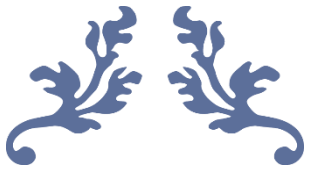


TU856-2
ALGORITHMS & DATA STRUCTURES

GRAPH TRAVERSAL, MST & SPT ALGORITHM ASSIGNMENT

COMPLETED BY: PAULINA CZARNOTA C21365726

DUE DATE: 28/04/2023

INTODUCTION

- ***KruskalTrees.java***

Explanation of Kruskal's Algorithm for Finding Minimum Spanning Tree in a Graph

In my code, I have implemented Kruskal's Algorithm, which is a popular algorithm utilized for discovering the Minimum Spanning Tree (MST) of a weighted, connected graph. The MST in this algorithm is a collection of edges of the graph that link all vertices while reducing the overall weight of the tree. Kruskal's Algorithm is one of the well-known algorithms used to solve the MST problem.

Kruskal's Algorithm

The algorithm works by starting with a graph containing all edges and repeatedly adding the smallest edge that does not create a cycle in the MST. The algorithm maintains a set of disjoint subsets of the vertices, initially one subset per vertex, and repeatedly merges these subsets by adding edges that connect vertices from different subsets.

Steps

Kruskal's algorithm can be summarized as follows:

1. Create a set S of all the edges in the graph.
2. Sort the edges in S by weight.
3. Initialize an empty set E of edges that will be part of the MST.
4. For each edge in S, starting from the smallest weight:
    a. If adding the edge to E does not create a cycle, add the edge to E.
    b. Otherwise, discard the edge.
5. When there are no more edges in S, the set E contains the edges of the MST.

Checking for Cycles

To check whether adding an edge creates a cycle, the algorithm uses the Union-Find data structure to keep track of which vertices are in the same subset.

Union-Find Data Structure

The Union-Find data structure is initialized with each vertex in its own subset. When an edge is added to the MST, the algorithm checks whether the vertices of the edge belong to the same subset. If they do, adding the edge would create a cycle, so the edge is discarded. If the vertices belong to different subsets, the subsets are merged by setting the parent of one subset to be the other subset's parent.

Implementation and Time Complexity

The algorithm then adds the edge to the MST. The time complexity of Kruskal's algorithm is O (m log m), where m is the number of edges in the graph. In the implementation shown in the code, the algorithm uses a heap data structure to efficiently find the edge with the smallest weight.

Java Implementation

Input Format

The Java implementation of Kruskal's algorithm takes as input a file containing the graph in edge list format, where each line contains three integers: the two vertices that form the edge, and the weight of the edge.

Classes

The program uses three classes: Edge, Heap, and UnionFindSets.

Edge

The Edge class represents an edge in the graph by storing its source vertex, destination vertex, and weight. It also includes methods for printing the edge and converting a vertex integer to a character for pretty printing.

Heap

The Heap class implements a binary heap to store the edges sorted by weight. The heap is initialized with all edges, and the edges are removed from the heap in order of increasing weight. The class includes methods for converting the heap to a proper heap, sifting down nodes in the heap, and removing the edge with the highest priority from the heap.

UnionFindSets

The UnionFindSets class implements the Union-Find data structure, which is used to keep track of a partition of a set of elements into disjoint subsets. This is used to check for cycles in the graph and to ensure that the minimum spanning tree does not contain any cycles. The class includes methods for finding the parent of a vertex, merging two subsets, and checking if two vertices belong to the same subset.

- ***PrimLists.java***

Explanation of Graph Traversal and Shortest Path Algorithms with DFS, BFS, Prim, and Dijkstra

In my code, I have utilized Graph traversal and Shortest Path Algorithms as crucial tools in graph theory to analyse the connections between nodes and edges in a graph. These algorithms have been helpful in identifying the best path between two nodes or traversing the entire graph in search of a specific node or property. Specifically, I have implemented Depth First Search (DFS) and Breadth First Search (BFS) as commonly used algorithms for graph traversal. Additionally, to find the shortest path in a weighted graph, I have utilized popular algorithms such as Dijkstra's algorithm and Prim's algorithm for minimum spanning trees. In my implementation, the graph is represented using adjacency lists, where each node in the adjacency list contains the adjacent vertex and the weight of the edge that connects it to the current vertex.

Graph Representation

The graph is represented using adjacency lists, where each node in the list represents an adjacent vertex and its weight. The graph is read from a text file, where the first line contains the number of vertices and edges in the graph, and subsequent lines represent each edge with its corresponding vertices and weight.

Implementation and Time Complexity

- GraphLists(String graphFile): This method reads in the graph data from the file and creates the adjacency list representation of the graph. The time complexity of this method depends on the size of the graph and the number of edges, and it is O(V+E), where V is the number of vertices and E is the number of edges.
- void display(): This method simply prints out the adjacency list of the graph. The time complexity of this method is O(V+E), where V is the number of vertices and E is the number of edges, because it needs to iterate through all the nodes in the adjacency list.
- void DF(int s): This method performs a depth-first traversal of the graph starting from the vertex s. The time complexity of this method 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.
- void MST_Prim(): This method finds the minimum spanning tree of the graph using Prim's algorithm. The time complexity of this method 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 findMinVertex(int[] wt, boolean[] visited): This method finds the vertex with the minimum weight that has not yet been visited. The time complexity of this method is O(V), where V is the number of vertices, because it needs to iterate through all the vertices in the graph.

Depth First Search (DFS)

DFS is a graph traversal algorithm that searches as deep as possible along each branch before backtracking. In other words, the algorithm visits every vertex in a graph by exploring each branch as far as possible before backtracking. DFS starts at a selected vertex, and when a vertex is visited, it is marked as such. DFS continues until all the vertices are marked. The DFS algorithm is implemented using a recursive approach. When applied to a weighted graph, DFS can also be used to find the shortest path between two vertices. However, it is not efficient and may not always find the optimal solution.

Breadth First Search (BFS)

BFS is a graph traversal algorithm that visits all the vertices in a graph by visiting each vertex's neighbours before moving on to the next level of vertices. In other words, BFS visits all the vertices at a particular depth or distance before moving to the next level. BFS starts at a selected vertex and visits all the vertices at the current level before moving to the next level. BFS is implemented using a queue data structure. BFS is commonly used to find the shortest path between two vertices in an unweighted graph.

Prim's Algorithm

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree of a weighted undirected graph. The algorithm starts by selecting any vertex as the root and considers all the vertices connected to the root. It selects the edge with the minimum weight, and the corresponding vertex is added to the minimum spanning tree. The algorithm repeats this process until all vertices are included in the minimum spanning tree.

Dijkstra's Algorithm

Dijkstra's algorithm is a popular shortest-path algorithm that can find the shortest path between two vertices in a weighted graph. The algorithm starts by assigning a tentative distance value to every vertex in the graph. It then selects the vertex with the smallest tentative distance and considers all the vertices connected to it. The algorithm calculates the tentative distance from the source vertex to each connected vertex and updates the tentative distance value if the calculated distance is less than the current tentative distance. The algorithm repeats this process until all the vertices in the graph have been processed. Dijkstra's algorithm uses a priority queue data structure to select the vertex with the smallest tentative distance.

ADJACENCY LISTS DIAGRAMS

Adjacency List:

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

Graph Representation of 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]

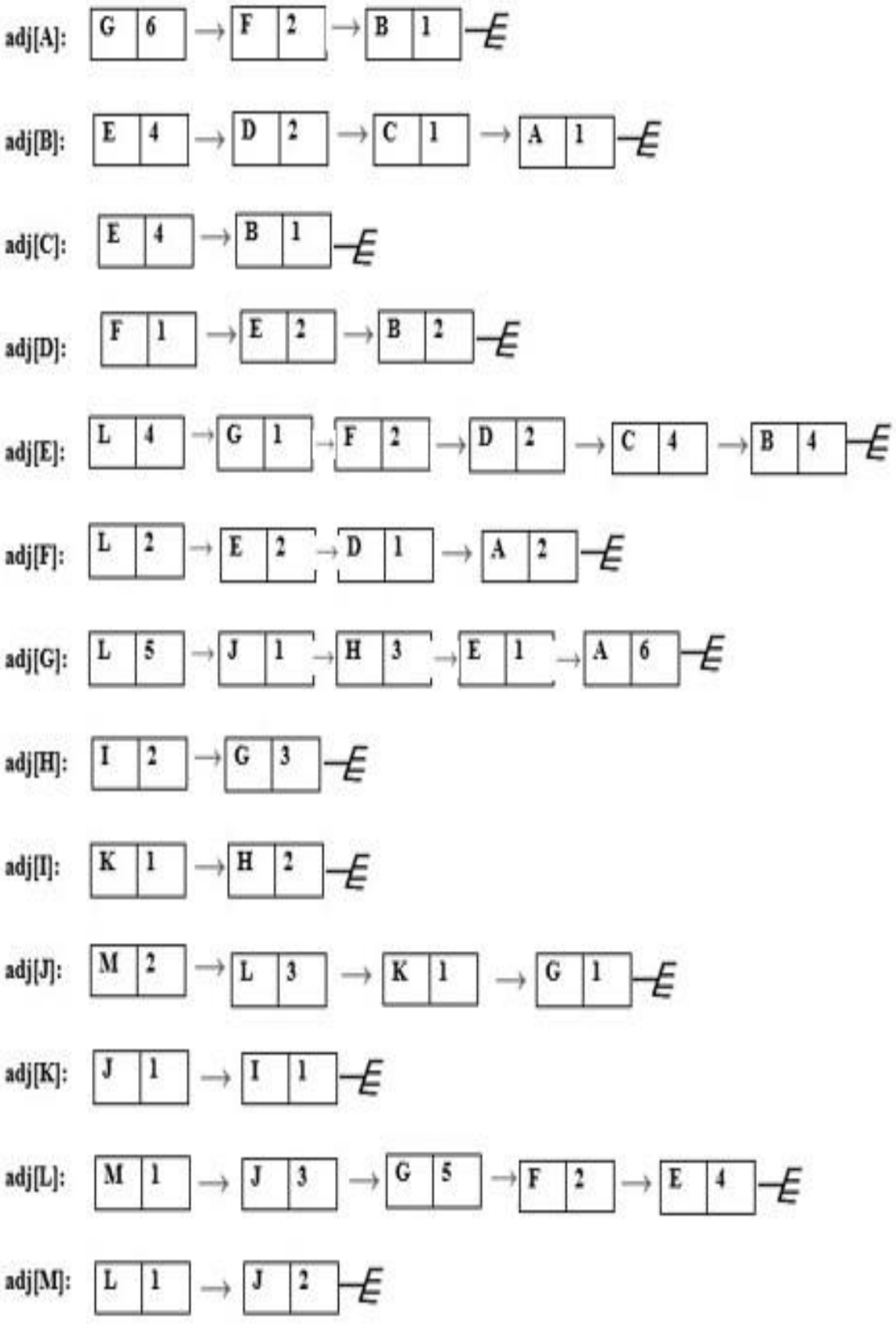
9 : [11, 1] → [8, 2]

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

CONSTRUCTION OF THE MST USING PRIM'S ALGORITHM

Initial State Starting From Vertex L:

Step 0: Start With L Vertex

Heap:

E4

F2

G5

J3

M1

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Dist		**0**	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Explanation:

At step 0, the algorithm starts with an initial state where all vertices except the starting vertex have a distance of infinity. In this case, the starting vertex is L. Then, the algorithm proceeds in steps to add vertices to the MST. At step 0, the algorithm starts with the L vertex and creates a heap of all its adjacent edges with their weights. The heap includes the edges E4, F2, G5, J3, and M1, where the number next to each edge represents its weight.

