# Algorithms and Data Structures

4/4/2024

## Graph Traversal and Minimum Spanning Tree Algorithms

Module Code: CMPU2001

Lecturer: Mr Richard Lawlor

STUDENT NAME: Aaron Baggot
STUDENT NUMBER: C22716399
TU856

# Table of Contents

**INTRODUCTION** ..... **2**

**ADJACENCY LISTS DIAGRAM:** ..... **4**

**PRIMS ALGORITHM** ..... **8**

*Prims Weighted undirected graph* ..... *8*
*Prims Algorithm* ..... *9*
*Initialisation* ..... *9*
*Priority Queue Setup* ..... *10*
*Update Distances* ..... *10*
*Repeat* ..... *10*
COMPLETE MINIMUM SPANNING TREE ..... 28
BREADTH FIRST SEARCH ..... 29
*Algorithm* ..... *29*
*Implementation* ..... *30*
*Node Handling* ..... *30*
DEPTH FIRST SEARCH ..... 35
*Algorithm* ..... *35*
*Implementation* ..... *36*
*Efficiency* ..... *36*
*Advantages* ..... *36*

**DIJKSTRA ALGORITHM SHORTEST PATH TREE** ..... **38**

MENU ..... 46

**KRUSKAL'S ALGORITHM** ..... **47**

*Program Code Explained* ..... *50*
*Time Complexity* ..... *50*
*Output* ..... *51*
UNION FIND PARTITION AND SET REPRESENTATIONS ..... 52
*Disjoint Sets* ..... *52*
*Kruskal's sets* ..... *56*
TESTING ..... 61

**LEARNING GAINED** ..... **63**

**REFERENCES & BIBLIOGRAPHY** ..... **64**

# Introduction

This Java program is designed to implement various graph algorithms, including Dijkstra's shortest path tree (SPT) algorithm. Graph algorithms are fundamental in computer science and find extensive applications in various domains, including network routing, social network analysis, and optimization problems.

Graphs are mathematical structures composed of nodes (vertices) and edges connecting these nodes. They are used to model relationships between entities in a wide range of scenarios. This program focuses on weighted, connected graphs, where each edge has a numerical weight representing the cost or distance between nodes.

Dijkstra's algorithm is a well-known graph algorithm used to find the shortest paths from a source node to all other nodes in a graph. It operates efficiently on graphs with non-negative edge weights and produces a shortest path tree (SPT) rooted at the source node.

The objectives of this program is to implement Dijkstra's algorithm to compute the shortest path tree (SPT) for a given graph. Provide a user-friendly interface for inputting graph data and selecting the source node. Display the resulting shortest path tree (SPT) to the console, along with any intermediate steps or workings for clarity and understanding.
This program will be structured to allow for easy extension to other graph algorithms, such as Prim's algorithm for minimum spanning trees (MST) and various graph traversal techniques like depth-first traversal (DFS) and breadth-first traversal (BFS).

The program will adhere to best practices in Java programming, including proper code organization, modular design, and extensive commenting for clarity and maintainability. Additionally, it will incorporate user input validation and error handling to ensure robustness and reliability.

Overall, this Java program aims to provide a comprehensive implementation of Dijkstra's algorithm for finding shortest paths in weighted graphs, along with supporting functionalities to facilitate experimentation and understanding of graph algorithms.

The assignment involves implementing and analysing various graph traversal algorithms like DFS, BFS, Prim's algorithm for MST, and Dijkstra's algorithm for SPT. It also requires representing the graph using an adjacency list and understanding the steps involved in constructing MST and SPT using Prim's and Dijkstra's algorithms respectively.

# Adjacency Lists Diagram:

The adjacency list representation shows how each vertex in the graph is connected to its neighbours along with the weights of the edges. This representation is essential for implementing graph algorithms efficiently. Each vertex is listed with its adjacent vertices and corresponding edge weights. Image 1 illustrates the expected output collaborated using whiteboard while image 2 reciprocates the output after running the code.

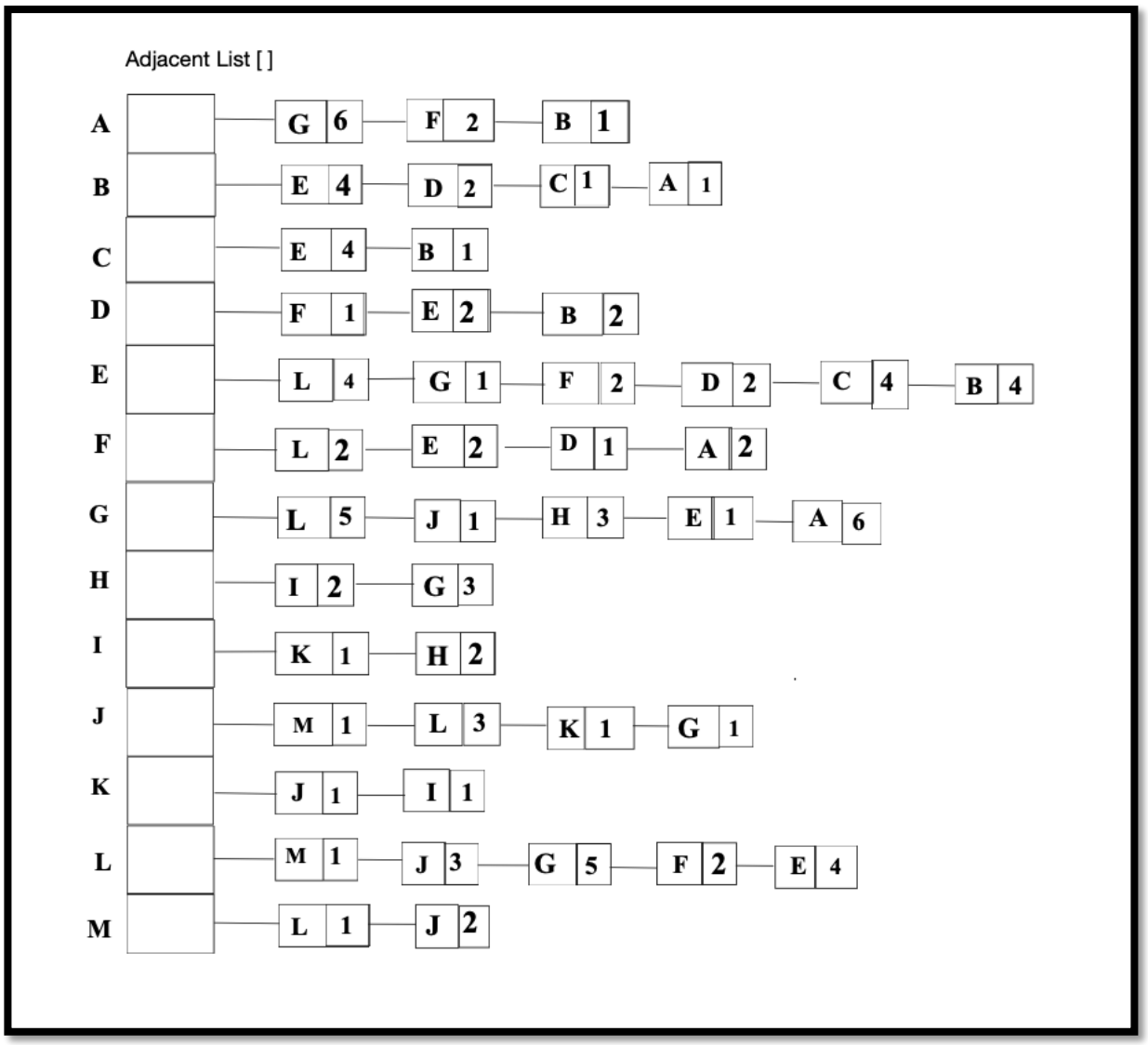



Image 1

```
Adjacency List Diagram:

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| ->
```

Image 2

A minimum spanning tree for a connected graph used in this assignment are illustrated in Image 3 and the output to the terminal in Image 4 included are the weights on each edge. Parts[ ] shows the number of Vertices being 13 and Edges 22. For example A is represented by 1, B by 2 and so forth. A is connected to B with a weight of 2.

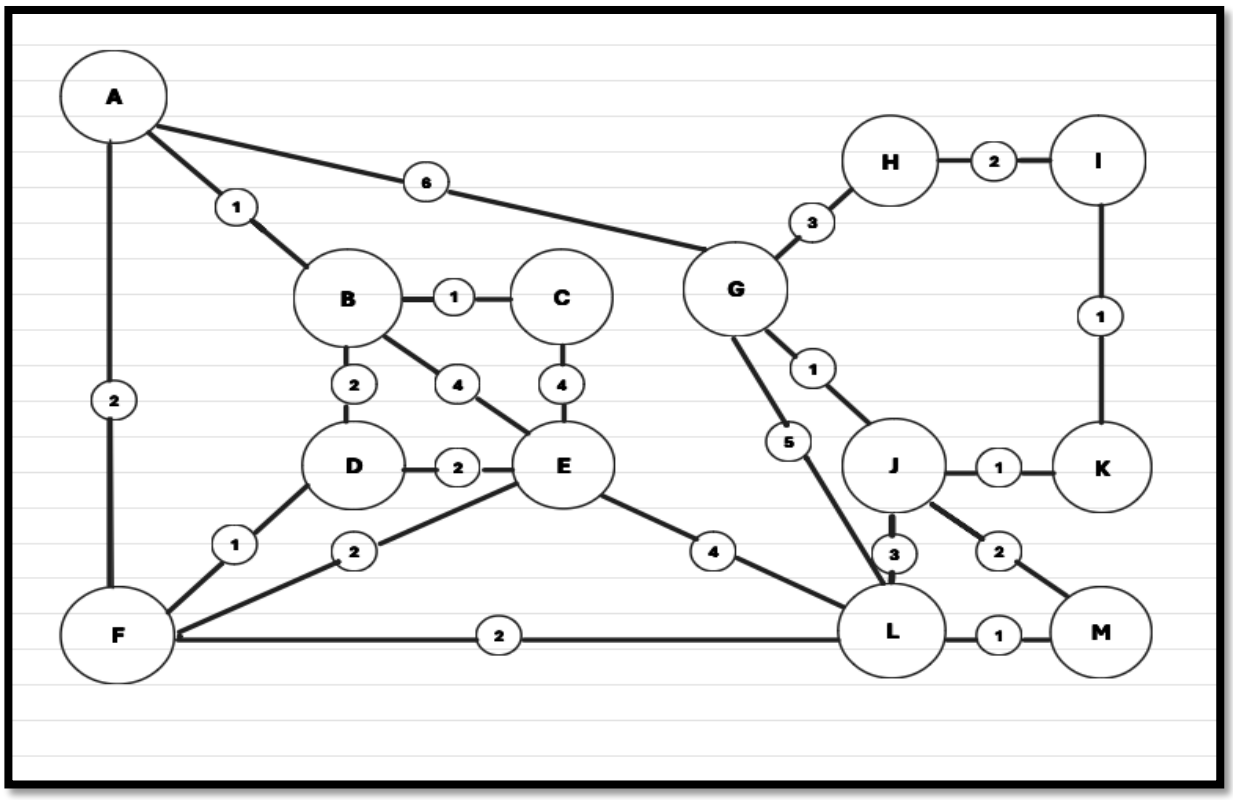



Image 3

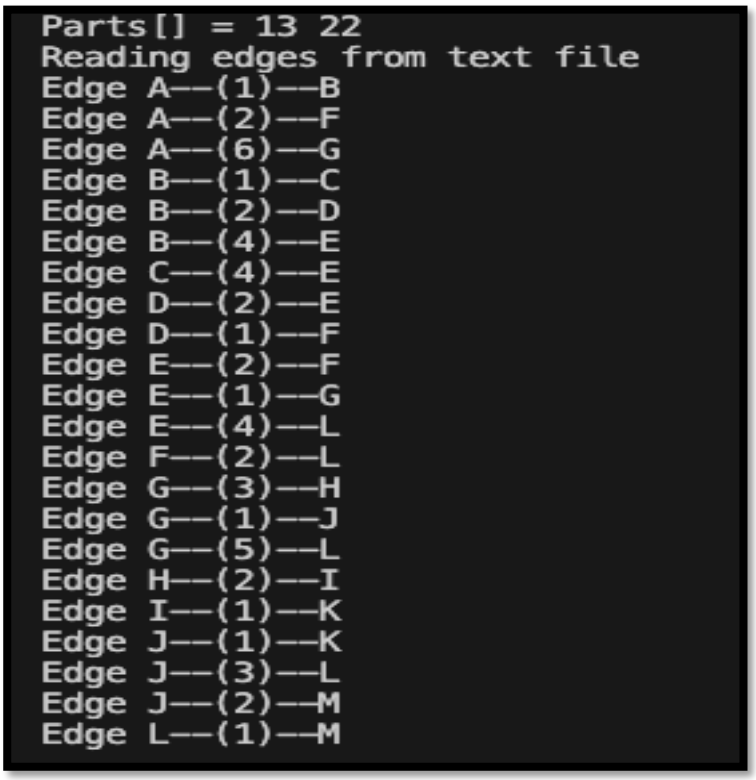

Parts[] = 13 22
Reading edges from text file
Edge A--(1)--B
Edge A--(2)--F
Edge A--(6)--G
Edge B--(1)--C
Edge B--(2)--D
Edge B--(4)--E
Edge C--(4)--E
Edge D--(2)--E
Edge D--(1)--F
Edge E--(2)--F
Edge E--(1)--G
Edge E--(4)--L
Edge F--(2)--L
Edge G--(3)--H
Edge G--(1)--J
Edge G--(5)--L
Edge H--(2)--I
Edge I--(1)--K
Edge J--(1)--K
Edge J--(3)--L
Edge J--(2)--M
Edge L--(1)--M

Image 4

In the provided code Image 4a, u and v are variables used to represent vertices (nodes) in a graph. Within the for loop that reads the edges of the graph from a text file, u and v are assigned the values of the vertices that form each edge, u represents one endpoint and v represents the other endpoint of the edge. This code is adding the edge to the adjacency list representation of the graph, where adj is an array of linked lists, with each element corresponding to a vertex in the graph. The code is essentially adding the edge between vertices u and v to both of their adjacency lists.

```
// read the edges
System.out.println(x:"Reading edges from text file");
for(e = 1; e <= E; ++e)
{ //start for
    line = reader.readLine();
    parts = line.split(splits);
    u = Integer.parseInt(parts[0]); // Extract the first vertex of the edge
    v = Integer.parseInt(parts[1]); // Extract the second vertex of the edge
    wgt = Integer.parseInt(parts[2]);

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

    // Add the edge to the adjacency linked list representation of the graph
    t = adj[u]; // Get the current head of the adjacency list for vertex u
    adj[u] = new Node(); // Create a new node for the edge
    adj[u].vert = v; // Set the vertex of the new node
    adj[u].wgt = wgt; // Set the weight of the new node
    adj[u].next = t; // Set the next pointer of the new node to the previous head

    t = adj[v]; // Get the current head of the adjacency list for vertex v
    adj[v] = new Node(); // Create a new node for the edge
    adj[v].vert = u; // Set the vertex of the new node
    adj[v].wgt = wgt; // Set the weight of the new node
    adj[v].next = t; // Set the next pointer of the new node to the previous head

} //end for
```

Image 4a

# Prims Algorithm

Prim's algorithm is a fundamental method for finding minimum spanning trees, like Kruskal's approach. It bears resemblance to Dijkstra's algorithm for shortest paths. The algorithm begins by creating a unified tree set A, starting from a chosen vertex and expanding until all vertices are included. Each step adds a lightweight edge to connect A to a new vertex, ensuring only safe edges are chosen. This process results in a minimum spanning tree once all vertices are encompassed. Efficiency in Prim's algorithm depends on swiftly selecting the next edge to add, which is outlined in the pseudocode. The algorithm utilizes a priority queue to manage vertices not yet part of the tree, ordered by their minimal weight edge connection. This method effectively manages the progression while adhering to the principles of the "*GENERIC-MST"* framework. Additionally, the assignment involves using a heap and linked list to create a graph from a text file. The program then traverses the heap to find the minimum spanning tree and calculates its weight. Time complexity of O(E + V log V).

## Prims Weighted undirected graph

- Prim's algorithm starts from vertex L, denoted as s, to initiate the construction of the Minimum Spanning Tree (MST).
- The algorithm iterates until all vertices are incorporated into the MST, selecting the smallest edge that connects a vertex from the MST to a vertex outside of it.

## Prims Algorithm

- MST_Prim(G,w,s) initializes arrays for storing minimum distances (d[]) and parent vertices (p[]) for each vertex in the graph.
- It iterates through the vertices, updating the minimum distance to reach the MST for each vertex and selecting the next vertex to add to the MST based on the smallest edge weight connecting it to the existing MST.

*MST_Prim(G,w,s) // N = |V|*
*1 int d[N], p[N]*
*2 For v = 0 -> N-1*
*3 d[v]=inf //min weight of edge connecting v to MST*
*4 p[v]=-1 //(p[v],v) in MST and w(p[v],v) =d[v]*
*5 d[s]=0*
*6 Q = PriorityQueue(G.V,d)*
*7 While notEmpty(Q)*
*8 u = removeMin(Q,d) //u is picked*
*9 for each v adjacent to u*
*10 if v in Q and w(u,v)<d[v]*
*11 p[v]=u*
*12 d[v] = w(u,v)*
*13 decreasedKeyFix(Q,v,d)*
(Cormen et al., 2009)

## Initialisation

Choose a starting vertex, often denoted as s, to begin the MST. Initialize two arrays: d[] to store the minimum weight of the edge connecting each vertex to the MST and p[] to store the parent of each vertex in the MST. Initially, set all elements of d[] to infinity (except for d[s] = 0) and set all elements of p[] to -1.

### Priority Queue Setup

Create a priority queue Q (implemented as a heap) to store the vertices not yet included in the MST, sorted by their corresponding d[] values.

Main Loop: While Q is not empty, repeatedly select the vertex u with the smallest d[u] from the priority queue.

### Update Distances

For each vertex v adjacent to u, if v is still in Q and the weight of the edge (u, v) is less than d[v], update d[v] to the weight of (u, v) and set p[v] to u. Then, adjust the priority queue accordingly to maintain the minimum d[] values.

### Repeat

Repeat steps 3 and 4 until all vertices are included in the MST.

By following these steps, Prim's algorithm constructs the minimum spanning tree of the given weighted undirected graph efficiently. It ensures that at each step, the edge added to the MST has the minimum weight among all edges connecting vertices in the MST to vertices outside of the MST.

A step-by-step construction of the MST using Prim's algorithm for the provided sample graph. Tracking the contents of the priority queue (heap), the parent[] array, and the dist[] array at each step to illustrate the progression of the algorithm. Image 5 illustrates the output on the terminal with edges and weight and Minimum Spanning Tree parent array (Image 5a) .

```
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
```

Image 5

```
Minimum Spanning tree parent array is:
A -> B

B -> A

C -> B

D -> F

E -> G

F -> D

G -> E

H -> I

I -> K

J -> G

K -> J

L -> @

M -> L
```

Image 5a

The diagram in image 6 represents the vertex, parent and distance implementing Prims algorithm and image 7 displays the commented code for Prims minimum spanning tree.

