Algorithms & Data Structures – Continuous Assessment 2: MST/SPT Assignment Report

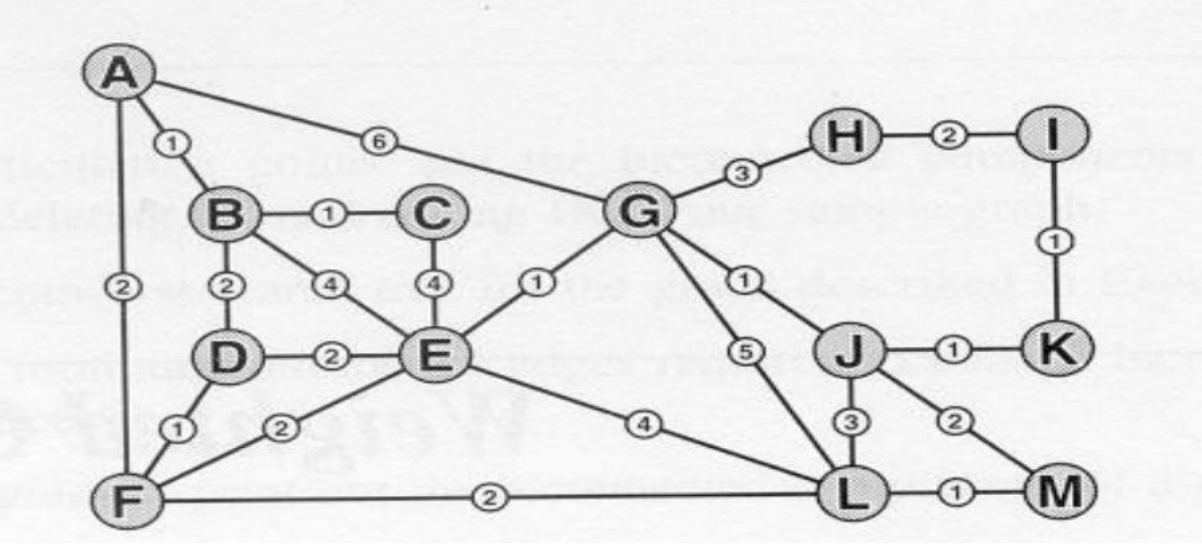
Deyor Abidjanov – C21446556

Introduction:

For this assignment, I've created a Java program to find the Minimum Spanning Tree and Shortest Path Tree of the graph below. I did this by implementing the following algorithms: Prim, Kruskal, Dijkstra, Depth-First, and Breadth-First

A minimum spanning tree (MST) of a weighted graph is a tree that connects all the vertices with minimum possible total edge weight. In other words, it is a subset of the edges that form a tree which includes every vertex and has the minimum possible total edge weight.

A shortest path tree (SPT) of a weighted graph is a tree that connects all the vertices with the minimum possible total distance. Essentially, it is a subset of the edges that form a tree which includes every vertex and has the minimum possible total distance. The distance between two vertices in a graph is the sum of the weights of the edges in the path between them.



Converting the Graph:

```
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
```

I first began by converting the graph above into a text file called wGraph1.txt, so that the program can read the graph. On the first line you'll see, "13 22". 13 is the number of Vertices and 11 is the amount of the Edges. The rest of the lines say the weight each node is connected to its neighbour, so for example on the second line you have 1 connected to 2 which is A to B with the weight of 1. On the 11th line, you see 5 connected to 6. This essentially is E to F with a weight of 2.

A -> 1: [(2, 1), (6, 2), (7, 6)]

B -> 2: [(1, 1), (3, 1), (4, 2), (5, 4)]

C -> 3: [(2, 1), (5, 4)]

D -> 4: [(2, 2), (5, 2), (6, 1)]

E -> 5: [(2, 4), (3, 4), (4, 2), (6, 2), (7, 1), (12, 4)]

F -> 6: [(1, 2), (4, 1), (5, 2), (12, 2)]

G ->7: [(1, 6), (5, 1), (8, 3), (10, 1), (12, 5)]

H -> 8: [(7, 3), (9, 2)]

I -> 9: [(8, 2), (11, 1)]

J -> 10: [(7, 1), (11, 1), (12, 3), (13, 2)]

K -> 11: [(9, 1), (10, 1)]

L -> 12: [(5, 4), (6, 2), (7, 5), (10, 3), (13, 1)]

M -> 13: [(10, 2), (12, 1)]

- The picture above is an adjacency lists diagram showing the graph representation of the sample graph.

