Graph Traversal, MST and SPT Assignment

Ella Fahey, C22396101

Introduction

Throughout the duration of the completion of this assignment, I implemented Prim's and Kruskal's' algorithm for finding the minimum spanning tree for a weighted connected graph and Dijkstra's shortest path tree algorithm. This was accomplished by creating two separate java files, one for Kruskal's algorithm called 'Kruskal.java' and another for Prim and Dijkstra's called 'GraphLists.java'. These algorithms serve as invaluable tools in refining optimal routes between nodes. I've integrated Depth-First Search(DFS) and Breadth-First Search (BFS) algorithms. For determining the shortest path within a weighted graph, Dijkstra's and Prim's algorithms have been employed. The graph representation relies on adjacency lists, where each node encapsulates the adjacent vertex along with the weight of the connecting edge displaying a clear representation of the graph.

wGraph1.txt

For the graph representation I've made use of adjacency lists. In these lists, each node contains information about its adjacent vertices and the corresponding edge weights, offering a compact and efficient way to capture the graphs connectivity. Graph traversal techniques like DFS and BFS, are made use of to navigate through the graph efficiently. Depth First Search, explores as far as possible along each branch before backtracking, making it subtle for tasks like cycle detection and topological sorting. Conversely, Breadth First Search explores all neighbour nodes at the present depth before moving onto nodes at the next depth level, making it ideal for tasks like finding the shortest path and discovering connected components within the graph.

How the graph works

Graph(String graphFile)

- I implemented this method to initialize a graph by reading is data from a text file. This was accomplished by file reading, parsing, initialization, edge processing and adjacency list construction.

- The graph is opened with the ‘GraphLists.java’ using ‘FileReader’ and ‘BuffferedReader’ to efficiently read from the file.
- The parsing method reads the first line of the file to determine the number of vertices (V) and edges(E) in the graph. It splits this line based on whitespace and parses the values accordingly.
- The initialization of a sentinel node ‘z’ creates an array of adjacency lists ‘adj’ with V+1 elements, each initialized to ‘z’.
- Edge processing is employed to read the subsequent lines of the file, each representing an edge in the graph. For each edge, it parses the source vertex ‘u’, destination vertex ‘v’ and edge weight ‘wgt’.
- Within the adjacency list construction, each edge creats a new node ‘t’ to represent the edge in the adjacency list. It sets the vertex and weight values for ‘t’ and links it appropriately in the adjacency list for both ‘u’ and ‘v’.

Public void display()

- The method, display() is responsible for printing out the adjacency list representation of the graph. This is done by looping through the vertices in the graph from 1 to V
- For each vertex it prints the label of the vertex (‘toChar(V)’) along with an arrow indicating the start of the adjacency list.
- It also iterates through the adjacency list, within each iteration it traverses through the adjacency list of the current vertex ‘v’ . This traversal is done using the ‘n’ pointer starting from the head of the adjacency list.
- Overall this method provides a clear and organized visualization of the graphs adjacency list, making it easier for users to understand the structure of connections withing the graph.

DFS(int s)

- This method performs a depth-first traversal of the graph starting from the vertex ‘s’.
- It recursively explores each vertex and its adjacent vertices in a depth-first manner, marking visited vertices along the way.
- The time complexity is O(V +E), where V is the number of vertices and E is the number of edges, because it needs to visit every vertex and edge in the graph

MST_Prim(int s)

- This method finds the minimum spanning tree(MST) of the graph using Prim's algorithm, starting from the vertex 's'.
- It iteratively grows the MST by adding the minimum weight edge that connects a vertex in the MST to a vertex outside the MST. This is achieved by maintaining a priority queue of vertices based on their edge weights
- The time complexity for this is O(E log V), where V is the number of vertices and E is the number of edges, because it uses a priority queue to select the next vertex with the minimum weight.

Private int minDistance(int[] dist, Boolean[] visited)

- This finds the vertex with the minimum weight that has not yet been visited, which is used in Dijkstra's algorithm and Prim's algorithm.
- This iterates through all the vertices in the graph and selects the vertex with the minimum distance value from the source vertex, excluding vertices that have already been visited,
- The time complexity for this is O(V), where V is the number of vertices, because it needs to iterate through all the vertices in the graph to find the minimum distance vertex.

Prim's Algorithm

Prims algorithm is implemented to fine the minimum spanning tree (MST) of a connected, undirected graph. It starts by selecting an arbitrary vertex. Then it incrementally constructs the MST by adding the edge with the smallest weight that connects a vertex within the MST to one outside of it. This process continues until all vertices are included in the MST. The algorithm maintains a priority queue of potential edges based on their weights, choosing the edge with the least weight at each step. The time complexity of Prim's algorithm is O(E log V), where V is the number of vertices and E is the number of edges in the graph.

Dijkstra's Algorithm

Dijkstra's algorithm is a renowned method for finding the shortest path from a single source vertex to all other vertices in a weighted graph. It maintains a asset of vertices whose shortest distance from the source is already known and continually expands this set by selecting the vertex with the smallest distance. It then updates the distances to its adjacent vertices if a shorter path is found. This process repeats until all vertices have been explored. Dijkstra's algorithm relies on a priority queue to efficiently select the next vertex with the shortest distance. The time complexity of Dijkstra's algorithm is O((V + E)log V), where V is the number of vertices and E is the number of edges in the graph.

Breadth-first Search(BFS)

Breadth-first search serves a crucial graph traversal technique for exploring all vertices within a graph systematically. It commences from a designated source vertex and thoroughly explores its adjacent vertices before moving on to the next level of vertices. BFS relies on a queue data structure to manage the vertices awaiting exploration. By methodically traversing vertices level by level, BFS ensures the discovery of the shortest paths from the source vertex to all others in unweighted graphs. The time complexity of BFS is O(V + E), with E representing the number of edges and V the vertices in the graph.

Depth-First Search(DFS)

Depth-first Search(DFS), however, represents another essential graph traversal approach used for systematically navigating through all vertices and edges. Unlike BFS, DFS delves as deeply as possible along each branch before retracting its steps. It commences from a chosen source vertex and probes as extensively as feasible along each branch before backtracking. DFS typically employs either a stack data structure or recursion to manage the vertices awaiting exploration. It fins applications in solving problems related to connected components, cycle detection and topological sorting. The time complexity of DFS is also O(V + E).

Kruskal's Algorithm

Kruskal's algorithm is a fundamental method for determining the minimum spanning tree (MST) of a connected, undirected graph. It begins by treating each vertex as an individual component and gradually merges them to form the MST. The algorithm iteratively selects the shortest edge that does not create a cycle when added to the MST. This process continues until all vertices are connected, resulting in the creation of the MST with the minimum total edge weight. This algorithm commonly employs a disjoint set data structure to efficiently manage component merging and cycle detection in the MST construction process.

Adjacency List:

0 : [ ]

1 : [7, 6] → [6, 2] → [2, 1]

2 : [5, 4] → [4, 2] → [3, 1] → [1, 1]

3 : [5, 4] → [2, 1]

4 : [6, 1] → [5, 2] → [2, 2]

5 : [12, 4] → [7, 1] → [6, 2] → [4, 2] → [3, 4] → [2, 4]

6 : [12, 2] → [5, 2] → [4, 1] → [1, 2]

7 : [12, 5] → [10, 1] → [8, 3] → [5, 1] → [1, 6]

8 : [9, 2] → [7, 3]

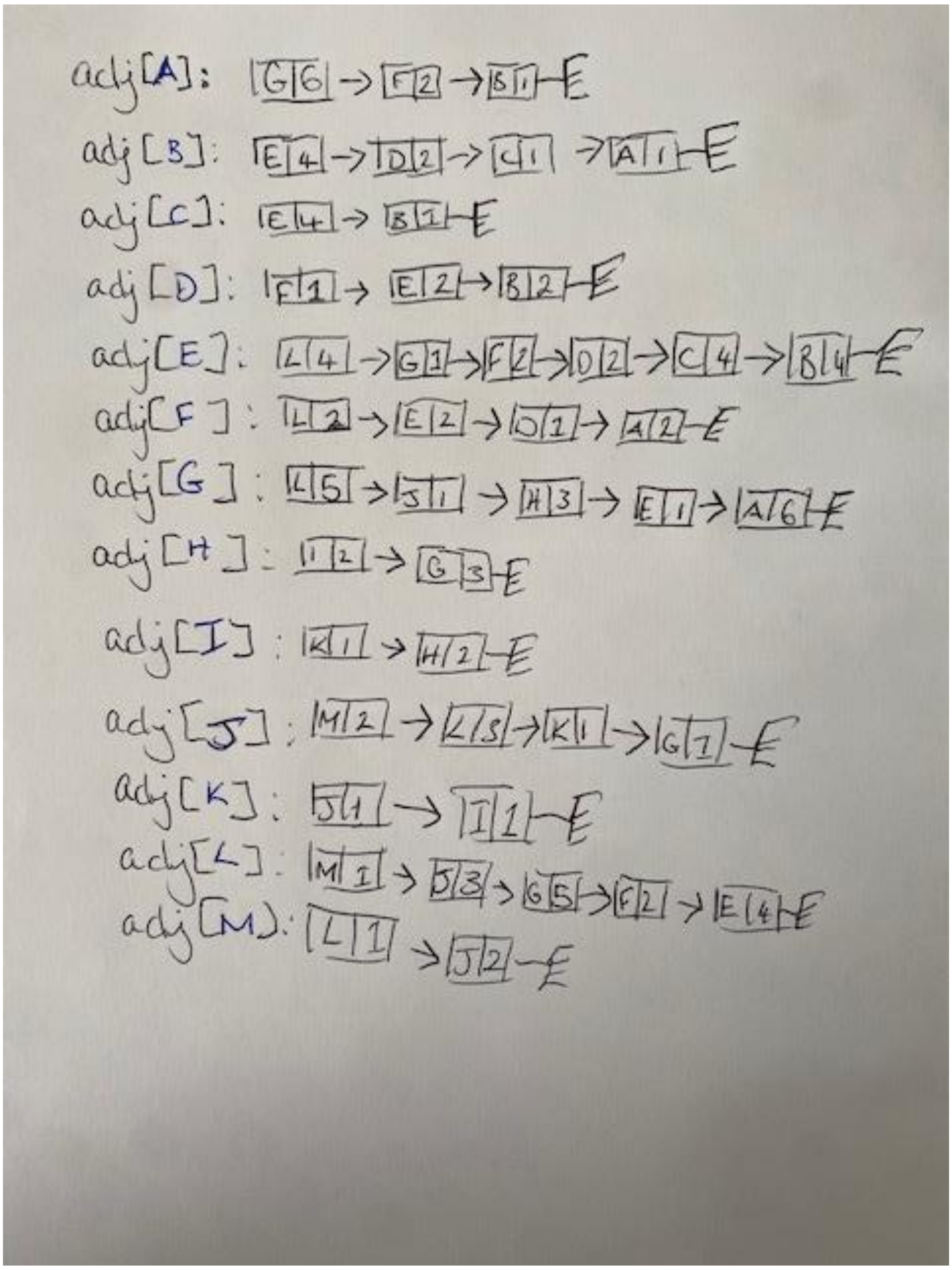
9 : [11, 1] → [8, 2]