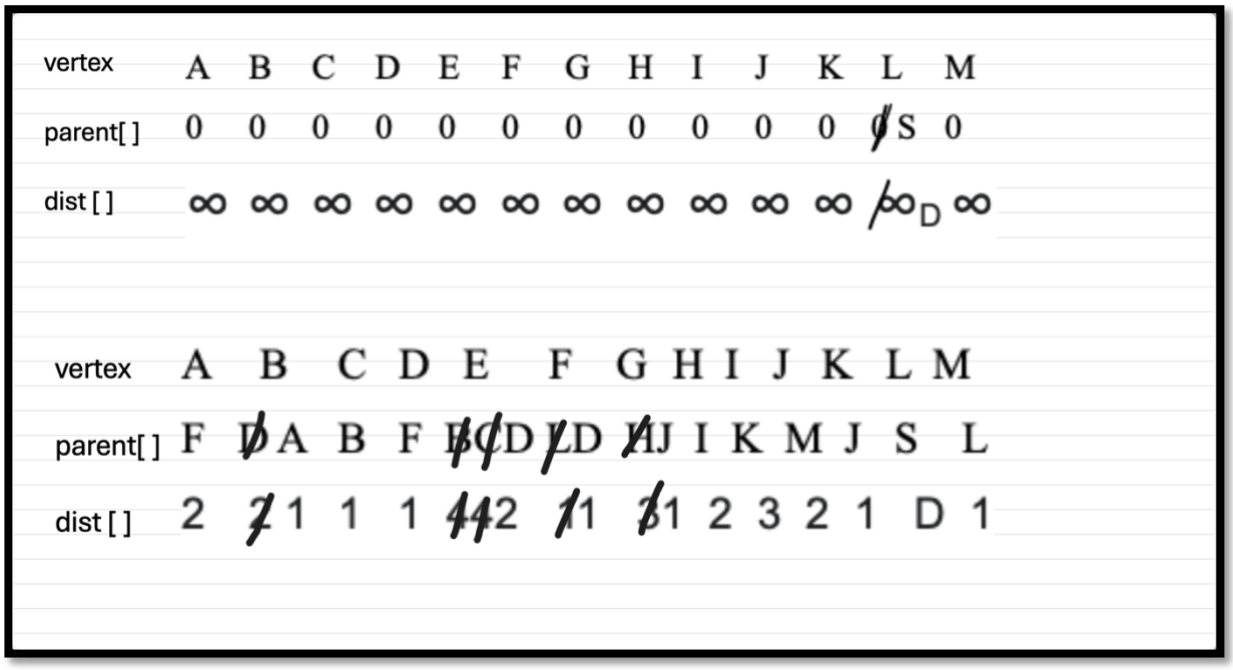
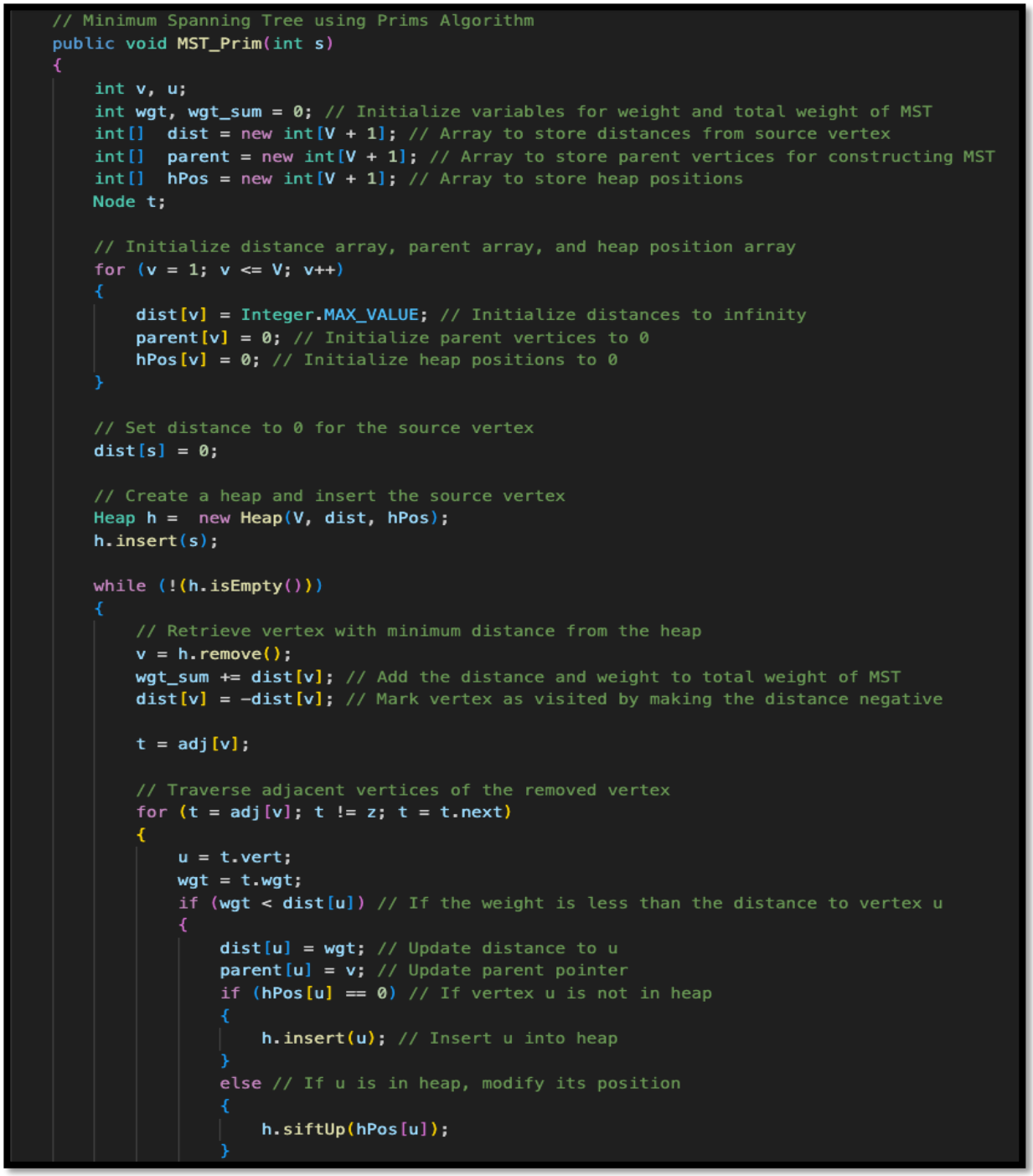



Image 6



```java
// Minimum Spanning Tree using Prims Algorithm
public void MST_Prim(int s)
{
    int v, u;
    int wgt, wgt_sum = 0; // Initialize variables for weight and total weight of MST
    int[]  dist = new int[V + 1]; // Array to store distances from source vertex
    int[]  parent = new int[V + 1]; // Array to store parent vertices for constructing MST
    int[]  hPos = new int[V + 1]; // Array to store heap positions
    Node t;

    // Initialize distance array, parent array, and heap position array
    for (v = 1; v <= V; v++)
    {
        dist[v] = Integer.MAX_VALUE; // Initialize distances to infinity
        parent[v] = 0; // Initialize parent vertices to 0
        hPos[v] = 0; // Initialize heap positions to 0
    }

    // Set distance to 0 for the source vertex
    dist[s] = 0;

    // Create a heap and insert the source vertex
    Heap h =  new Heap(V, dist, hPos);
    h.insert(s);

    while (!(h.isEmpty()))
    {
        // Retrieve vertex with minimum distance from the heap
        v = h.remove();
        wgt_sum += dist[v]; // Add the distance and weight to total weight of MST
        dist[v] = -dist[v]; // Mark vertex as visited by making the distance negative

        t = adj[v];

        // Traverse adjacent vertices of the removed vertex
        for (t = adj[v]; t != z; t = t.next)
        {
            u = t.vert;
            wgt = t.wgt;
            if (wgt < dist[u]) // If the weight is less than the distance to vertex u
            {
                dist[u] = wgt; // Update distance to u
                parent[u] = v; // Update parent pointer
                if (hPos[u] == 0) // If vertex u is not in heap
                {
                    h.insert(u); // Insert u into heap
                }
                else // If u is in heap, modify its position
                {
                    h.siftUp(hPos[u]);
                }
```

Image 7

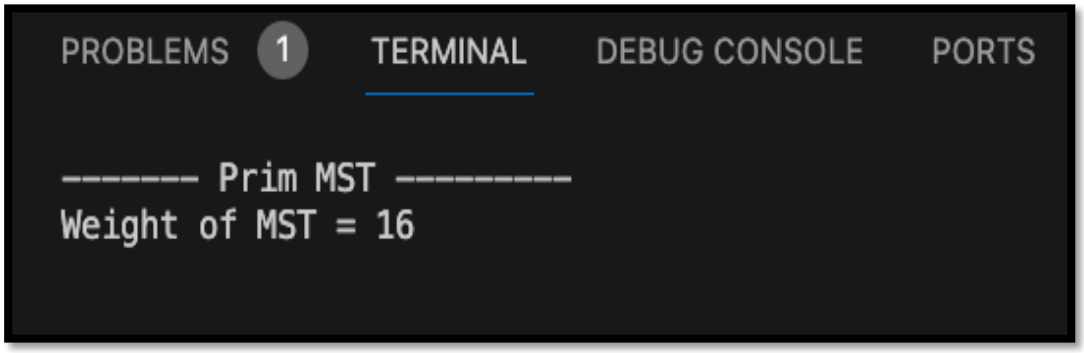



Image 8

The following is a step-by-step construction of the MST using Prim's algorithm for the provided sample graph. Tracking the contents of the priority queue (heap), the parent[] array, and the dist[] array at each step to illustrate the progression of the algorithm.

Red - current MST

Green - potential edges and vertices

Blue – unprocessed edges and vertices.

*Step 1:*

Start at Vertex L as with Prims a starting Vertex is selected.

Vertex L-M = 1

Vertex L-E = 4

Vertex L-J = 3

Vertex L-G = 5

Vertex L-F = 2

The Vertex with the lowest edge is chosen L-M – 1

The Heap is: L-E – 4, **L-M – 1**, L-G – 5, L – F -2, L–J-3

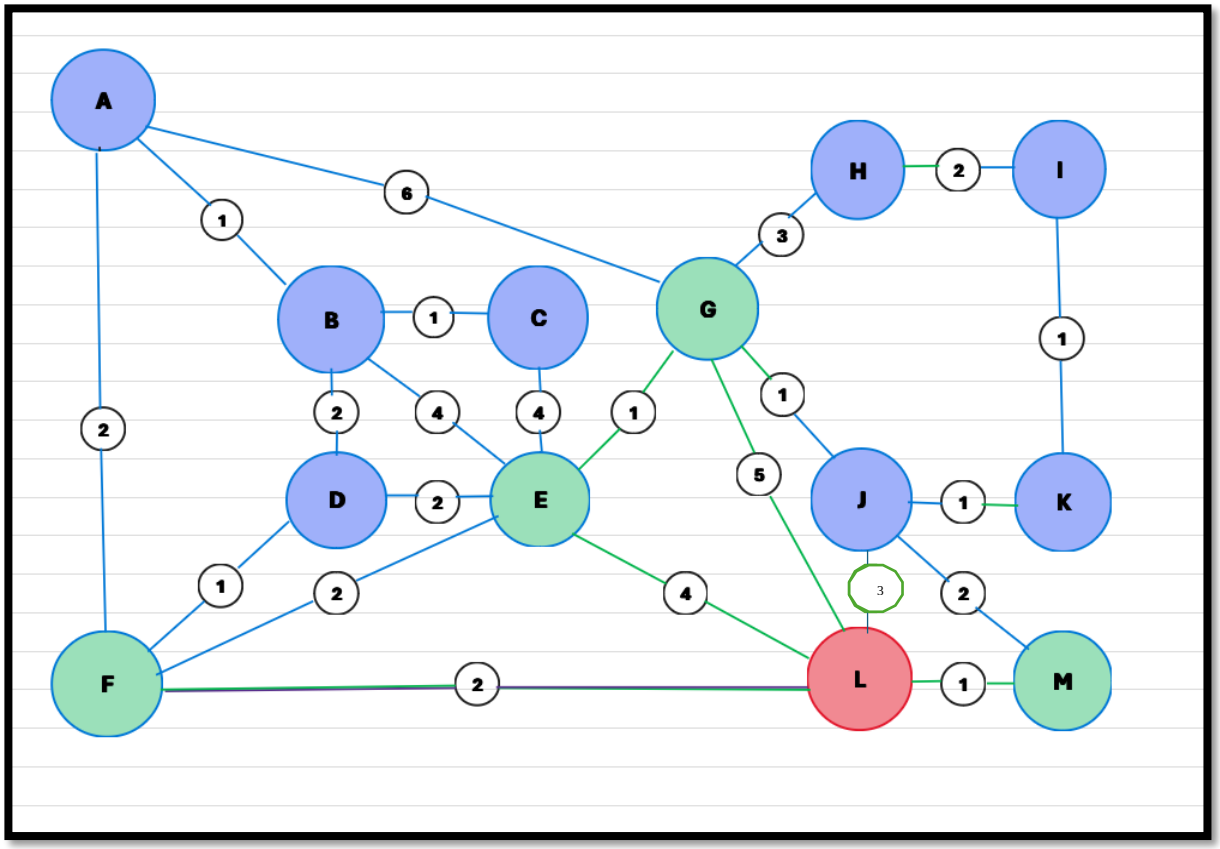



Fig 1.1

*Step 2:*

L and M are now connected.

Vertex J is now added.

Vertex M-J = 2

Vertex L-J = 3

Vertex L-E = 4

Vertex L-G = 5

Vertex L-F = 2

The Vertex with the lowest edge is chosen M-J - 2

The Heap is: L-E – 4, **M-J -2**, L-G – 5, L – F -2, L–J-3

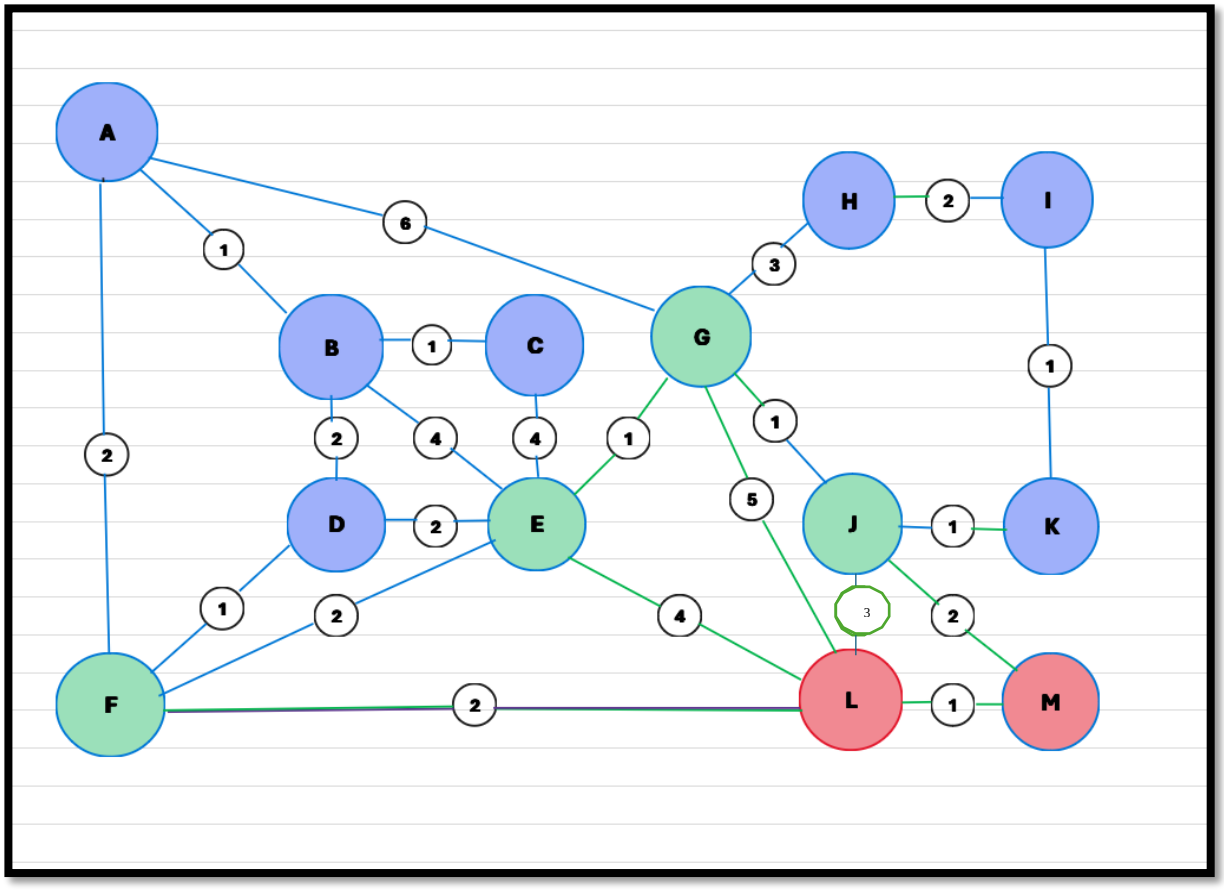



Fig 1.2

*Step 3:*

Vertex L M and J are connected.

Vertex K is now added.

Vertex J-G = 1

Vertex J-K = 1

Vertex L-E = 4

Vertex L-G = 5

Vertex L-F = 2

The Vertex with the lowest edge is chosen J-K - 1

The Heap is: L-E – 4, **J-K -1**, L-G – 5, L – F -2, J – G - 1

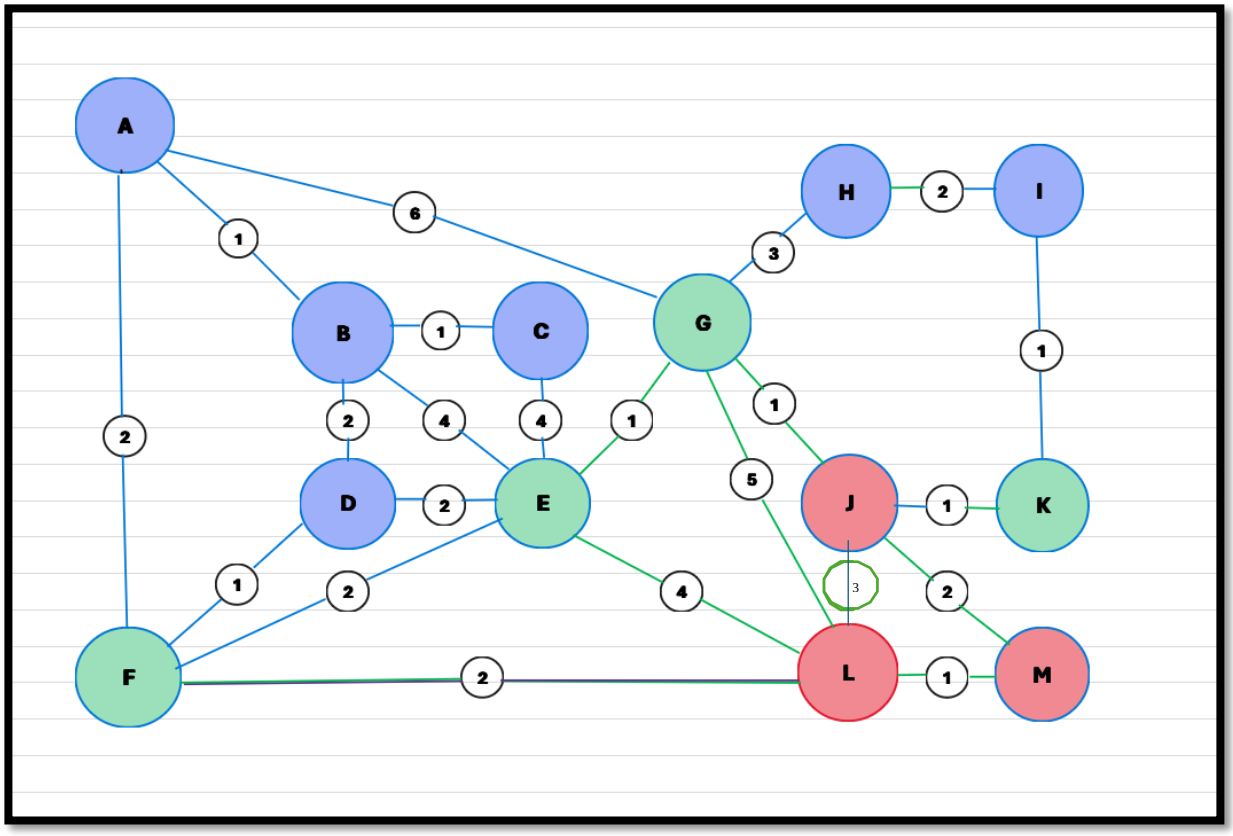



Fig 1.3

*Step 4:*

Vertex L M J and K are connected.

Vertex I is now added.