MST Prim’s Algorithm:

Prim's Algorithm solves the minimum spanning tree (MST) problem. It is a greedy algorithm that starts with a single vertex and adds edges to create a minimum spanning tree. The algorithm operates on a connected, weighted, undirected graph, and it finds the subset of edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

The algorithm is vertex-based, meaning it builds the MST from a given vertex, adding the smallest edge to the MST. It works by maintaining two disjoint sets of vertices: one set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At each step, the algorithm chooses the vertex in the set of not yet included vertices that is closest to the set of included vertices and adds it to the MST.

Prim's algorithm is like Kruskal's algorithm, but instead of building the MST by adding the smallest edge, Kruskal's algorithm builds the MST by adding the smallest edge that does not create a cycle.

For my program, I use the “Eager” implementation of Prim's MST algorithm. In the Eager implementation, the priority queue contains all vertices that have not yet been added to the MST, and the distance from each of these vertices to the MST is known. The algorithm iteratively adds the vertex with the smallest distance to the MST and updates the distance values of adjacent vertices.

Construction of the Minimum Spanning Tree using Prim’s Algorithm

- Down below are screenshots of the output of my program showing the contents of the heap i.e. the contents of the parent[] & dist[] arrays after each traverse, starting from Vertex L

```
MST Prim's Algorithm starting at vertex L:

Heap: 13
Distance: 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 0 1
Parent: 0 0 0 0 0 0 0 0 0 0 0 12


Heap: 13 10
Distance: 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 3 2147483647 0 1
Parent: 0 0 0 0 0 0 0 0 0 12 0 12


Heap: 13 10 7
Distance: 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647 5 2147483647 2147483647 3 2147483647 0 1
Parent: 0 0 0 0 0 0 12 0 0 12 0 12


Heap: 13 6 7 10
Distance: 2147483647 2147483647 2147483647 2147483647 2147483647 2 5 2147483647 2147483647 3 2147483647 0 1
Parent: 0 0 0 0 0 12 12 0 0 12 0 12


Heap: 13 6 7 10 5
Distance: 2147483647 2147483647 2147483647 2147483647 4 2 5 2147483647 2147483647 3 2147483647 0 1
Parent: 0 0 0 0 12 12 12 0 0 12 0 12


Heap: 6 10 7 5
Distance: 2147483647 2147483647 2147483647 2147483647 4 2 5 2147483647 2147483647 3 2147483647 0 -1
Parent: 0 0 0 0 12 12 12 0 0 12 0 12
```

```
Heap: 6 10 7 5
Distance: 2147483647 2147483647 2147483647 2147483647 4 2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 0 0 0 0 12 12 12 0 0 13 0 0 12


Heap: 10 5 7
Distance: 2147483647 2147483647 2147483647 2147483647 4 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 0 0 0 0 12 12 12 0 0 13 0 0 12


Heap: 10 5 7
Distance: 2147483647 2147483647 2147483647 2147483647 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 0 0 0 0 6 12 12 0 0 13 0 0 12


Heap: 4 10 7 5
Distance: 2147483647 2147483647 2147483647 1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 0 0 0 6 6 12 12 0 0 13 0 0 12


Heap: 4 10 7 5 1
Distance: 2 2147483647 2147483647 1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 0 0 6 6 12 12 0 0 13 0 0 12


Heap: 1 10 7 5
Distance: 2 2147483647 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 0 0 6 6 12 12 0 0 13 0 0 12


Heap: 1 10 7 5
Distance: 2 2147483647 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 0 0 6 6 12 12 0 0 13 0 0 12
```

```
Heap: 1 10 7 5 2
Distance: 2 2 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 4 0 6 6 12 12 0 0 13 0 0 12


Heap: 2 10 7 5
Distance: -2 2 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 4 0 6 6 12 12 0 0 13 0 0 12


Heap: 2 10 7 5
Distance: -2 2 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 4 0 6 6 12 12 0 0 13 0 0 12


Heap: 2 10 7 5
Distance: -2 1 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 1 0 6 6 12 12 0 0 13 0 0 12


Heap: 5 10 7
Distance: -2 -1 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 1 0 6 6 12 12 0 0 13 0 0 12


Heap: 5 10 7
Distance: -2 -1 2147483647 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 1 0 6 6 12 12 0 0 13 0 0 12