10 : [13, 2] → [12, 3] → [11, 1] → [7, 1]

11 : [10, 1] → [9, 1]

12 : [13, 1] → [10, 3] → [7, 5] → [6, 2] → [5, 4]

13 : [12, 1] → [10, 2]



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] ⏚

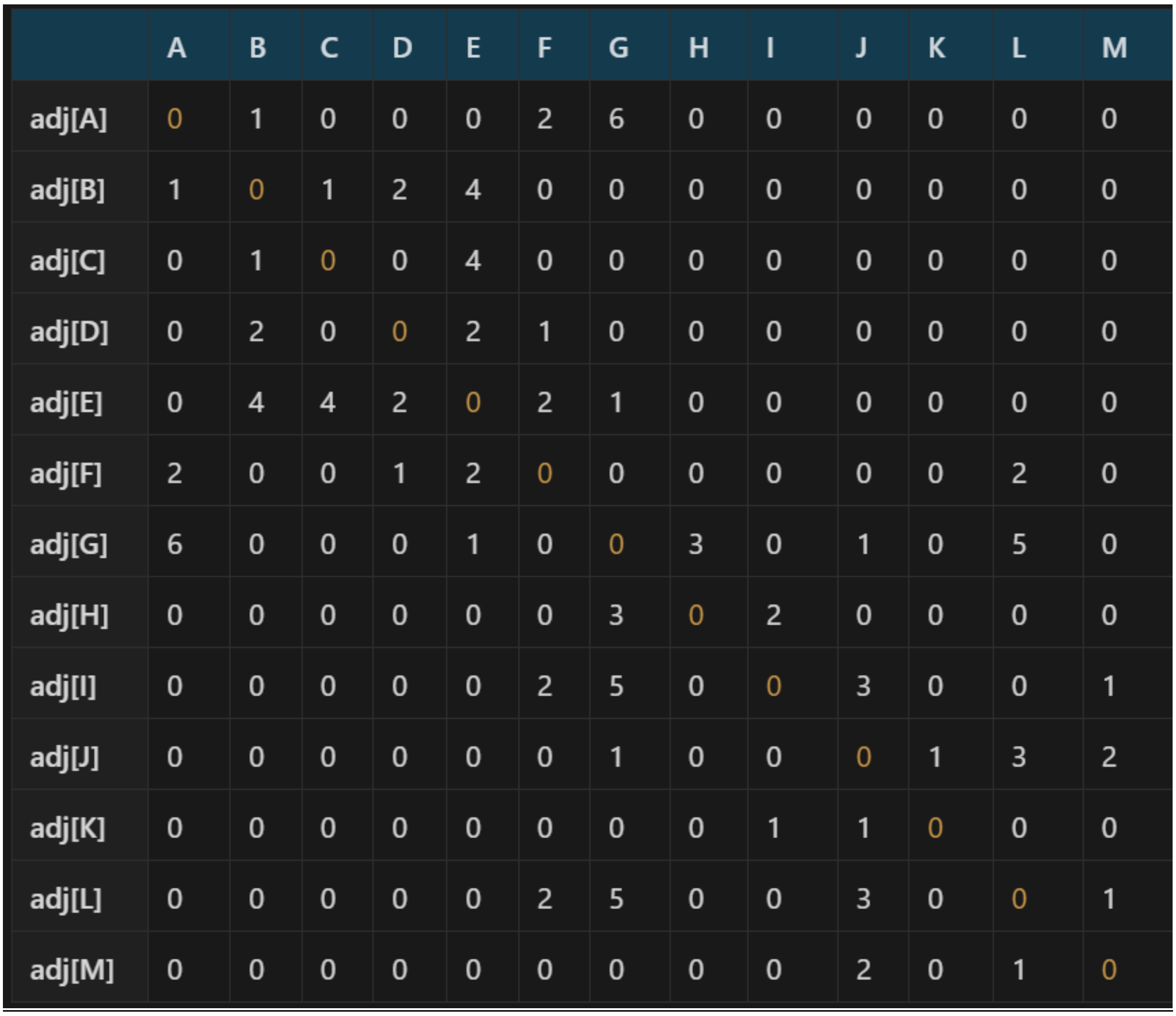
adj[I]: [K|1] → [H|2] ⏚

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] ⏚



	A	B	C	D	E	F	G	H	I	J	K	L	M
adj[A]	0	1	0	0	0	2	6	0	0	0	0	0	0
adj[B]	1	0	1	2	4	0	0	0	0	0	0	0	0
adj[C]	0	1	0	0	4	0	0	0	0	0	0	0	0
adj[D]	0	2	0	0	2	1	0	0	0	0	0	0	0
adj[E]	0	4	4	2	0	2	1	0	0	0	0	0	0
adj[F]	2	0	0	1	2	0	0	0	0	0	0	2	0
adj[G]	6	0	0	0	1	0	0	3	0	1	0	5	0
adj[H]	0	0	0	0	0	0	3	0	2	0	0	0	0
adj[I]	0	0	0	0	0	2	5	0	0	3	0	0	1
adj[J]	0	0	0	0	0	0	1	0	0	0	1	3	2
adj[K]	0	0	0	0	0	0	0	0	1	1	0	0	0
adj[L]	0	0	0	0	0	2	5	0	0	3	0	0	1
adj[M]	0	0	0	0	0	0	0	0	0	2	0	1	0

Construction of the MST using Prim’s Algorithm

Initial Starting point from vertex L (number 12):

Heap:

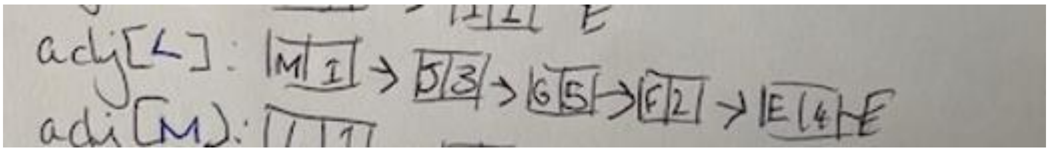
E4

F2

G5

J3

M1



Heap

L0

Parent[]:

L(0)

Dist[]:

L(1)

Everything else(Infinity)

Explanation

In step 0, the algorithm commences with vertex L as the starting point. The heap is initialized with L and its distance set to 0. All other vertices are marked as unvisited with their distances set to infinity, except for L, which has a distance of 0. Additionally, both the parent and distance arrays are initialized: every vertex is assigned 0 as its parent, indicating that none are yet connected, and infinity is assigned as the distance from L to all other vertices except L itself, where the distance is 0.

Sep 2: First Iteration:

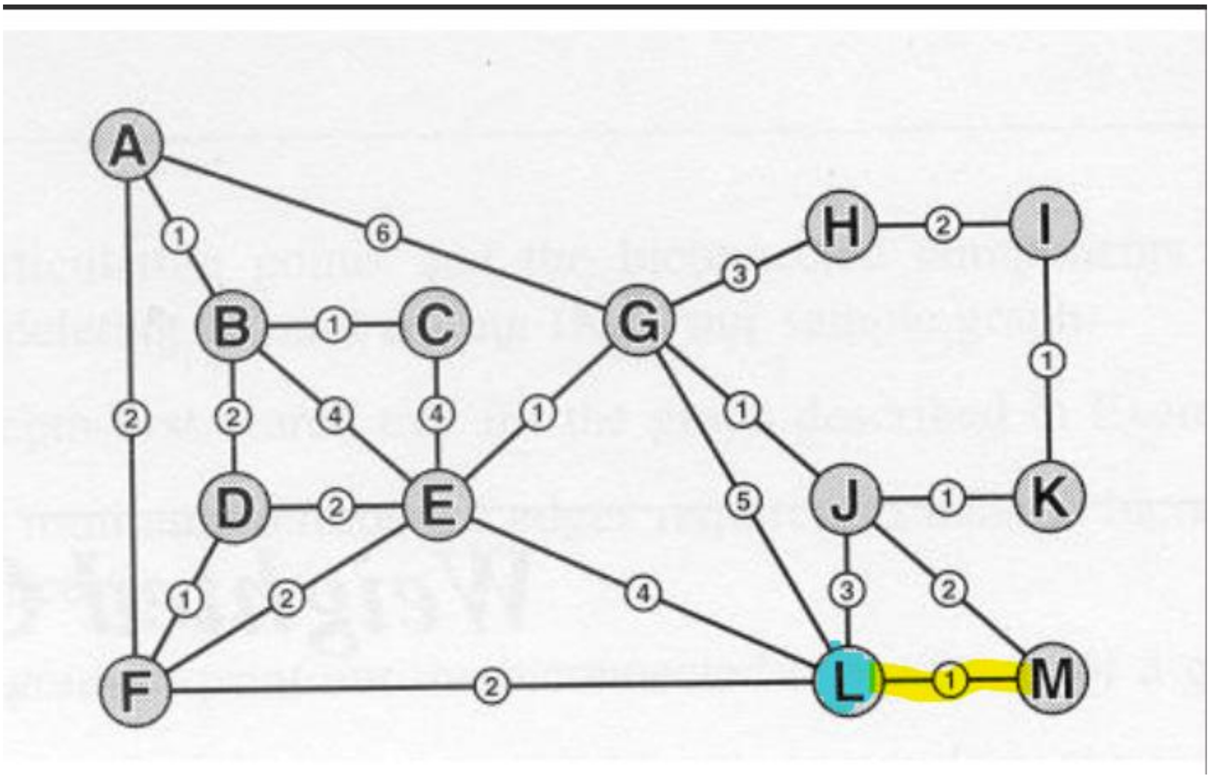
- Select vertex M with minimum heap value from the heap.
- Explore its adjacent vertices and update their distances if necessary.