Step 1: Next is M

Heap:

E4

F2

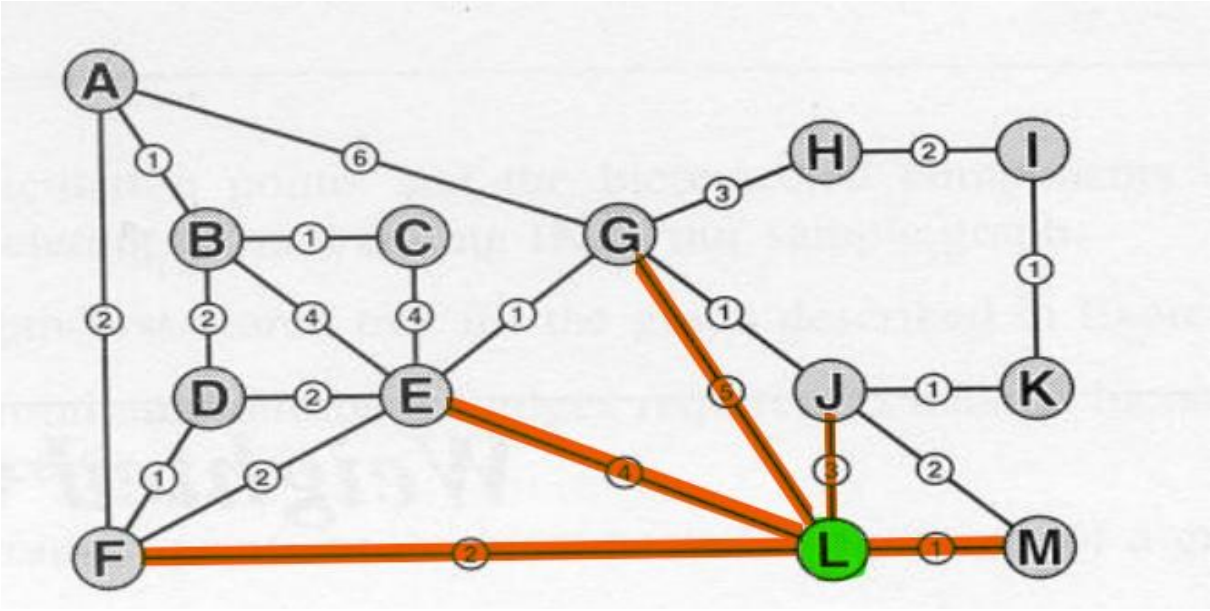
G5

J2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**0**	**0**	**0**	**0**	**L**	**L**	**L**	**0**	**0**	**L**	**0**	**@**	**L**
Dist[]		**0**	∞	∞	∞	**4**	**2**	**5**	∞	∞	**3**	∞	**0**	**1**

Graph Representation:



Explanation:

At step 1, the algorithm has added the vertex M to the MST. The heap now includes the edges E4, F2, G5, J2, where the number next to each edge represents its weight. The algorithm will choose the edge with the smallest weight, which is J2, and add the vertex J to the MST.

Step 2: Next is F

Heap:

A2

D1

E2

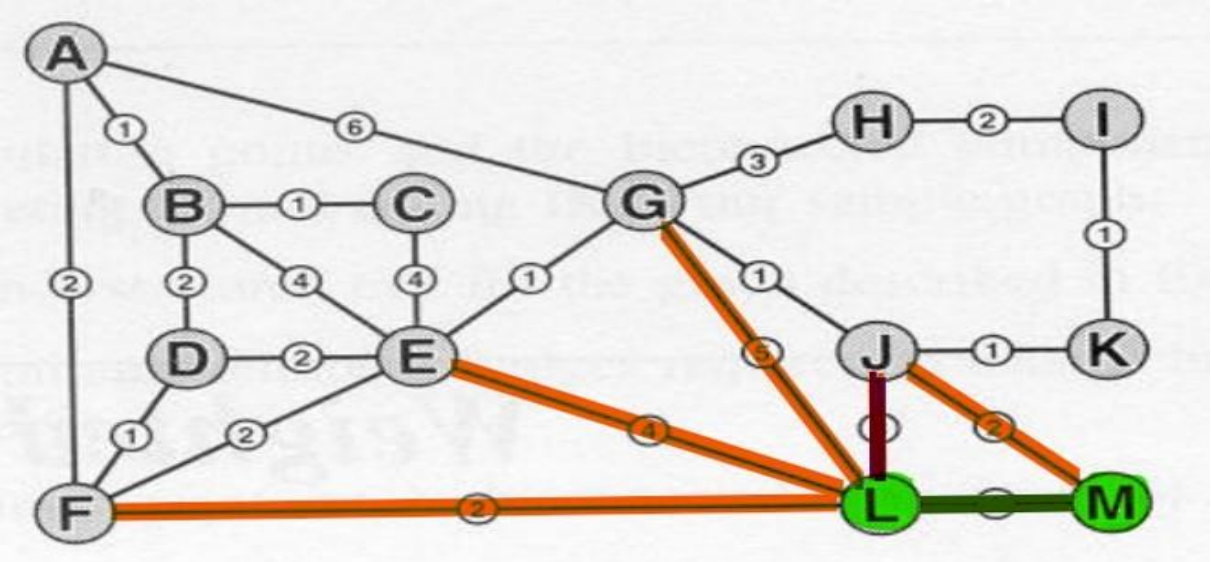
G5

J2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**0**	**0**	**0**	**0**	**L**	**L**	**L**	**0**	**0**	~~**L**~~ **M**	**0**	**@**	**L**
Dist[]		**0**	∞	∞	∞	**4**	**2**	**5**	∞	∞	~~**3**~~ **2**	∞	**0**	**1**

Graph Representation:



Explanation:

At step 2, the algorithm has added the vertex F to the MST. The heap now includes the edges A2, D1, E2, G5, J2, where the number next to each edge represents its weight. The algorithm will choose the edge with the smallest weight, which is D1, and add the vertex D to the MST. The heap is updated, and the algorithm continues to add vertices to the MST until all vertices are included.

Step 3: Next is D

Heap:

A2

B2

E2

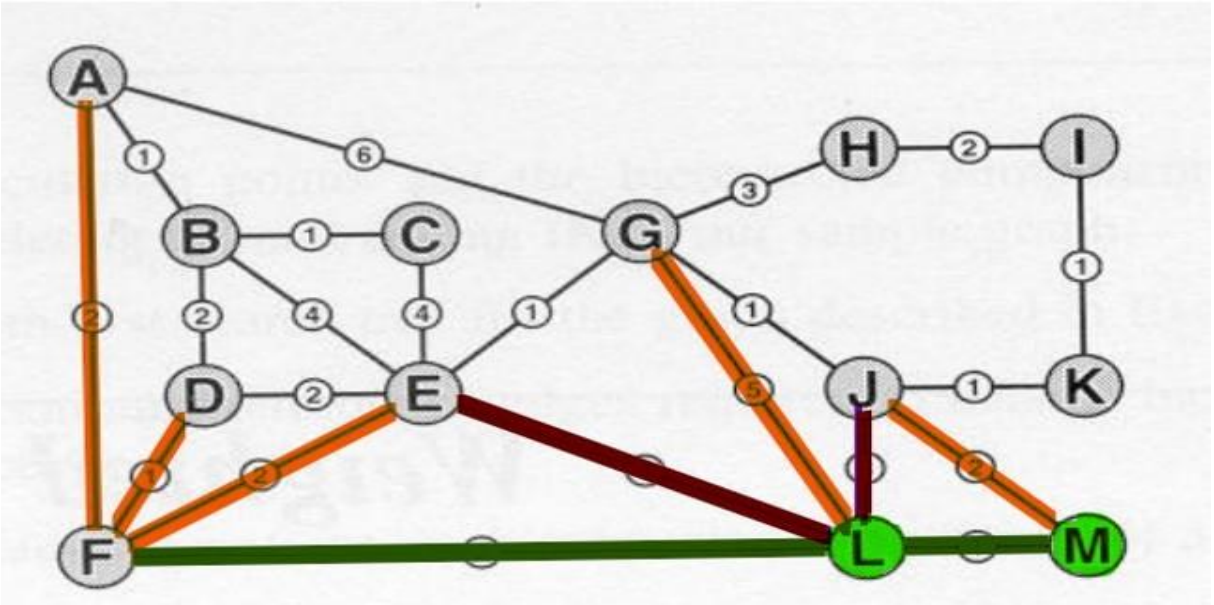
G5

J2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	**0**	**0**	**F**	~~**L**~~ **F**	**L**	**L**	**0**	**0**	~~**L**~~ **M**	**0**	**@**	**L**
Dist[]		**2**	∞	∞	**1**	~~**4**~~ **2**	**2**	**5**	∞	∞	~~**3**~~ **2**	∞	**0**	**1**

Graph Representation:



Explanation:

At step 3, the algorithm adds the vertex D to the MST. The heap now includes the edges A2, B2, E2, G5, J2, where the number next to each edge represents its weight. The algorithm will choose the edge with the smallest weight, which is D1, and add the vertex D to the MST. The heap is updated, and the algorithm continues to add vertices to the MST until all vertices are included.

The updated Parent and Distance Arrays show the state of the algorithm after adding F and D to the MST. The heap at step 3 includes the edges A2, B2, E2, G5, J2. The algorithm chooses the edge with the smallest weight, which is D1, and adds the vertex D to the MST. The heap is updated, and the algorithm continues to add vertices to the MST until all vertices are included.

Step 4: Next is A

Heap:

B1

E2

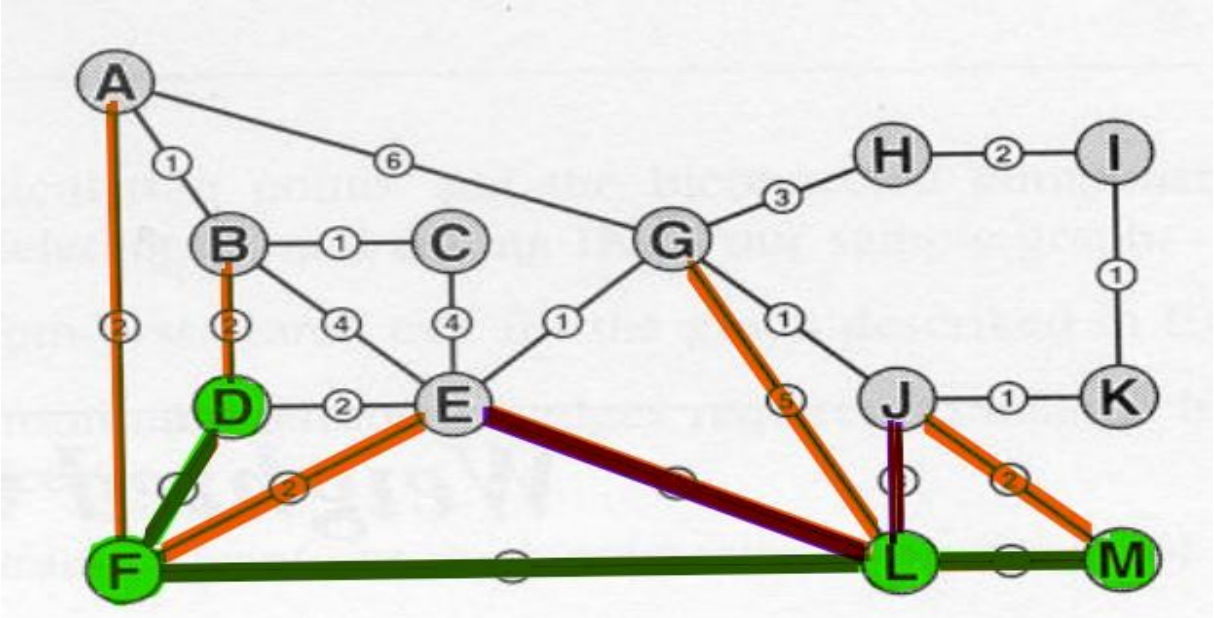
G5

J2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	**D**	**0**	**F**	~~**L**~~ **F**	**L**	**L**	**0**	**0**	~~**L**~~ **M**	**0**	**@**	**L**
Dist[]		**2**	**2**	∞	**1**	~~**4**~~ **2**	**2**	**5**	∞	∞	~~**3**~~ **2**	∞	**0**	**1**

Graph Representation:



Explanation:

At step 4, the algorithm for constructing the Minimum Spanning Tree (MST) using Prim's algorithm selects the next vertex to add to the MST. In this case, the algorithm has already selected vertex L as the starting point and has updated the parent and distance arrays accordingly. The next vertex to select is determined by finding the minimum distance value in the heap.