Heap: 3 5 7 10
Distance: -2 -1 1 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12
```

```
Heap: 3 5 7 10
Distance: -2 -1 1 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12


Heap: 10 5 7
Distance: -2 -1 -1 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12


Heap: 10 5 7
Distance: -2 -1 -1 -1 2 -2 5 2147483647 2147483647 2 2147483647 0 -1
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12


Heap: 5 7
Distance: -2 -1 -1 -1 2 -2 5 2147483647 2147483647 -2 2147483647 0 -1
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12


Heap: 5 7
Distance: -2 -1 -1 -1 2 -2 5 2147483647 2147483647 -2 2147483647 0 -1
Parent: 6 1 2 6 6 12 12 0 0 13 0 0 12


Heap: 11 7 5
Distance: -2 -1 -1 -1 2 -2 5 2147483647 2147483647 -2 1 0 -1
Parent: 6 1 2 6 6 12 12 0 0 13 10 0 12


Heap: 11 7 5
Distance: -2 -1 -1 -1 2 -2 1 2147483647 2147483647 -2 1 0 -1
Parent: 6 1 2 6 6 12 10 0 0 13 10 0 12
```

```
Heap: 7 5
Distance: -2 -1 -1 -1 2 -2 1 2147483647 2147483647 -2 -1 0 -1
Parent: 6 1 2 6 6 12 10 0 0 13 10 0 12


Heap: 7 5 9
Distance: -2 -1 -1 -1 2 -2 1 2147483647 1 -2 -1 0 -1
Parent: 6 1 2 6 6 12 10 0 11 13 10 0 12


Heap: 9 5
Distance: -2 -1 -1 -1 2 -2 -1 2147483647 1 -2 -1 0 -1
Parent: 6 1 2 6 6 12 10 0 11 13 10 0 12


Heap: 9 5
Distance: -2 -1 -1 -1 2 -2 -1 2147483647 1 -2 -1 0 -1
Parent: 6 1 2 6 6 12 10 0 11 13 10 0 12


Heap: 9 5 8
Distance: -2 -1 -1 -1 2 -2 -1 3 1 -2 -1 0 -1
Parent: 6 1 2 6 6 12 10 7 11 13 10 0 12


Heap: 9 5 8
Distance: -2 -1 -1 -1 1 -2 -1 3 1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 7 11 13 10 0 12


Heap: 9 5 8
Distance: -2 -1 -1 -1 1 -2 -1 3 1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 7 11 13 10 0 12
```

```
Heap: 5 8
Distance: -2 -1 -1 -1 1 -2 -1 3 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 7 11 13 10 0 12


Heap: 5 8
Distance: -2 -1 -1 -1 1 -2 -1 2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12


Heap: 8
Distance: -2 -1 -1 -1 -1 -2 -1 2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12


Heap: 8
Distance: -2 -1 -1 -1 -1 -2 -1 2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12


Heap: 8
Distance: -2 -1 -1 -1 -1 -2 -1 2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12


Heap: 8
Distance: -2 -1 -1 -1 -1 -2 -1 2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12


Heap: 8
Distance: -2 -1 -1 -1 -1 -2 -1 2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12
```

```
Heap: 8
Distance: -2 -1 -1 -1 -1 -2 -1 2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12


Heap:
Distance: -2 -1 -1 -1 -1 -2 -1 -2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12


Heap:
Distance: -2 -1 -1 -1 -1 -2 -1 -2 -1 -2 -1 0 -1
Parent: 6 1 2 6 7 12 10 9 11 13 10 0 12

v = 0 distance = -2147483648 parent = 0
v = 1 distance = -2 parent = 6
v = 2 distance = -1 parent = 1
v = 3 distance = -1 parent = 2
v = 4 distance = -1 parent = 6
v = 5 distance = -1 parent = 7
v = 6 distance = -2 parent = 12
v = 7 distance = -1 parent = 10
v = 8 distance = -2 parent = 9
v = 9 distance = -1 parent = 11
v = 10 distance = -2 parent = 13
v = 11 distance = -1 parent = 10
v = 12 distance = 0 parent = 0
v = 13 distance = -1 parent = 12
```

Output of Program showing Prim's MST

```
Enter the name of the graph you wish to test:
wGraph1.txt
Enter the vertex you wish to start on:
12
13 22
Parts[] = 13 22
Reading edges from text file
A-(1)-B
A-(2)-F
A-(6)-G
B-(1)-C
B-(2)-D
B-(4)-E
C-(4)-E
D-(2)-E
D-(1)-F
E-(2)-F
E-(1)-G
E-(4)-L
F-(2)-L
G-(3)-H
G-(1)-J
G-(5)-L
H-(2)-I
I-(1)-K
J-(1)-K
J-(3)-L
J-(2)-M
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| ->
Depth First Traversal starting at vertex L:
L M J K I H G E F D B C A
Breadth First Traversal starting at vertex L:
L M J G F E K H A D C B I
```

```
Weight of 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
```

SPT Dijkstra’s Algorithm:

Dijkstra's algorithm is a graph traversal algorithm that finds the shortest path between a source vertex and all other vertices in a weighted graph. It works by maintaining a set of visited vertices and a set of unvisited vertices, with the distance to each vertex from the source vertex initially set to infinity except for the source vertex itself, which is set to 0.