Heap contents:
E4
F2
G5
J3

Parent[]:
L0
ML

Dist[]:
L1
M1
Everything else (Infinity)

Graph representation



Step 3: Second Iteration

- Select vertex F with minimum heap value from the heap
- Explore its adjacent vertices and update their distances if necessary

Heap contents:

E4

G5

J3

Parent[]:

L0

ML

FM

Dist[]:

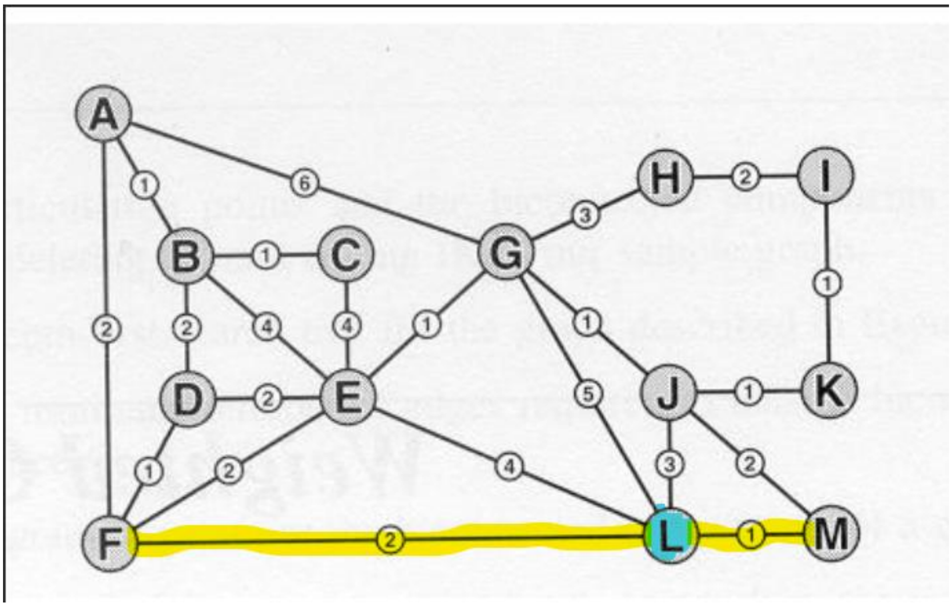
L1

M1

F2

Everything else (Infinity)

Graph representation:



Step 4: Third Iteration

- Select vertex J with minimum heap value from the heap
- Explore its adjacent vertices and update their distances if necessary

Heap contents:

E4
G5

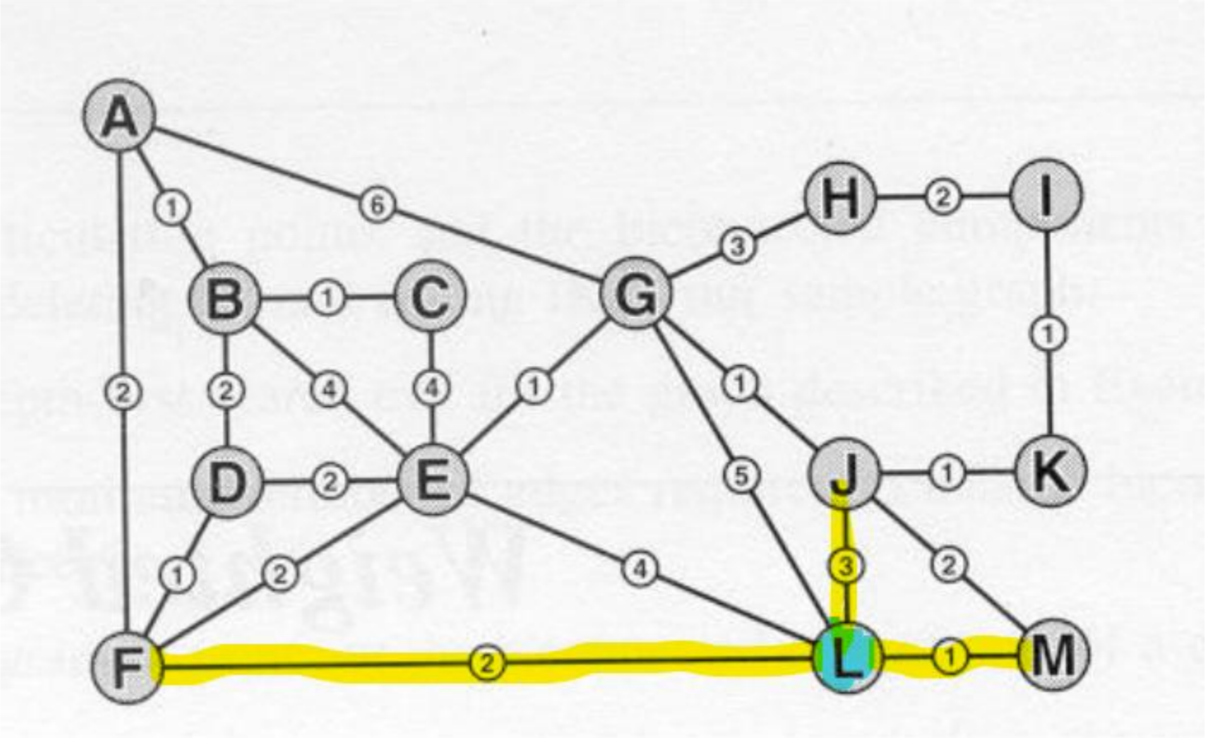
Parent[]:

L0
ML
FM
JF

Dist[]:

L1
M1
F2
J3
Everything else (Infinity)

Graph representation:



Step 5: Fourth Iteration

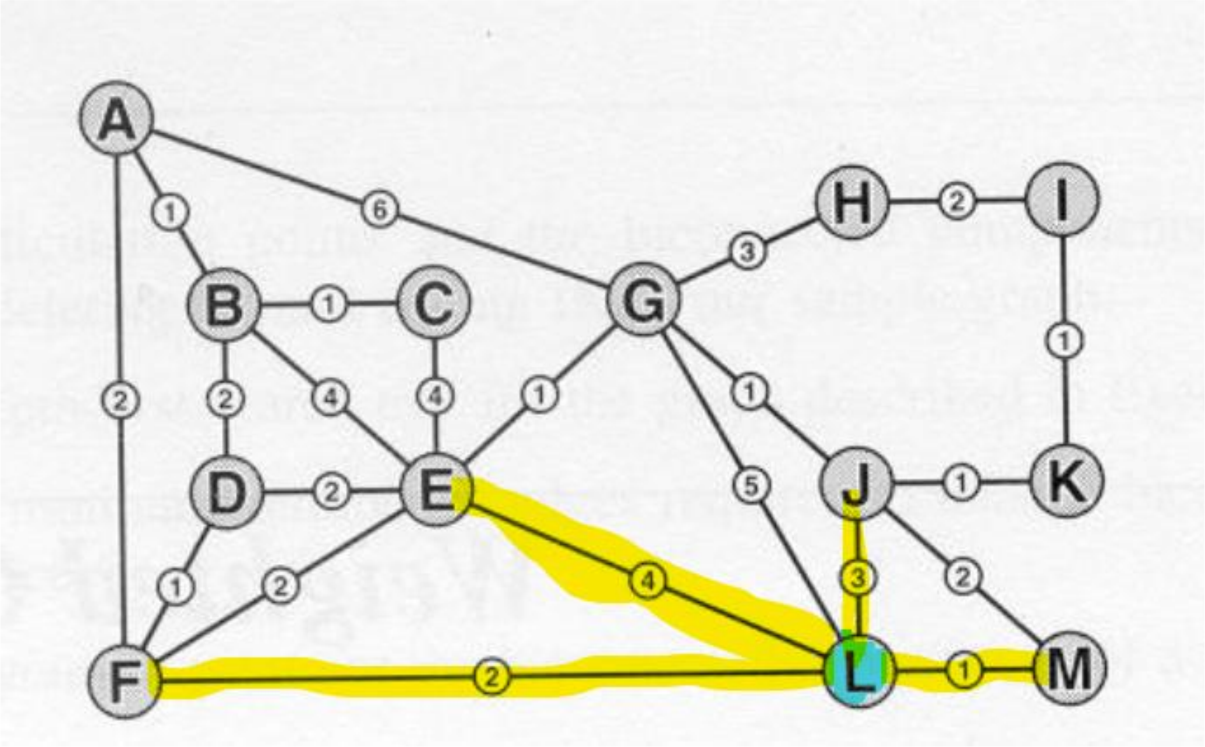
- Select vertex E with minimum heap value from the heap
- Explore its adjacent vertices and update their distances if necessary

Heap contents:
G5

Parent[]:
L0
ML
FM
JF
EJ

Dist[]:
L1
M1
F2
J3
E4
Everything else (infinity)

Graph Representation:



Step 6: Fifth Iteration

- Select vertex G with minimum heap value from the heap
- Explore its adjacent vertices and update their distance if necessary