Vertex J-G = 1

Vertex K-I = 1

Vertex L-E = 4

Vertex L-G = 5

Vertex L-F = 2

The Vertex with the lowest edge is chosen K-I - 1

The Heap is: L-E – 4, **K-I -1**, L-G – 5, L – F -2, J – G -1

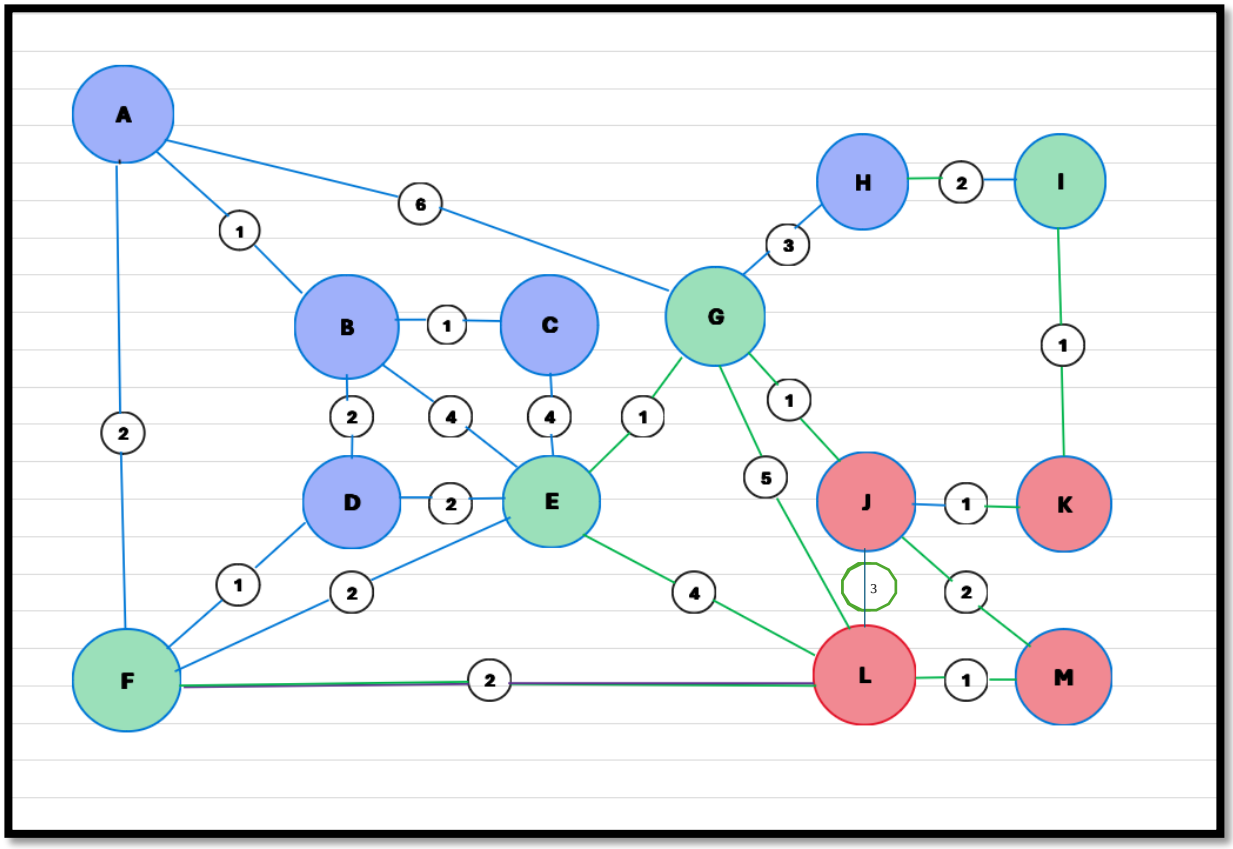



Fig 1.4

*Step 5:*

Vertex L M J K and I are connected.

Vertex G is now added.

Vertex J-G = 1

Vertex I-H = 2

Vertex L-E = 4

Vertex L-G = 5

Vertex L-F = 2

The Vertex with the lowest edge is chosen J-G - 1

The Heap is: L-E – 4, I-H -2, L-G – 5, L – F -2, **J-G - 1**

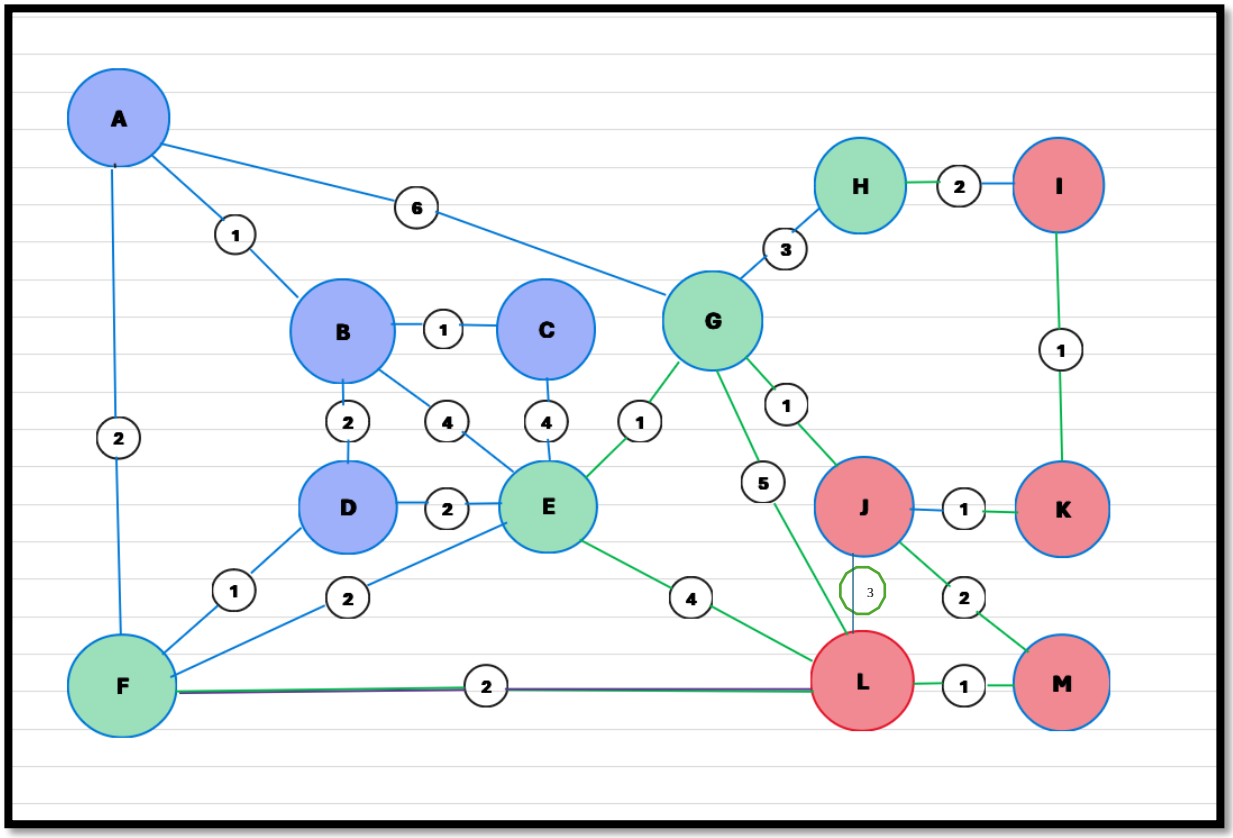



Fig 1.5

*Step 6:*

Vertex L M J K I and G are connected.

Vertex E is now added.

Vertex G-A = 6

Vertex G-E = 1

Vertex I-H = 2

Vertex L-E = 4

Vertex L-F = 2

The Vertex with the lowest edge is chosen **G-E - 1**

The Heap is: L-E – 4, G-A -6, I-H -2, L – F -2, **G-E - 1**

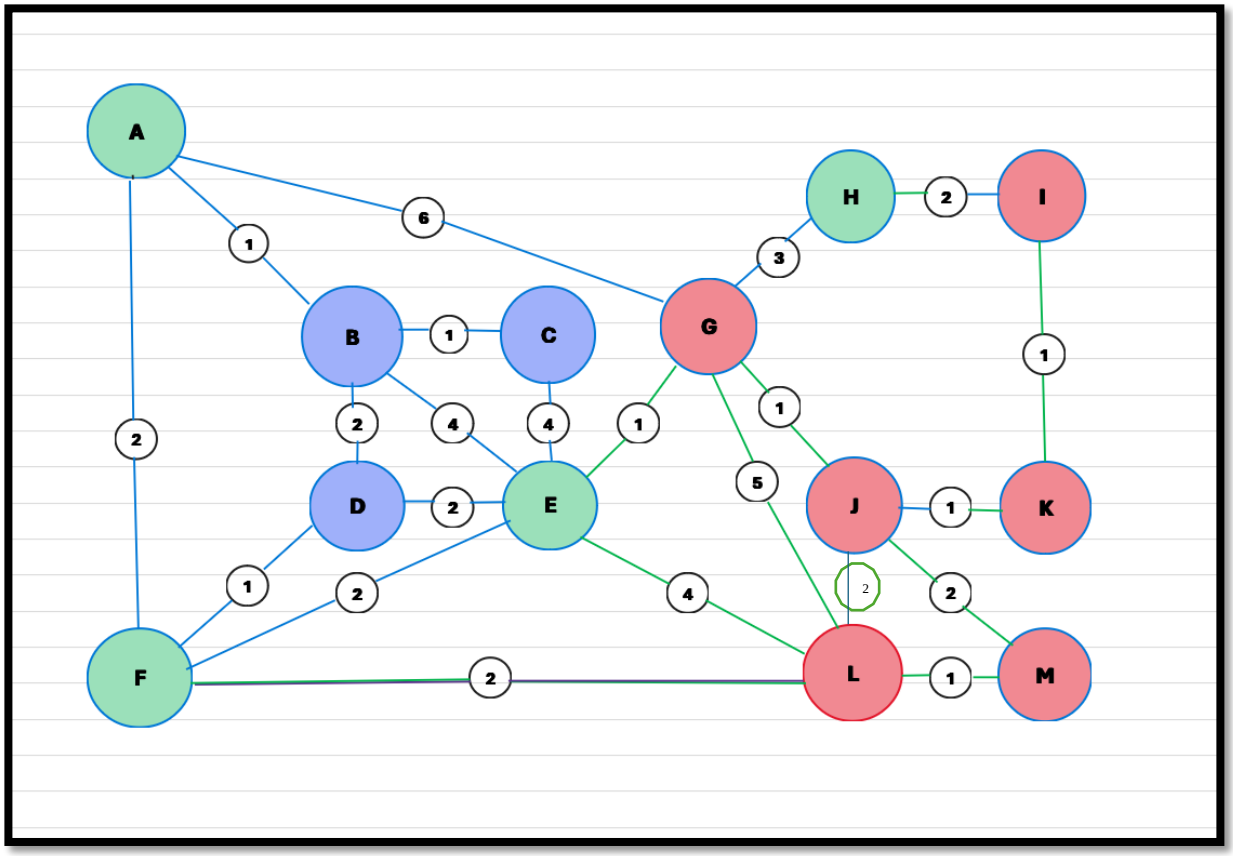



Fig 1.6

*Step 7:*

Vertex L M J K I G and E are connected.

Vertex D is now added.

Vertex G-A = 6

Vertex E-C = 4

Vertex E-D = 2

Vertex E-B= 4

Vertex E-F = 2

Vertex L-F = 2

Vertex I-H = 2

The Vertex with the lowest edge is chosen **E-D-2**

The Heap is: E-C – 4, G-A -6, E-B -4, **E – D -2**, E- F -2, L-F -2, I-H -2

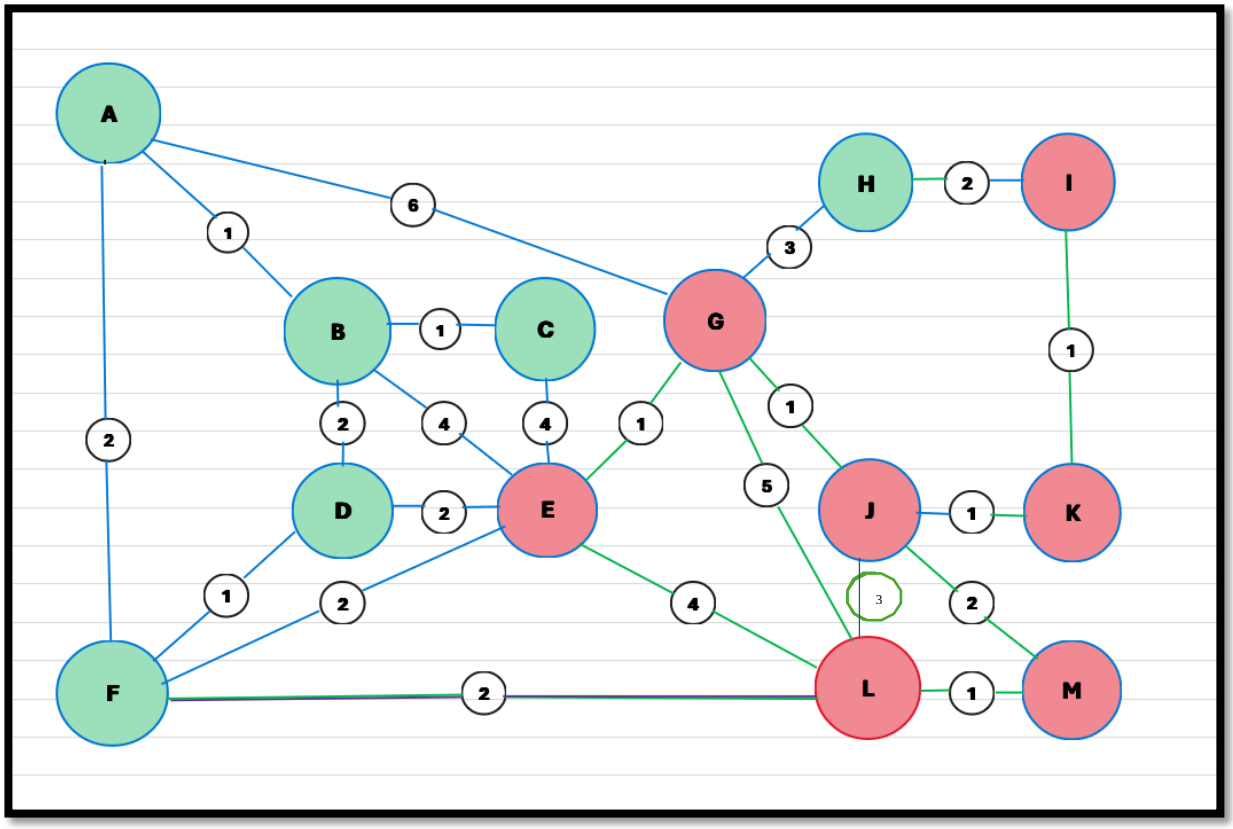



Fig 1.7

*Step 8:*

Vertex L M J K I G E and D are connected.

Vertex F is now added.

Vertex G-A = 6

Vertex E-C = 4

Vertex E-B= 4

Vertex E-F = 2

Vertex L-F = 2

Vertex I-H = 2

Vertex D-F = 1

Vertex D-B = 2

The Vertex with the lowest edge is chosen **D-F - 1**

The Heap is: E-C – 4, G-A -6,  E-B -4, **D-F -1**, E- F -2, L-F -2, I-H -2,D-B-2

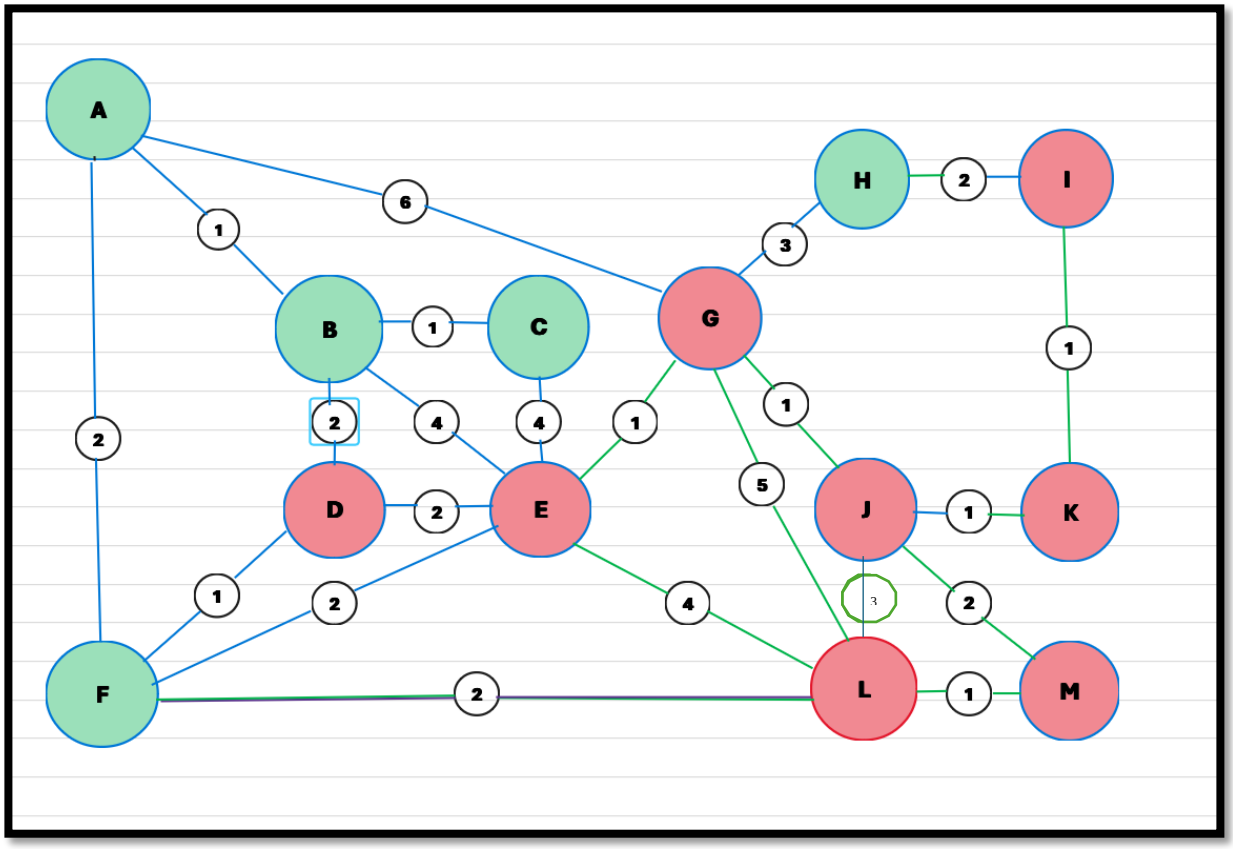



Fig 1.8

*Step 9:*

Vertex L M J K I G E D and F are connected.

Vertex A is now added.