At each iteration, the algorithm selects the unvisited vertex with the smallest distance from the source vertex and adds it to the set of visited vertices. Then, it examines all the adjacent vertices to the newly visited vertex and updates their distance values if the distance to the newly visited vertex plus the weight of the connecting edge is less than the current distance value.

The algorithm continues until all vertices have been visited or until the destination vertex has been reached. Once all vertices have been visited, the resulting shortest path tree can be constructed by tracing back from each vertex to the source vertex along the path with the smallest distance value.

The Dijkstra's algorithm is a greedy algorithm, meaning it always chooses the shortest path available at each iteration. However, it only works on graphs with non-negative weights, as negative weights can cause loops and make it difficult to determine the shortest path.

Construction of the Shortest Path Tree using Dijkstra's Algorithm

- Down below are screenshots of the output of my program showing the contents of the heap i.e. the contents of the parent[] & dist[] arrays after each traverse, starting from Vertex L.

```
SPT Dijkstra's Algorithm starting at vertex L:

Heap: 13 6 10 7 5
Distance: 2147483647 2147483647 2147483647 2147483647 4 2 5 2147483647 2147483647 3 2147483647 0 1
Parent: 0 0 0 0 12 12 12 0 0 12 0 0 12


Heap: 6 5 10 7
Distance: 2147483647 2147483647 2147483647 2147483647 4 2 5 2147483647 2147483647 3 2147483647 0 1
Parent: 0 0 0 0 12 12 12 0 0 12 0 0 12


Heap: 10 5 7
Distance: 2147483647 2147483647 2147483647 2147483647 4 2 5 2147483647 2147483647 3 2147483647 0 1
Parent: 0 0 0 0 12 12 12 0 0 12 0 0 12


Heap: 4 1 7 5
Distance: 4 2147483647 2147483647 3 4 2 5 2147483647 2147483647 3 2147483647 0 1
Parent: 6 0 0 6 12 12 12 0 0 12 0 0 12


Heap: 11 1 7 5
Distance: 4 2147483647 2147483647 3 4 2 4 2147483647 2147483647 3 4 0 1
Parent: 6 0 0 6 12 12 10 0 0 12 10 0 12


Heap: 1 5 7 2
Distance: 4 5 2147483647 3 4 2 4 2147483647 2147483647 3 4 0 1
Parent: 6 4 0 6 12 12 10 0 0 12 10 0 12


Heap: 5 9 7 2
Distance: 4 5 2147483647 3 4 2 4 2147483647 5 3 4 0 1
Parent: 6 4 0 6 12 12 10 0 11 12 10 0 12
```

```
Heap: 7 9 2
Distance: 4 5 2147483647 3 4 2 4 2147483647 5 3 4 0 1
Parent: 6 4 0 6 12 12 10 0 11 12 10 0 12


Heap: 9 3 2
Distance: 4 5 8 3 4 2 4 2147483647 5 3 4 0 1
Parent: 6 4 5 6 12 12 10 0 11 12 10 0 12


Heap: 2 8 3
Distance: 4 5 8 3 4 2 4 7 5 3 4 0 1
Parent: 6 4 5 6 12 12 10 7 11 12 10 0 12


Heap: 8 3
Distance: 4 5 8 3 4 2 4 7 5 3 4 0 1
Parent: 6 4 5 6 12 12 10 7 11 12 10 0 12


Heap: 8
Distance: 4 5 6 3 4 2 4 7 5 3 4 0 1
Parent: 6 4 2 6 12 12 10 7 11 12 10 0 12


Heap:
Distance: 4 5 6 3 4 2 4 7 5 3 4 0 1
Parent: 6 4 2 6 12 12 10 7 11 12 10 0 12
```

```
v = @ parent = @ distance = -2147483648
v = A parent = F distance = 4
v = B parent = D distance = 5
v = C parent = B distance = 6
v = D parent = F distance = 3
v = E parent = L distance = 4
v = F parent = L distance = 2
v = G parent = J distance = 4
v = H parent = G distance = 7
v = I parent = K distance = 5
v = J parent = L distance = 3
v = K parent = J distance = 4
v = L parent = @ distance = 0
v = M parent = L distance = 1
```

Output of Program showing Dijkstra's SPT

```
Enter the name of the graph you wish to test:
wGraph1.txt
Enter the vertex you wish to start on:
12
13 22
Parts[] = 13 22
Reading edges from text file
```