In this specific example, the heap contains three vertices: B with a distance of 1, E with a distance of 2, and G with a distance of 5. The algorithm selects the vertex with the minimum distance value, which is B with a distance of 1. Therefore, the algorithm adds B to the MST and updates the parent and distance arrays accordingly. The heap is then updated to include the adjacent vertices of B that are not already in the MST, which are A and C.

Step 5: Next is B

Heap:

C1

E2

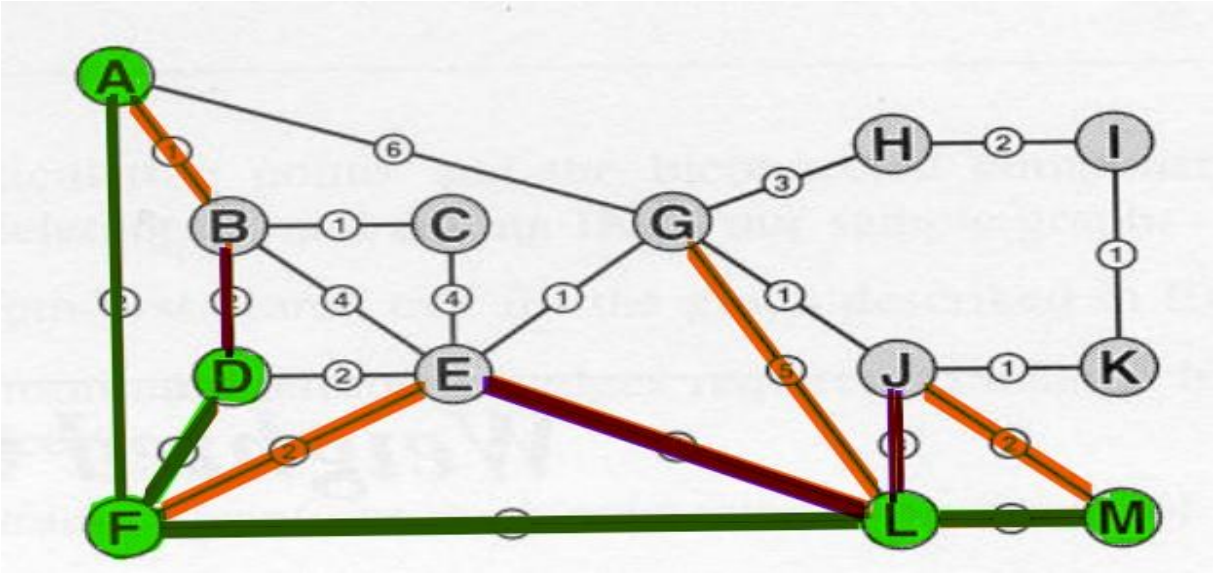
G5

J2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**0**	**F**	~~**L**~~ **F**	**L**	**L**	**0**	**0**	~~**L**~~ **M**	**0**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	∞	**1**	~~**4**~~ **2**	**2**	**5**	∞	∞	~~**3**~~ **2**	∞	∞	**1**

Graph Representation:



Explanation:

At step 5, the algorithm for constructing the Minimum Spanning Tree (MST) using Prim's algorithm selects the next vertex B to add to the MST. The algorithm maintains a priority queue (heap) of vertices that are adjacent to the MST and selects the vertex with the smallest edge weight. In this case, the vertex B has the smallest edge weight among the vertices adjacent to the MST.

After selecting the vertex B, the algorithm updates the parent and distance arrays to reflect the new vertex added to the MST. The parent of vertex B is set to L, which is the vertex that added it to the MST, and the distance from B to its parent is 2. The distances to the other adjacent vertices are updated if they are smaller than the previous distance. The heap is also updated to reflect the new distances to the adjacent vertices.

The algorithm continues in this manner, selecting the next smallest edge and adding the corresponding vertex to the MST until all vertices are included in the tree. The final result is the MST, which is a subset of the original graph that connects all vertices with the minimum total edge weight.

Step 6: Next is C

Heap:

E2

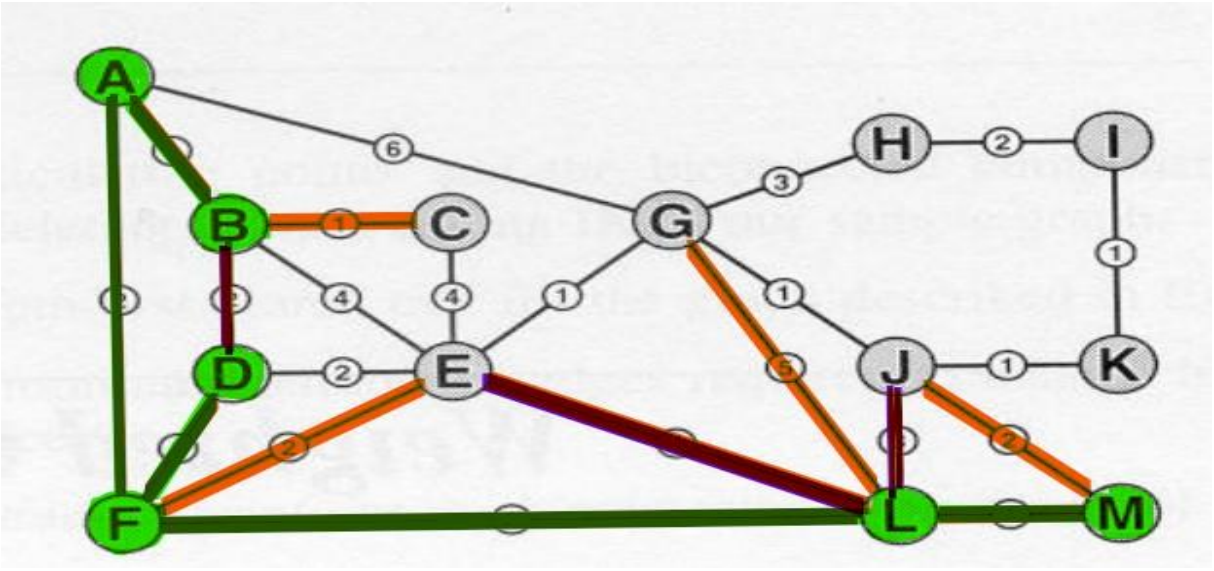
G5

J2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ **F**	**L**	**L**	**0**	**0**	~~**L**~~ **M**	**0**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ **2**	**2**	**5**	∞	∞	~~**3**~~ **2**	∞	∞	**1**

Graph Representation:



Explanation:

At step 6, the algorithm considers the vertex C as the next potential candidate for the minimum spanning tree (MST). The algorithm adds the adjacent edges of C to the priority queue (heap) and updates the parent and distance arrays accordingly. The heap at this step contains three entries: E2, G5, and J2. The algorithm selects the edge with the minimum weight, which is the edge between C and E with a weight of 1, and adds it to the MST. The parent and distance arrays are updated to reflect the addition of the new edge to the tree.

Step 7: Next is J

Heap:

E2

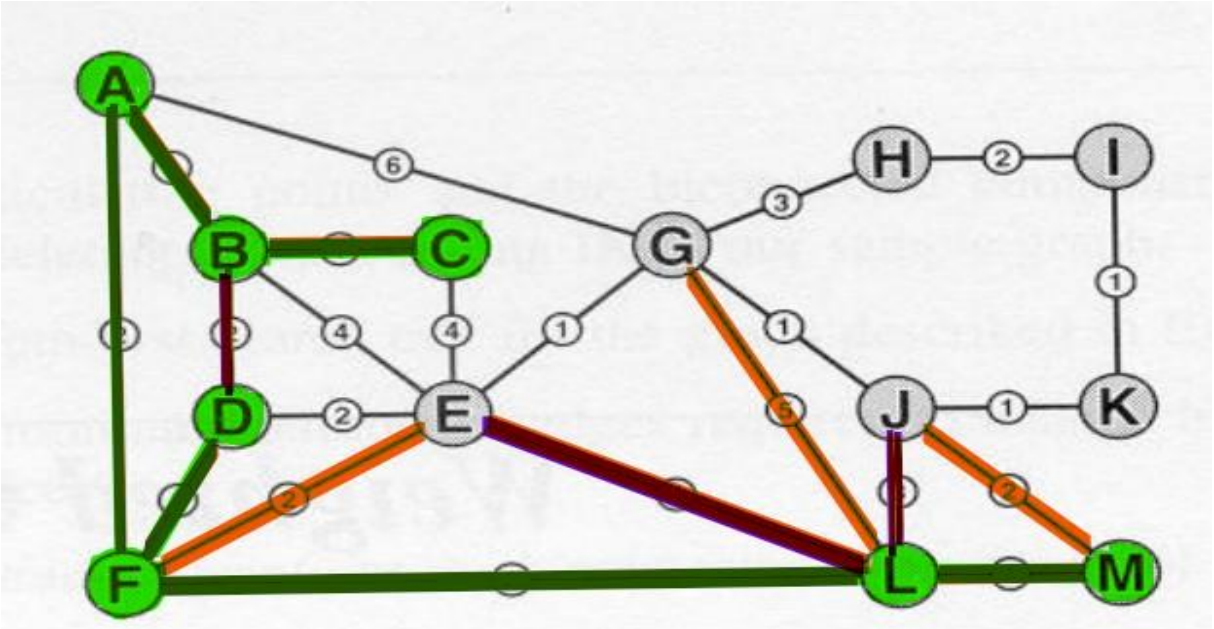
G1

K1

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ **F**	**L**	**L**	**0**	**0**	~~**L**~~ **M**	**0**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ **2**	**2**	**5**	∞	∞	~~**3**~~ **2**	∞	∞	**1**

Graph Representation:



Explanation:

At step 7, the algorithm considers the vertex J as the next potential candidate for the minimum spanning tree (MST). The algorithm adds the adjacent edges of J to the priority queue (heap) and updates the parent and distance arrays accordingly. The heap at this step contains three entries: E2, G1, and K1. The algorithm selects the edge with the minimum weight, which is the edge between J and G with a weight of 1, and adds it to the MST. The parent and distance arrays are updated to reflect the addition of the new edge to the tree.

