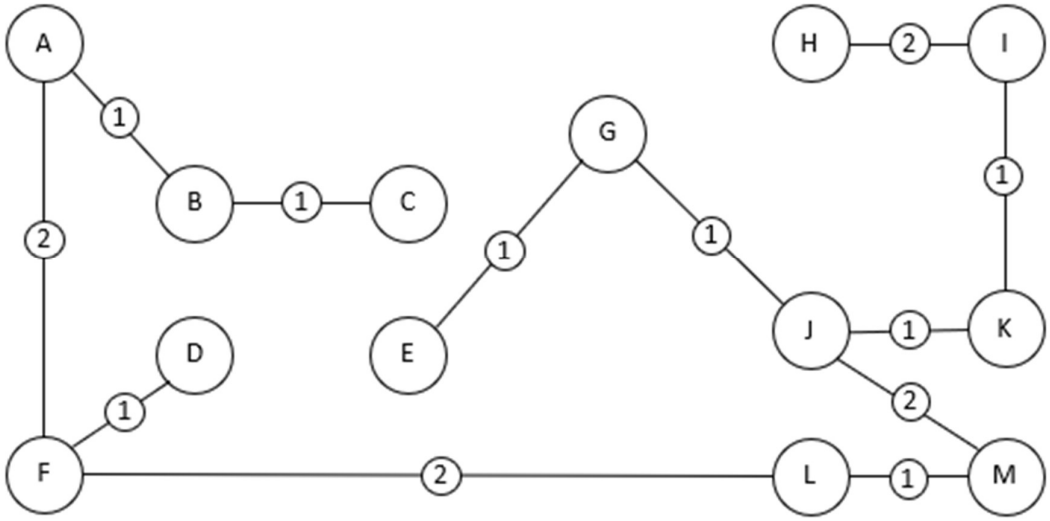
Parent[]	**Dist[]**	**Heap**
A -> 6	**A -> -2**	
B -> 1	**B -> -1**	
C -> 2	**C -> -1**	
D -> 6	**D -> -1**	
E -> 7	**E -> -1**	
F -> 12	**F -> -2**	
G -> 10	**G -> -1**	
H -> 9	**H -> -2**	
I -> 11	**I -> -1**	
J -> 13	**J -> -2**	
K -> 10	**K -> -1**	
L -> 0	**L -> 0**	
M -> 12	**M -> -1**	

The Minimum Spanning Tree:

A -> 6(F)
B -> 1 (A)
C -> 2 (B)
D -> 6 (F)
E -> 7 (G)
F -> 12 (L)
G -> 10 (J)
H -> 9 (I)
I -> 11 (K)
J -> 13 (M)
K -> 10 (J)
L -> 0
M -> 12 (L)

Graph Diagram:





The weight of the minimum spanning tree is: 16.

Screen Captures of Outputs:

```
Enter the name of the input file:
wGraph1.txt
Starting Vertex:
12
Parts[] = 13 22
Reading edges from text file
Edge A--(1)--B
Edge A--(2)--F
Edge A--(6)--G
Edge B--(1)--C
Edge B--(2)--D
Edge B--(4)--E
Edge C--(4)--E
Edge D--(2)--E
Edge D--(1)--F
Edge E--(2)--F
Edge E--(1)--G
Edge E--(4)--L
Edge F--(2)--L
Edge G--(3)--H
Edge G--(1)--J
Edge G--(5)--L
Edge H--(2)--I
Edge I--(1)--K
Edge J--(1)--K
Edge J--(3)--L
Edge J--(2)--M
Edge L--(1)--M

Adjacency List

adj[A] -> |G | 6| -> |F | 2| -> |B | 1| ->
adj[B] -> |E | 4| -> |D | 2| -> |C | 1| -> |A | 1| ->
adj[C] -> |E | 4| -> |B | 1| ->
adj[D] -> |F | 1| -> |E | 2| -> |B | 2| ->
adj[E] -> |L | 4| -> |G | 1| -> |F | 2| -> |D | 2| -> |C | 4| -> |B | 4| ->
adj[F] -> |L | 2| -> |E | 2| -> |D | 1| -> |A | 2| ->
adj[G] -> |L | 5| -> |J | 1| -> |H | 3| -> |E | 1| -> |A | 6| ->
adj[H] -> |I | 2| -> |G | 3| ->
adj[I] -> |K | 1| -> |H | 2| ->
adj[J] -> |M | 2| -> |L | 3| -> |K | 1| -> |G | 1| ->
adj[K] -> |J | 1| -> |I | 1| ->
adj[L] -> |M | 1| -> |J | 3| -> |G | 5| -> |F | 2| -> |E | 4| ->
adj[M] -> |L | 1| -> |J | 2| ->
```

```
Prim's Minimum Spanning Tree Algorithm


Starting from vertex 12

Vertex 12 removed from heap.
Heap: Heap =
Parent: 0 0 0 0 0 0 0 0 0 0 0 0 0
Distance: 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 0 2147483647
Examining vertex 13 with the weight 1.
Vertex 13 inserted into heap.
Examining vertex 10 with the weight 3.
Vertex 10 inserted into heap.
Examining vertex 7 with the weight 5.
Vertex 7 inserted into heap.
Examining vertex 6 with the weight 2.
Vertex 6 inserted into heap.
Examining vertex 5 with the weight 4.
Vertex 5 inserted into heap.

Vertex 13 removed from heap.
Heap: Heap = 6 10 7 5
Parent: 0 0 0 0 12 12 12 0 0 12 0 0 12
Distance: 2147483647 2147483647 2147483647 2147483647 4 2 5 2147483647 2147483647 3 2147483647 0 -1
Examining vertex 12 with the weight 1.
Examining vertex 10 with the weight 2.
Vertex 10 updated in heap.

Vertex 6 removed from heap.
Heap: Heap = 10 5 7
Parent: 0 0 0 0 12 12 12 0 0 13 0 0 12
Distance: 2147483647 2147483647 2147483647 2147483647 4 -2 5 2147483647 2147483647 2 2147483647 0 -1
Examining vertex 12 with the weight 2.
Examining vertex 5 with the weight 2.
Vertex 5 updated in heap.
Examining vertex 4 with the weight 1.
Vertex 4 inserted into heap.
Examining vertex 1 with the weight 2.
Vertex 1 inserted into heap.

Vertex 4 removed from heap.
Heap: Heap = 1 10 7 5
Parent: 6 0 0 6 6 12 12 0 0 13 0 0 12
Distance: 2 2147483647 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Examining vertex 6 with the weight 1.
Examining vertex 5 with the weight 2.
Examining vertex 2 with the weight 2.
Vertex 2 inserted into heap.
```

```
Vertex 1 removed from heap.
Heap: Heap = 2 10 7 5
Parent: 6 4 0 6 6 12 12 0 0 13 0 0 12
Distance: -2 2 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Examining vertex 7 with the weight 6.
Examining vertex 6 with the weight 2.
Examining vertex 2 with the weight 1.
Vertex 2 updated in heap.

Vertex 2 removed from heap.
Heap: Heap = 5 10 7
Parent: 6 1 0 6 6 12 12 0 0 13 0 0 12
Distance: -2 -1 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Examining vertex 5 with the weight 4.
Examining vertex 4 with the weight 2.
Examining vertex 3 with the weight 1.
Vertex 3 inserted into heap.
Examining vertex 1 with the weight 1.

Vertex 3 removed from heap.
Heap: Heap = 10 5 7
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12
Distance: -2 -1 -1 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Examining vertex 5 with the weight 4.
Examining vertex 2 with the weight 1.

Vertex 10 removed from heap.
Heap: Heap = 5 7
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12
Distance: -2 -1 -1 -1 2 -2 5 2147483647 2147483647 -2 2147483647 0 -1
Examining vertex 13 with the weight 2.
Examining vertex 12 with the weight 3.
Examining vertex 11 with the weight 1.
Vertex 11 inserted into heap.
Examining vertex 7 with the weight 1.
Vertex 7 updated in heap.

Vertex 11 removed from heap.
Heap: Heap = 7 5
Parent: 6 1 2 6 6 12 10 0 0 13 10 0 12
Distance: -2 -1 -1 -1 2 -2 1 2147483647 2147483647 -2 -1 0 -1
Examining vertex 10 with the weight 1.
Examining vertex 9 with the weight 1.
Vertex 9 inserted into heap.
```

```
Vertex 7 removed from heap.
Heap: Heap = 9 5
Parent: 6 1 2 6 6 12 10 0 11 13 10 0 12
Distance: -2 -1 -1 -1 2 -2 -1 2147483647 1 -2 -1 0 -1
Examining vertex 12 with the weight 5.
Examining vertex 10 with the weight 1.
Examining vertex 8 with the weight 3.
Vertex 8 inserted into heap.
Examining vertex 5 with the weight 1.
Vertex 5 updated in heap.
Examining vertex 1 with the weight 6.

Vertex 9 removed from heap.
Heap: Heap = 5 8
Parent: 6 1 2 6 7 12 10 7 11 13 10 0 12
Distance: -2 -1 -1 -1 1 -2 -1 3 -1 -2 -1 0 -1
Examining vertex 11 with the weight 1.
Examining vertex 8 with the weight 2.
Vertex 8 updated in heap.

Vertex 5 removed from heap.
Heap: Heap = 8
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12
Distance: -2 -1 -1 -1 -1 -2 -1 2 -1 -2 -1 0 -1
Examining vertex 12 with the weight 4.
Examining vertex 7 with the weight 1.
Examining vertex 6 with the weight 2.
Examining vertex 4 with the weight 2.
Examining vertex 3 with the weight 4.
Examining vertex 2 with the weight 4.

Vertex 8 removed from heap.
Heap: Heap =
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12
Distance: -2 -1 -1 -1 -1 -2 -1 -2 -1 -2 -1 0 -1
Examining vertex 9 with the weight 2.
Examining vertex 7 with the weight 3.
```

```
The Weight of the MST = 16


Minimum Spanning tree parent array is:
A -> F
B -> A
C -> B
D -> F
E -> G
F -> L
G -> J
H -> I
I -> K
J -> M
K -> J
L -> @
M -> L
```

Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm is very similar to Prim's minimum spanning tree algorithm as both algorithms use priority or best search for vertices approach and is a greedy algorithm. Dijkstra's shortest path algorithm is a graph traversal algorithm that finds the shortest path from a source vertex to all the other vertices in a weighted graph. Dijkstra's algorithm works on a connected, directed or undirected graph with the edges having non-negative weights. It iteratively selects vertices with the minimum distance from the source vertex and examines neighboring vertices to update their distance (dist[]) if a shorter path is found. Its Time Complexity is: O(V*logV+E).

The code for Dijkstra's Shortest Path Tree Algorithm in my program:

```
    public void SPT_Dijkstra(int s) {

        int v, u;
        int wgt, wgt_sum = 0;
        int[] dist, parent, hPos;

        //create arrays to store the distances, parents, and heap positions of each vertex
        dist = new int[V + 1];
        parent = new int[V + 1];
        hPos = new int[V + 1];

        //initialise dist, parent, and hPos arrays
        for(v = 0; v <= V; v++) {
            dist[v] = Integer.MAX_VALUE;
            parent[v] = 0;
            hPos[v] = 0;
        }

        Heap h = new Heap(V, dist, hPos); //create a heap to store the vertices by distance from the
source vertex
        h.insert(s);
        dist[s] = 0;

        System.out.print("\n\nDijkstra's Shortest Path Tree Algorithm\n\n");
        System.out.print("Starting from vertex " + s + "\n\n"); //print the source vertex

        //while the heap is not empty, remove the vertex with the smallest distance from the heap
        while(!h.isEmpty()){
            u = h.remove();

            System.out.print("\nVertex " + u + " removed from heap.\n");

            // Show contents of heap, parent[], and dist[] arrays
            System.out.print("Heap: ");
            h.showHeap();
```

```
            System.out.print("Parent: ");
            showArray(parent);
            System.out.print("Dist: ");
            showArray(dist);
            System.out.print("\n");

            //for each adjacent vertex, update its distance and parent if a shorter path is found
            for(Node n = adj[u]; n != z; n = n.next){
                v = n.vert;
                wgt = n.wgt;

                System.out.print("Examining vertex " + v + " with the weight " + wgt + ".\n");

                if(dist[u] + wgt < dist[v]){
                    dist[v] = dist[u] + wgt;
                    parent[v] = u;
                    if(hPos[v] == 0){
                        h.insert(v);
                        System.out.println("Vertex " + v + " inserted into heap.");
                    }
                    else{
                        h.siftUp(hPos[v]);
                        System.out.println("Vertex " + v + " updated in heap.");
                    }
                }
            }
        }

        System.out.println("SPT Using Dijkstra's Algorithm From Source Vertex " + s + ":"); //print
the shortest path tree
        System.out.println("Vertex   Parent  Distance");
        for (v = 1; v <= V; v++) {
            System.out.println(toChar(v) + "         " + toChar(parent[v]) + "       " + dist[v]);
        }

        int totalWeight = 0; //calculate and print the weight of the shortest path tree
        for (int i = 1; i <= V; i++) {
            if (dist[i] != Integer.MAX_VALUE) {
                totalWeight += dist[i];
            }
        }
        System.out.println("\nWeight of SPT: " + totalWeight + "\n");
        System.out.print("\n-------------------------------------------------------------------------------
-------------------------------------------------------------------------------\n\n");

    }
```