Vertex F-A = 2

Vertex I-H = 2

Vertex G-H = 3

Vertex G-A = 6

Vertex E-C = 4

Vertex E-B = 4

Vertex D-B = 2

The Vertex with the lowest edge is chosen **F-A - 2**

The Heap is: **F-A – 2**, I-H -2, G-H-3, G-A-6, E-C-4,E-B-4, D-B-2

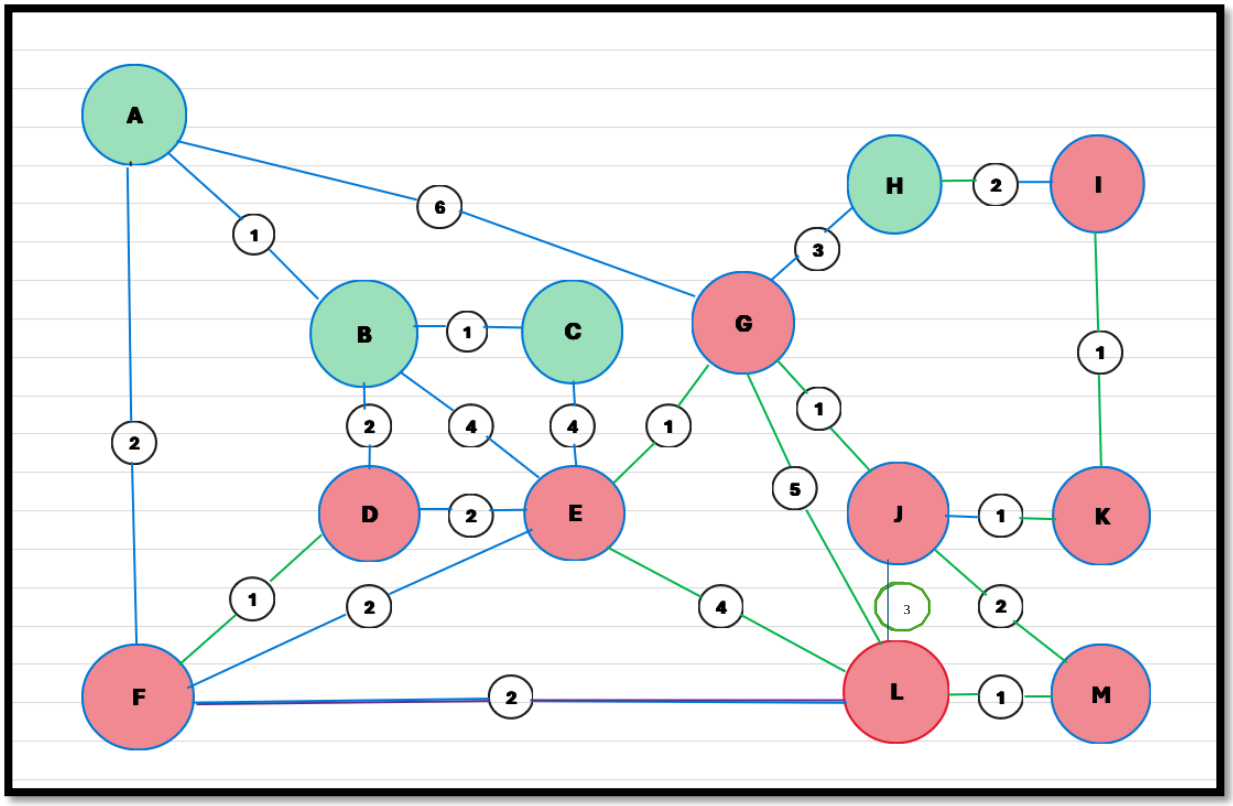



Fig 1.9

*Step 10:*

Vertex L M J K I G E D F and A are connected.

Vertex B is now added.

Vertex A-B = 1

Vertex I-H = 2

Vertex G-H = 3

Vertex E-C = 4

Vertex E-B = 4

Vertex D-B = 2

The Vertex with the lowest edge is chosen **A-B - 1**

The Heap is: **A-B – 1**, I-H -2, G-H-3, E-C-4,E-B-4, D-B-2

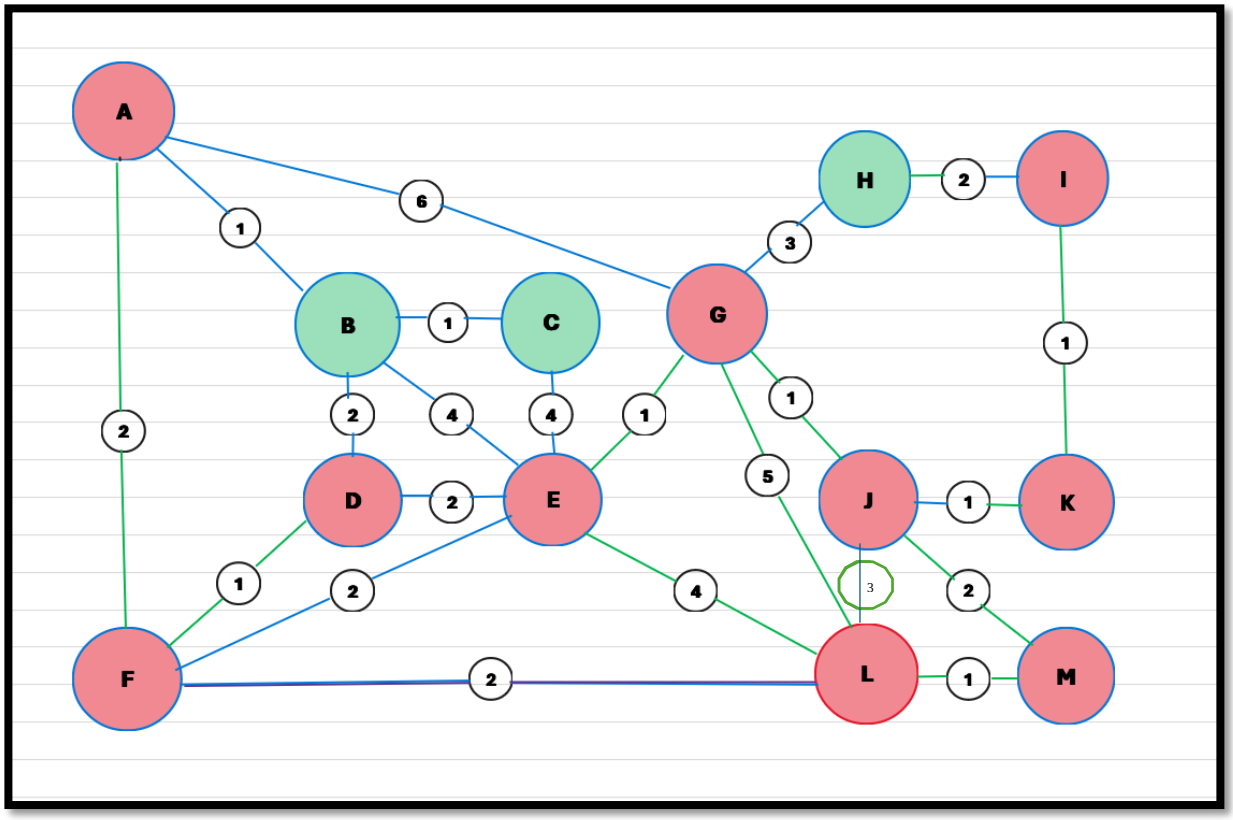



Fig 1.10

*Step 11:*

Vertex L M J K I G E D F A and B are connected.

Vertex C is now added.

Vertex B-C = 1

Vertex E-C = 4

Vertex I-H = 2

Vertex G-H = 3

The Vertex with the lowest edge is chosen **B-C - 1**

The Heap is: **B-C – 1**, G-H-3, I-H-2

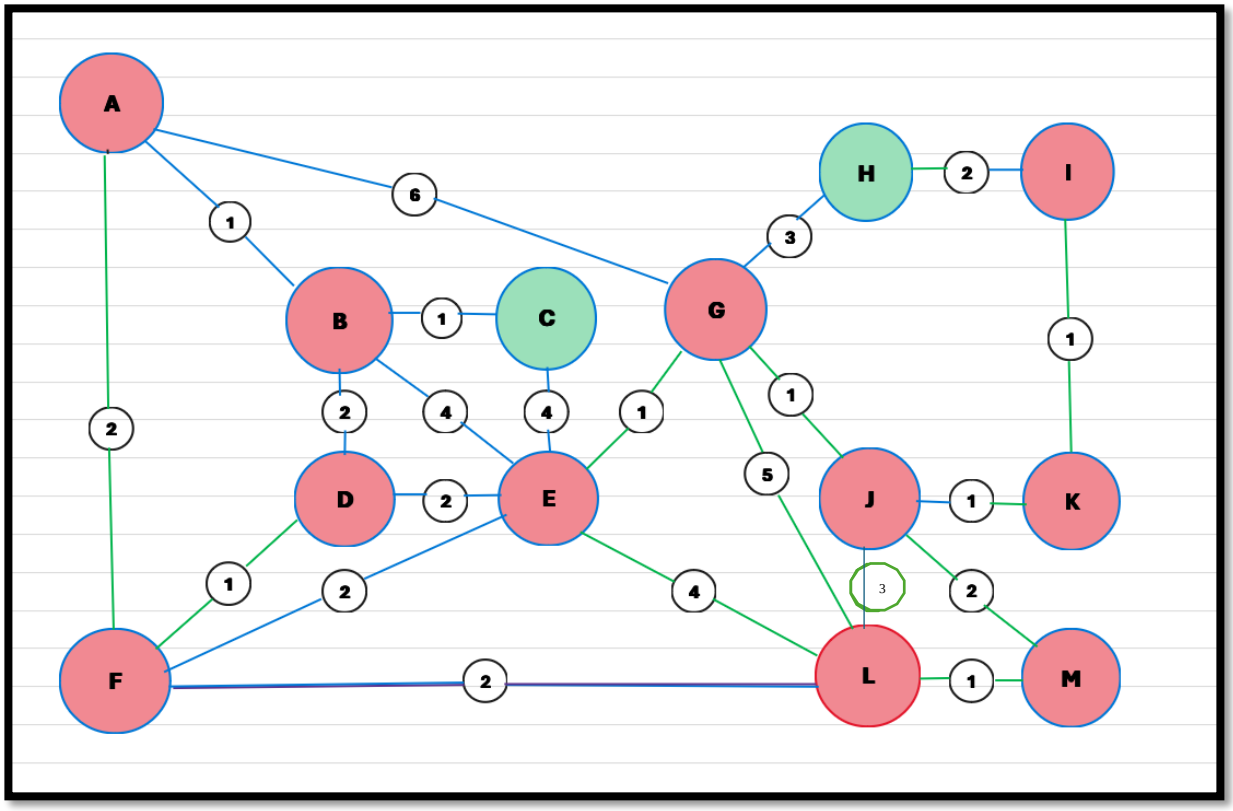



Fig 1.11

*Step 12:*

Vertex L M J K I G E D F A B and C are connected.

Vertex H is now added.

Vertex I-H = 2

Vertex G-H = 3

The Vertex with the lowest edge is chosen **I-H - 2**

The Heap is: G-H-3, **I-H-2**

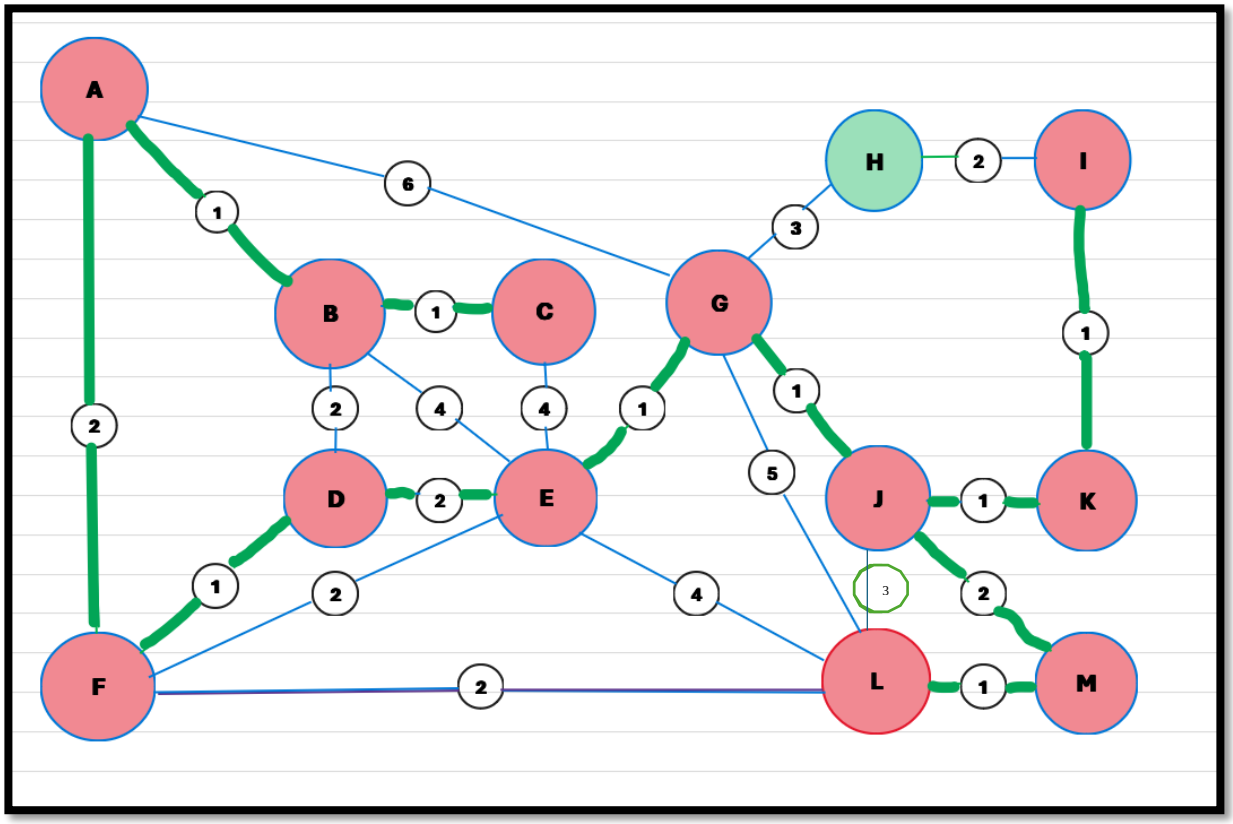



Fig 1.12

*Step 13:*

Vertex L M J K I G E D F A B C and H are connected. This gives the complete Minimum Spanning Tree

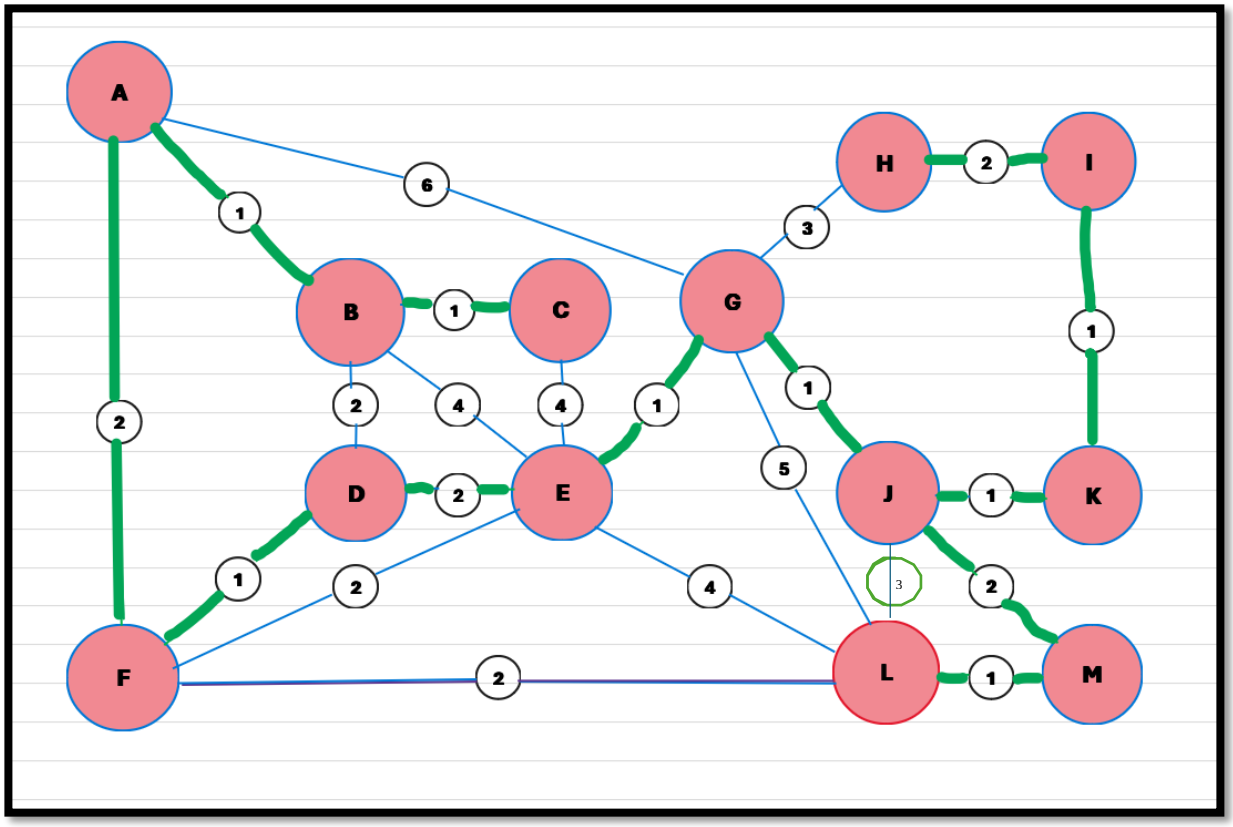



Fig 1.13

## Complete Minimum Spanning Tree

The following image depicts the complete Minimum Spanning Tree (MST) comprising all vertices and edges necessary to achieve the minimum total weight. MSTs play a crucial role in everyday applications such as construction and cabling, where minimizing cost over distance is paramount. By selecting the optimal subset of edges from a graph, MSTs efficiently connect all vertices while minimizing the overall expense, making them invaluable in various practical scenarios.

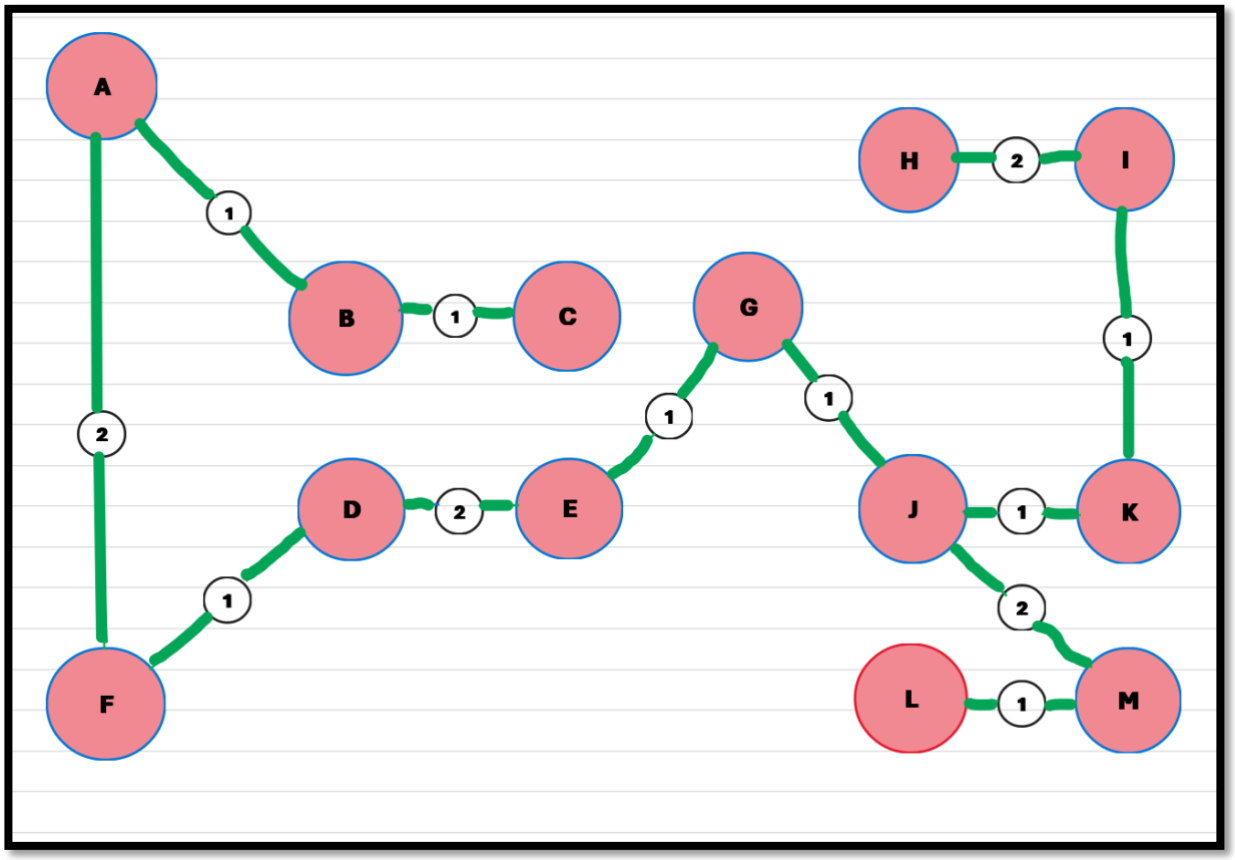



Fig 1.14

Total Weight of Minimum Spanning Tree is 16 (Fig 1.14).