Step 8: Next is K

Heap:

E2

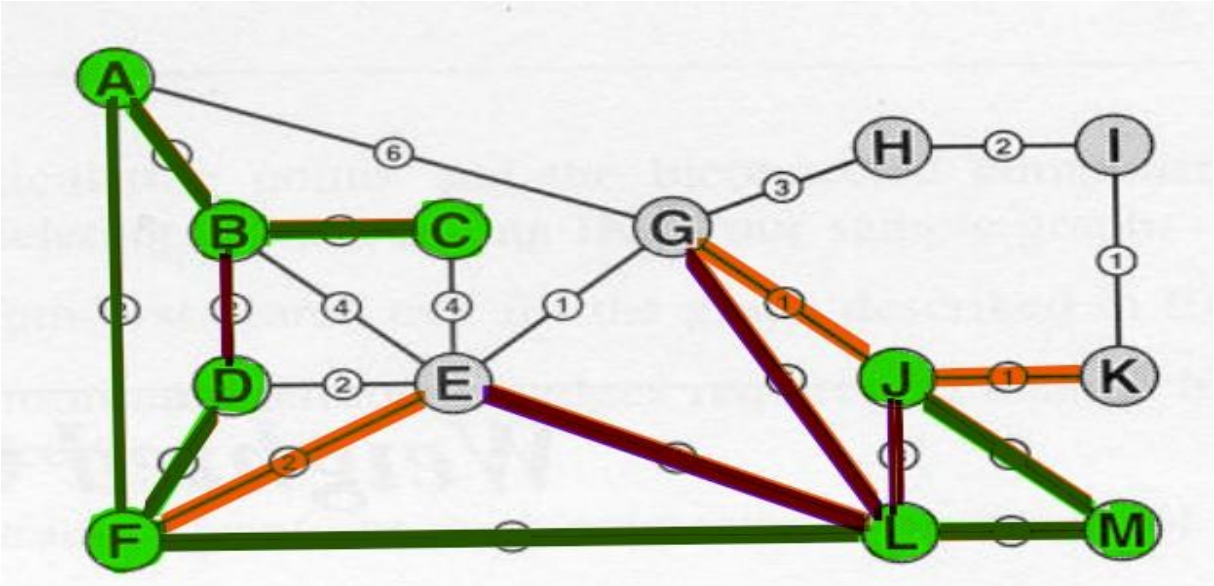
G1

I1

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ **F**	**L**	~~**L**~~ **J**	**0**	**0**	~~**L**~~ **M**	**J**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ **2**	**2**	~~**5**~~ **1**	∞	∞	~~**3**~~ **2**	**1**	∞	**1**

Graph Representation:



Explanation:

At step 8, the algorithm considers the vertex K as the next potential candidate for the minimum spanning tree (MST). The algorithm adds the adjacent edges of K to the priority queue (heap) and updates the parent and distance arrays accordingly. The heap at this step contains three entries: E2, G1, and I1. The algorithm selects the edge with the minimum weight, which is the edge between K and I with a weight of 1, and adds it to the MST. The parent and distance arrays are updated to reflect the addition of the new edge to the tree.

Step 9: Next is G

Heap:

E1

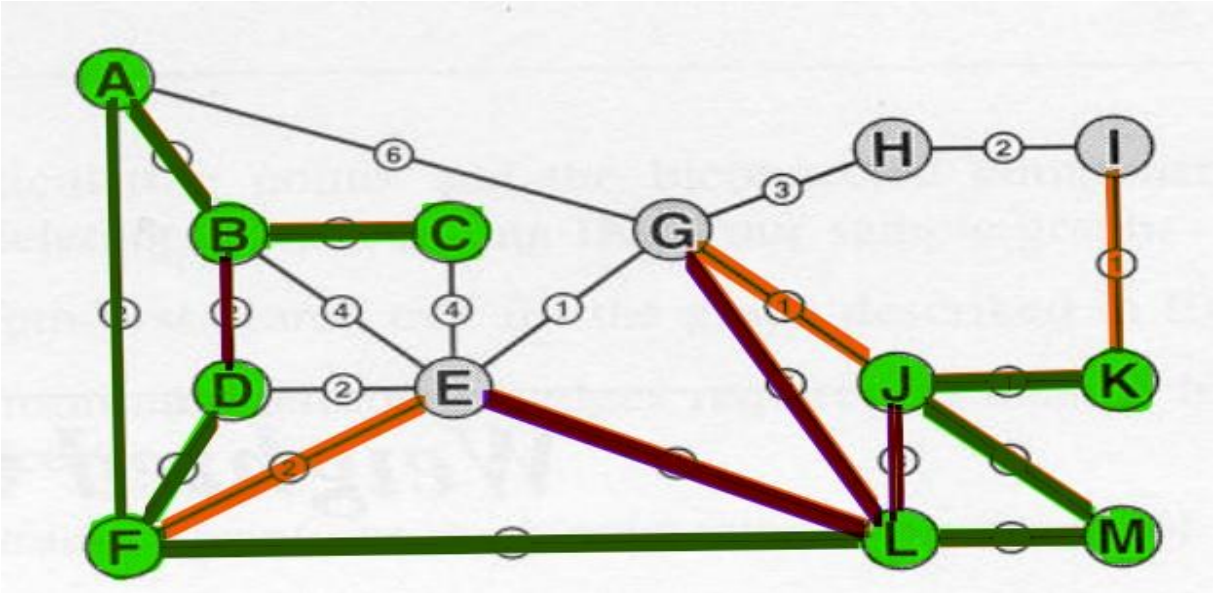
I1

H3

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ **F**	**L**	~~**L**~~ **J**	**0**	**K**	~~**L**~~ **M**	**J**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ **2**	**2**	~~**5**~~ **1**	∞	**1**	~~**3**~~ **2**	**1**	∞	**1**

Graph Representation:



Explanation:

At step 9, the algorithm considers the vertex G as the next potential candidate for the minimum spanning tree (MST). The algorithm adds the adjacent edges of G to the priority queue (heap) and updates the parent and distance arrays accordingly. The heap at this step contains three entries: E1, I1, and H3. The algorithm selects the edge with the minimum weight, which is the edge between G and E with a weight of 1, and adds it to the MST. The parent and distance arrays are updated to reflect the addition of the new edge to the tree.

Step 10: Next is I

Heap:

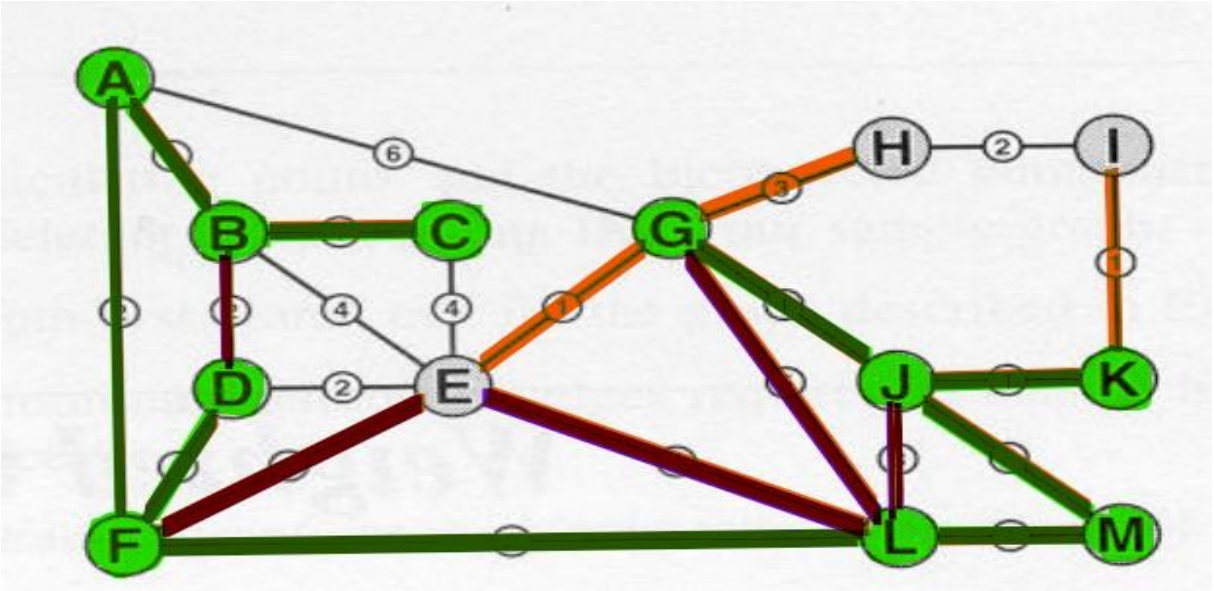
E1

H2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ ~~**F**~~ **G**	**L**	~~**L**~~ **J**	**G**	**K**	~~**L**~~ **M**	**J**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ ~~**2**~~ **1**	**2**	~~**5**~~ **1**	**3**	**1**	~~**3**~~ **2**	**1**	∞	**1**

Graph Representation:



Explanation:

At step 10, the algorithm considers the vertex I as the next potential candidate for the minimum spanning tree (MST). The algorithm adds the adjacent edges of I to the priority queue (heap) and updates the parent and distance arrays accordingly. The heap at this step contains two entries: E1 and H2. The algorithm selects the edge with the minimum weight, which is the edge between I and H with a weight of 2, and adds it to the MST. The parent and distance arrays are updated to reflect the addition of the new edge to the tree.

Step 11: Next is E

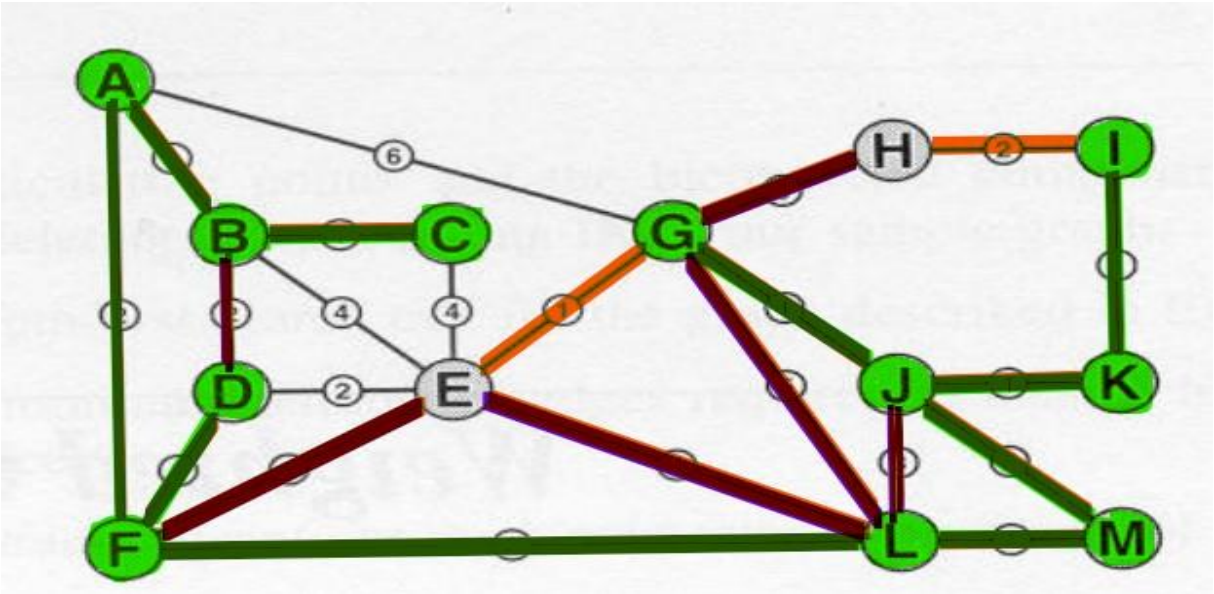
Heap:

H2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ ~~**F**~~ **G**	**L**	~~**L**~~ **J**	~~**G**~~ **I**	**K**	~~**L**~~ **M**	**J**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ ~~**2**~~ **1**	**2**	~~**5**~~ **1**	~~**3**~~ **2**	**1**	~~**3**~~ **2**	**1**	∞	**1**

Graph Representation:



Explanation:

At step 11, the algorithm selects vertex E with distance 1 from vertex H, the vertex currently in the MST. The parent of E is set to H, and its distance is updated to 1 in the distance array. Since there are no other vertices left to explore, the algorithm terminates.