Step-By-Step Explanation of Code Provided Above:

1. The method “SPT_Dijkstra (int s)” takes an integer value “s” as input. This integer “s” represents the starting vertex the user inputs for Prim’s algorithm to find the minimum spanning tree, in case the starting vertex is “12” / ”L”.
2. Variables are then initialized these being, “v” and “u”, which represent the vertices. The variable “wgt” represents the weight of the edge. Arrays are then initialized as “dist”, “parent” and “hPos”, they are used to keep track of the distance, parent and heap position of each vertex.
3. The arrays are then initialized with values “dist[v] = Integer.MAX_VALUE”, “parent[v] = 0” and “hPos[v] = 0”. A heap object “h” with the parameters “V”, “dist” and “hPos” are then created. This heap will be used to store the vertices based on their distance from the source vertex, which in this case will be L or 12.
4. The “insert” method is then used to insert the source vertex into the heap “h” and “dist[s]” is then set to 0, this indicates that the distance of the source to itself is 0.
5. The program then enters the while loop which continues until the heap is empty.
6. The “remove” method is used to remove vertices with the smallest distance from the heap and then assigns them to variable “u”. Messages will be printed into the terminal indicating what vertices have been removed in each iteration of the while loop. As well as that, the contents of the “heap”, “parent[]” and “dist[]” are also printed out into the terminal using the “showHeap” and “showArray” methods.
7. The adjacent vertices of “u” are then iterated using a for loop and the adjacency list of vertex “u” is stored in “adj[u]”.
8. The distance and parent are updated for each adjacent vertex “v” if a shorter path is found to exist. An if statement is also used to ensure that if the distance from the source to “u” and the weight of the edge “(u, v)” is less than or smaller than the current distance to “v”, the program will update the “dist[v]” with the new distances and set the “parent[v]” to “u”.
9. An inner if statement is used so that if “hPos[v]” is equal to “0” then it means that “v” is not yet in the heap and using the “insert” method “v” will be inserted into the heap. Otherwise, if “v” is already in the heap then its position is updated using the “siftUp” method. A print message is used to indicate this.
10. When the heap is empty the while loop stops executing and the shortest path tree is then printed and displayed in the terminal.
11. The shortest path tree weight is also calculated and displayed in the terminal.

Step-By-Step Construction of The Shortest Path Tree

MV = Max Value

Starts at Vertex L or 12

1st Traverse:

Vertex L is removed from the heap. Vertices E, F, G, J, M are inserted into the heap.

Parent[]	Dist[]	Heap
A -> @	A -> MV	E -> 4
B -> @	B -> MV	F -> 2
C -> @	C -> MV	G -> 5
D -> @	D -> MV	J -> 3
E -> 12	E -> 4	M -> 1
F -> 12	F -> 2	
G -> 12	G -> 5	
H -> @	H -> MV	
I -> @	I -> MV	
J -> 12	J -> 3	
K -> @	K -> MV	
L -> 0	L -> 0	
M -> 12	M -> 1	

2nd Traverse:

Vertex M is removed from the heap.

Parent[]	Dist[]	Heap
A -> @	A -> MV	E -> 4
B -> @	B -> MV	F -> 2
C -> @	C -> MV	G -> 5
D -> @	D -> MV	J -> 3
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 12	G -> 5	
H -> @	H -> MV	
I -> @	I -> MV	
J -> 12	J -> 3	
K -> @	K -> MV	
L -> 0	L -> 0	
M -> 12	M -> 1	

3rd Traverse:

Vertex F is removed from the heap. Vertices A and D are inserted into the heap.

Parent[]	Dist[]	Heap
A -> @	A -> MV	E -> 4
B -> @	B -> MV	G -> 5
C -> @	C -> MV	J -> 3
D -> @	D -> MV	A -> 2
E -> 12	E -> 4	D - > 1
F -> 12	F -> 2	
G -> 12	G -> 5	
H -> @	H -> MV	
I -> @	I -> MV	
J -> 12	J -> 3	
K -> @	K -> MV	
L -> 0	L -> 0	
M -> 12	M -> 1	

4th Traverse:

Vertex J is removed from the heap. Vertex K is inserted into the heap. Vertex G is updated in the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	E -> 4
B -> @	B -> MV	G -> 1
C -> @	C -> MV	A -> 2
D -> 6	D -> 3	D - > 1
E -> 12	E -> 4	K -> 1
F -> 12	F -> 2	
G -> 12	G -> 5	
H -> @	H -> MV	
I -> @	I -> MV	
J -> 12	J -> 3	
K -> @	K -> MV	
L -> 0	L -> 0	
M -> 12	M -> 1	

5th Traverse:

Vertex D is removed from the heap. Vertex **B** is inserted into the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	E -> 4
B -> @	B -> MV	G -> 1
C -> @	C -> MV	A -> 2
D -> 6	D -> 3	K -> 1
E -> 12	E -> 4	B -> 2
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> @	H -> MV	
I -> @	I -> MV	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

6th Traverse:

Vertex K is removed from the heap. Vertex I is inserted into the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	E -> 4
B -> 4	B -> 5	G -> 1
C -> @	C -> MV	A -> 2
D -> 6	D -> 3	B -> 2
E -> 12	E -> 4	I -> 1
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> @	H -> MV	
I -> @	I -> MV	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

7th Traverse:

Vertex A is removed from the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	E -> 4
B -> 4	B -> 5	G -> 1
C -> @	C -> MV	B -> 2
D -> 6	D -> 3	I -> 1
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> @	H -> MV	
I -> 11	I -> 5	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

8th Traverse:

Vertex E is removed from the heap. Vertex C is inserted into the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	G -> 1
B -> 4	B -> 5	B -> 2
C -> @	C -> MV	I -> 1
D -> 6	D -> 3	C -> 4
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> @	H -> MV	
I -> 11	I -> 5	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

9th Traverse:

Vertex G is removed from the heap. Vertex H is inserted into the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	B -> 2
B -> 4	B -> 5	I -> 1
C -> 5	C -> 8	C -> 4
D -> 6	D -> 3	H -> 3
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> @	H -> MV	
I -> 11	I -> 5	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

10th Traverse:

Vertex I is removed from the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	B -> 2
B -> 4	B -> 5	C -> 4
C -> 5	C -> 8	H -> 3
D -> 6	D -> 3	
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> 7	H -> 7	
I -> 11	I -> 5	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

11th Traverse:

Vertex B is removed from the heap. Vertex C is updated in the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	C -> 1
B -> 4	B -> 5	H -> 3
C -> 5	C -> 8	
D -> 6	D -> 3	
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> 7	H -> 7	
I -> 11	I -> 5	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

12th Traverse:

Vertex C is removed from the heap.