## Breadth First Search

The Queue data structure and the implementation of the breadth-first search (BFS) algorithm are essential components for performing a shortest path spanning tree on an unweighted graph. BFS systematically explores all neighbouring nodes at the current depth level before moving on to nodes at the next depth level. This traversal strategy is crucial for finding the shortest path spanning tree on unweighted graphs, with applications ranging from network routing to social network analysis and solving shortest path problems.

### Algorithm

The BFS algorithm, as defined below, explores vertices in a graph starting from a given source vertex ‘s’

BFS(G, s)

```
1   for each vertex u in G:V
2   u.color = WHITE
3   u.d = ∞
4   u:π = NIL
5   s.color = GRAY
6   s.d = 0
7   s. π = NIL
8   Q = Ø
9   ENQUEUE(Q, s)
10  while Q ≠ Ø
11          u = DEQUEUE(Q)
12          for each v ∈ G:Adj[u]
13                  if v.color == WHITE
14                  v.color = GRAY
15                  v.d = u.d + 1
16                  v.π = u
17                  ENQUEUE(Q, v)
18          u.color = BLACK
```

(Cormen et al., 2009)

## Implementation

Breadth-First Search (BFS) is a graph traversal algorithm that explores all the neighbouring nodes at the present depth before moving on to the nodes at the next depth level. BFS is implemented to find the shortest path spanning tree on an unweighted graph, which is useful for applications like network routing, social network analysis, and shortest path problems.
BFS uses a queue data structure to keep track of the nodes to be visited. The Queue class implemented provides the necessary functionality for enqueuing and dequeuing nodes efficiently, ensuring that nodes are visited in the order they were discovered, as required by BFS.

## Node Handling

The Queue data structure efficiently handles the insertion and removal of nodes, making it suitable for BFS, where nodes are processed in a first-in-first-out (FIFO) manner. This allows BFS to systematically explore all nodes at a given level before moving on to the next level.
Within the BFS algorithm, each node is marked as visited once it is enqueued, ensuring that nodes are not processed multiple times. The output demonstrates the traversal process of BFS starting from a specified vertex L. The “Linked List enQueue” lines indicate the addition of vertices to the queue during the traversal (Image 9, 9a and 9b).
As BFS visits each vertex, it enqueues its neighbouring vertices, exploring them in subsequent iterations. The “BFS visited vertex” lines indicate the order in which vertices are visited, forming the shortest path spanning tree (Image 10 and 10a)

By following the BFS traversal, the algorithm systematically explores the graph, ensuring that all reachable vertices are visited while maintaining the shortest path spanning tree structure.

```
Breadth First Search:
BFS visited vertex L
BFS visited vertex M
BFS visited vertex J
BFS visited vertex G
BFS visited vertex F
BFS visited vertex E
BFS visited vertex L
BFS visited vertex K
BFS visited vertex H
BFS visited vertex A
BFS visited vertex D
BFS visited vertex C
BFS visited vertex B
BFS visited vertex I
```

Image 9

```
DFS visited vertex A along B--A

Breadth First Search:
Linked List enQueue: 12 ->
12


Linked List enQueue: 13 ->
13


Linked List enQueue: 10 ->
13  10


Linked List enQueue: 7 ->
13  10  7


Linked List enQueue: 6 ->
13  10  7  6


Linked List enQueue: 5 ->
13  10  7  6  5


BFS visited vertex L
Linked List enQueue: 12 ->
10  7  6  5  12


Linked List enQueue: 10 ->
10  7  6  5  12  10


BFS visited vertex M
Linked List enQueue: 12 ->
7  6  5  12  10  12


Linked List enQueue: 11 ->
7  6  5  12  10  12  11


Linked List enQueue: 7 ->
7  6  5  12  10  12  11  7


BFS visited vertex J
Linked List enQueue: 12 ->
6  5  12  10  12  11  7  12


Linked List enQueue: 8 ->
6  5  12  10  12  11  7  12  8


Linked List enQueue: 5 ->
6  5  12  10  12  11  7  12  8  5


Linked List enQueue: 1 ->
6  5  12  10  12  11  7  12  8  5  1


BFS visited vertex G
Linked List enQueue: 12 ->
5  12  10  12  11  7  12  8  5  1  12


Linked List enQueue: 5 ->
5  12  10  12  11  7  12  8  5  1  12  5
```

Image 9a

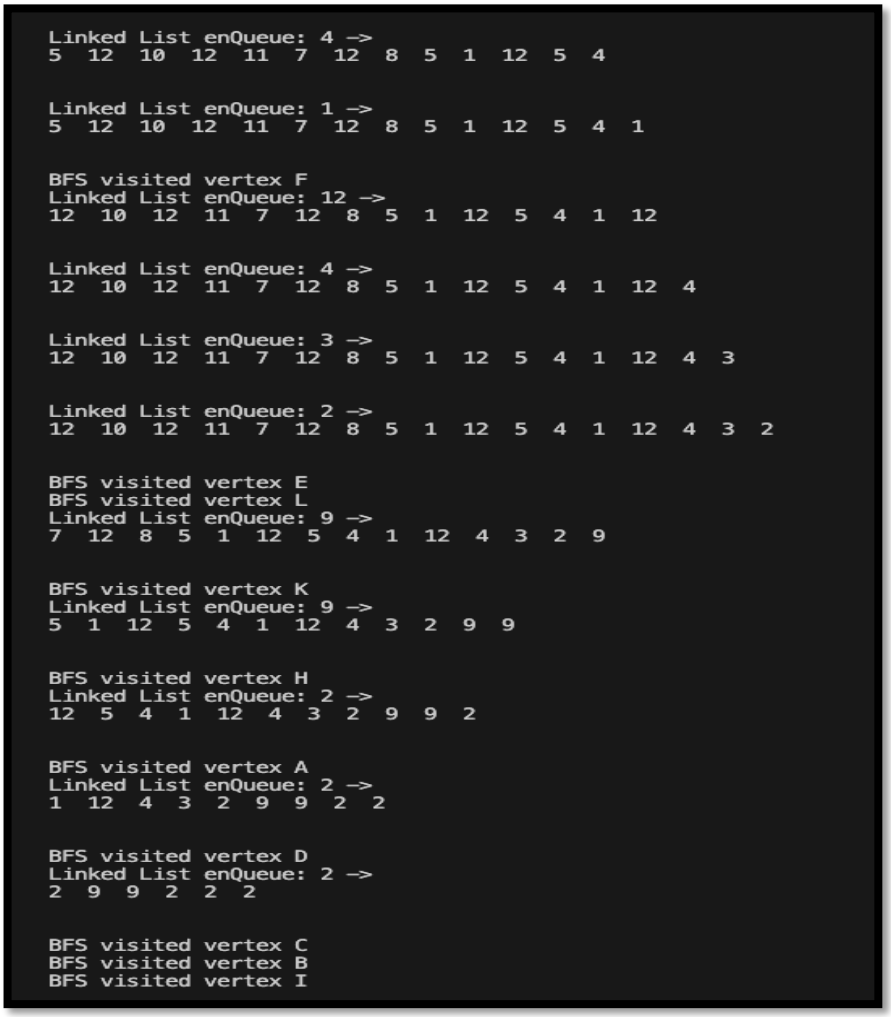

Linked List enQueue: 4 ->
5 12 10 12 11 7 12 8 5 1 12 5 4
Linked List enQueue: 1 ->
5 12 10 12 11 7 12 8 5 1 12 5 4 1
BFS visited vertex F
Linked List enQueue: 12 ->
12 10 12 11 7 12 8 5 1 12 5 4 1 12
Linked List enQueue: 4 ->
12 10 12 11 7 12 8 5 1 12 5 4 1 12 4
Linked List enQueue: 3 ->
12 10 12 11 7 12 8 5 1 12 5 4 1 12 4 3
Linked List enQueue: 2 ->
12 10 12 11 7 12 8 5 1 12 5 4 1 12 4 3 2
BFS visited vertex E
BFS visited vertex L
Linked List enQueue: 9 ->
7 12 8 5 1 12 5 4 1 12 4 3 2 9
BFS visited vertex K
Linked List enQueue: 9 ->
5 1 12 5 4 1 12 4 3 2 9 9
BFS visited vertex H
Linked List enQueue: 2 ->
12 5 4 1 12 4 3 2 9 9 2
BFS visited vertex A
Linked List enQueue: 2 ->
1 12 4 3 2 9 9 2 2
BFS visited vertex D
Linked List enQueue: 2 ->
2 9 9 2 2 2
BFS visited vertex C
BFS visited vertex B
BFS visited vertex I

Image 9b

```
    // BreathFirst search - Shortest path Spanning Tree - Unweighted graph
    public void breadthFirst(int s)
    { // Start breadth-first operation
        Queue q = new Queue(); // Initialize a queue for BFS traversal
        id = 0; // Initialize ID for marking visited vertices
        int u, v; // Variables for vertices

        System.out.print(s:"\n\nBreadth First Search: ");

        // Mark all vertices as not visited
        for (int i = 1; i <= V; i++)
        { // Loop through all vertices

            visited[i] = 0; // Mark vertex as not visited

        } // End for loop

        q.enQueue(s); // Queue the root node

        // Repeat until the queue is empty
        while (!q.isEmpty())
        { // Start while loop
            v = q.deQueue(); // Dequeue a vertex from the queue

            if (visited[v] == 0)
            { // If vertex is not visited
                visited[v] = id++; // Mark vertex as visited and assign ID
                for (Node t = adj[v]; t != z; t = t.next)
                { // Loop through all neighbors of v
                    u = t.vert; // Get the neighbor vertex
                    if (visited[u] == 0)
                    { // If neighbour vertex is not visited
                        q.enQueue(u); // Enqueue the neighbour vertex
                    } // End inner if statement
                } // End for loop
                System.out.print("\nBFS visited vertex " + toChar(v));
            } // End outer if statement
        } // End while loop
    } // End breadth-first operation
}
```

Image 10

```
// Add values to Queue
public void enQueue(int x)
{ // Start enQueue operation

    Node t;

    t = new Node();
    t.data = x;
    t.next = z;

    if (head == z) // If Empty list
    { // Start if
        head = t;
    } // End if
    else // List not empty
    { // Start else
        tail.next = t;
    } // End else

    tail = t; // New node is now at the tail

    System.out.println("\nLinked List enQueue: " + x + " -> ");
    display();
}

// Removing values from the Queue
public int deQueue() { // Start deQueue

    if(!isEmpty()){
        int temp = head.data;
        head = head.next;
        return temp;
    }
    if(isEmpty()){
        tail = null;
    } else {
        System.out.println(x:"Queue is Empty");

    }
    return -1;

    }

// Check if the Queue is empty
public boolean isEmpty() { // Start isEmpty operation

    return head == head.next;

} // End isEmpty opetion

// Display contents of queue
public void display() {

    Node t = head;
    while (t != t.next) {

        System.out.print(t.data + "  ");
```

Image 10a

## Depth first search

Depth-First Search (DFS) is applied to traverse the graph and visit all vertices in a systematic manner. DFS was chosen in this assignment for its efficiency.

### Algorithm

```
DFS.G
    1       for each vertex u  ∈G:V
    2               u.color = WHITE
    3               u: π D NIL
    4       time D 0
    5       for each vertex u ∈G:V
    6               if  u.color == WHITE
    7                       DFS-VISIT (G; u)
DFS-VISIT (G, u)
     1 time = time + 1 // white vertex u has just been discovered
    2   u.d = time
    3   u.color =  GRAY
    4   for each vertex v in G.Adj[u]       // explore each edge (u; v)
    5       if  v.color == WHITE
    6               v. π = u
    7               DFS-VISIT (G, v)
    8    time = time + 1
    9   u.f = time
    10  u.color =  BLACK                    // blacken u it is finished
```

(Cormen et al., 2009)

### Implementation

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It's well-suited for traversing and searching graphs and is particularly useful for exploring deep paths in graphs or trees. In this case, DFS is used to explore all reachable vertices from a starting vertex L (Image 11) and commented code (Image 12).

### Efficiency

The time complexity of DFS is O(V + E), where V is the number of vertices and E is the number of edges. This is because DFS visits each vertex and edge once.
The space complexity of DFS is O(V), where V is the number of vertices. This is because DFS uses a stack to keep track of vertices, and in the worst case, the stack can contain all vertices.

### Advantages

Depth-First Search is relatively simple to implement and understand, making it a popular choice for graph traversal.

- Uses less memory compared to breadth-first search (BFS) because it only needs to keep track of vertices along the current path
- Suitable for applications such as finding connected components, detecting cycles in graphs, and traversing tree structures
- Chosen for its simplicity, efficiency in memory usage, and suitability for exploring graphs deeply. Its time complexity is reasonable for many practical applications, making it a versatile choice for graph traversal tasks

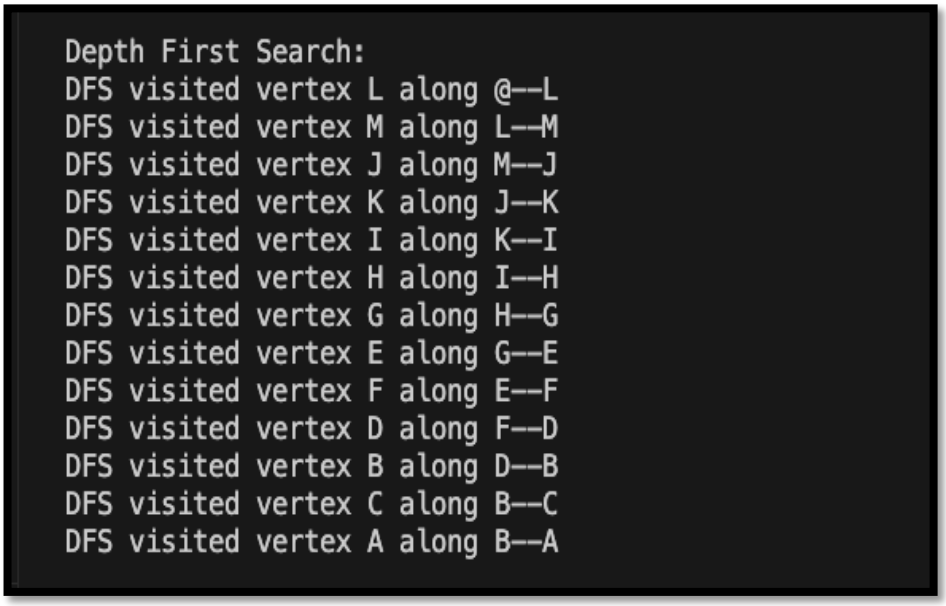



Image 11

```
public void DF(int s)
{ // Start Depth First Search operation
    System.out.print(s:"\nDepth First Search: ");
    id = 0;

    // Initialize visited array
    for (int v = 1; v <= V; v++) { // Loop through all vertices
        visited[v] = 0; // Mark vertex as not visited
    }

    // Call recursive depth-first search
    dfVisit(id, s);
} // End Depth First Search operation

tabnine: test | explain | document | ask
// Recursive Depth First Traversal
private void dfVisit(int parent, int v)
{ // Start Depth First Visit operation
    Node u;

    // Initialize adjacency node
    u = adj[v];

    // Mark vertex as visited
    visited[v] = 1;

    // Print visited vertex
    System.out.print("\nDFS visited vertex " + toChar(v) + " along " + toChar(parent) + "--" + toChar(v));

    // Traverse adjacent vertices
    for (u = adj[v]; u != z; u = u.next)
    { // Loop through adjacent vertices
        if (visited[u.vert] == 0)
        { // If vertex is not visited
            dfVisit(v, u.vert); // Recursively visit adjacent vertex
        }
    }

} // End Depth First Visit operation
```

Image 12

# Dijkstra Algorithm Shortest Path Tree

The provided diagrams illustrate a step-by-step construction of the Shortest Path Tree (SPT) using Dijkstra's algorithm, starting from a designated vertex, L, for the given sample graph. Dijkstra's algorithm is a popular method used to find the shortest paths from a single source vertex to all other vertices in a weighted graph, with non-negative edge weights.

Each step in the construction process involves selecting the next vertex to add to the SPT, ensuring that the shortest path from the source vertex to each vertex is progressively determined. The algorithm maintains priority queues (heaps) to efficiently select the next vertex based on its tentative distance from the source vertex.

The diagrams visually represent the progression of Dijkstra's algorithm, showcasing the vertices being considered for inclusion in the SPT, the growing tree, and the updates to the tentative distances and parent vertices. Additionally, the contents of the priority queue (heap), parent[] array, and dist[] array are depicted at each step, providing insight into how the algorithm operates and how it efficiently computes the shortest paths.

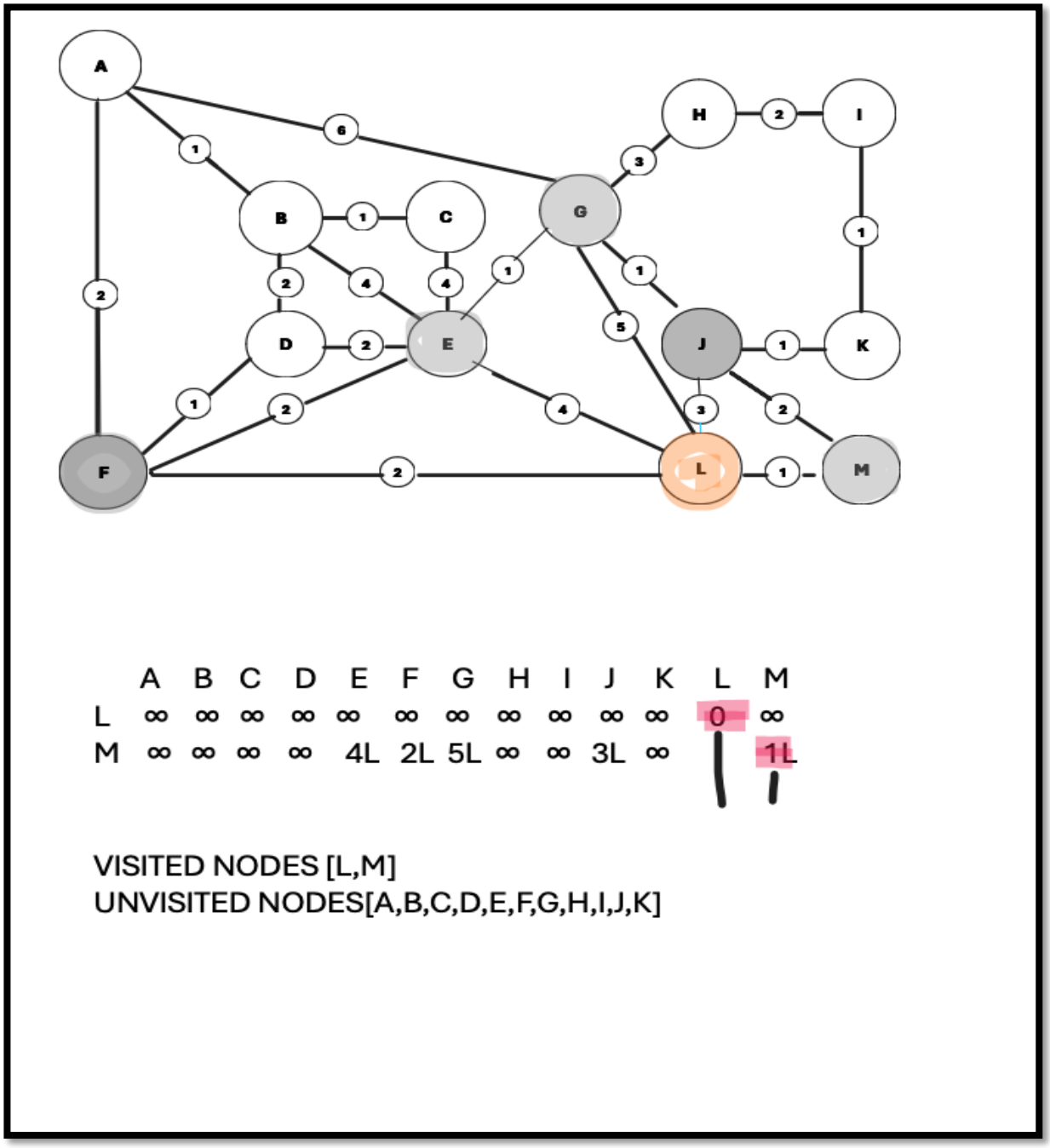

A
B
C
D
E
F
G
H
I
J
K
L
M
6
1
2
1
2
4
4
1
3
1
5
1
2
1
1
2
4
3
2
2
1
A B C D E F G H I J K L M
L ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞
M ∞ ∞ ∞ ∞ 4L 2L 5L ∞ ∞ 3L ∞ 1L
VISITED NODES [L,M]
UNVISITED NODES[A,B,C,D,E,F,G,H,I,J,K]