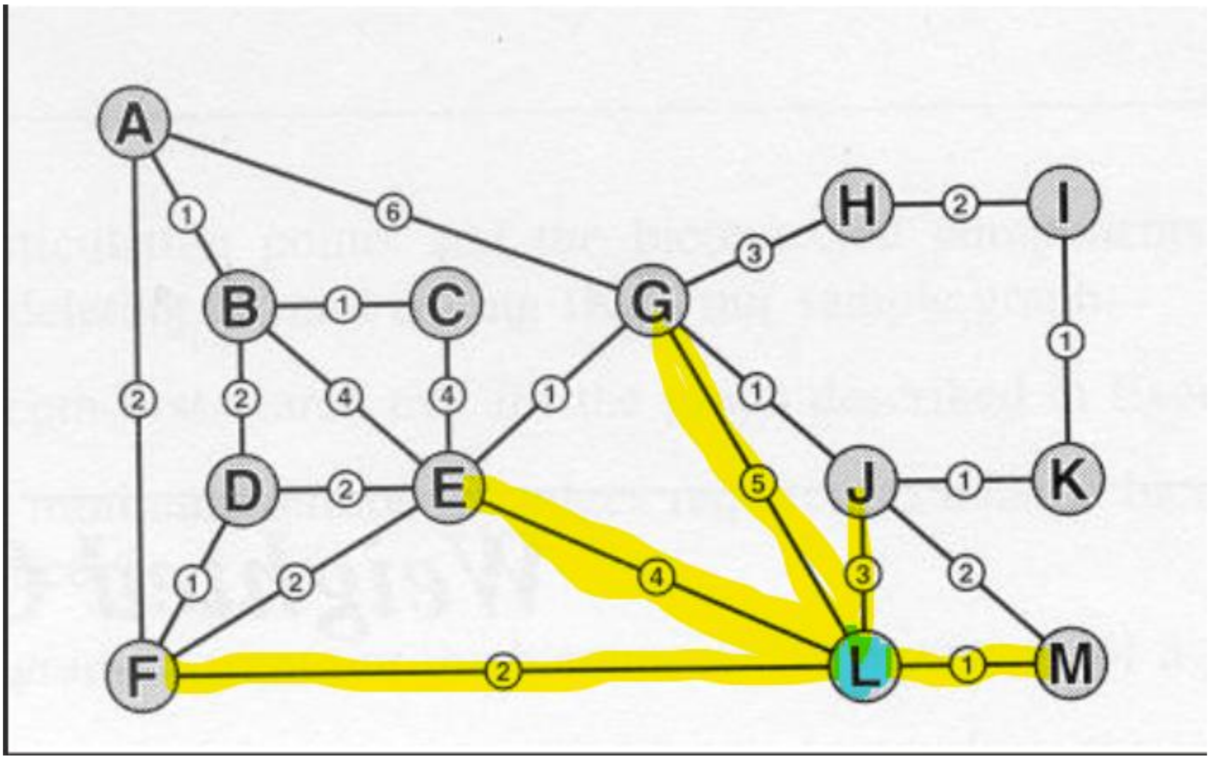
Heap contents:
Empty

Parent[]:
L0
ML
FM
JF
EJ
GE

Dist[]:
L1
M1

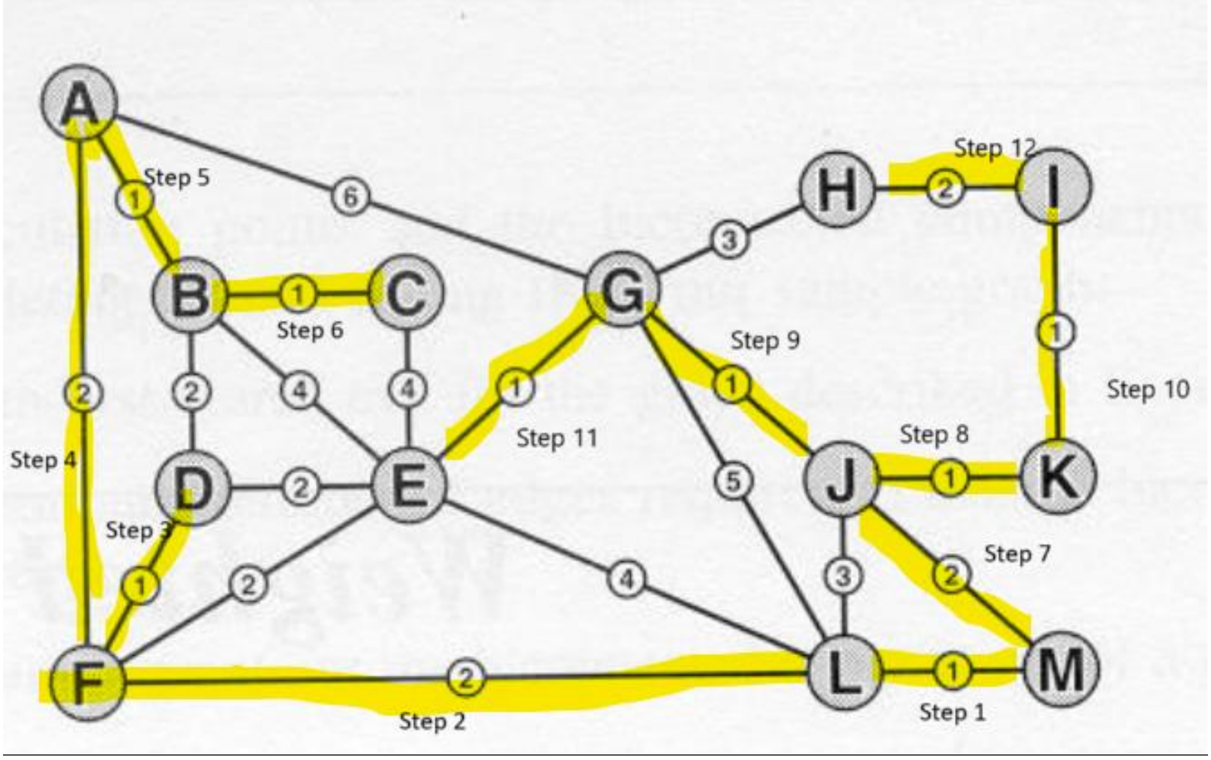
F2
J3
E4
G5
Everything else (infinity)

Graph representation:



This process continues until all vertices are included in the MST. These 6 steps demonstrates the construction of the MST using Prims algorithm with vertex L as the starting point.

Diagram Showing Prim's MST Superimposed On The Graph



SPT using Dinkstra's algorithm

Step 0

- Initialize the heap with L and its distance 0
- All other vertices are unvisited, and their distances are set to infinity except for L, which has a distance of 0.
- Initialize the parent[] array with 0 as the parent[] of all vertices and infinity as the distance of all vertices from L except for L, which has a distance of 0.

Heap:

L0

Parent Array:

L: 0,

A: 0,

B: 0,

C: 0,

D: 0,

E: 0,

F: 0,

G: 0,

H: 0,

I: 0,

J: 0,

K: 0,

M: 0

Dist[] array:

L: 0,

Everything else(infinity)

Step 1

- Select the vertex with the smallest distance from the heap, which is L
- Explore the neighbours of L(A,B,G) and update their distances if a shorter path is found
- Remove L from the heap since its visited

Heap contents:

A4

B2

G5

Parent[]:

L: 0

A: L

B: L,

C: 0

D: 0

E: 0

F: 0

G: L

H: 0

I: 0

J: 0

K: 0

M: 0

Dist[]:

L: 0

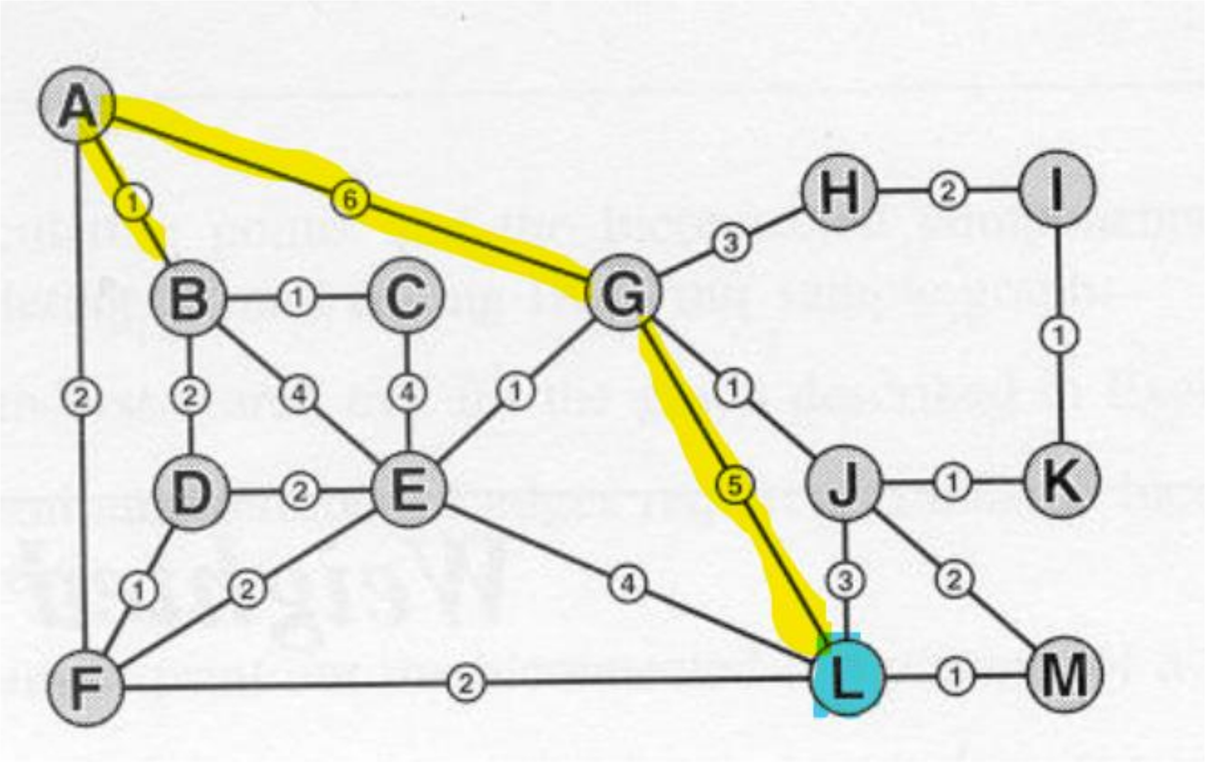
A: 4

B: 2

G: 5

Everything else(infinity)

Graph representaion:



Step 2

- Select the vertex with the smallest distance from the heap which is B
- Explore the neighbours of B(A,D,F) and update their distances if a shorter path is found
- Remove B from the heap since its visited

Heap contents:

A3

D5

F4

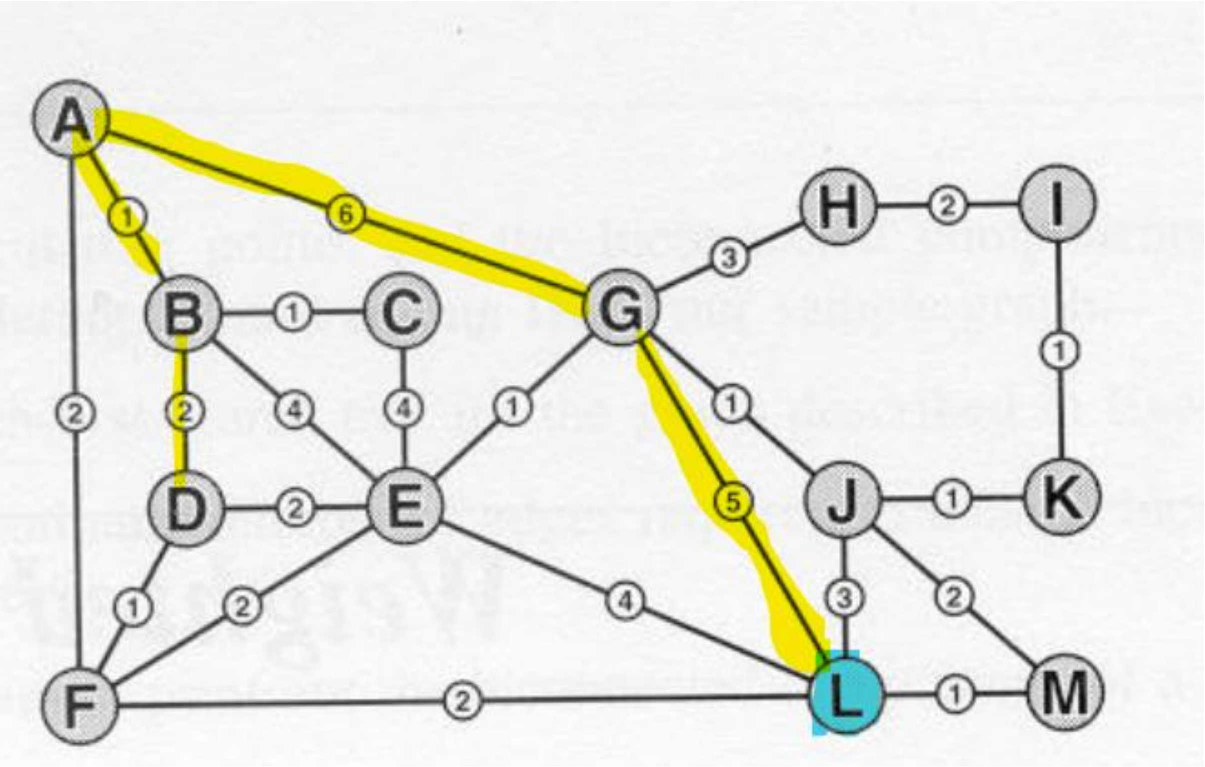
G5

Parent[]:
L: 0
A: B
B: L
C: 0
D: B

E: 0
F: B
G: L
H: 0
I: 0
J: 0
K: 0
M: 0

Dist[]:
L: 0
A: 3
B: 2
D: 7
F: 6
G: 5
Everything else(infinity)

Graph representaion:



Step 3

- Select the vertex with the smallest distance from the heap, which is A

- Explore the neighbour of A© and update its distance if a shorter path is found
- Remove A from the heap since its visited

Heap contents:

C5
D5
F4
G5

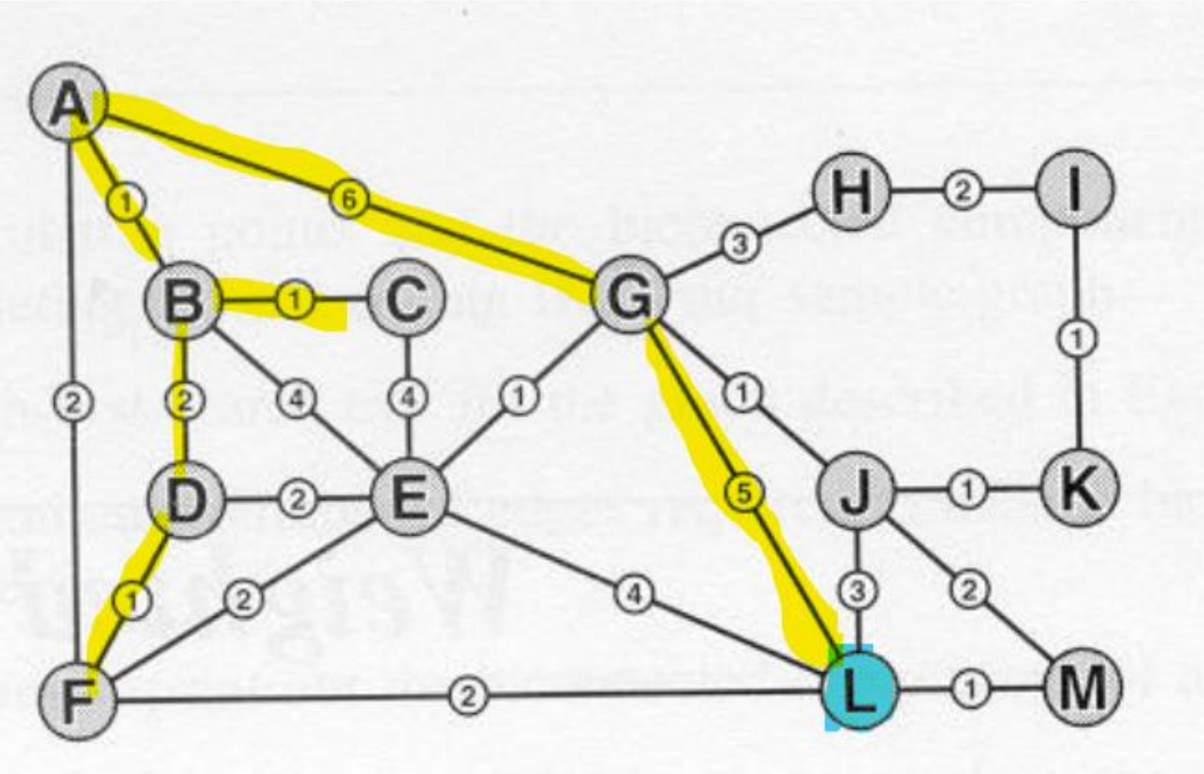
Parent[]:

L: 0
A: B
B: L
C: A
D: B
E: 0
F: B
G: L
H: 0
I: 0
J: 0
K: 0
M: 0

Dist[]:

L: 0
A: 3
B: 2
C: 8
D: 7
F: 6
G: 5
Everything else(infinity)

Graph representation:



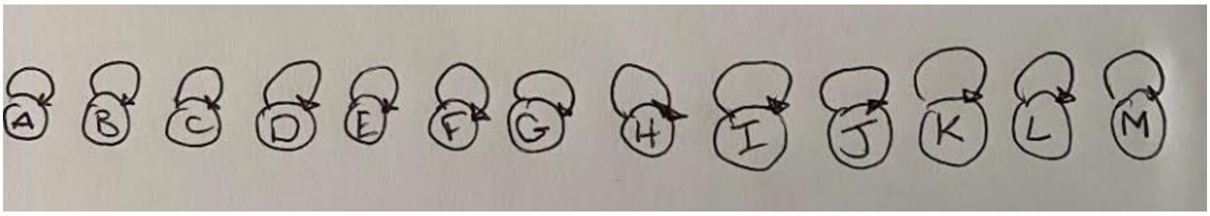
This process continues until all vertices are visited and the shortest paths from vertex L to all other vertices are computed.

CONSTRUCTION OF THE MST USING KRUSKAL ALGORITHM WITH SET REPRESENTATION AND UNION-FIND PARTITION

Initial State:

Sets ; { A} {B} {C} {D} {E} {F} {G} {H} {I} {J} {K} {L} {M}

Union-Find partition

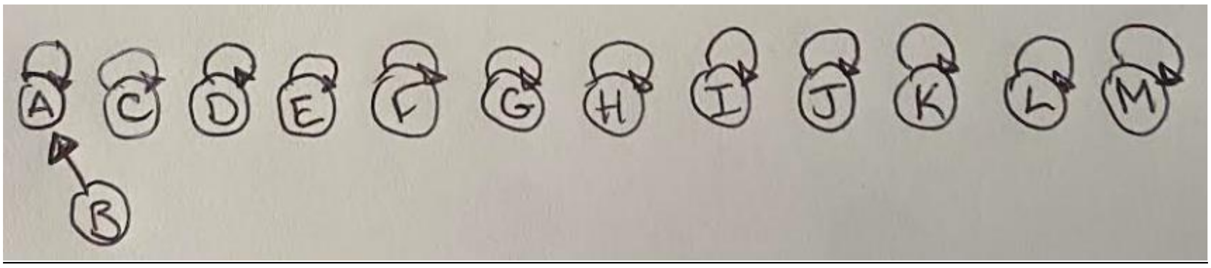


Step 1: MST_Kruskal

A 1 B

Sets: {AB} {C} {D} {E} {F} {G} {H} {I} {J} {K} {L} {M}

Union Find partition

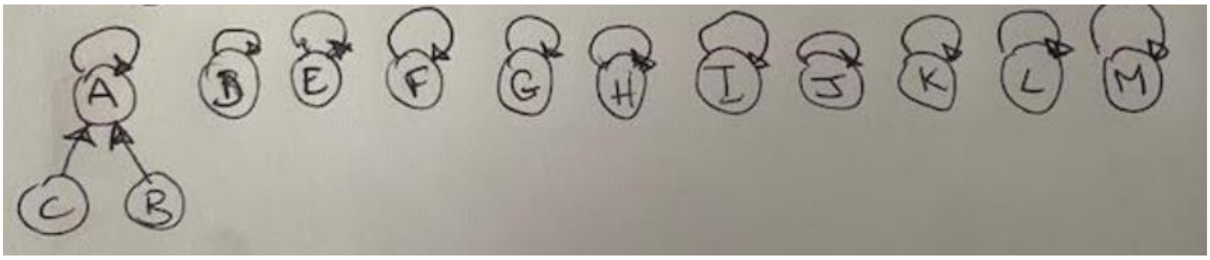


Step 2:

B 1 C

Sets: {ABC} {D} {E} {F} {G} {H} {I} {J} {K} {L} {M}

Union find partition

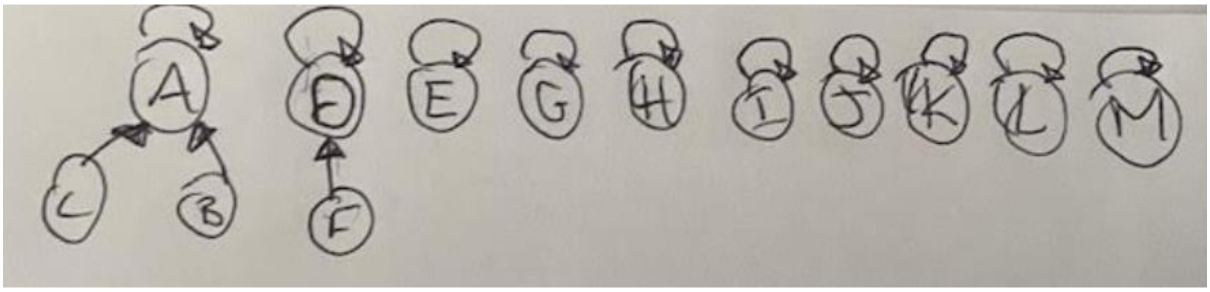


Step 3:

D 1 F

Sets: {ABC} {DF} {E} {G} {H} {I} {J} {K} {L} {M}

Union find partition



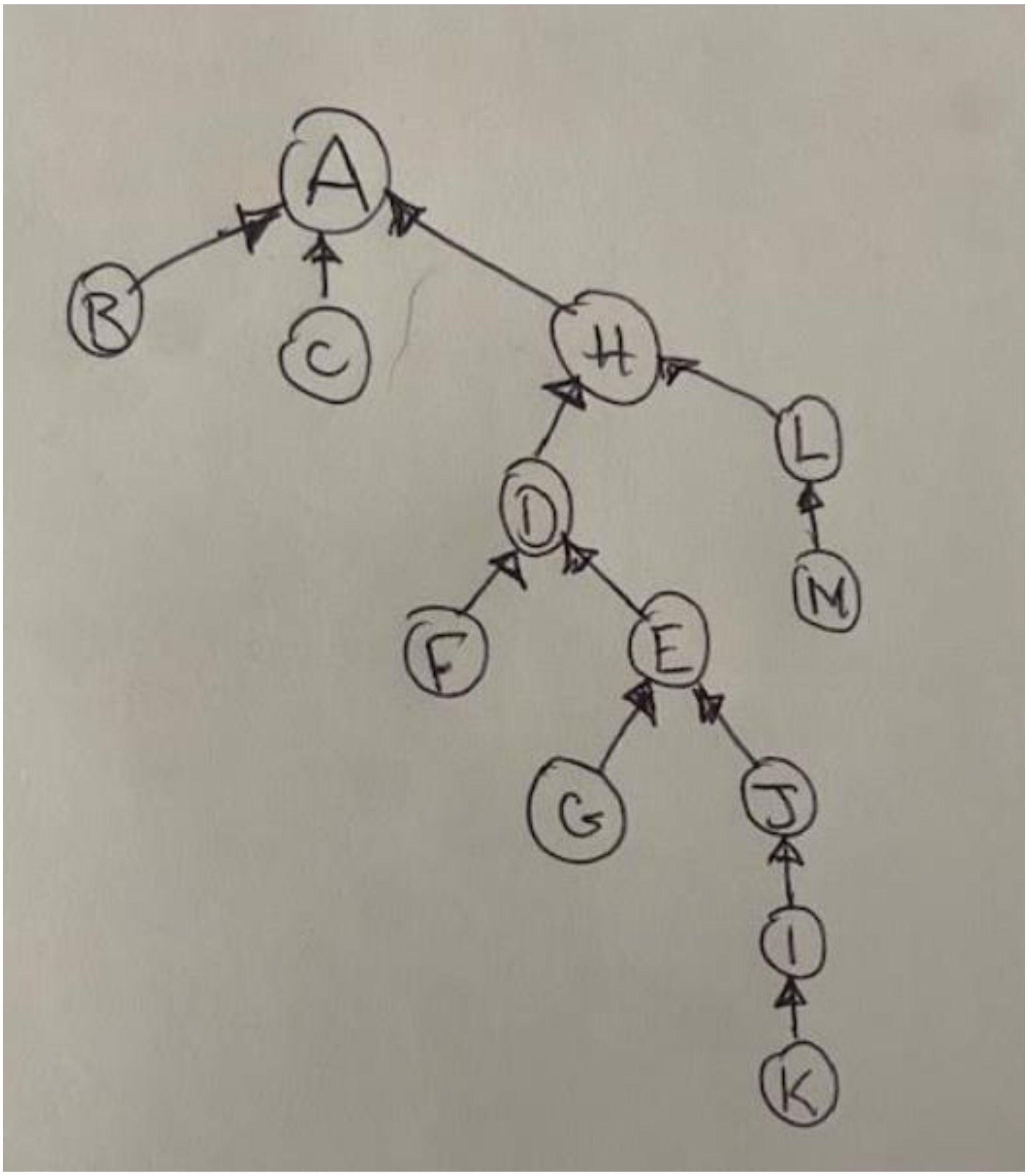
This continues until all letters are in the same set. The final output would look like the following:

MST_Kruskal

A-2-F

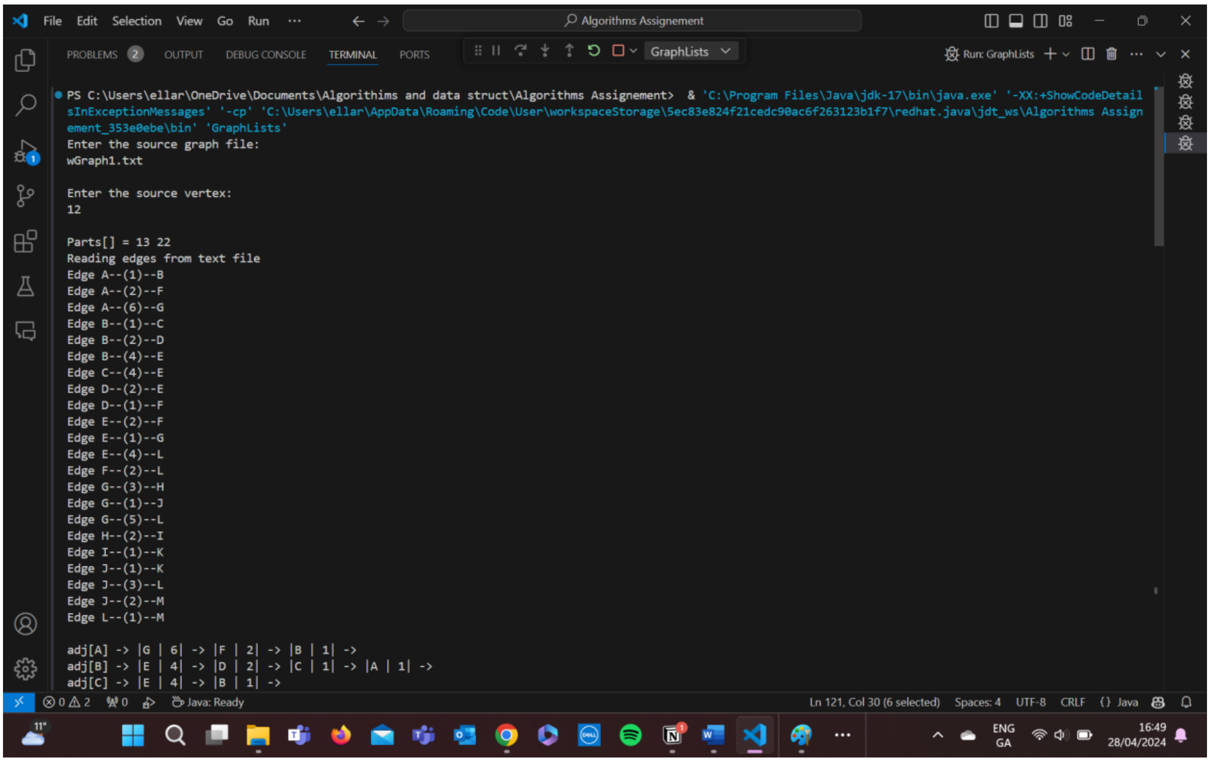
Sets: { A B C D E F G H I J K L M }

Union find partition



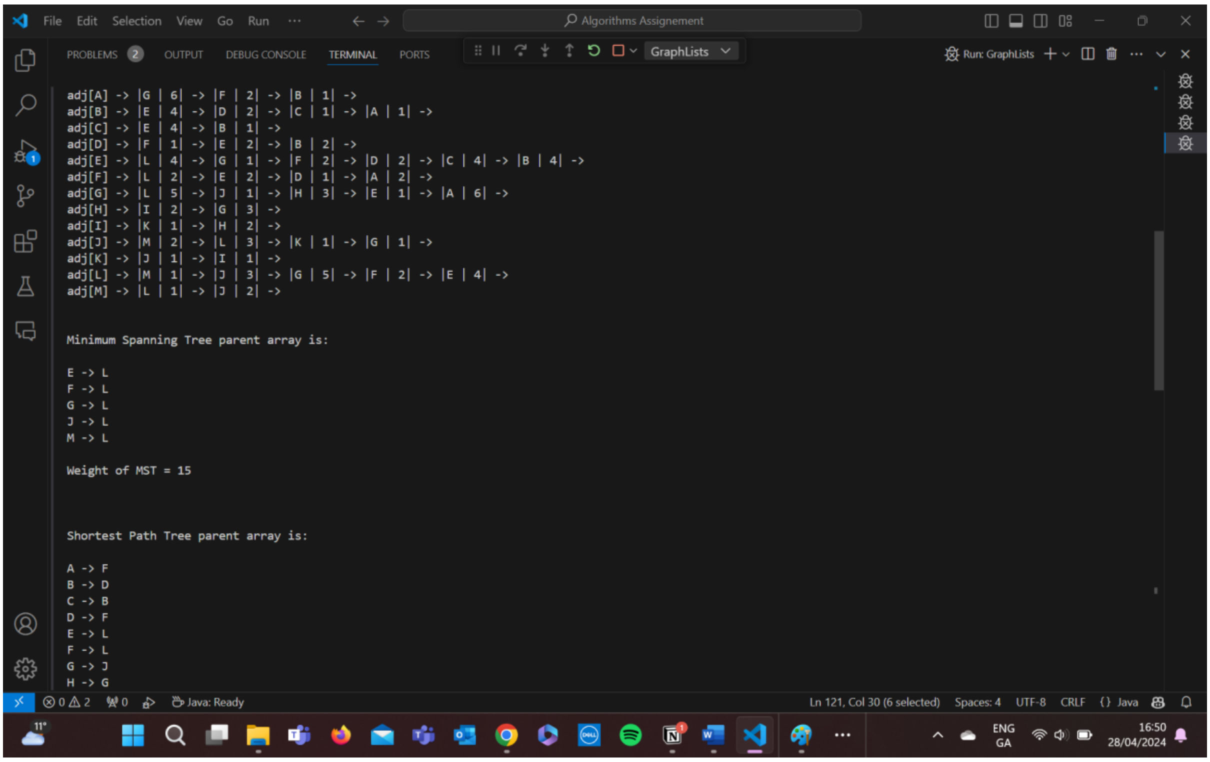
Screenshots of code output

GraphLists.java



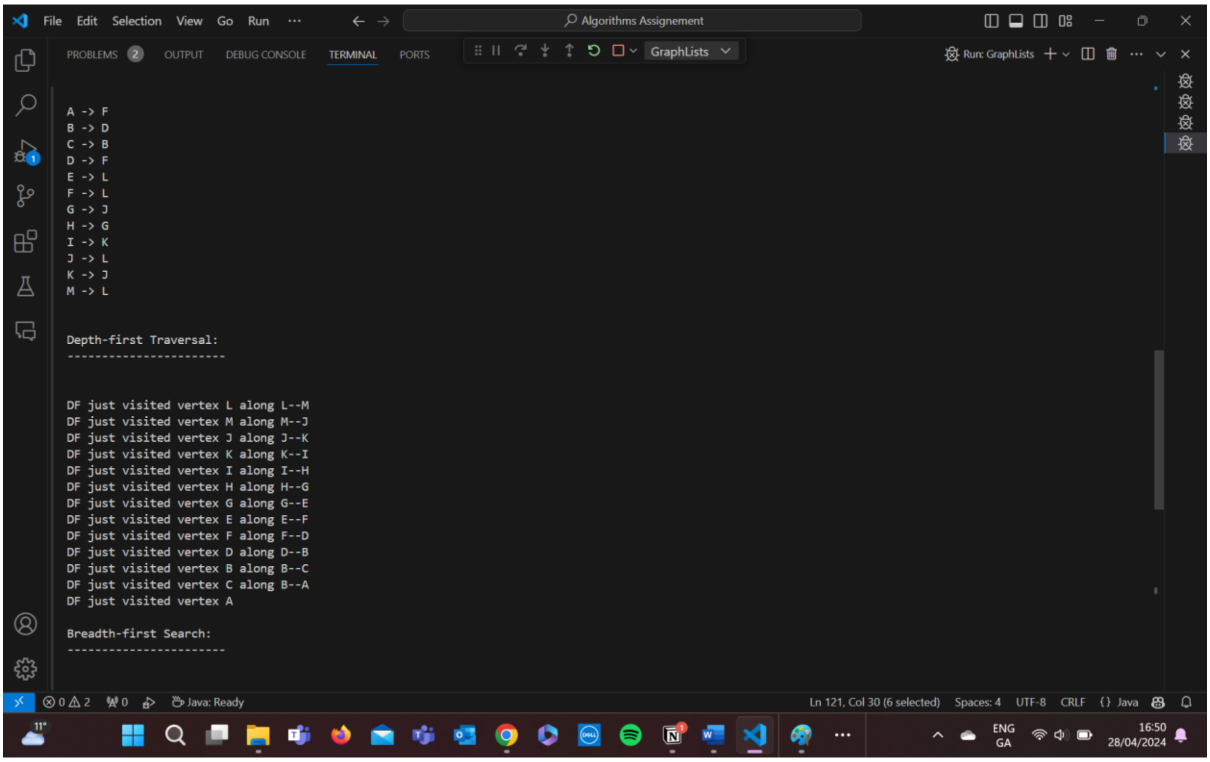
File Edit Selection View Go Run ···
Algorithms Assignement
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS
GraphLists
Run: GraphLists
Ln 121, Col 30 (6 selected) Spaces: 4 UTF-8 CRLF {} Java
Java: Ready
ENG GA 16:49 28/04/2024

```
PS C:\Users\ellar\OneDrive\Documents\Algorithims and data struct\Algorithms Assignement>  & 'C:\Program Files\Java\jdk-17\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\ellar\AppData\Roaming\Code\User\workspaceStorage\5ec83e824f21cedc90ac6f263123b1f7\redhat.java\jdt_ws\Algorithms Assignement_353e0ebe\bin' 'GraphLists'
Enter the source graph file:
wGraph1.txt

Enter the source 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

adj[A] -> |G | 6| -> |F | 2| -> |B | 1| ->
adj[B] -> |E | 4| -> |D | 2| -> |C | 1| -> |A | 1| ->
adj[C] -> |E | 4| -> |B | 1| ->
```



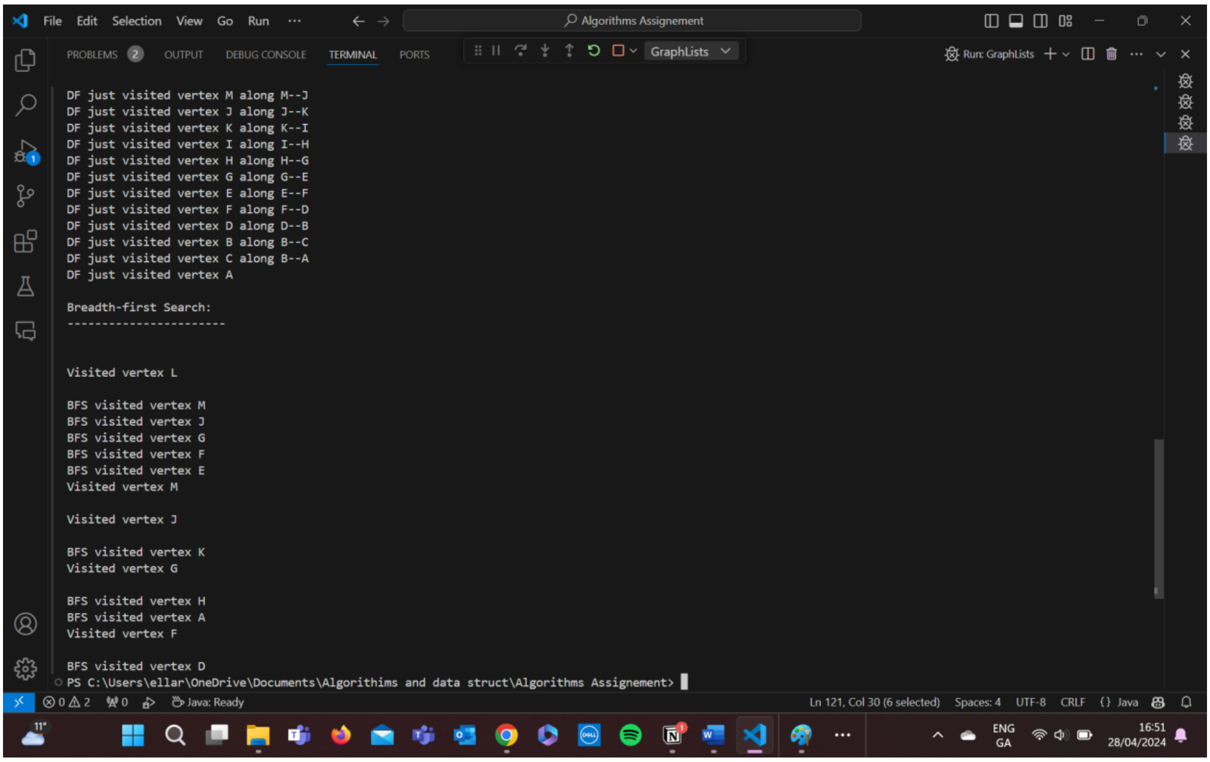
File Edit Selection View Go Run ···
Algorithms Assignement
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS
GraphLists
Run: GraphLists
Ln 121, Col 30 (6 selected) Spaces: 4 UTF-8 CRLF {} Java
Java: Ready
ENG GA 16:50 28/04/2024

```
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| ->


Minimum Spanning Tree parent array is:

E -> L
F -> L
G -> L
J -> L
M -> L

Weight of MST = 15



Shortest Path Tree parent array is:

A -> F
B -> D
C -> B
D -> F
E -> L
F -> L
G -> J
H -> G
```



File Edit Selection View Go Run ...
Algorithms Assignement
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS
GraphLists
Run: GraphLists
Ln 121, Col 30 (6 selected) Spaces: 4 UTF-8 CRLF {} Java
Java: Ready
16:50
28/04/2024

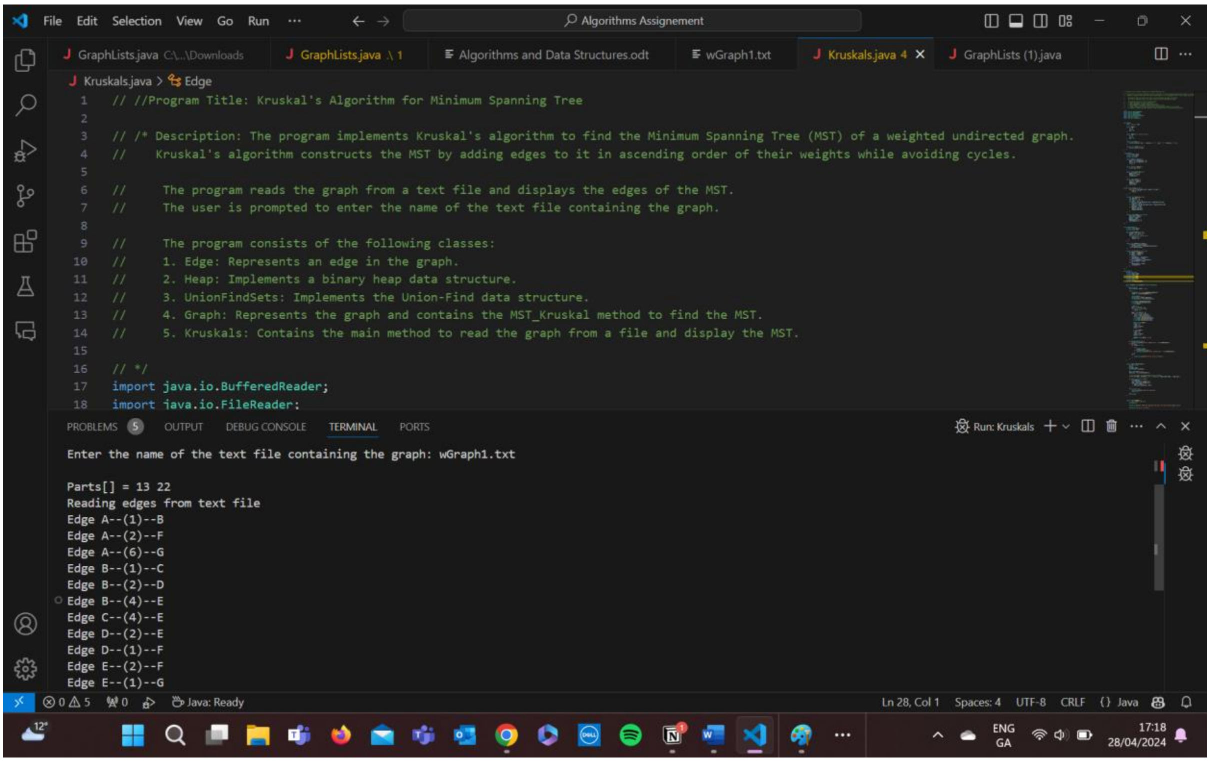
```
A -> F
B -> D
C -> B
D -> F
E -> L
F -> L
G -> J
H -> G
I -> K
J -> L
K -> J
M -> L


Depth-first Traversal:
----------------------


DF just visited vertex L along L--M
DF just visited vertex M along M--J
DF just visited vertex J along J--K
DF just visited vertex K along K--I
DF just visited vertex I along I--H
DF just visited vertex H along H--G
DF just visited vertex G along G--E
DF just visited vertex E along E--F
DF just visited vertex F along F--D
DF just visited vertex D along D--B
DF just visited vertex B along B--C
DF just visited vertex C along B--A
DF just visited vertex A

Breadth-first Search:
----------------------
```