Parent[]	Dist[]	Heap
A -> 6	A -> 4	H -> 3
B -> 4	B -> 5	
C -> 2	C -> 6	
D -> 6	D -> 3	
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> 7	H -> 7	
I -> 11	I -> 5	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

13th Traverse:

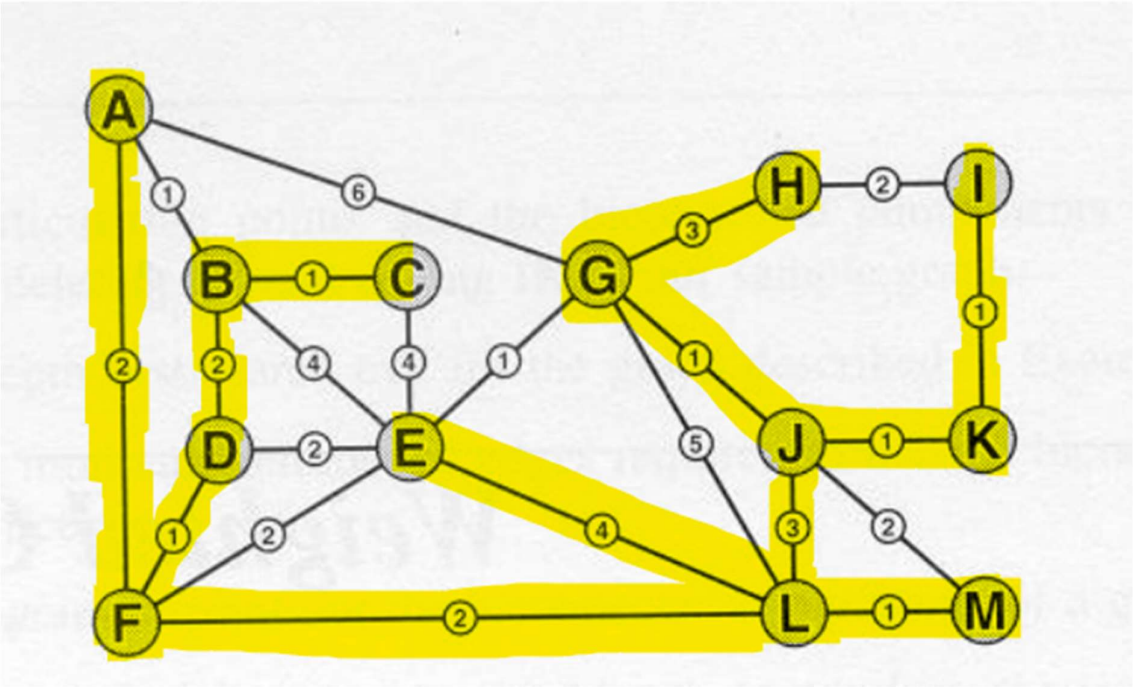
Vertex H is removed from the heap.

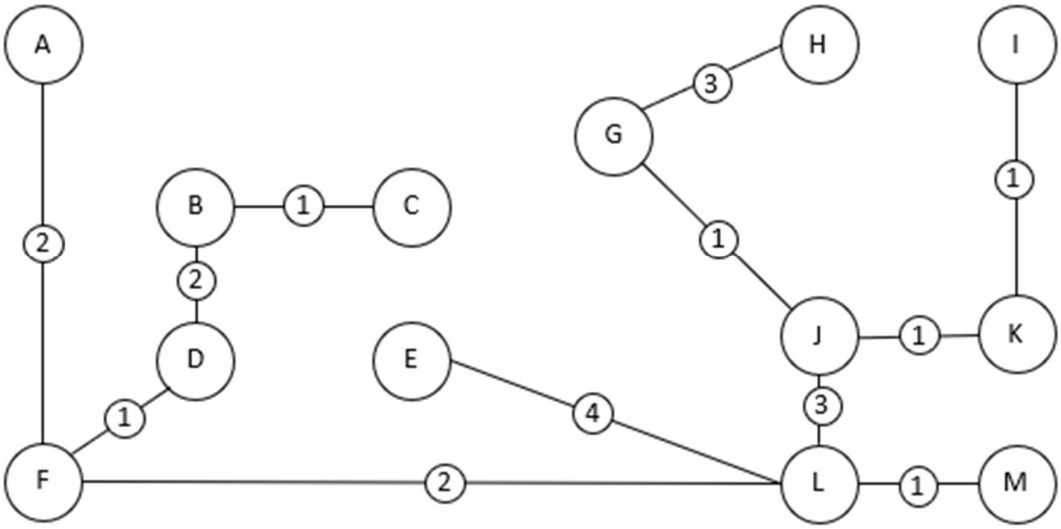
Parent[]	Dist[]	Heap
A -> 6	A -> 4	
B -> 4	B -> 5	
C -> 2	C -> 6	
D -> 6	D -> 3	
E -> 12	E -> 4	
F -> 12	F -> 2	
G -> 10	G -> 4	
H -> 7	H -> 7	
I -> 11	I -> 5	
J -> 12	J -> 3	
K -> 10	K -> 4	
L -> 0	L -> 0	
M -> 12	M -> 1	

The Shortest Path Tree:

Vertex	Parent	Distance
A	F	4
B	D	5
C	B	6
D	F	3
E	L	4
F	L	2
G	J	4
H	G	7
I	K	5
J	L	3
K	J	4
L	@	0
M	L	1

The Graph Diagram:





The Weight of The Shortest Path Tree: 48

Kruskal's Minimum Spanning Tree Algorithm, Union-Find Partition and Set Representation

Kruskal's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted graph. It finds a subset of edges that forms a tree that includes every vertex and the total weight of the all the edges in the tree are minimized. Kruskal's algorithm sorts the edges according to their weights. Similarly, to Prim's and Dijkstra's algorithm it is also a greedy algorithm. There are also many ways to implement Kruskal's algorithm, for example by using a heap, quicksort or heapsort to sort the edges. You can also improve the algorithm by implementing union by rank and path compression. Kruskal's algorithm's Time Complexity is: O(E Log V).