Fig 2.1

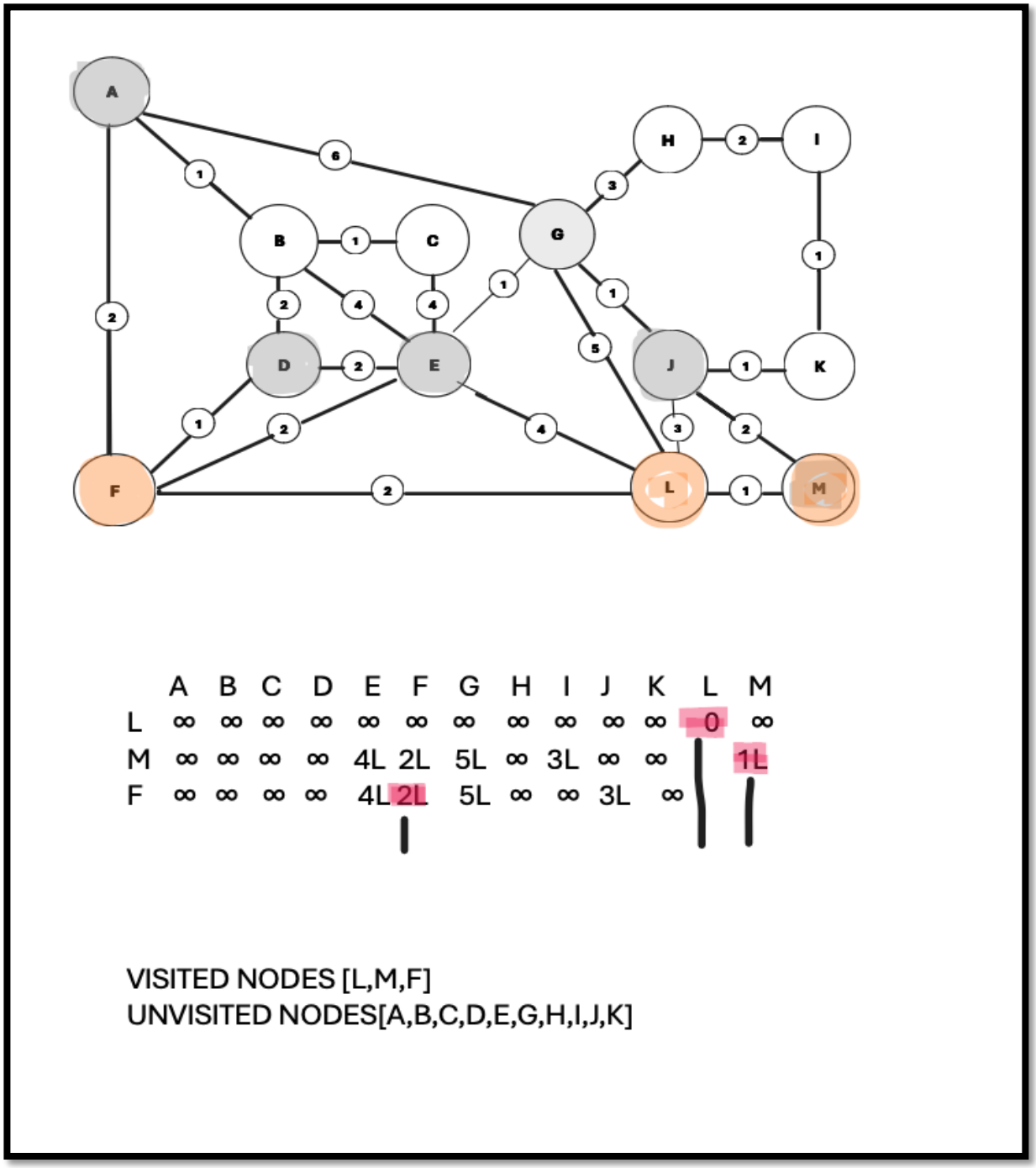

A
B
C
D
E
F
G
H
I
J
K
L
M
6
1
2
1
2
4
4
1
3
2
1
1
5
1
2
1
4
3
2
2
1
A B C D E F G H I J K L M
L ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞
M ∞ ∞ ∞ ∞ 4L 2L 5L ∞ 3L ∞ ∞ 1L
F ∞ ∞ ∞ ∞ 4L 2L 5L ∞ ∞ 3L ∞
VISITED NODES [L,M,F]
UNVISITED NODES[A,B,C,D,E,G,H,I,J,K]

Fig 2.2

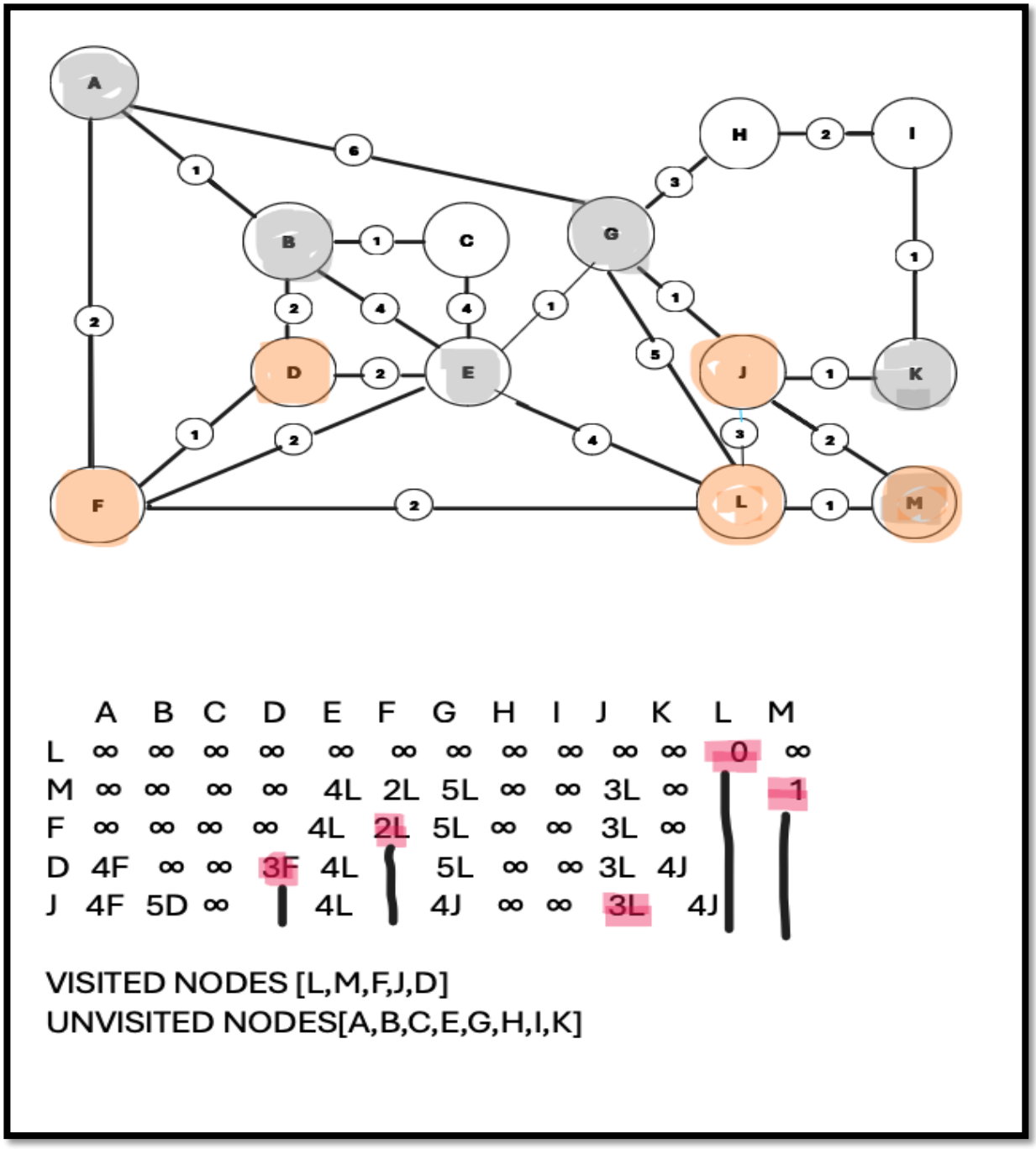

A
H
I
B
C
G
D
E
J
K
F
L
M
1
6
3
2
1
1
2
4
4
1
1
5
2
1
1
2
4
3
2
2
1
A B C D E F G H I J K L M
L ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞
M ∞ ∞ ∞ ∞ 4L 2L 5L ∞ ∞ 3L ∞ 1
F ∞ ∞ ∞ ∞ 4L 2L 5L ∞ ∞ 3L ∞
D 4F ∞ ∞ 3F 4L 5L ∞ ∞ 3L 4J
J 4F 5D ∞ 4L 4J ∞ ∞ 3L 4J
VISITED NODES [L,M,F,J,D]
UNVISITED NODES[A,B,C,E,G,H,I,K]

Fig 2.3

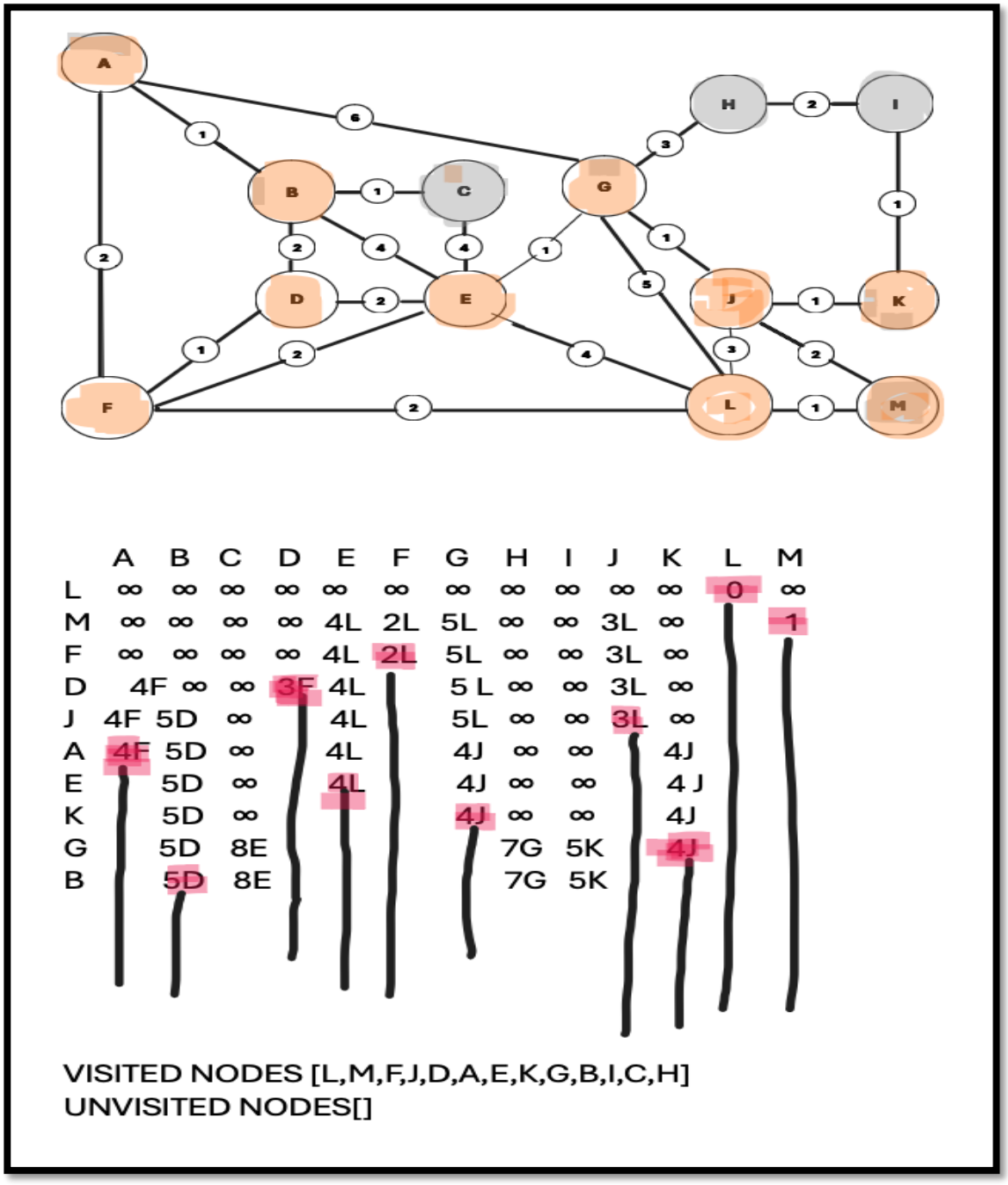

A
B
C
D
E
F
G
H
I
J
K
L
M
A B C D E F G H I J K L M
L ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞
M ∞ ∞ ∞ ∞ 4L 2L 5L ∞ ∞ 3L ∞ 1
F ∞ ∞ ∞ ∞ 4L 2L 5L ∞ ∞ 3L ∞
D 4F ∞ ∞ 3F 4L 5 L ∞ ∞ 3L ∞
J 4F 5D ∞ 4L 5L ∞ ∞ 3L ∞
A 4F 5D ∞ 4L 4J ∞ ∞ 4J
E 5D ∞ 4L 4J ∞ ∞ 4 J
K 5D ∞ 4J ∞ ∞ 4J
G 5D 8E 7G 5K 4J
B 5D 8E 7G 5K
VISITED NODES [L,M,F,J,D,A,E,K,G,B,I,C,H]
UNVISITED NODES[]

Fig 2.4

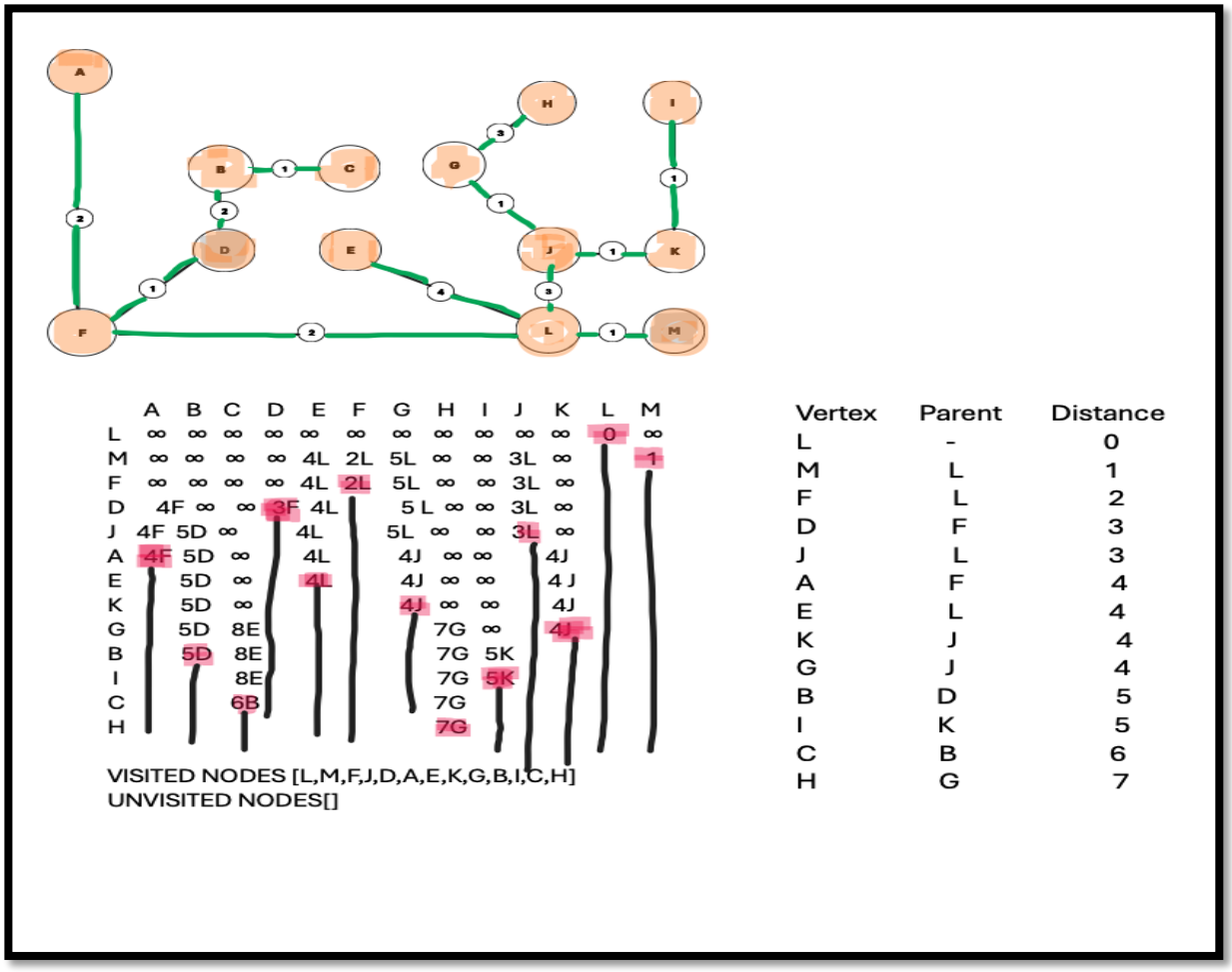



Fig 2.5

The provided code implements Dijkstra's algorithm for finding the shortest paths from a source vertex to all other vertices in a weighted graph (Image 13). It initializes arrays to store distances,

parent vertices, and heap positions, then proceeds to find the shortest paths using a priority queue.

Initialization involves setting distances to infinity, setting the distance to the source vertex as 0, and creating a heap with the source vertex. The main loop continues until the heap is empty, retrieving vertices with minimum distances, updating distances of adjacent vertices if shorter paths are found, and adjusting their positions in the heap.