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

Entire Code implementing Algorithms Prim & Dijkstra:

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

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

class Heap
{
    private int[] a;       // heap array
    private int[] hPos;    // hPos[a[k]] == k
    private int[] dist;    // dist[v] = priority of v
    private int max;

    private int N;         // heap size

    // The heap constructor gets passed from the Graph:
    //    1. maximum heap size
    //    2. reference to the dist[] array
    //    3. reference to the hPos[] array
    public Heap(int maxSize, int[] _dist, int[] _hPos)
    {
        N = 0; //Assume size of heap is 0.
        max = maxSize;
        a = new int[maxSize + 1];
        dist = _dist; //Given as parameters and you intiliasie them.
        hPos = _hPos;
    }

    //Checks if the heap is empty.
    public boolean isEmpty()
    {
        return N == 0;
    }


    public void siftUp( int k)
    {
        int v = a[k]; //Current position of the vertex.

        a[0] = 0;
        dist[0] = Integer.MIN_VALUE;

        /*While distance value at the current element. Is less than the
distance value at k / 2.
        Keep dividing going up the list to insert the element at the correct
place*/
```

```
        while(dist[v] < dist[a[k/2]]){

            a[k] = a[k/2];

            hPos[a[k]] = k;

            k = k/2;
        }

        a[k] = v;

        hPos[v] = k;
    }


    //removing the vertex at top of heap
    //passed the index of the smallest value in heap
    //siftdown resizes and resorts heap

    public void siftDown( int k)
    {
        int v, j;

        v = a[k];

        while(k <= N/2){

            j = 2 * k;

            if(j < N && dist[a[j]] > dist[a[j + 1]]){
                j++;
            }

            if(dist[v] <= dist[a[j]]){
                break;
            }

            a[k] = a[j];

            hPos[a[k]] = k;

            k = j;
        }
        a[k] = v;
        hPos[v] = k;
    }

    public void printHeap()
```

```
    {
        System.out.print("Heap: ");
        for (int i = 1; i <= N; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }

    public void insert( int x)
    {
        a[++N] = x;
        siftUp( N);
    }


    public int remove()
    {
        int v = a[1];
        hPos[v] = 0; // v is no longer in heap
        a[N+1] = 0;  // put null node into empty spot

        a[1] = a[N--];
        siftDown(1);

        return v;
    }

}

class GraphQueue {
    private int[] q;
    private int capacity;
    private int size;
    private int first;
    private int last;

    public GraphQueue(int capacity) {
        this.capacity = capacity;
        this.size = 0;
        this.first = 0;
        this.last = capacity - 1;
        q = new int[this.capacity];
    }

    public void printQueue() {
        for (int i = first; i <= last; i++) {
            System.out.print(q[i] + " ");
        }
        System.out.println();
```

```
    }

    public void enqueue(int v) {
        if (isFull()) {
            System.out.println("Queue is full");
            return;
        }
        last = (last + 1) % capacity;
        q[last] = v;
        size++;
    }

    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        int v = q[first];
        first = (first + 1) % capacity;
        size--;
        return v;
    }

    public boolean isEmpty() {
        return (size == 0);
    }

    public boolean isFull() {
        return (size == capacity);
    }
}

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

        public Node() {
            this.vert = 0;
            this.wgt = 0;
            this.next = null;
        }

        public Node(int v, int w, Node n) {
            this.vert = v;
            this.wgt = w;
            this.next = n;
```

```
        }
    }

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

    // used for traversing graph
    private int[] visited;
    private int id;

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

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

            t = new Node(u, wgt, 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("");
    }

    public void printArray(int[] a) { //modification
        for (int i = 1; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }

    public void MST_Prim(int s) {
        int v, u;
        int wgt, wgt_sum = 0;
```

```
        int[] dist, parent, hPos;
        Node t;

        // code here
        dist = new int[V + 1];
        hPos = new int[V + 1];
        parent = new int[V + 1];

        System.out.println("MST Prim's Algorithm starting at vertex " +
toChar(s) + ":");

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

        dist[s] = 0;

        Heap h = new Heap(V, dist, hPos);
        h.insert(s);

        while (!h.isEmpty()) {

            v = h.remove();
            wgt_sum += dist[v];
            dist[v] = -dist[v];

            for (t = adj[v]; t != z; t = t.next) {
                u = t.vert;
                wgt = t.wgt;
                if (wgt < dist[u]) {
                    dist[u] = wgt;
                    parent[u] = v;
                    if (hPos[u] == 0) {
                        h.insert(u);
                    } else {
                        h.siftUp(hPos[u]);
                    }
                }
                System.out.println();
                h.printHeap();
                System.out.print("Distance: ");
                printArray(dist);
                System.out.print("Parent: ");
                printArray(parent);
                System.out.println();
            }
```