Step 12: Next is H

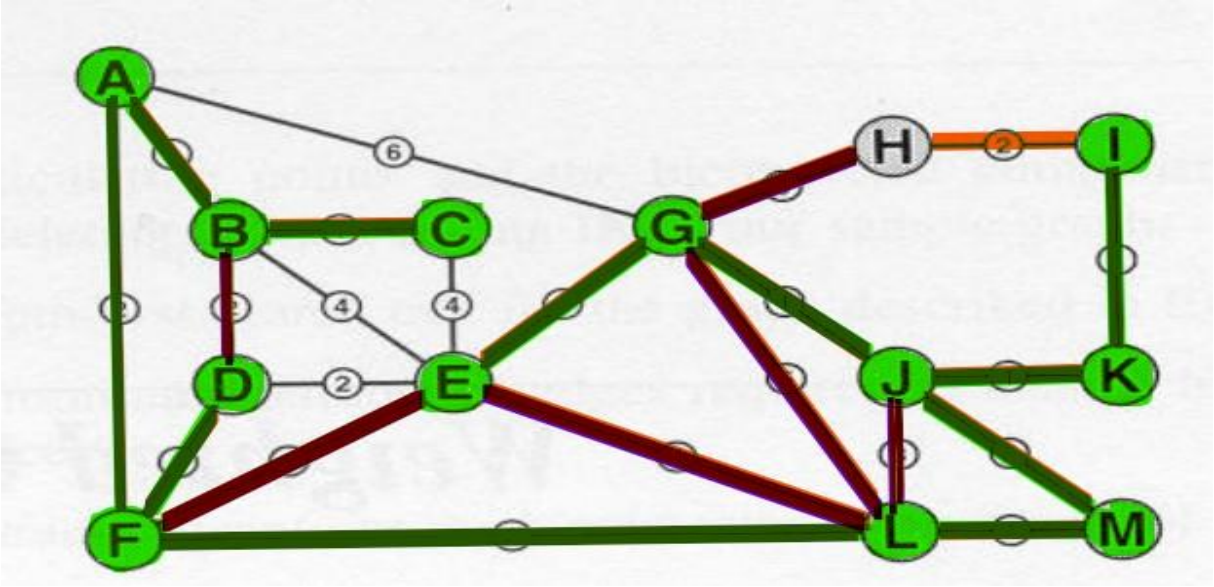
Heap:

Empty

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ ~~**F**~~ **G**	**L**	~~**L**~~ **J**	~~**G**~~ **I**	**K**	~~**L**~~ **M**	**J**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ ~~**2**~~ **1**	**2**	~~**5**~~ **1**	~~**3**~~ **2**	**1**	~~**3**~~ **2**	**1**	∞	**1**

Graph Representation:



Explanation:

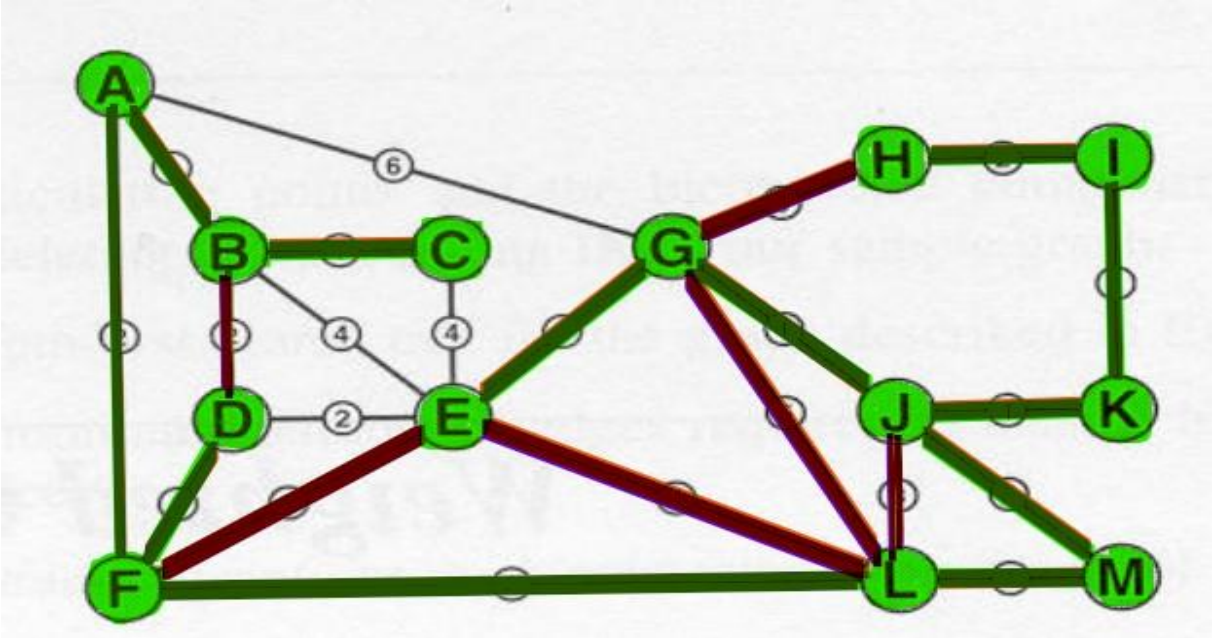
At step 12, the algorithm has visited all the vertices in the graph and the heap is empty. Therefore, the algorithm terminates and the resulting Minimum Spanning Tree (MST) is formed. The parent and distance arrays contain the information about the MST. The MST starts from vertex L and its parent array represents the path to each vertex in the MST from

the starting vertex L. The distance array contains the distances of each vertex from the starting vertex L in the MST.

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	~~**D**~~ **A**	**B**	**F**	~~**L**~~ ~~**F**~~ **G**	**L**	~~**L**~~ **J**	~~**G**~~ **I**	**K**	~~**L**~~ **M**	**J**	**0**	**L**
Dist[]		**2**	~~**2**~~ **1**	**1**	**1**	~~**4**~~ ~~**2**~~ **1**	**2**	~~**5**~~ **1**	~~**3**~~ **2**	**1**	~~**3**~~ **2**	**1**	∞	**1**

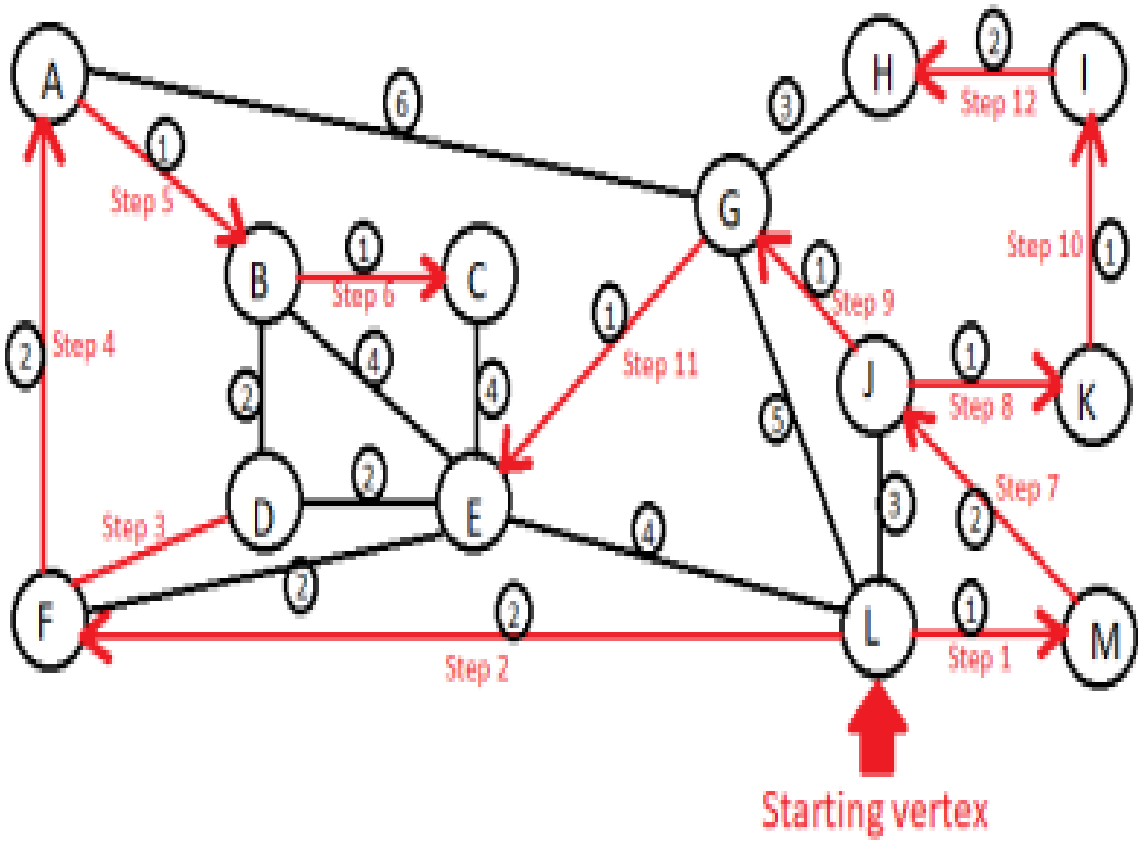
Graph Representation:



Prim's MST Final Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent[]		**F**	**A**	**B**	**F**	**G**	**L**	**J**	**I**	**K**	**M**	**J**	**0**	**L**
Dist[]		**2**	**1**	**1**	**1**	**1**	**2**	**1**	**2**	**1**	**2**	**1**	∞	**1**

Diagram Showing Prim's MST Superimposed On The Graph



Explanation:

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a given graph. An MST is a subset of edges that connects all vertices of the graph and has the minimum possible total edge weight.

The construction of the MST using Prim's algorithm begins by selecting an arbitrary vertex as the starting vertex. In this case, the initial state starts from vertex L. The algorithm maintains two arrays: the parent array and the distance array.

The parent array keeps track of the parent vertex of each vertex in the MST. The parent of the starting vertex is set to itself. The distance array keeps track of the minimum distance from each vertex to the MST. The distance of the starting vertex is set to 0, and the distance of all other vertices is set to infinity.

The algorithm then repeatedly selects the vertex with the minimum distance that is not yet in the MST and adds it to the MST. For each selected vertex, the algorithm updates the parent and distance arrays for its adjacent vertices that are not yet in the MST.

In this case, after running Prim's algorithm, the final parent and distance arrays are given. The MST can be constructed by using the information in the parent array. Starting from any vertex, we can follow the parent pointers to construct the MST. For example, starting from vertex A, we follow the parent pointers to vertices F, B, D, C, E, and finally L. This gives us the edges {AF, FB, BD, DC, CE, EL} with a total weight of 11.

CONSTRUCTION OF THE SPT USING DIJKSTRA'S ALGORITHM

Initial State Starting From Vertex L:

Step 0: Start With L Vertex

Heap:

L0

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Dist		**0**	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Explanation:

At step 0, we start with vertex L and 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. We also initialize the Parent and Dist arrays 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.