After completion, the algorithm outputs the total weight of the Shortest Path Tree (SPT) and stores the parent array for constructing the SPT. This efficient computation provides valuable insights into network connectivity and optimization. The output includes the total weight of the SPT (Image 14).

```java
public void SPT_Dijkstra(int s)
{
    int v, u;
    int wgt, wgt_sum = 0; // Initialize variables for weight and total weight of SPT
    int[] dist = new int[V + 1]; // Array to store distances from source vertex
    int[] parent = new int[V + 1]; // Array to store parent vertices for constructing SPT
    int[] hPos = new int[V + 1]; // Array to store heap positions
    Node t;

    // Initialize distance array, parent array, and heap position array
    for (v = 1; v <= V; v++)
    {
        dist[v] = Integer.MAX_VALUE; // Initialize distances to infinity
        parent[v] = 0; // Initialize parent vertices to 0
        hPos[v] = 0; // Initialize heap positions to 0
    }

    // Set distance to source vertex as 0
    dist[s] = 0;

    // Create a heap and insert the source vertex
    Heap h = new Heap(V, dist, hPos);
    h.insert(s);

    while (!h.isEmpty()) // Continue until the heap is empty
    {
        // Retrieve vertex with minimum distance from the heap
        v = h.remove();

        wgt_sum += dist[v]; // Add the distance and weight to total weight of SPT
        t = adj[v];

        // Traverse adjacent vertices of the removed vertex
        for (t = adj[v]; t != z; t = t.next)
        {
            u = t.vert; // Get adjacent vertex
            wgt = t.wgt; // Get weight of edge to adjacent vertex
            if (wgt < dist[u]) // If the weight is less than the distance to vertex u
            {
                dist[u] = wgt; // Update distance to u
                parent[u] = v; // Update parent pointer
                if (hPos[u] == 0) // If vertex u is not in heap
                {
                    h.insert(u); // Insert u into heap
                }
                else // If u is in heap, modify its position
                {
                    h.siftUp(hPos[u]);
                }
            }
        }
    }

    // Output the total weight of SPT
    System.out.println(x:"\n\n------- Dijkstra SPT ---------");
    System.out.print("Weight of SPT = " + wgt_sum + "\n");

    mst = parent; // Store parent array for constructing SPT
}
```

Image 13

```
PROBLEMS 1   TERMINAL   DEBUG CONSOLE   PORTS

------- Dijkstra SPT ---------
Weight of SPT = 19
```

Image 14

## Menu

The program offers a user-friendly menu option, guiding users through graph data input and visualization. Initially, users are prompted to input the filename containing graph data. This file should adhere to a specific format, typically detailing the vertices and edges of the graph. Subsequently, users are prompted to specify the starting vertex for the graph, with vertices represented by numerical values. Notably, the assignment designates vertex "L" as having a predefined numerical value of 12. Upon receiving the filename and starting vertex, the program initializes a graph object based on the provided file. It then proceeds to display comprehensive details about the graph, showcasing its vertices, edges, and other pertinent information regarding its representation. Additionally, the program includes robust exception handling to gracefully manage any potential errors encountered during file input or processing. If an exception occurs, such as an invalid filename or starting vertex input, the program promptly notifies the user with an error message, guiding them to re-enter valid information (Image 15 and 16).

```
public class GraphLists {
    Run | Debug | tabnine: test | fix | explain | document | ask
    public static void main(String[] args) throws IOException
    {
        // int s = 2;
        // String fname = "wGraph1.txt";

        // Graph g = new Graph(fname);


       try { // start try

            try (// prompt user to enter name of file
                    Scanner input = new Scanner(System.in)) {
                System.out.println(x:"Enter the name of the file: ");
                String fname = input.nextLine();

                File v = new File(fname);

                // prompt user to enter starting vertex of the graph
                System.out.println(x:"Enter the starting vertex of the graph: ");
                System.out.println(x:"Vertex A = 1, B = 2, C = 3, D = 4, ..... M = 13\n");
                System.out.println(x:"Starting Vertex for this assignment is L = 12");
                int s = Integer.parseInt(input.nextLine());

                Graph g = new Graph(fname);

                g.display();
                g.DF(s);
                g.breathFirst(s);
                g.MST_Prim(s);
                g.SPT_Dijkstra(s);
                g.showMST();
            }

        } // end try
        catch (Exception e) { // start catch
            System.out.println(
                    x:"\nPlease enter a valid file name with extension .txt, and a valid number of your starting vertex afterwards");
        } // end catch

    } // end main
} // end GraphLists class
```

Image 15

```
Enter the name of the file:
wGraph1.txt

Enter the starting vertex of the graph:
Vertex A = 1, B = 2, C = 3, D = 4, ..... M = 13

Starting Vertex for this assignment is L = 12


12
Parts[] = 13 22
```

Image 16

# Kruskal's Algorithm

Kruskal's algorithm operates by gradually building a forest of trees, where each tree initially contains a single vertex. It selects edges to connect these trees, ensuring that each added edge is of minimum weight among all edges connecting different trees. This approach guarantees that the chosen edge is safe to include in the growing forest. Since Kruskal's algorithm consistently chooses the edge with the least weight available at each step, it fits the definition of a greedy algorithm.

In this Java program implementation of Kruskal's algorithm (Image 17), it employs a disjoint set data structure to manage several disjoint sets of elements, each representing a tree in the current forest. Each set contains the vertices belonging to one tree. The *'findSet'* operation helps identify the representative element of the set to which a vertex belongs. By comparing the representative elements of two vertices, this determines whether they belong to the same tree. To merge trees, Kruskal's algorithm utilizes the *'union'* procedure.

MST-KRUSKAL (G, w)

```
1  A = ∅
2  for each vertex ∈ G.V
3          Make-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge(u,v)  ∈ G:E, taken in nondecreasing order by weight
6          if FIND SET(v) ≠ FIND-SET(v)
7                  A = A ∪ {(u, v)}
8                  Union(u,v)
9  Return A
```

(Cormen et al., 2009)

```java
// Class to support union-find operations
class UnionFindSets
{
    private int[] treeParent;
    private int N;

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

        //Initially each element is its own parent
        for(int i = 0;i <=V; i++)
        {
            treeParent[i] = i;
        }
    } // end union find data

    public int findSet(int vertex)
    {
        // Traverse through the parent pointers until finding the root node
        while (vertex != treeParent[vertex])
        { // Start while
            vertex = treeParent[vertex];
        } // End while

        return vertex;
    }

    public void union(int set1, int set2)
    {
        // Joins two subsets into single set
        treeParent[findSet(set2)] = findSet(treeParent[set1]);
        System.out.print("Union: (" + toChar(set1) + "," + toChar(set2) + "): ");
    }

    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)
```

Image 17

## Program Code Explained

- The program reads the input file containing the graph data and creates a linked list to store vertices and their connections along with edge weights
- It displays the vertices connected to each other along with their corresponding edge weights
- The program cycles through the edges of the graph to find the Minimum Spanning Tree using Kruskal's Algorithm. It iteratively selects the smallest edge that does not form a cycle in the current partial MST
- Throughout the algorithm's execution, the Union-Find data structure efficiently manages partitions and set representations of vertices, facilitating cycle detection, and set merging operations

## Time Complexity

Kruskal's Algorithm is known to run in O(E log V) time, where E is the number of edges and V is the number of vertices in the graph.

Kruskal's Algorithm is edge-based and involves sorting the edges according to their weights. While various sorting algorithms such as Merge-Sort or Quick-Sort can be used, this implementation utilizes the Union-Find data structure for efficiency. The algorithm builds the MST separately, adding the smallest edge to the tree if it does not form a cycle. Unlike Prim's Algorithm, which is vertex-based, Kruskal's Algorithm focuses on selecting edges based on their weights.

## Output

The output illustrates the execution of Kruskal's algorithm on a graph to determine its minimum spanning tree. Initially, individual vertices of the graph are isolated in distinct sets. Through the algorithm's progression, these sets are gradually merged by prioritizing edges of minimal weight. Each line in the output denotes the sets' configurations following each union operation. The "Union" lines signify the amalgamation of sets through the addition of edges to the minimum spanning tree, such as “(A,B)” indicating the fusion of sets containing vertices A and B. Once all unions are completed, the output showcases the edges of the minimum spanning tree alongside their respective weights (Image 18). These edges establish the most concise connections covering all graph vertices without forming cycles. Each line under “Minimum spanning tree build from following edges” provides notation for an edge within the minimum spanning tree, shown by “Edge A--1--B” representing an edge linking vertex A to vertex B with a weight of 1 (Image 19).


Image 18


Image 19

## Union Find Partition and Set Representations

This section provides insights into the Union-Find partition and set representations at every traversal during the execution of Kruskal's Algorithm. Unlike selecting a starting vertex, Kruskal's Algorithm operates by comparing edges based on their weights, starting from the lowest and progressing upwards.

The graph used to determine the Minimum Spanning Tree (MST) through Kruskal's Algorithm is presented below. Initially, the edges are sorted in ascending order: 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 4, 4, 4, 5, 6. Each traversal scrutinizes the possibility of adding a new edge without introducing a cycle.

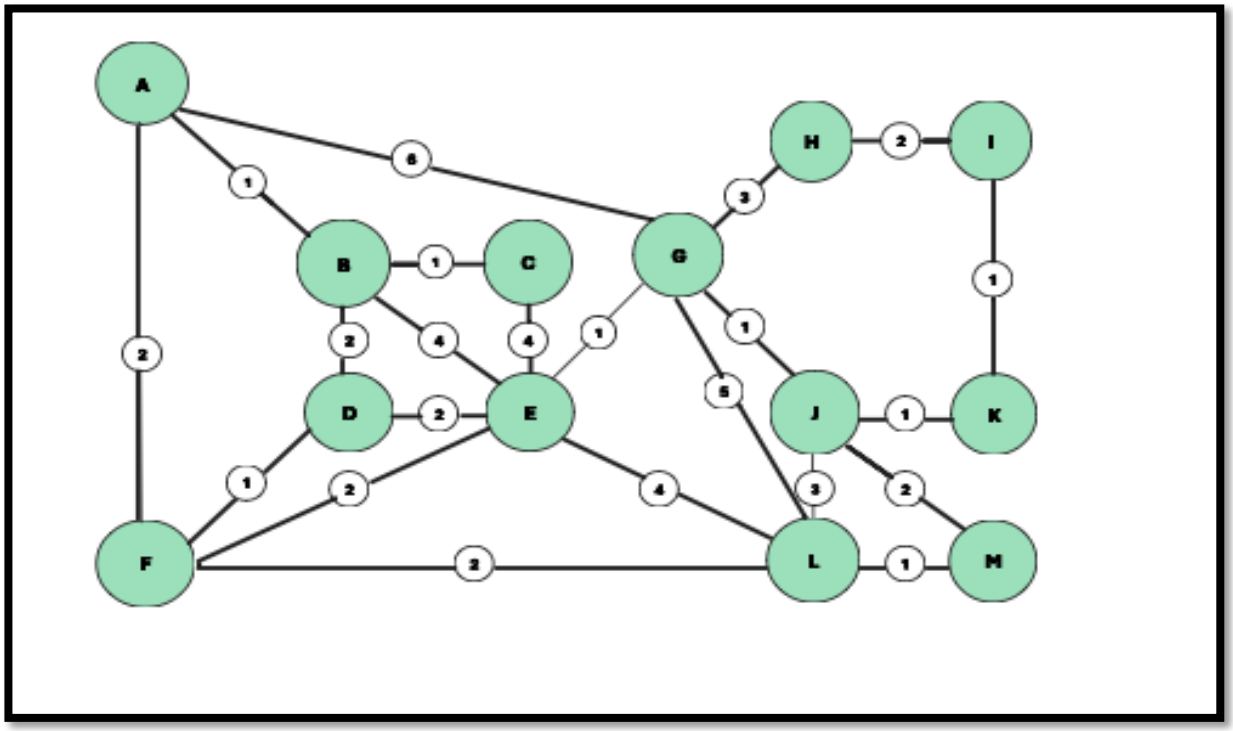



Fig 3.1

### Disjoint Sets

At the outset, the number of sets equals the number of vertices. With each edge addition, two sets are merged. The algorithm halts when only a single set remains.

The disjoint sets in this graph are represented as follows:

A, B, C, D, E, F, G, H, I, J, K, L, M.

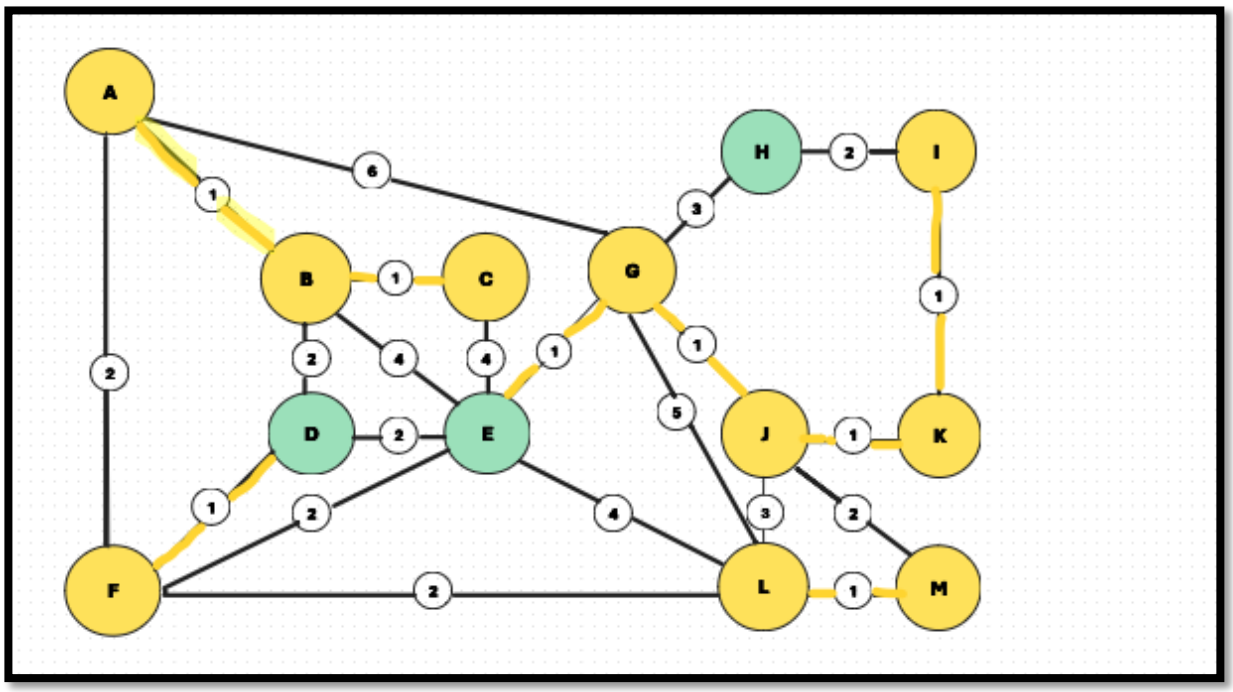



Fig 3.2

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

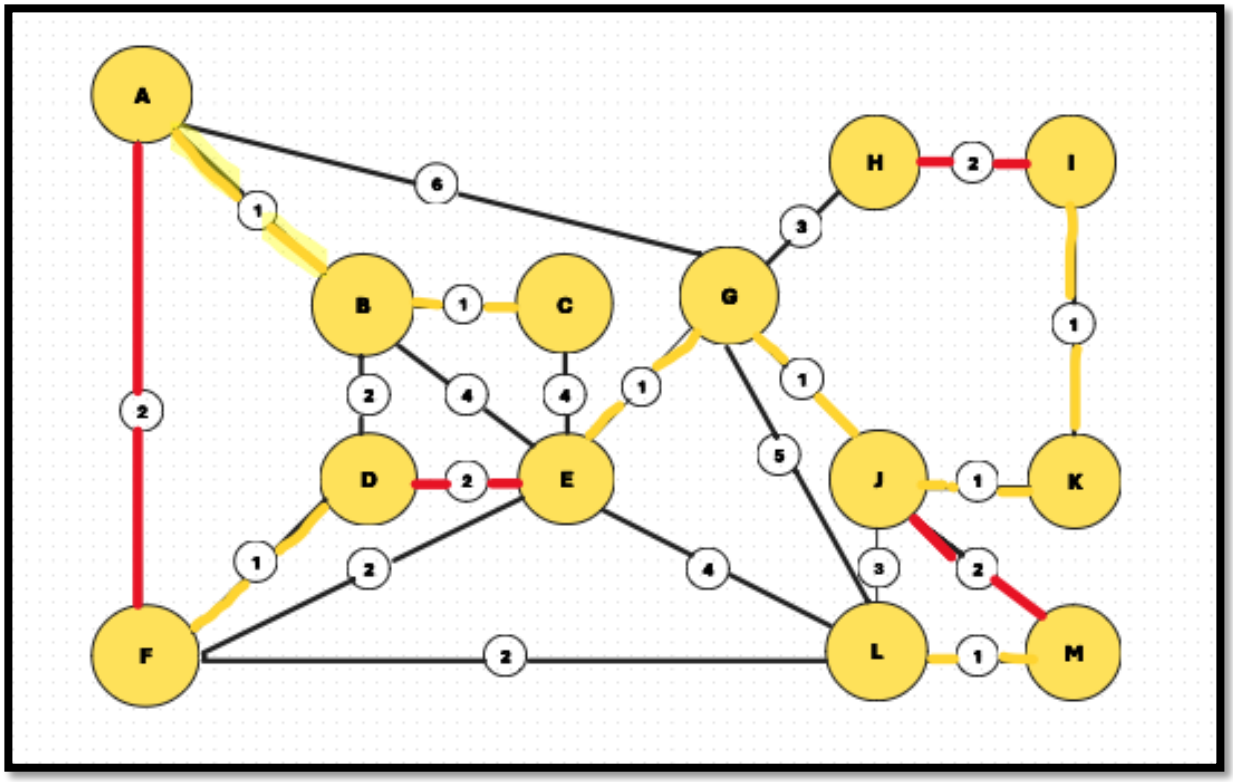



Fig 3.3

Edge D--2--E

Edge H--2--I

Edge J--2--M

Edge A--2--F

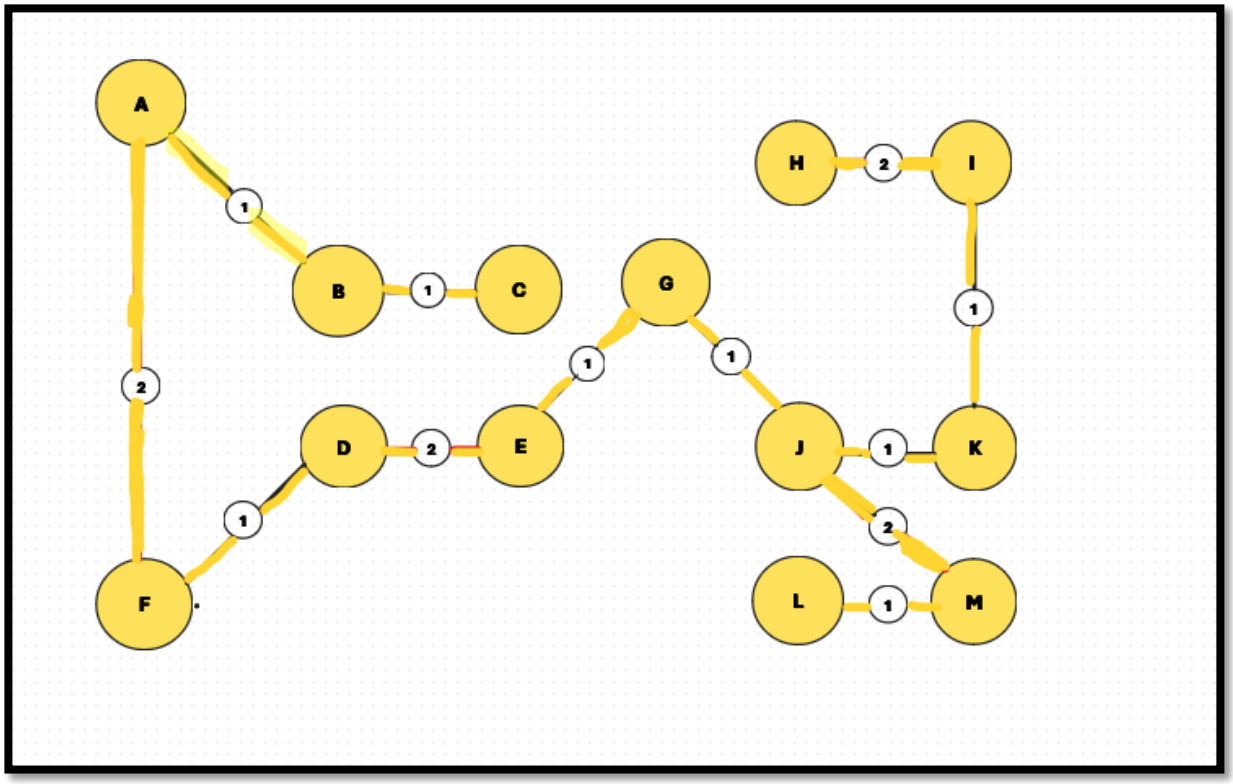



Fig 3.4

## Kruskal's sets

In an optimized approach to implementing disjoint sets, rooted trees represent sets, where each node corresponds to a member, and each tree represents a distinct set (Fig 4.1 – Fig 4.4). Within a disjoint-set forest, each member solely points to its parent, and the root of each tree contains the representative element, acting as its own parent. While the basic algorithms employing this structure may not initially offer faster performance compared to those using a linked-list representation, the integration of two key heuristics namely, "union by rank" and "path compression" allows to attain an asymptotically optimal disjoint-set data structure.

Regarding the three primary disjoint-set operations, the "*MAKE-SET"* operation straightforwardly generates a tree consisting of a single node. The "*FIND-SET"* operation entails traversing parent pointers until reaching the root of the tree, with the nodes encountered along this path forming the find path. Finally, the "*UNION"* operation involves directing the root of one tree to point to the root of another tree, effectively merging the two sets (Cormen et al., 2009).