```
        }

        for (v = 0; v <= V; v++) {
            System.out.println("v = " + v + " distance = " + dist[v] + "
parent = " + parent[v]);
        }
        System.out.print("\n\nWeight of MST = " + wgt_sum + "\n");

        mst = parent;
        showMST();
    }

    public void showMST() {
        System.out.print("\n\nMinimum Spanning tree parent array is:\n");
        for (int v = 1; v <= V; ++v)
            System.out.println(toChar(v) + " -> " + toChar(mst[v]));
        System.out.println("");
    }

    public void SPT_Dijkstra(int s) {
        int v, u;
        int wgt;
        int[] dist, parent, hPos;
        Node t;
        dist = new int[V + 1];
        hPos = new int[V + 1];
        parent = new int[V + 1];

        System.out.println("SPT Dijkstra's Algorithm starting at vertex " +
toChar(s) + ":");

        for (v = 1; v <= V; v++) {
            dist[v] = Integer.MAX_VALUE;
            parent[v] = 0;
            hPos[v] = 0;
        }



        dist[s] = 0;
        v = s;

        Heap h = new Heap(V, dist, hPos);
        h.insert(s);
        while (!h.isEmpty()){
            for(t = adj[v]; t != z; t = t.next){
                u = t.vert;
```

```
                wgt = t.wgt;

                if (dist[u] > dist[v] + wgt){

                    dist[u] = dist[v] + wgt;
                    parent[u] = v;

                    if (hPos[u] == 0){
                        h.insert(u);
                    } else {
                        h.siftUp(hPos[u]);
                    }

                }
            }
            v = h.remove();
            System.out.println();
            h.printHeap();
            System.out.print("Distance: ");
            printArray(dist);
            System.out.print("Parent: ");
            printArray(parent);
            System.out.println();
        }

        for (v = 0; v <= V; v++) {
            System.out.println("v = " + toChar(v) + " parent = " +
toChar(parent[v]) + " distance = " + dist[v]);

        }
        spt = parent;
        showSPT();
    }
    public void showSPT() {
        System.out.print("\n\nShortest Path tree parent array is:\n");
        for (int v = 1; v <= V; ++v)
            System.out.println(toChar(v) + " -> " + toChar(spt[v]));
        System.out.println("");
    }
    public void DF(int s) {
        int v;
        visited = new int[V + 1];
        System.out.println("Depth First Traversal starting at vertex " +
toChar(s) + ":");
        for (v = 1; v <= V; ++v)
            visited[v] = 0;

        DFTraverse(s);
```

```
    }

    public void DFTraverse(int v) {
        Node t;
        visited[v] = 1;
        //Goes through the neighbors of the vertex and if not visited it will
call the method again
        System.out.print(toChar(v) + " ");
        for (t = adj[v]; t != z; t = t.next) {
            if (visited[t.vert] == 0) {
                DFTraverse(t.vert);
            }
        }
    }

    public void breadthFirst(int v) {
        int u;
        Node t;
        GraphQueue q = new GraphQueue(V);
        visited = new int[V + 1];

        System.out.println("Breadth First Traversal starting at vertex " +
toChar(v) + ":");

        //Sets all the vertices to not visited
        for (u = 1; u <= V; ++u) {
            visited[u] = 0;
        }

        visited[v] = 1;
        System.out.print(toChar(v) + " ");

        q.enqueue(v);

        while (!q.isEmpty()) {
            //Goes through the neighbors of the dequeued vertex and if not
visted it will enqueue it
            v = q.dequeue();
            for (t = adj[v]; t != z; t = t.next) {
                u = t.vert;
                if (visited[u] == 0) {
                    visited[u] = 1;
                    System.out.print(toChar(u) + " ");
                    q.enqueue(u);
                }
            }
        }
    }
```

```
}

public class GraphLists {
    public static void main(String[] args) throws IOException {

        Scanner keyInput = new Scanner(System.in);
        System.out.println("Enter the name of the graph you wish to test: ");
        String fname = keyInput.nextLine();

        System.out.println("Enter the vertex you wish to start on: ");
        int startVertex = keyInput.nextInt();