File Edit Selection View Go Run ...
Algorithms Assignement
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS
GraphLists
Run: GraphLists
Ln 121, Col 30 (6 selected) Spaces: 4 UTF-8 CRLF {} Java
Java: Ready
16:51
28/04/2024

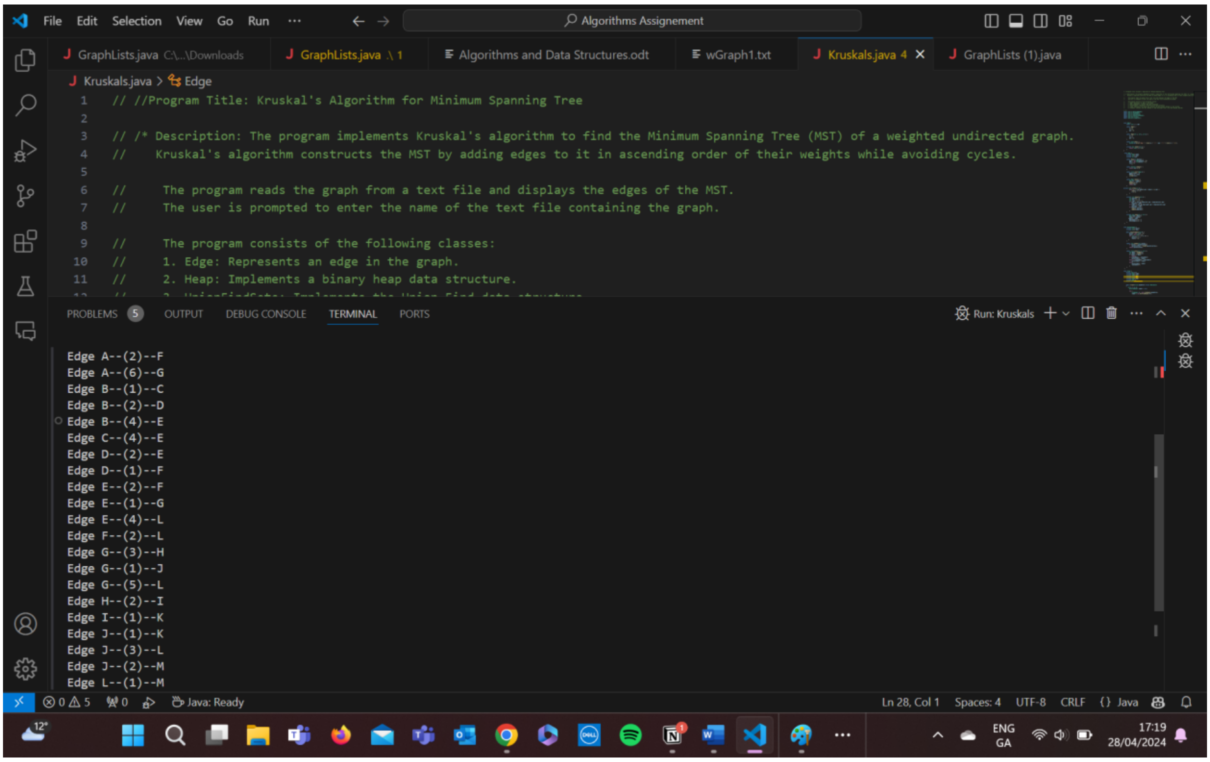
```
DF just visited vertex M along M--J
DF just visited vertex J along J--K
DF just visited vertex K along K--I
DF just visited vertex I along I--H
DF just visited vertex H along H--G
DF just visited vertex G along G--E
DF just visited vertex E along E--F
DF just visited vertex F along F--D
DF just visited vertex D along D--B
DF just visited vertex B along B--C
DF just visited vertex C along B--A
DF just visited vertex A

Breadth-first Search:
----------------------


Visited vertex L

BFS visited vertex M
BFS visited vertex J
BFS visited vertex G
BFS visited vertex F
BFS visited vertex E
Visited vertex M

Visited vertex J

BFS visited vertex K
Visited vertex G

BFS visited vertex H
BFS visited vertex A
Visited vertex F

BFS visited vertex D
PS C:\Users\ellar\OneDrive\Documents\Algorithims and data struct\Algorithms Assignement>
```

Kruskals.java



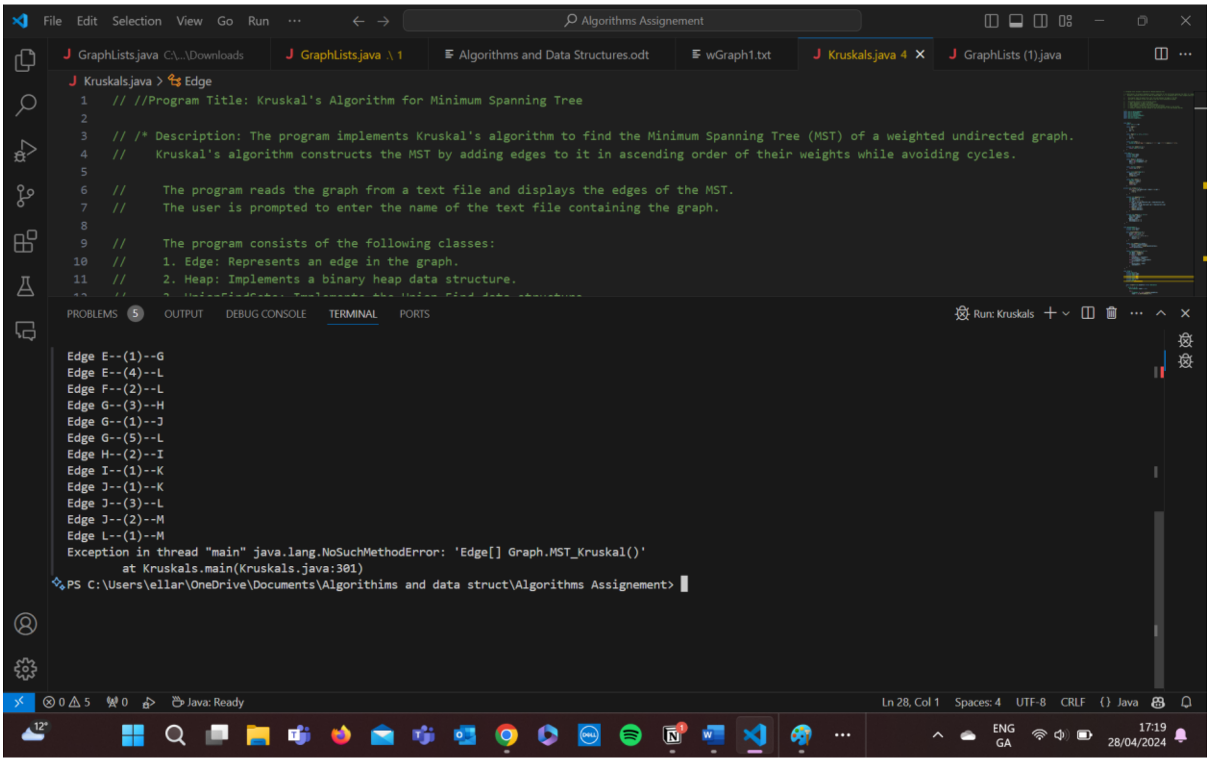
```java
// //Program Title: Kruskal's Algorithm for Minimum Spanning Tree

// /* Description: The program implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a weighted undirected graph.
//    Kruskal's algorithm constructs the MST by adding edges to it in ascending order of their weights while avoiding cycles.

//     The program reads the graph from a text file and displays the edges of the MST.
//     The user is prompted to enter the name of the text file containing the graph.

//     The program consists of the following classes:
//     1. Edge: Represents an edge in the graph.
//     2. Heap: Implements a binary heap data structure.
//     3. UnionFindSets: Implements the Union-Find data structure.
//     4. Graph: Represents the graph and contains the MST_Kruskal method to find the MST.
//     5. Kruskals: Contains the main method to read the graph from a file and display the MST.

// */
import java.io.BufferedReader;
import java.io.FileReader;
```

```
Enter the name of the text file containing the graph: 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
```



```java
// //Program Title: Kruskal's Algorithm for Minimum Spanning Tree

// /* Description: The program implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a weighted undirected graph.
//    Kruskal's algorithm constructs the MST by adding edges to it in ascending order of their weights while avoiding cycles.

//     The program reads the graph from a text file and displays the edges of the MST.
//     The user is prompted to enter the name of the text file containing the graph.

//     The program consists of the following classes:
//     1. Edge: Represents an edge in the graph.
//     2. Heap: Implements a binary heap data structure.
```

```
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
```



```
// //Program Title: Kruskal's Algorithm for Minimum Spanning Tree

// /* Description: The program implements Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a weighted undirected graph.
//    Kruskal's algorithm constructs the MST by adding edges to it in ascending order of their weights while avoiding cycles.

//     The program reads the graph from a text file and displays the edges of the MST.
//     The user is prompted to enter the name of the text file containing the graph.

//     The program consists of the following classes:
//     1. Edge: Represents an edge in the graph.
//     2. Heap: Implements a binary heap data structure.
```

```
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
Exception in thread "main" java.lang.NoSuchMethodError: 'Edge[] Graph.MST_Kruskal()'
        at Kruskals.main(Kruskals.java:301)
PS C:\Users\ellar\OneDrive\Documents\Algorithims and data struct\Algorithms Assignement>
```

Reflection

Reflecting on this assignment, I found the exploration of various graph algorithms like Prims, Dijkstra’s and Kruskal’s to be enlightening and valuable. Implementing these algorithms helped deepen my understanding of graph theory and refine my problem-solving abilities.

Regarding the encountered error with the Kruskal algorithm implementation, the “NoSuchMethodError” is usually output when a method expected is missing however, the methods were there and should have ran as expected. This proved to be difficult and I was however unable to overcome this error in time. Encountering challenges such as this, provided valuable learning opportunities for myself. It reinforced the importance of thorough testing, code review and understanding dependencies in software development.

One of the highlights of this assignment was the hands on construction of the MST and SPT using Prim’s and Dijkstra’s algorithms. This allowed me to enhance my ability to think algorithmically and visualize complex processes.

Overall this assignment has deepened my appreciation for graph algorithms and their significance in computer science and fostered a greater curiosity for exploring advanced topics in algorithms and data structures.