The code for Kruskal's Algorithm in my program:

```java
/*
 * Author: Ian Hipolito
 * StudentID: C21436494
 * Class: TU856/2 - Algorithms & Data Structures
 * Assignment: Graph Traversal, MST & SPT Algorithm Assignment
 */

// Simple weighted graph representation
// Uses an Adjacency Linked Lists, suitable for sparse graphs

import java.io.*;
import java.util.Scanner;

class Graph {
    class Node {
        public int vert;
        public int wgt;
        public Node next;
    }

    // V = number of vertices
    // E = number of edges
    // adj[] is the adjacency lists array
    private int V, E;
    private Node[] adj;
    private Node z;
    int[] parent;

    // default constructor
    public Graph(String graphFile) throws IOException {
        int u, v;
        int e, wgt;
        Node t;
```

```
        FileReader fr = new FileReader(graphFile);
        BufferedReader reader = new BufferedReader(fr);

        String splits = " +"; // multiple whitespace as delimiter
        String line = reader.readLine();
        String[] parts = line.split(splits);
        System.out.println("Parts[] = " + parts[0] + " " + parts[1]);

        V = Integer.parseInt(parts[0]);
        E = Integer.parseInt(parts[1]);

        // create sentinel node
        z = new Node();
        z.next = z;

        // create adjacency lists, initialised to sentinel node z
        adj = new Node[V + 1];
        for (v = 1; v <= V; ++v)
            adj[v] = z;

        // read the edges
        System.out.println("Reading edges from text file");
        for (e = 1; e <= E; ++e) {
            line = reader.readLine();
            parts = line.split(splits);
            u = Integer.parseInt(parts[0]);
            v = Integer.parseInt(parts[1]);
            wgt = Integer.parseInt(parts[2]);

            System.out.println("Edge " + toChar(u) + "--(" + wgt + ")--" + toChar(v));

            // write code to put edge into adjacency matrix
            t = new Node();
            t.vert = v;
            t.wgt = wgt;
            t.next = adj[u];
            adj[u] = t;

            t = new Node();
            t.vert = u;
            t.wgt = wgt;
            t.next = adj[v];
            adj[v] = t;
        }
    }

    // convert vertex into char for pretty printing
    private char toChar(int u) {
```

```
        return (char) (u + 64);
    }

    // method to display the graph representation
    public void display() {
        int v;
        Node n;

        for (v = 1; v <= V; ++v) {
            System.out.print("\nadj[" + toChar(v) + "] ->");
            for (n = adj[v]; n != z; n = n.next)
                System.out.print(" |" + toChar(n.vert) + " | " + n.wgt + "| ->");
        }
        System.out.println("");
    }

    //Edge class to store edges and their weights
    class Edge{
        public int u;
        public int v;
        public int wgt;

        //constructor
        public Edge(){
            u = 0;
            v = 0;
            wgt = 0;
        }

        public Edge(int wgt, int u, int v){
            this.u = u;
            this.v = v;
            this.wgt = wgt;
        }

        //display edge
        public void show() {
            System.out.println("Edge {" + toChar(u) + "--(" + wgt + ")--" + toChar(v) + "}");
        }
    }

    //QuickSort method takes an array of edges, left and right integers
    public void QS(Edge[] arr, int left, int right){
        //if left is less than right
        if (left < right){
            int pivot = partition(arr, left, right); //declares pivot and assigns it to partition
method
            QS(arr, left, pivot - 1); //recursively calls QuickSort method
```

```
        QS(arr, pivot + 1, right); //recursively calls QuickSort method
    }
}

//Partition method takes an array of edges, left and right integers
private int partition(Edge[] arr, int left, int right){
    Edge pivot = arr[right]; //assigns pivot to to right index of array
    int i = left - 1; //assigns i to left - 1

    //for loop to iterate from left to right
    for (int j = left; j < right; j++){

        //checks if weight of edge is less than or equal to pivot
        if (arr[j].wgt <= pivot.wgt){
            i++;
            //swaps arr[i] and arr[j]
            Edge temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    //swaps arr[i + 1] and arr[right]
    Edge temp = arr[i + 1];
    arr[i + 1] = arr[right];
    arr[right] = temp;

    return i + 1; //returns index of pivot after partition
}

//findRoot the root of a set using path compression
public int FindRoot(int u) {
    //checks of u is in the parent array
    if (parent[u] != u) {
        parent[u] = FindRoot(parent[u]); //recursively calls FindRoot method
    }
    return parent[u]; //returns value of parent[u] after path compression
}

//Union By Rank method takes an array of parents, an array of ranks, and two integers
public void UBR(int[] parent, int[] rank, int uSet, int vSet) {
    //checks if rank of uSet is less than rank of vSet
    if (rank[uSet] > rank[vSet]) {
        parent[vSet] = uSet; //assigns parent of vSet to uSet
    }
    //checks if rank of uSet is less than rank of vSet
    else if (rank[uSet] < rank[vSet]) {
        parent[uSet] = vSet; //assigns parent of uSet to vSet
```

```
    }
    //checks if rank of uSet is equal to rank of vSet
    else {
        parent[vSet] = uSet; //assigns parent of vSet to uSet
        rank[uSet] = rank[uSet] + 1; //increments rank of uSet
    }
}

//showSets method displays the sets
public void showSets()
{
    int u, root; //u is vertex, root is root of set
    int[] shown = new int[V+1]; //array to keep track of shown sets
    //for loop to iterate through vertices
    for (u=1; u<=V; ++u)
    {
        root = FindRoot(u); //assigns root to FindRoot method
        //checks if shown[root] is not equal to 1
        if(shown[root] != 1) {
            showSet(root); //calls showSet method
            shown[root] = 1; //assigns shown[root] to 1
        }
    }
    System.out.print("\n");
}

//showSet method takes int root as parameter and displays the set
private void showSet(int root)
{
    int v;
    System.out.print("Set{");

    //for loop to iterate through vertices
    for(v=1; v<=V; ++v)
        //checks if FindRoot of v is equal to root
        if(FindRoot(v) == root){
            System.out.print(toChar(v) + " ");
        }

    System.out.print("}  ");

}

//KruskalMST method finds the minimum spanning tree
public void KruskalMST() {
    Edge[] edges = new Edge[E]; //array of edges
    Edge[] mst = new Edge[V - 1]; //array of edges in MST
```

```
        int i = 0;

        //for loop to iterate through vertices
        for(int u = 1; u <= V; u++){
            //for loop to iterate through adjacent vertices
            for(Node n = adj[u]; n != z; n = n.next){
                int v = n.vert; //assigns v to vertex
                int wgt = n.wgt; //assigns wgt to weight

                //checks if u is less than v
                if(u < v){
                    edges[i++] = new Edge(wgt, u, v); //assigns edges[i] to new Edge
                }
            }
        }

        QS(edges, 0, edges.length - 1); //calls QuickSort method

        System.out.println("Sorted edges");

        //for loop to iterate through edges
        for(i = 0; i < edges.length; i++)
        {
            edges[i].show(); //calls show method
        }

        parent = new int[V + 1]; //array of parents
        int[] rank = new int[V + 1]; //array of ranks
        int mstWeight = 0; //weight of MST

        //for loop to iterate through vertices
        for(int v = 1; v <= V; v++){
            parent[v] = v; //assigns parent[v] to v
        }

        System.out.println("\n\n");

        int counter = 0; //counter for mst array

        //for loop to iterate through edges
        for(int j = 0; j < E; j++){
            Edge e = edges[j]; //assigns e to edges[j]
            int u = e.u; //assigns u to e.u
            int v = e.v; //assigns v to e.v
            int wgt = edges[j].wgt; //assigns wgt to edges[j].wgt
            int uSet = FindRoot(u); //assigns uSet to FindRoot of u
            int vSet = FindRoot(v); //assigns vSet to FindRoot of v
```

```
            //checks if uSet is not equal to vSet
            if(uSet != vSet){
                showSets(); //calls showSets method
                UBR(parent, rank, uSet, vSet); //calls UBR method
                mstWeight += wgt; //increments mstWeight by wgt
                mst[counter++] = e; //assigns mst[counter] to e
            }
        }

        //for loop to iterate through mst array
        for(i = 0; i < mst.length; i++){
            mst[i].show(); //calls show method
        }

        System.out.println("\nWeight of MST: " + mstWeight);
    }


    public class GraphLists2 {
        public static void main(String[] args) throws IOException {
            // String fname = "wGraph1.txt";


            String fname;

            System.out.println("Enter the name of the input file: ");
            Scanner Name = new Scanner(System.in);
            fname = Name.nextLine();

            Graph g = new Graph(fname);

            g.display();

            g.KruskalMST();

        }
    }

}
```