        Graph g = new Graph(fname);

        g.display();

        g.DF(startVertex);
        System.out.println();
        g.breadthFirst(startVertex);
        System.out.println();

        g.MST_Prim(startVertex);
        g.SPT_Dijkstra(startVertex);
    }
}
```

MST Kruskal's Algorithm:

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree of a connected weighted graph. The algorithm works by sorting all the edges in the graph by weight and then adding them to the minimum spanning tree in increasing order of weight, as long as adding the edge will not create a cycle. The algorithm uses a Union-Find data structure to keep track of the connected components of the graph and to determine if adding an edge will create a cycle.

In my implementation, it uses the Union-Find data structure for implementing the UnionFindSets class, which is the implementation of the Union-Find algorithm, that uses the Discrete Set Trees, which is Union by Rank. In my implementation, path compression is not implemented.

The Kruskal's algorithm consists of creating an empty graph, sorting the edges in the input graph by their weights, iterating over each edge, and checking if it creates a cycle by checking if the vertices of the edge are already in the same set in the Union-Find data structure. If they are not in the same set, the edge is added to the minimum spanning tree, and the sets are merged. The algorithm continues until all vertices are in the same set.

The given code implements the Union-Find data structure, the Heap data structure for storing the edges, and Kruskal's algorithm. The main method of the Graph class reads in the graph from a text file and initializes the data structures. Kruskal's algorithm is then run, and the minimum spanning tree is stored in the MST array.

Construction of the MST using Dijkstra's Algorithm

- Down below are screenshots of the output of my program showing the union-find partition and set representations for Kruskal at every traverse.

- The algorithm compares the vertices by the weight of the edges, starting with the lowest edge and moving up.

```
Beginning Kruskal's MST Algorithm
Trees:
A->A  B->A  C->C  D->D  E->E  F->F  G->G  H->H  I->I  J->J  K->K  L->L  M->M
Sets:
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 }

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

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

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

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

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

Trees:
A->A  B->A  C->A  D->D  E->E  F->D  G->E  H->H  I->J  J->J  K->I  L->L  M->L
Sets:
Set{A B C }  Set{D F }  Set{E G }  Set{H }  Set{I J K }  Set{L M }
```

```
Trees:
A->A  B->A  C->A  D->D  E->E  F->D  G->E  H->H  I->J  J->E  K->I  L->L  M->L
Sets:
Set{A B C }  Set{D F }  Set{E G I J K }  Set{H }  Set{L M }

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

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

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

Trees:
A->A  B->A  C->A  D->H  E->D  F->D  G->E  H->A  I->J  J->E  K->I  L->H  M->L
Sets:
Set{A B C D E F G H I J K L M }
```

Output of Program showing Kruskal's MST

```
Enter the name of the file you wish to test:
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
Beginning Kruskal's MST Algorithm
```

```
Weight of MST = 16

Minimum spanning tree build from following edges:
Edge A--1--B
Edge B--1--C
Edge D--1--F
Edge I--1--K
Edge J--1--K
Edge E--1--G
Edge L--1--M
Edge G--1--J
Edge D--2--E
Edge H--2--I
Edge J--2--M
Edge A--2--F
```

Entire Code implementing Kruskal's Algorithm:

```
// Kruskal's Minimum Spanning Tree Algorithm
// Union-find implemented using disjoint set trees without compression

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

class Edge {
    public int u, v, wgt;

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

    public Edge( int x, int y, int w) {
        // missing lines
        this.u = x;
        this.v = y;
        this.wgt = w;
    }

    public void show() {
        System.out.print("Edge " + toChar(u) + "--" + wgt + "--" + toChar(v) +
"\n") ;
    }

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

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


class Heap
{
    private int[] h;
    int N, Nmax;
    Edge[] edge;


    // Bottom up heap construct
    public Heap(int _N, Edge[] _edge) {
        int i;
        Nmax = N = _N;
        h = new int[N+1];
        edge = _edge;

        // initially just fill heap array with
        // indices of edge[] array.
        for (i=0; i <= N; ++i)
            h[i] = i;

        // Then convert h[] into a heap
        // from the bottom up.
        for(i = N/2; i > 0; --i){
            siftDown(i); //missing line
        }
    }

    private void siftDown( int k) {
        int e, j;

        e = h[k];
        while( k <= N/2) {
            //missing lines
            j = 2*k;
            if (j < N && edge[h[j]].wgt > edge[h[j+1]].wgt){
                j++;
            }
            if (edge[e].wgt <= edge[h[j]].wgt){
                break;
            }

            h[k] = h[j];
            k = j;
```

```
        }
        h[k] = e;
    }


    public int remove() {
        h[0] = h[1];
        h[1] = h[N--];
        siftDown(1);
        return h[0];
    }

    public void printHeap(){
        int i;
        System.out.print("Heap: ");
        for(i=1; i<=N; ++i){
            System.out.print(edge[h[i]].wgt + " ");
        }
        System.out.println();
    }