Step 1: Extract the vertex with the minimum distance from the heap, which is vertex L with a distance of 0

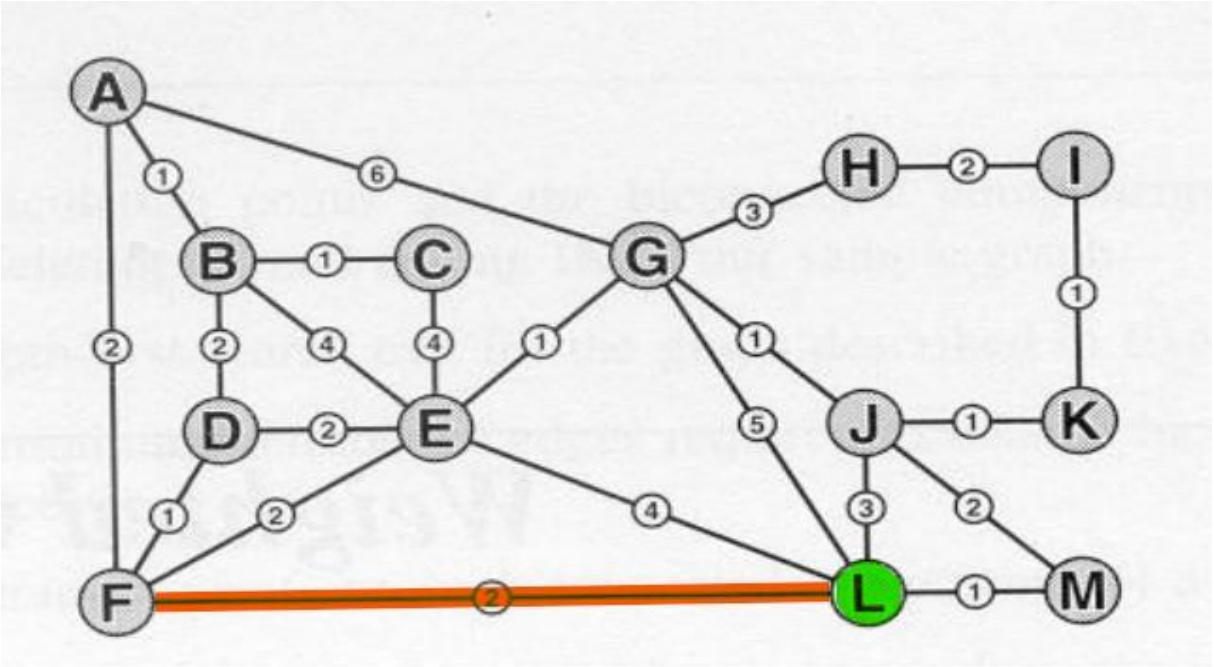
Heap:

F2

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		**0**	**0**	**0**	**0**	**0**	~~**0**~~ **L**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Dist		~~**0**~~ **2**	∞	∞	∞	∞	~~∞~~ **0**	∞	∞	∞	∞	∞	~~∞~~ **0**	∞

Graph Representation:



Explanation:

At step 1, vertex L is extracted from the heap as it has the minimum distance, which is 0. This means that the shortest path from L to L has been found, and vertex L is marked as visited. We now add the neighbours of L to the heap and update their distances in the Dist array.

Step 2: Extract the vertex with the minimum distance from the heap, which is vertex F with a distance of 2

Heap:

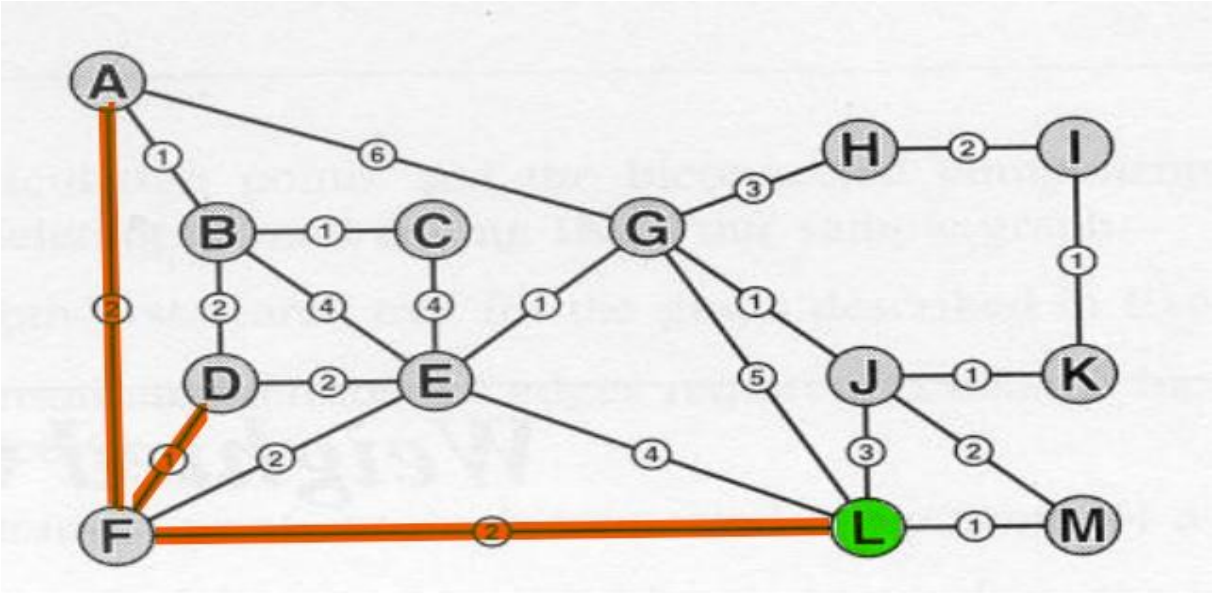
A2

D1

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		**0**	~~**0**~~ **F**	**0**	~~**0**~~ **D**	**0**	~~**0**~~ **L**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Dist		~~**0**~~ **2**	~~∞~~ **2**	∞	~~∞~~ **1**	∞	~~∞~~ **0**	∞	∞	∞	∞	∞	~~∞~~ **0**	∞

Graph Representation:



Explanation:

At step 2, vertex F is extracted from the heap as it has the minimum distance, which is 2. This means that the shortest path from L to F has been found, and vertex F is marked as visited. We now add the neighbours of F to the heap and update their distances in the Dist array.

Step 3: Extract the vertex with the minimum distance from the heap, which is vertex D with a distance of 2

Heap:

B2

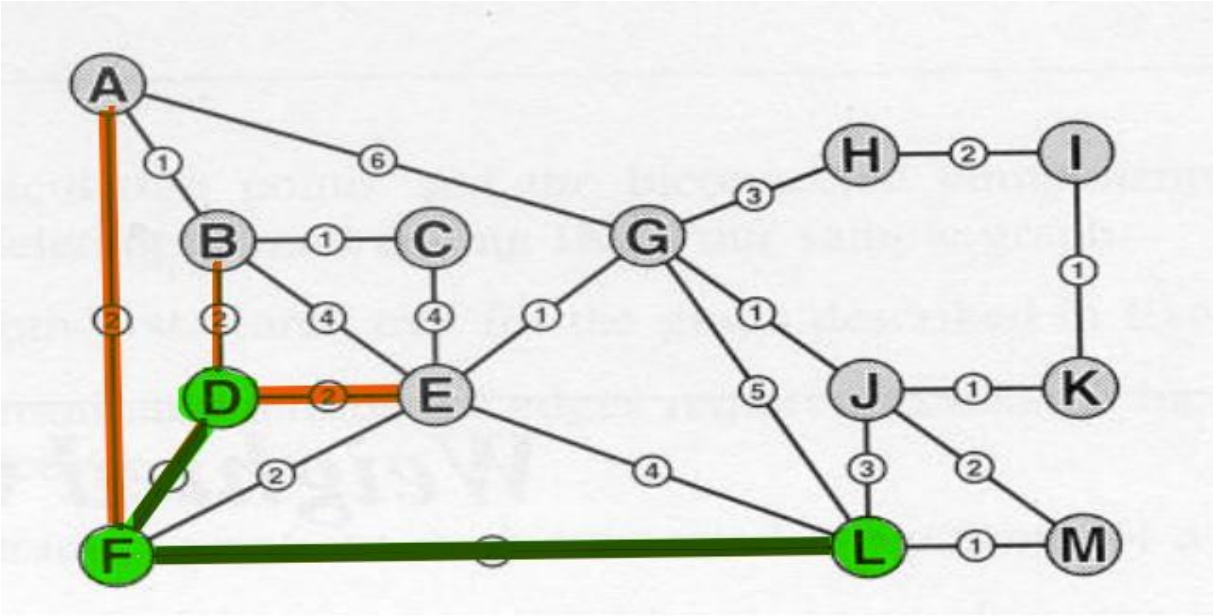
C4

E4

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		~~**0**~~ ~~**F**~~ **0**	~~**0**~~ ~~**A**~~ **F**	~~**0**~~ **B**	~~**0**~~ **D**	**0**	~~**0**~~ **L**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Dist		~~**0**~~ **2**	~~∞~~ ~~**1**~~ **2**	~~∞~~ **2**	~~∞~~ **2**	~~∞~~ **4**	~~∞~~ ~~**2**~~ **0**	∞	∞	∞	∞	∞	~~∞~~ **0**	∞

Graph Representation:



Explanation:

At step 3, vertex D is extracted from the heap as it has the minimum distance, which is 2. This means that the shortest path from L to D has been found, and vertex D is marked as visited. We now add the neighbours of D to the heap and update their distances in the Dist array.

Step 4: Extract the vertex with the minimum distance from the heap, which is vertex E with a distance of 2

Heap:

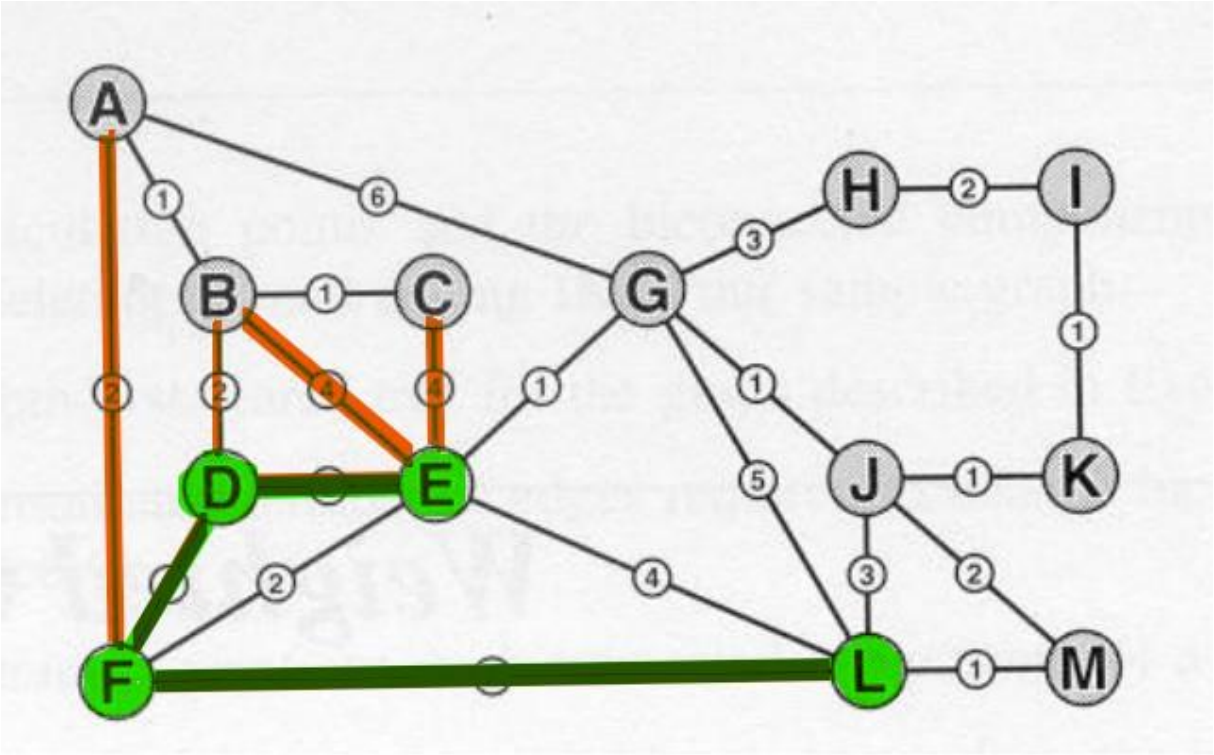
B2

C4

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		~~**0**~~ ~~**F**~~ **0**	~~**0**~~ ~~**A**~~ **F**	~~**0**~~ **B**	~~**0**~~ **D**	**0**	~~**0**~~ **L**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Dist		~~**0**~~ **2**	~~∞~~ ~~**1**~~ **2**	~~∞~~ **2**	~~∞~~ **2**	~~∞~~ **4**	~~∞~~ ~~**2**~~ **0**	∞	∞	∞	∞	∞	~~∞~~ **0**	∞