Step-By-Step Explanation of Code Provided Above:

1. The “Edge” class is defined and initialized in the “Graph” class. It stores the source vertex - “u”, destination vertex - “v” , and the weight of the edges - “wgt”. The “show” method is used to display information about the edges into the terminal.
2. The “QS” method executes the QuickSort algorithm implemented to sort an array of edges on the graph based on their weights. It uses partition, and recursively sorts the subarrays to the left and right of the pivot.
3. The "Partition" method is employed by the QuickSort algorithm to partition the array of edges. It returns the index of the last element after partitioning has occurred, which serves as the pivot for further sorting.
4. The "FindRoot" method recursively finds the roots of the sets using path compression. It takes a vertex "u", and returns the root of the corresponding set to which the vertex belongs.
5. The “UBR” method, which is Union By Rank, operates on two sets represented by their parent and ranks. It also takes two vertices “uSet” and “vSet” and merges the sets together based on their respective ranks.
6. The “showSets” method serves the purpose of displaying the sets that are formed and created as a result of executing the Union By Rank method. It iterates through the vertices of the graph and it finds and identifies the root of each set. Once the root has been found/identified, it then calls the “showSet” method.
7. The “showSet” method is used to display the vertices within the specific sets. It iterates through the vertices while checking and confirming if the root of each vertex is equal to the given root. This finds/identifies the vertices belonging to the set.
8. The “KruskalMST” method is where the Kruskal’s algorithm is implemented to find/identify the minimum spanning tree of the given graph. It iterates through all the vertices and the adjacency list. This creates/forms an “Edge” object for each edge and adds it to the “edges” array. The QuickSort algorithm is then used to sort the edges according to their weights.
9. The “KruskalMST” method then initializes two arrays called “parent” and “rank”, these two arrays are used to track the parent of each vertex and the rank of each set. The sorted “edges” array is then iterated over and then checks if the source vertex and destination vertex belong to different sets for each edge, this is done by using the “FindRoot” method mentioned above. If they happen to belong to different sets, then the sets are merged together using the Union By Rank method. The weight of the edge is then added to the MST weight and the edge is added to the “mst” array.
10. Finally, when the program is finished running, the minimum spanning tree/MST edges and their total weight are then showcased/displayed in the terminal. Also, a step-by-step construction of the sets is displayed in the terminal.

Step-By-Step Construction of The Minimum Spanning Tree and The Union-Find Partition and Set Representations

Set{A}		**A -> A**
Set{B}		**B -> B**
Set{C}		**C -> C**
Set{D}		**D -> D**
Set{E}		**E -> E**
Set{F}		**F -> F**
Set{G}		**G -> G**
Set{H}		**H -> H**
Set{I}		**I -> I**
Set{J}		**J -> J**
Set{K}		**K -> K**
Set{L}		**L -> L**
Set{M}		**M -> M**

1st Traverse:

Edges A to B with weight 1.

Set{AB}		**A -> A**
Set{C}		**B -> A**
Set{D}		**C -> C**
Set{E}		**D -> D**
Set{F}		**E -> E**
Set{G}		**F -> F**
Set{H}		**G -> G**
Set{I}		**H -> H**
Set{J}		**I -> I**
Set{K}		**J -> J**
Set{L}		**K -> K**
Set{M}		**L -> L**
		M -> M

2nd Traverse:

Edges B to C with weight 1.

Set{ABC}		**A -> A**
Set{D}		**B -> A**
Set{E}		**C -> B**
Set{F}		**D -> D**
Set{G}		**E -> E**
Set{H}		**F -> F**
Set{I}		**G -> G**
Set{J}		**H -> H**
Set{K}		**I -> I**
Set{L}		**J -> J**
Set{M}		**K -> K**
		L -> L
		M -> M

3rd Traverse:

Edges D to F with weight 1.

Set{ABC}		**A -> A**
Set{DF}		**B -> A**
Set{E}		**C -> B**
Set{G}		**D -> D**
Set{H}		**E -> E**
Set{I}		**F -> D**
Set{J}		**G -> G**
Set{K}		**H -> H**
Set{L}		**I -> I**
Set{M}		**J -> J**
		K -> K
		L -> L
		M -> M

4th Traverse:

Edges E to G with weight 1.

Set{ABC}		**A -> A**
Set{DF}		**B -> A**
Set{EG}		**C -> B**
Set{H}		**D -> D**
Set{I}		**E -> E**
Set{J}		**F -> D**
Set{K}		**G -> E**
Set{L}		**H -> H**
Set{M}		**I -> I**
		J -> J
		K -> K
		L -> L
		M -> M

5th Traverse:

Edges G to J with weight 1.