}

/************************************************
*
*       UnionFind partition to support union-find operations
*       Implemented simply using Discrete Set Trees
*
************************************************/

class UnionFindSets
{
    private int[] treeParent;
    private int N;

    public UnionFindSets( int V)
    {
        N = V;
        treeParent = new int[V+1]; //missing line

    }

    public int findSet( int vertex)
    {
        if(treeParent[vertex] == vertex)
            return vertex;
        else
            return findSet(treeParent[vertex]);
    }
```

```
    public void union( int set1, int set2)
    {
        treeParent[set2] = set1;
    }

    public void makeSet( int vertex)
    {
        treeParent[vertex] = vertex;
    }



    public void showTrees()
    {
        int i;
        for(i=1; i<=N; ++i)
            System.out.print(toChar(i) + "->" + toChar(treeParent[i]) + "  "
);
        System.out.print("\n");
    }

    public void showSets()
    {
        int u, root;
        int[] shown = new int[N+1];
        for (u=1; u<=N; ++u)
        {
            root = findSet(u);
            if(shown[root] != 1) {
                showSet(root);
                shown[root] = 1;
            }
        }
        System.out.print("\n");
    }

    private void showSet(int root)
    {
        int v;
        System.out.print("Set{");
        for(v=1; v<=N; ++v)
            if(findSet(v) == root)
                System.out.print(toChar(v) + " ");
        System.out.print("}  ");

    }
```

```
    private char toChar(int u)
    {
        return (char)(u + 64);
    }
}

class Graph
{
    private int V, E;
    private Edge[] edge;
    private Edge[] mst;

    public Graph(String graphFile) throws IOException
    {
        int u, v;
        int w, e;

        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 edge array
        edge = new Edge[E+1];

       // 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]);
            w = Integer.parseInt(parts[2]);

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

            // create Edge object
            edge[e] = new Edge(u, v, w);
        }
    }
```

```
/**********************************************************
*
*       Kruskal's minimum spanning tree algorithm
*
**********************************************************/
public Edge[] MST_Kruskal()
{
    int ei, i = 0;
    Edge e;
    int uSet, vSet;
    UnionFindSets partition;
    int wgt_sum = 0;

    // create edge array to store MST
    // Initially it has no edges.
    mst = new Edge[V-1];

    // priority queue for indices of array of edges
    Heap h = new Heap(E, edge);

    System.out.println("Beginning Kruskal's MST Algorithm");

    // create partition of singleton sets for the vertices
    partition = new UnionFindSets(V);
    for(i=1; i<=V; ++i){
        partition.makeSet(i);
    }


    i = 0;
    while(i < V-1) {

        ei = h.remove();
        e = edge[ei];

        uSet = partition.findSet(e.u);
        vSet = partition.findSet(e.v);

        if(uSet != vSet) {

            mst[i++] = e;
            wgt_sum += e.wgt;

            partition.union(uSet, vSet);
        }
        System.out.println("Trees: ");
        partition.showTrees();
```

```
        System.out.println("Sets: ");
        partition.showSets();
        System.out.println();
    }

    System.out.println("Weight of MST = " + wgt_sum);

    return mst;
}


    // convert vertex into char for pretty printing
    private char toChar(int u)
    {
        return (char)(u + 64);
    }

    public void showMST()
    {
        System.out.print("\nMinimum spanning tree build from following
edges:\n");
        for(int e = 0; e < V-1; ++e) {
            mst[e].show();
        }
        System.out.println();

    }

} // end of Graph class

    // test code
class KruskalTrees {
    public static void main(String[] args) throws IOException
    {

        Scanner keyInput = new Scanner(System.in);
        System.out.println("Enter the name of the file you wish to test: ");
        String fname = keyInput.nextLine();



        Graph g = new Graph(fname);

        g.MST_Kruskal();

        g.showMST();

    }
}
```

Learning Outcomes:

- From doing this assignment, I learned how to convert graphs into text files so that programs can read from them. Knowing this will help me with working with other algorithms in the future which deals with graphs.

- After further research, I learned more about the real-life applications of these algorithms. The algorithms are used in various fields such as transportation and network design. For example, I learned that Kruskal's algorithm is used to build efficient electrical power grids by connecting power stations to cities with the minimum amount of transmission lines.

- I also learned that Kruskal performs better than Prim with more sparse graphs. or when the MST is more forest-like. This is because Kruskal's algorithm processes the edges in increasing order of weight, adding edges to the MST as long as they do not form cycles. This guarantees that the resulting MST is a forest. Kruskal's Algorithm found the MST in fewer iterations compared to Prim's Algorithm.