Start with a forest which is a collection of trees (Lawlor, 2024).
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 }

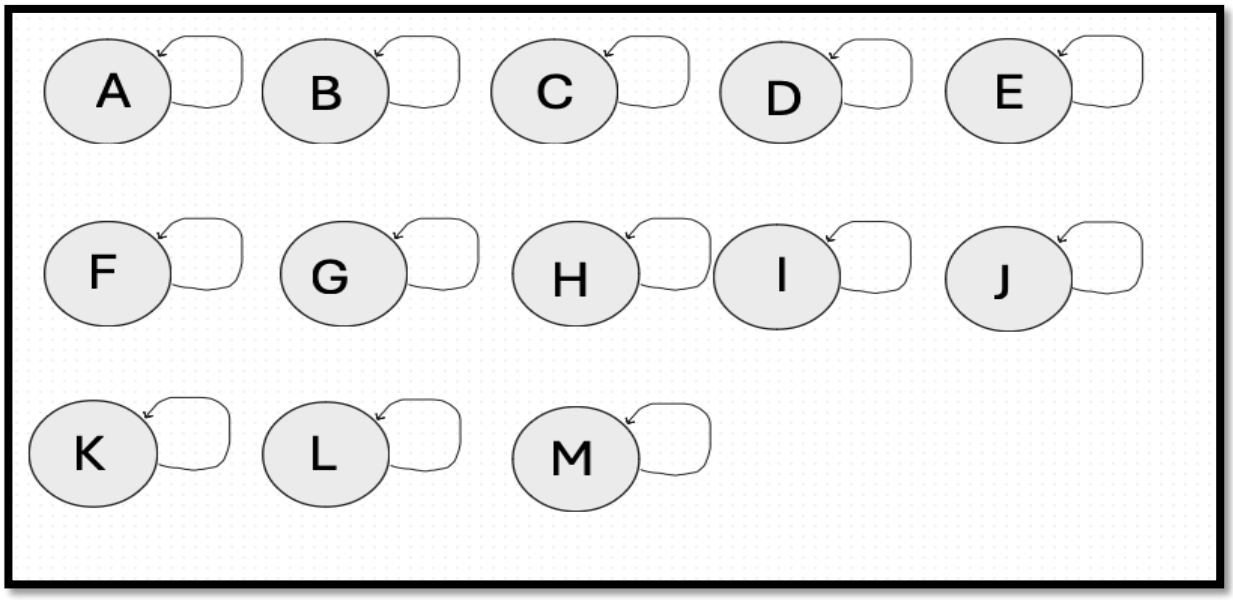



Fig 4.1

1st Traverse

Select two

Edge A and B weight 1

Union: (A,B): **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 }

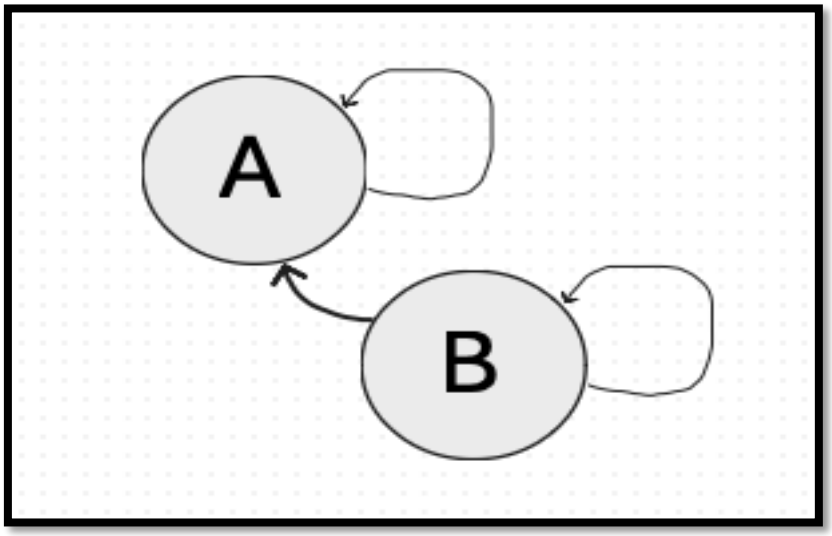



Fig 4.2

A links to itself therefore is the root of the tree.

2nd Traverse

Edge B and C weight 1

Union: (B,C): **Set{A B C }** Set{D } Set{E } Set{F } Set{G } Set{H } Set{I } Set{J } Set{K } Set{L } Set{M }

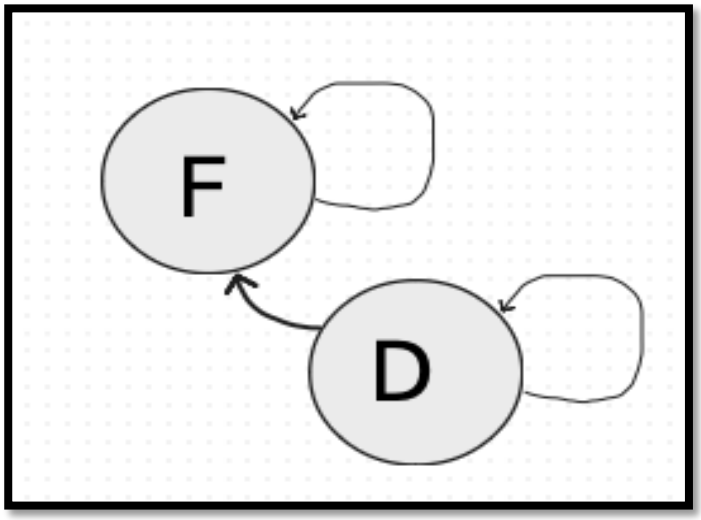



Fig 4.3

3rd Traverse

Edge D and F weight 1

Union: (D,F): **Set{A B C }** **Set{D F }** Set{E } Set{G } Set{H } Set{I } Set{J } Set{K } Set{L } Set{M }

4th Traverse

Edge I and K weight 1

Union: (I,K): **Set{A B C }** Set{D F } Set{E } Set{G } Set{H } **Set{I K }** Set{J } Set{L } Set{M }

5th Traverse

Edge J and K weight 1

Union: (J,K): **Set{A B C }** Set{D F } Set{E } Set{G } Set{H } **Set{I J K }** Set{L } Set{M }

6th Traverse

Edge E and G weight 1

Union: (E,G): Set{A B C } Set{D F } Set{E G } Set{H } Set{I J K } Set{L } Set{M }

7th Traverse

Edge L and M weight 1

Union: (L,M): Set{A B C } Set{D F } Set{E G } Set{H } Set{I J K } Set{L M }

8th Traverse

Edge G and J weight 1

Union: (G,J): Set{A B C }  Set{D F }  Set{E G I J K }  Set{H }  Set{L M }

9th Traverse

Edge D and E weight 2

Union: (D,E): Set{A B C }  Set{D E F G I J K }  Set{H }  Set{L M }

10th Traverse

Edge H and I weight 2

Union: (H,I): Set{A B C }  Set{D E F G H I J K }  Set{L M }

11th Traverse

Edge J and M weight 2

Union: (J,M): Set{A B C }  Set{D E F G H I J K L M }

12th Traverse

Edge A and F weight 2

Union: (A,F): Set{A B C D E F G H I J K L M }

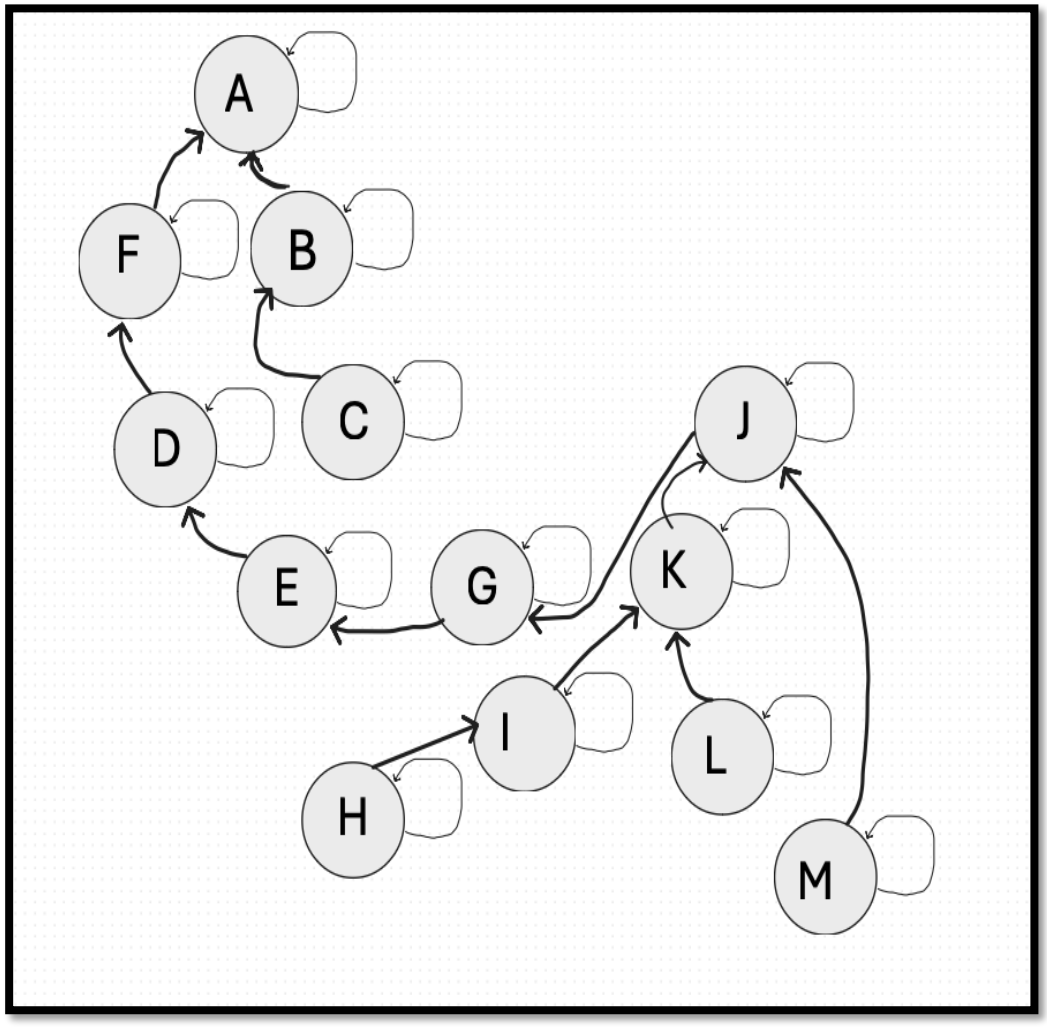



Fig 4.4

# Testing

This larger graph now contains 26 vertices and 44 edges, providing a more extensive network for testing purposes. This graph is used to assess the performance and scalability of the algorithms effectively.

This expanded graph represents an extended network of towns or locations with increased complexity. With a larger number of vertices and edges, this graph offers more opportunities to test the scalability and efficiency of graph algorithms. The edge weights continue to indicate distances or costs associated with traveling between towns, providing a diverse set of scenarios for algorithm analysis (Image 20 and 21).

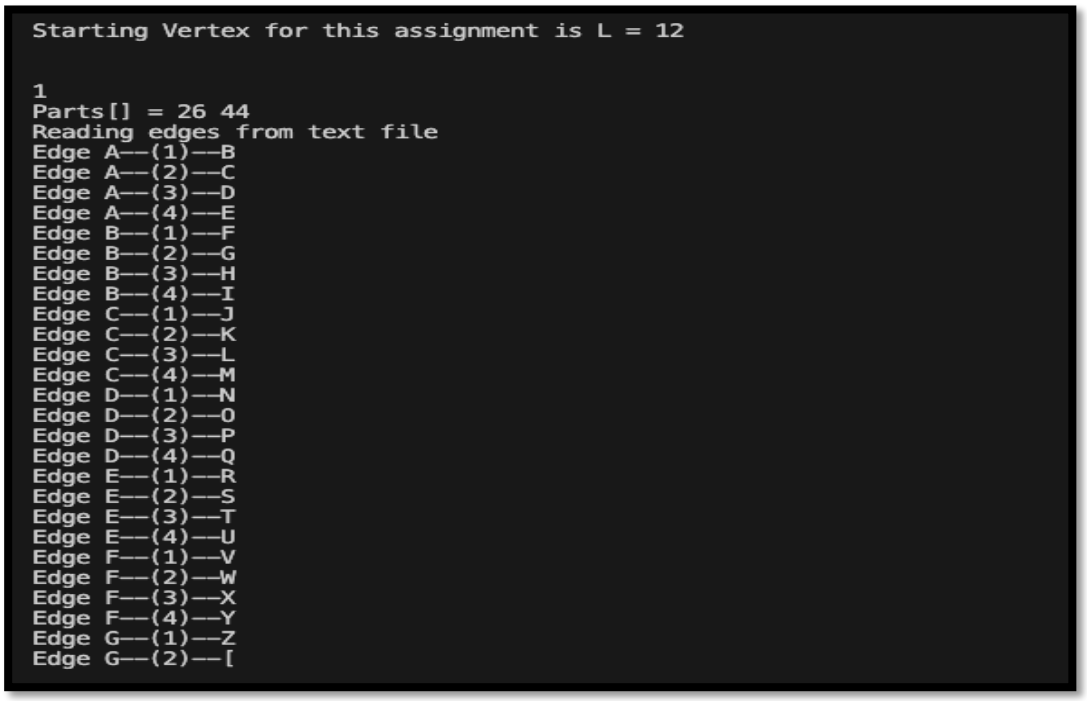



Image 20

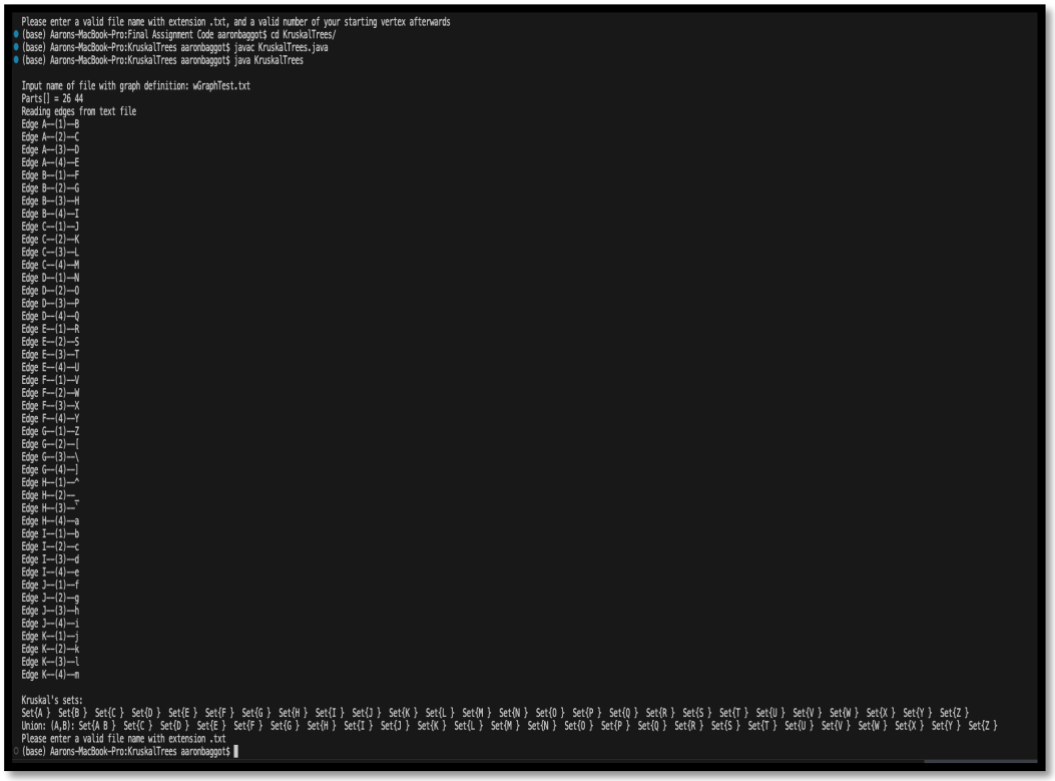

Please enter a valid file name with extension .txt, and a valid number of your starting vertex afterwards
(base) Aarons-MacBook-Pro:Final Assignment Code aaronbaggot$ cd KruskalTrees/
(base) Aarons-MacBook-Pro:KruskalTrees aaronbaggot$ javac KruskalTrees.java
(base) Aarons-MacBook-Pro:KruskalTrees aaronbaggot$ java KruskalTrees
Input name of file with graph definition: wGraphTest.txt
Parts[] = 26 44
Reading edges from text file
Edge A—(1)—B
Edge A—(2)—C
Edge A—(3)—D
Edge A—(4)—E
Edge B—(1)—F
Edge B—(2)—G
Edge B—(3)—H
Edge B—(4)—I
Edge C—(1)—J
Edge C—(2)—K
Edge C—(3)—L
Edge C—(4)—M
Edge D—(1)—N
Edge D—(2)—O
Edge D—(3)—P
Edge D—(4)—Q
Edge E—(1)—R
Edge E—(2)—S
Edge E—(3)—T
Edge E—(4)—U
Edge F—(1)—V
Edge F—(2)—W
Edge F—(3)—X
Edge F—(4)—Y
Edge G—(1)—Z
Edge G—(2)—[
Edge G—(3)—\
Edge G—(4)—]
Edge H—(1)—^
Edge H—(2)—_
Edge H—(3)—`
Edge H—(4)—a
Edge I—(1)—b
Edge I—(2)—c
Edge I—(3)—d
Edge I—(4)—e
Edge J—(1)—f
Edge J—(2)—g
Edge J—(3)—h
Edge J—(4)—i
Edge K—(1)—j
Edge K—(2)—k
Edge K—(3)—l
Edge K—(4)—m
Kruskal's sets:
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{N } Set{O } Set{P } Set{Q } Set{R } Set{S } Set{T } Set{U } Set{V } Set{W } Set{X } Set{Y } Set{Z }
Union: (A,B): 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{N } Set{O } Set{P } Set{Q } Set{R } Set{S } Set{T } Set{U } Set{V } Set{W } Set{X } Set{Y } Set{Z }
Please enter a valid file name with extension .txt
(base) Aarons-MacBook-Pro:KruskalTrees aaronbaggot$

Image 21

# Learning Gained

Throughout this assignment, I had the opportunity to delve deeply into various graph traversal and minimum spanning tree algorithms, including Prim's, Dijkstra's, and Kruskal's algorithms. As someone who is a kinaesthetic learner working on this assignment provided an excellent opportunity to apply theoretical knowledge to practical implementations in a way that concretise the learning for me.

One of the most valuable aspects of this assignment was the hands-on coding experience. Implementing each algorithm with the help of skeleton code allowed me to gain a deeper understanding of their inner workings, including how data structures such as heaps and disjoint sets are utilized to optimize their efficiency. Through iterative testing and debugging, I was able to refine my implementations and gain insight into algorithmic complexities and edge cases.

Additionally, the detailed whiteboard descriptions provided by the lecturer during lectures were immensely helpful in visualizing the algorithms and understanding their step-by-step execution. These visual aids, combined with the practical coding exercises, reinforced key concepts, and facilitated a deeper comprehension of graph theory and algorithm design principles.

Moreover, the process of testing the programs was crucial in ensuring their correctness and efficiency. I conducted extensive testing using various input graph configurations, including different sizes and edge cases, to validate the algorithms behaviour and performance. This iterative testing approach allowed me to identify and rectify any issues or inefficiencies in my implementations, contributing to a more robust and reliable final solution.

Overall, this assignment has been a rewarding learning experience, providing valuable insights into graph traversal algorithms and their real-world applications. By combining theoretical knowledge with practical implementation and rigorous testing, I have developed a deeper understanding of algorithmic design and problem-solving techniques, which will undoubtedly benefit me in future academic.

# References & Bibliography

Cormen, Thomas H et al. (2009) ‘Graph Algorithms’, in Introduction To Algorithms. Third. Cambridge, Massachusetts: Mit Press, pp. 589–671.

Lawlor, R. (2024) ‘Graph Traversal, MST & SPT Algorithm’.