Set{ABC}		**A -> A**
Set{DF}		**B -> A**
Set{EGJ}		**C -> B**
Set{H}		**D -> D**
Set{I}		**E -> E**
Set{K}		**F -> D**
Set{L}		**G -> E**
Set{M}		**H -> H**
		I -> I
		J -> G
		K -> K
		L -> L
		M -> M

6th Traverse:

Edges I to K with weight 1.

Set{ABC}		**A -> A**
Set{DF}		**B -> A**
Set{EGJ}		**C -> B**
Set{H}		**D -> D**
Set{IK}		**E -> E**
Set{L}		**F -> D**
Set{M}		**G -> E**
		H -> H
		I -> K
		J -> G
		K -> K
		L -> L
		M -> M

7th Traverse:

Edges J to K with weight 1.

Set{ABC}		**A -> A**
Set{DF}		**B -> A**
Set{EGIJK}		**C -> B**
Set{H}		**D -> D**
Set{L}		**E -> E**
Set{M}		**F -> D**
		G -> E
		H -> H
		I -> K
		J -> G
		K -> J
		L -> L
		M -> M

8th Traverse:

Edges L to M with weight 1.

Set{ABC}		**A -> A**
Set{DF}		**B -> A**
Set{EGIJK}		**C -> B**
Set{H}		**D -> D**
Set{LM}		**E -> E**
		F -> D
		G -> E
		H -> H
		I -> K
		J -> G
		K -> J
		L -> L
		M -> L

9th Traverse:

Edges D to E with weight 2.

Set{ABC}		**A -> A**
Set{DEFGIJK}		**B -> A**
Set{H}		**C -> B**
Set{LM}		**D -> D**
		E -> D
		F -> D
		G -> E
		H -> H
		I -> K
		J -> G
		K -> J
		L -> L
		M -> L

10th Traverse:

Edges F to L with weight 2.

Set{ABC}		**A -> A**
Set{DEFGIJKLM }		**B -> A**
Set{H}		**C -> B**
		D -> D
		E -> D
		F -> D
		G -> E
		H -> H
		I -> K
		J -> G
		K -> J
		L -> F
		M -> L

11th Traverse:

Edges B to D with weight 2.

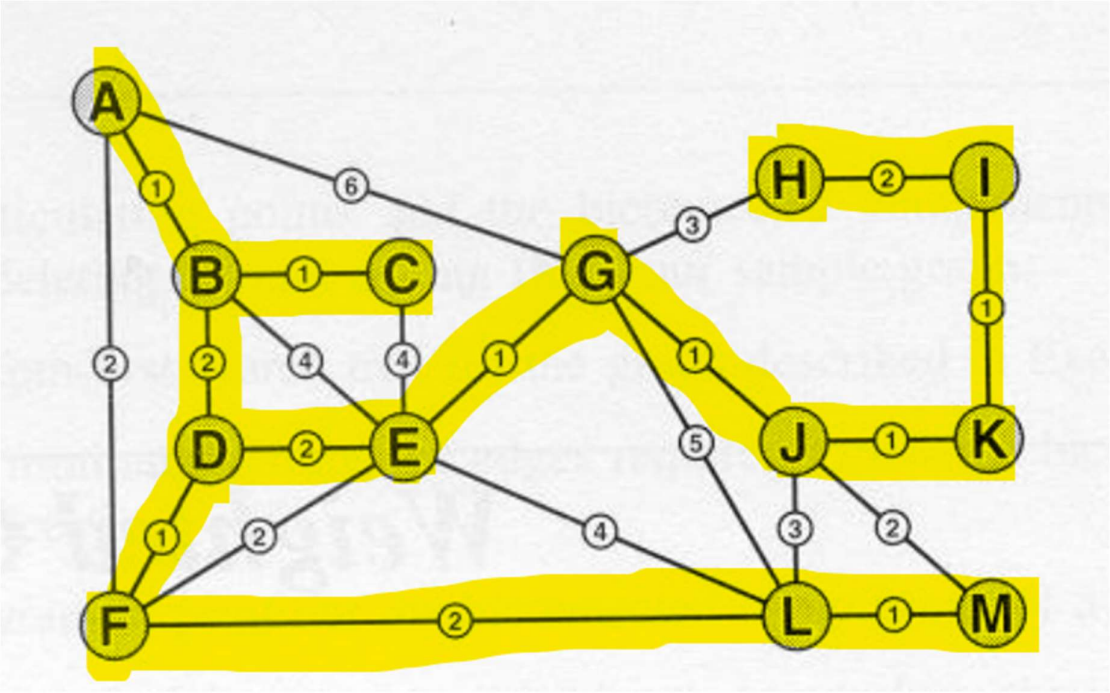
Set{ABCDEFGIJKLM }		**A -> A**
Set{H}		**B -> A**
		C -> B
		D -> B
		E -> D
		F -> D
		G -> E
		H -> H
		I -> K
		J -> G
		K -> J
		L -> F
		M -> L

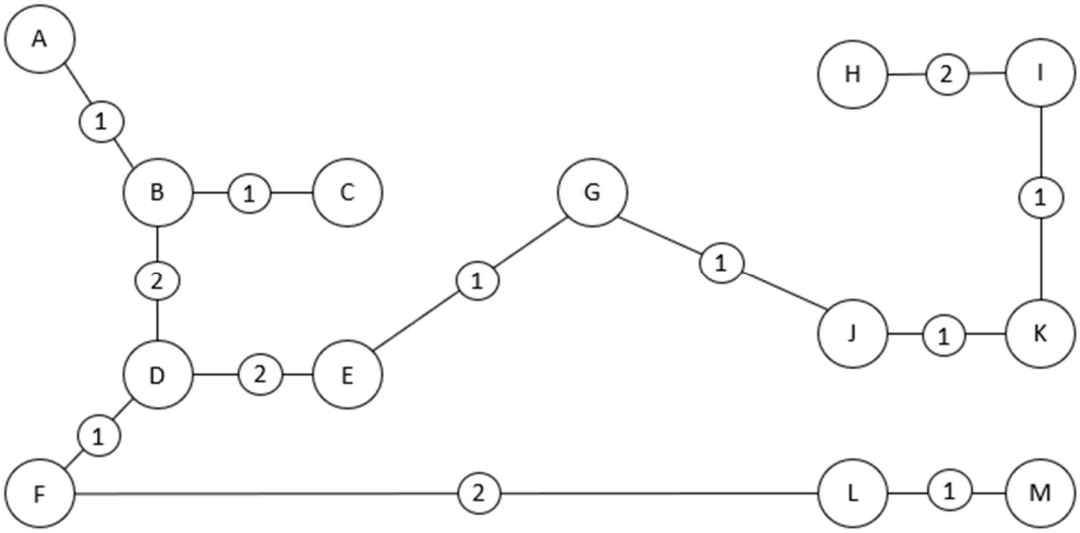
12th Traverse:

Edges I to H with weight 2.

Set{ABCDEFGHIJKLM}		**A -> A**
		B -> A
		C -> B
		D -> B
		E -> D
		F -> D
		G -> E
		H -> I
		I -> K
		J -> G
		K -> J
		L -> F
		M -> L

The Graph Diagram:





The Minimum Spanning Tree weight is 16.

Outputs of Kruskal's Algorithm:

```
Enter the name of the input file:
wGraph1.txt
Parts[] = 13 22
Reading edges from text file
Edge A--(1)--B
Edge A--(2)--F
Edge A--(6)--G
Edge B--(1)--C
Edge B--(2)--D
Edge B--(4)--E
Edge C--(4)--E
Edge D--(2)--E
Edge D--(1)--F
Edge E--(2)--F
Edge E--(1)--G
Edge E--(4)--L
Edge F--(2)--L
Edge G--(3)--H
Edge G--(1)--J
Edge G--(5)--L
Edge H--(2)--I
Edge I--(1)--K
Edge J--(1)--K
Edge J--(3)--L
Edge J--(2)--M
Edge L--(1)--M

adj[A] -> |G | 6| -> |F | 2| -> |B | 1| ->
adj[B] -> |E | 4| -> |D | 2| -> |C | 1| -> |A | 1| ->
adj[C] -> |E | 4| -> |B | 1| ->
adj[D] -> |F | 1| -> |E | 2| -> |B | 2| ->
adj[E] -> |L | 4| -> |G | 1| -> |F | 2| -> |D | 2| -> |C | 4| -> |B | 4| ->
adj[F] -> |L | 2| -> |E | 2| -> |D | 1| -> |A | 2| ->
adj[G] -> |L | 5| -> |J | 1| -> |H | 3| -> |E | 1| -> |A | 6| ->
adj[H] -> |I | 2| -> |G | 3| ->
adj[I] -> |K | 1| -> |H | 2| ->
adj[J] -> |M | 2| -> |L | 3| -> |K | 1| -> |G | 1| ->
adj[K] -> |J | 1| -> |I | 1| ->
adj[L] -> |M | 1| -> |J | 3| -> |G | 5| -> |F | 2| -> |E | 4| ->
adj[M] -> |L | 1| -> |J | 2| ->
```

```
Sorted edges
Edge {A--(1)--B}
Edge {B--(1)--C}
Edge {D--(1)--F}
Edge {E--(1)--G}
Edge {G--(1)--J}
Edge {I--(1)--K}
Edge {J--(1)--K}
Edge {L--(1)--M}
Edge {D--(2)--E}
Edge {E--(2)--F}
Edge {F--(2)--L}
Edge {B--(2)--D}
Edge {H--(2)--I}
Edge {A--(2)--F}
Edge {J--(2)--M}
Edge {G--(3)--H}
Edge {J--(3)--L}
Edge {E--(4)--L}
Edge {B--(4)--E}
Edge {C--(4)--E}
Edge {G--(5)--L}
Edge {A--(6)--G}



Set{A }  Set{B }  Set{C }  Set{D }  Set{E }  Set{F }  Set{G }  Set{H }  Set{I }  Set{J }  Set{K }  Set{L }  Set{M }
Set{A B }  Set{C }  Set{D }  Set{E }  Set{F }  Set{G }  Set{H }  Set{I }  Set{J }  Set{K }  Set{L }  Set{M }
Set{A B C }  Set{D }  Set{E }  Set{F }  Set{G }  Set{H }  Set{I }  Set{J }  Set{K }  Set{L }  Set{M }
Set{A B C }  Set{D F }  Set{E }  Set{G }  Set{H }  Set{I }  Set{J }  Set{K }  Set{L }  Set{M }
Set{A B C }  Set{D F }  Set{E G }  Set{H }  Set{I }  Set{J }  Set{K }  Set{L }  Set{M }
Set{A B C }  Set{D F }  Set{E G J }  Set{H }  Set{I }  Set{K }  Set{L }  Set{M }
Set{A B C }  Set{D F }  Set{E G J }  Set{H }  Set{I K }  Set{L }  Set{M }
Set{A B C }  Set{D F }  Set{E G I J K }  Set{H }  Set{L }  Set{M }
Set{A B C }  Set{D F }  Set{E G I J K }  Set{H }  Set{L M }
Set{A B C }  Set{D E F G I J K }  Set{H }  Set{L M }
Set{A B C }  Set{D E F G I J K L M }  Set{H }
Set{A B C D E F G I J K L M }  Set{H }
Edge {A--(1)--B}
Edge {B--(1)--C}
Edge {D--(1)--F}
Edge {E--(1)--G}
Edge {G--(1)--J}
Edge {I--(1)--K}
Edge {J--(1)--K}
Edge {L--(1)--M}
Edge {D--(2)--E}
Edge {F--(2)--L}
Edge {B--(2)--D}
Edge {H--(2)--I}

Weight of MST: 16
```

Discussion/Analysis/Reflection

I have learnt a massive amount of knowledge and gained an understanding of Prim's, Dijkstra's and Kruskal's algorithms from doing this assignment on graph traversal, MST and SPT algorithms. I also learnt about depth first and breath first.

I learnt that Prim's, Dijkstra's and Kruskal's algorithms all belong to the family of greedy algorithms, which means that their approach for solving something is by selecting the best possible option at that moment of time.

While doing this assignment I learnt a lot about adjacency lists and how they work. I learnt that adjacency lists are a data structure that consists of an array of linked lists and every vertex has a linked list is associated with it which just records the vertices connected to it with an edge and that each node in the linked list represents an edge in a graph.

I also learnt that if a graph is sparse and has fewer edges then Prim's or Dijkstra's algorithm would be more efficient than Kruskal's algorithm. On the other hand, if the graph is more dense and has a larger number of edges then Kruskal's algorithm is more efficient than Prim's or Dijkstra's algorithms.

I have a good understanding of the programs I have written as I have done a lot of research and wrote the code for all the algorithms needed for this assignment.

I now understand that there are many ways to write Kruskal's algorithm and that you don't need to use a heap and can use the QuickSort algorithm as I have done, etc. I also have realized that Prim's and Dijkstra's algorithm are very similar in terms of implementing their code.

I now know how to construct a graph using a text file in order for the program to read it correctly. I have learnt and now understand how to construct minimum spanning trees and shortest path trees step-by-step from a graph and how to read the graphs to understand what is happening and what is needed to be done.

Overall, I have gained a massive amount of knowledge and understanding of all these algorithms that I did not know or have before starting this assignment.