Graph Representation:



Explanation:

At step 4, vertex E is extracted from the heap as it has the minimum distance, which is 4. This means that the shortest path from L to E has been found, and vertex E is marked as visited. We now add the neighbours of E to the heap and update their distances in the Dist array.

Step 5: Extract the vertex with the minimum distance from the heap, which is vertex B with a distance of 4

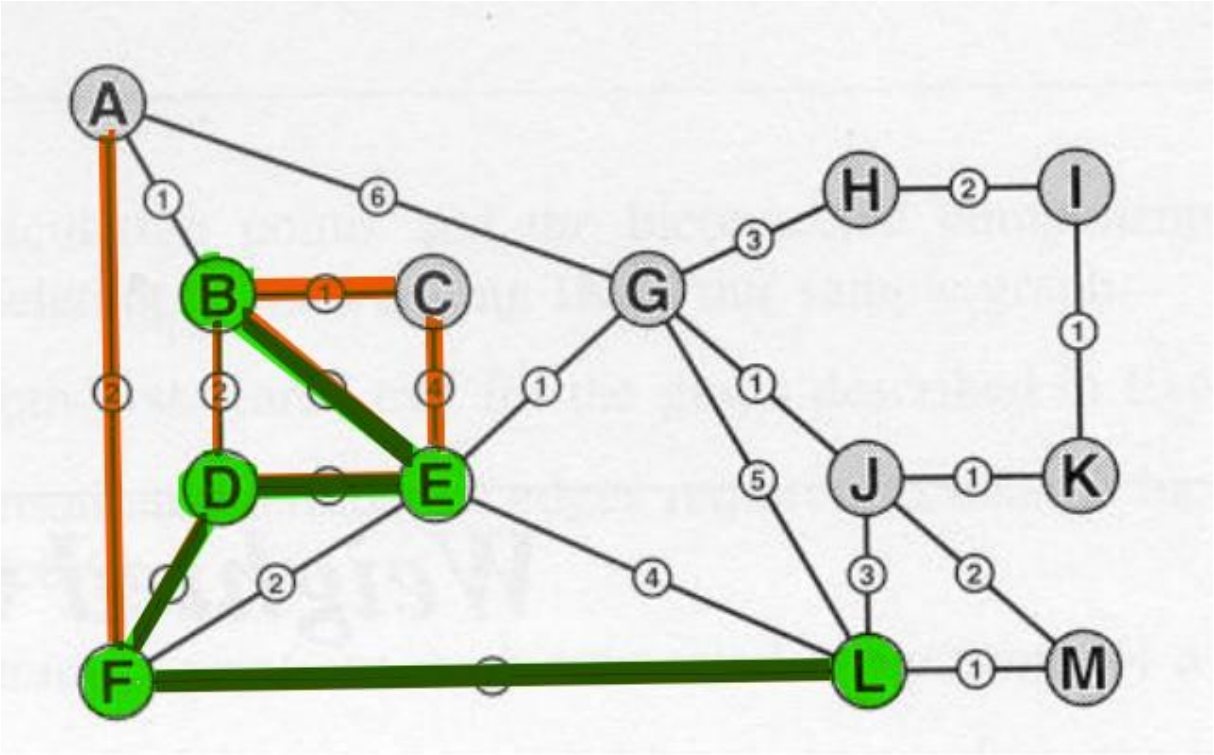
Heap:

C4

Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		~~**0**~~ ~~**F**~~ **0**	~~**0**~~ ~~**A**~~ **F**	~~**0**~~ **B**	~~**0**~~ **D**	**0**	~~**0**~~ **L**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Dist		~~**0**~~ **2**	~~∞~~ ~~**1**~~ **2**	~~∞~~ **2**	~~∞~~ **2**	~~∞~~ **4**	~~∞~~ ~~**2**~~ **0**	∞	∞	∞	∞	∞	~~∞~~ **0**	∞

Graph Representation:



Explanation:

At step 5, vertex B is extracted from the heap as it has the minimum distance, which is 4. This means that the shortest path from L to B has been found, and vertex B is marked as visited. We now add the neighbours of B to the heap and update their distances in the Dist array.

Step 6: Extract the vertex with the minimum distance from the heap, which is vertex C with a distance of 4

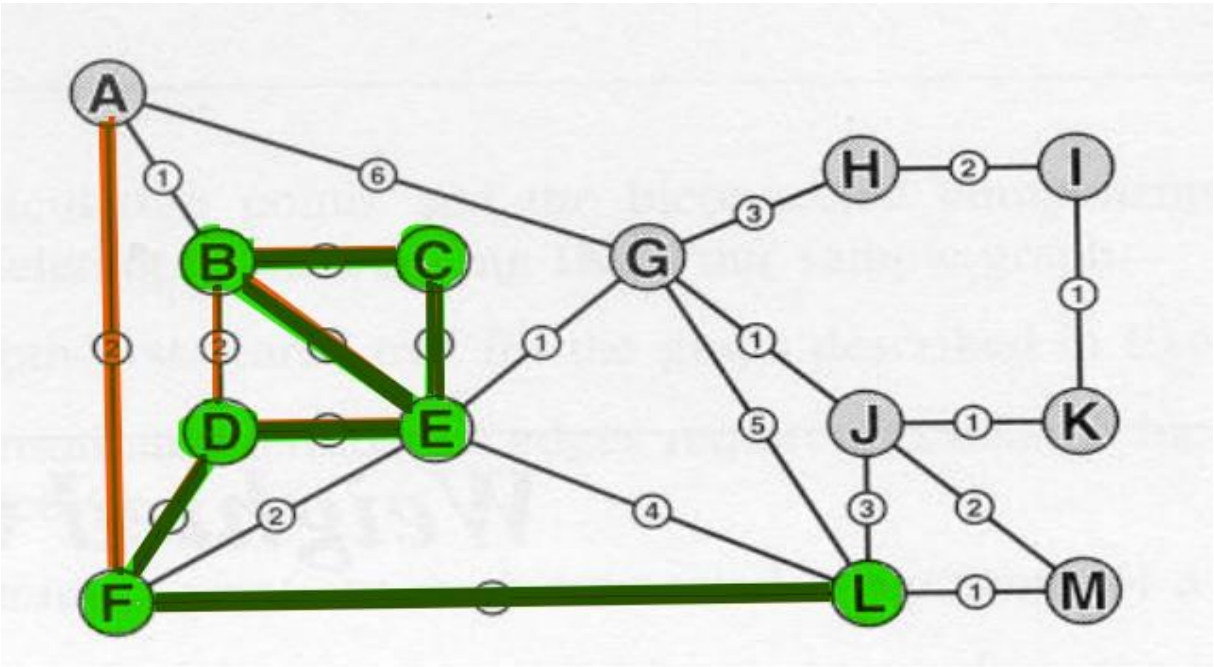
Heap:

Empty

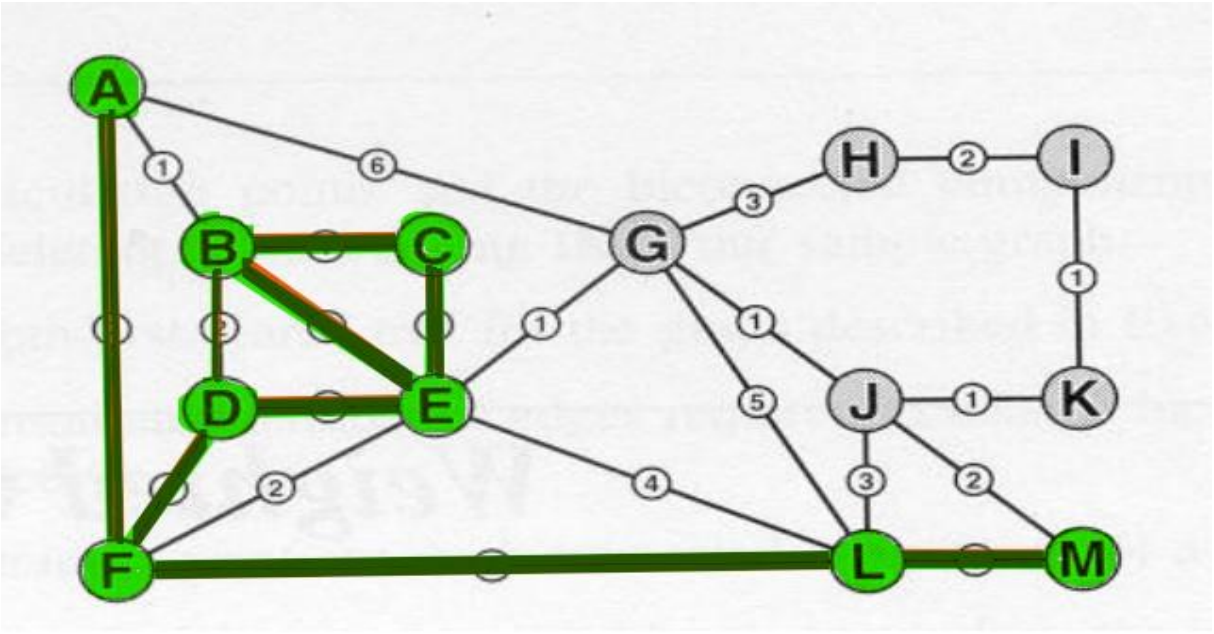
Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		~~**0**~~ **F**	~~**0**~~ **A**	~~**0**~~ **B**	**0**	~~**0**~~ **L**	~~**0**~~ **L**	**0**	**0**	**0**	**0**	**0**	**0**	~~**0**~~ **L**
Dist		~~**0**~~ **2**	~~∞~~ **1**	~~∞~~ **4**	∞	~~∞~~ **4**	~~∞~~ **0**	∞	∞	∞	∞	∞	~~∞~~ **0**	~~∞~~ **1**

Graph Representation:



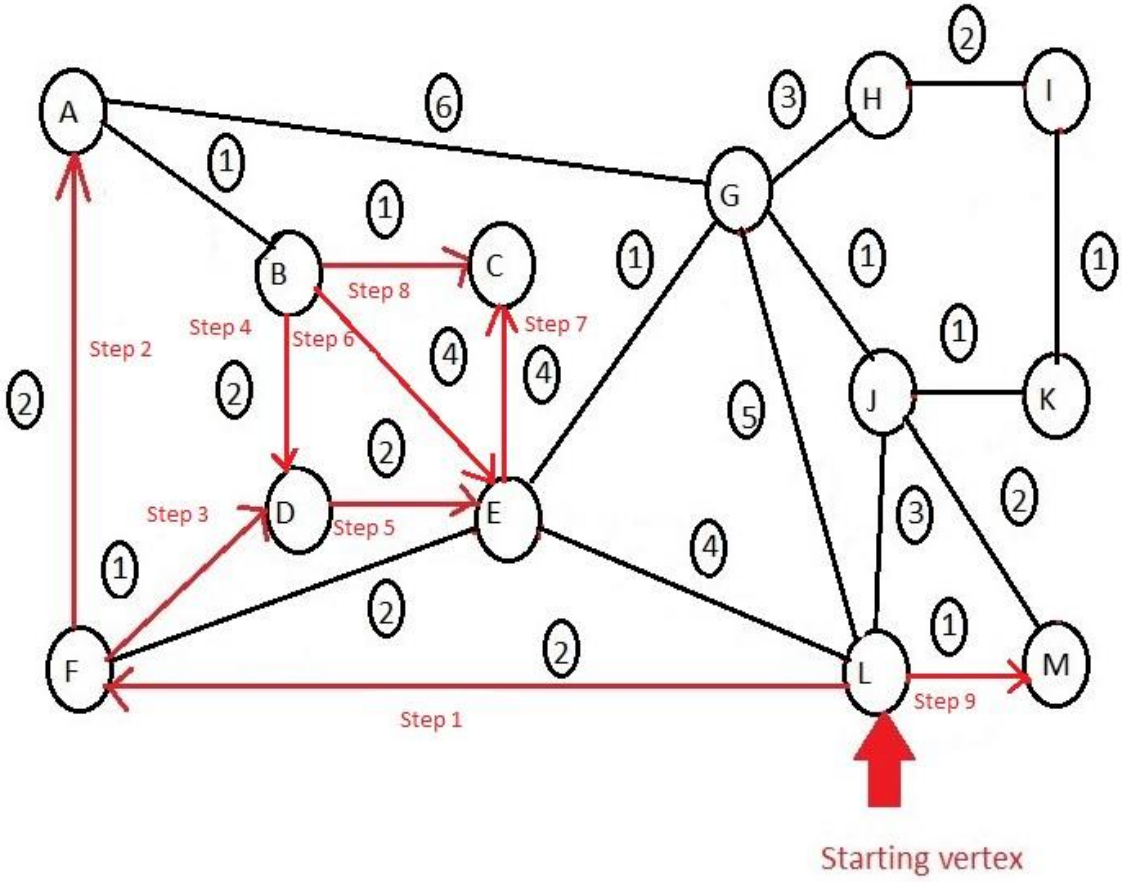
Graph Representation:



Dijkstra's SPT Final Parent And Distance Arrays:

		A	B	C	D	E	F	G	H	I	J	K	L	M
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**11**	**12**
Parent		**F**	**A**	**B**	**0**	**L**	**L**	**0**	**0**	**0**	**0**	**0**	**0**	**L**
Dist		**2**	**1**	**4**	∞	**4**	**0**	∞	∞	∞	∞	∞	**0**	**1**

Diagram Showing Dijkstra's SPT Superimposed On The Graph



Explanation:

Finally, vertex C is extracted from the heap as it has the minimum distance, which is 4. This means that the shortest path from L to C has been found, and vertex C is marked as visited. We now add the neighbors of C to the heap and update their distances in the Dist array. Since there are no more vertices in the heap, the algorithm terminates, and we have constructed the SPT.

The Parent array tracks the parent of each vertex in the SPT, which is the vertex that has the shortest path to the current vertex. In this case, the parent of vertex C is vertex B, the parent of vertex B is vertex E, the parent of vertex E is vertex F, the parent of vertex F is vertex L, and the parent of vertex D is vertex F.

The Distance array tracks the distance of each vertex from the starting vertex (L) in the SPT. In this case, the distance of vertex C from L is 4, the distance of vertex B from L is 4, the distance of vertex E from L is 2, the distance of vertex F from L is 2, the distance of vertex D from L is 2, and the distance of vertex L from L is 0.

Finally, the SPT consists of the vertices L, F, D, E, B, and C, and the edges (L, F), (F, D), (F, E), (E, B), and (B, C). These edges represent the shortest paths from L to each of the other vertices in the graph. The remaining edges in the original graph that are not part of the SPT are not part of the shortest paths and are not included in the SPT.

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 –

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

Union-Find Partition –

Step 2:

MST_Kruskal -

B-1-C

Sets –

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

Union-Find Partition –

Step 3:

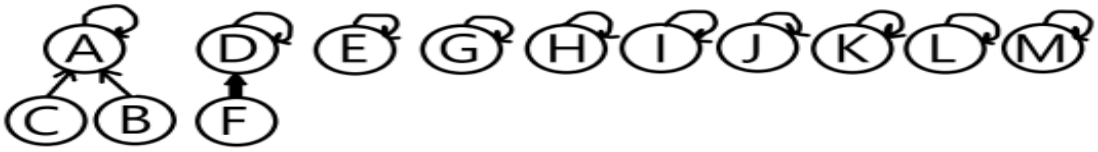
MST_Kruskal -

D-1-F

Sets –

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

Union-Find Partition



Step 4:

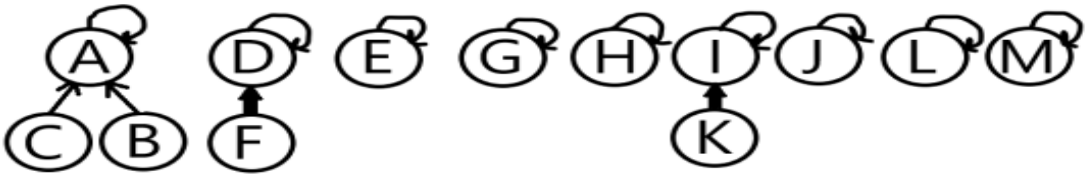
MST_Kruskal –

I-1-K

Sets –

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

Union-Find Partition –



Step 5:

MST_Kruskal -

J-1-K

Sets –

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

Union-Find Partition –

Step 6:

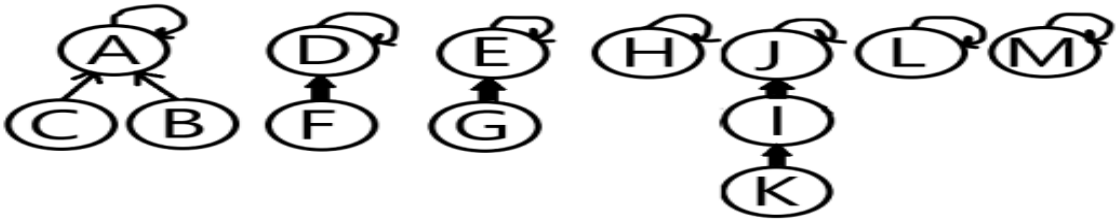
MST_Kruskal -

E-1-G

Sets –

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

Union-Find Partition –



Step 7:

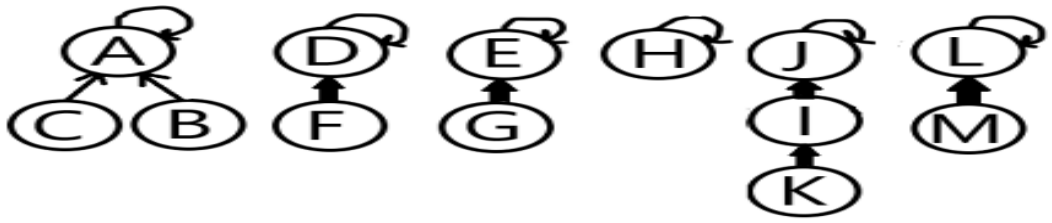
MST_Kruskal –

L-1-M

Sets –

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

Union-Find Partition –



Step 8:

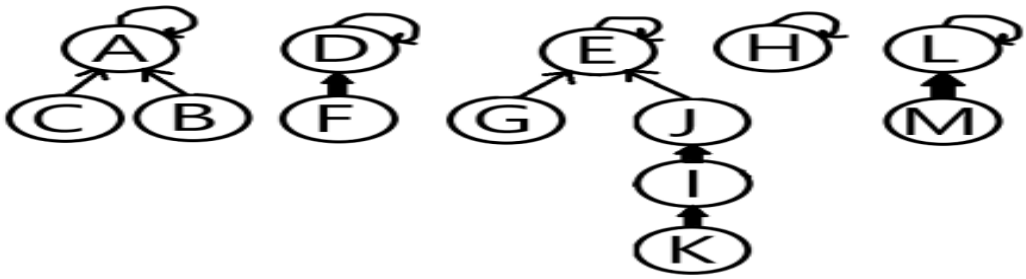
MST_Kruskal -

G-1-J

Sets –

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

Union-Find Partition –



Step 9:

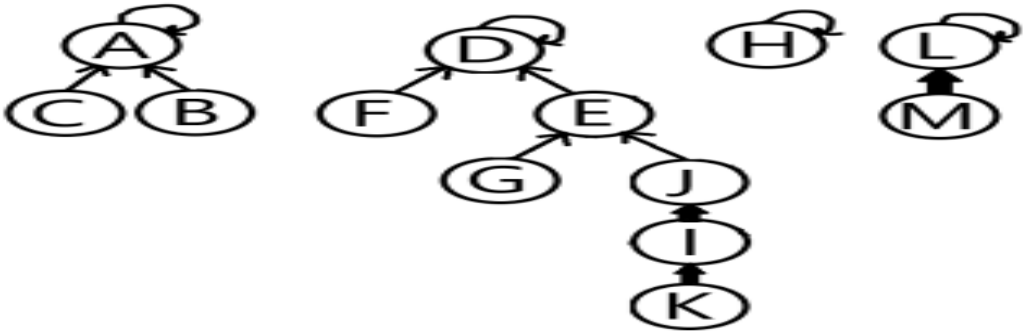
MST_Kruskal -

D-2-E

Sets –

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

Union-Find Partition –



Step 10:

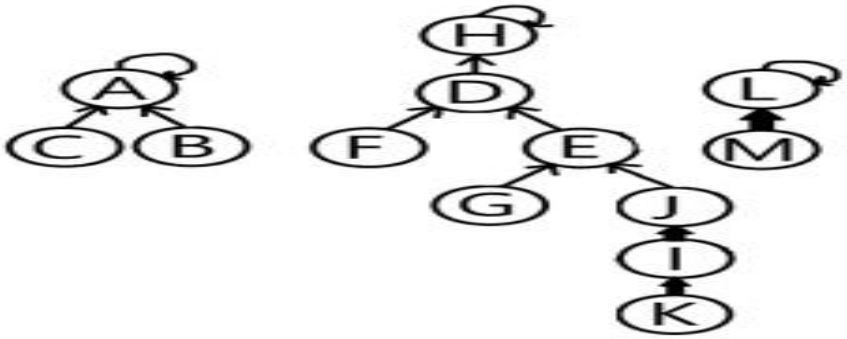
MST_Kruskal -

H-2-I

Sets –

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

Union-Find Partition –



Step 11:

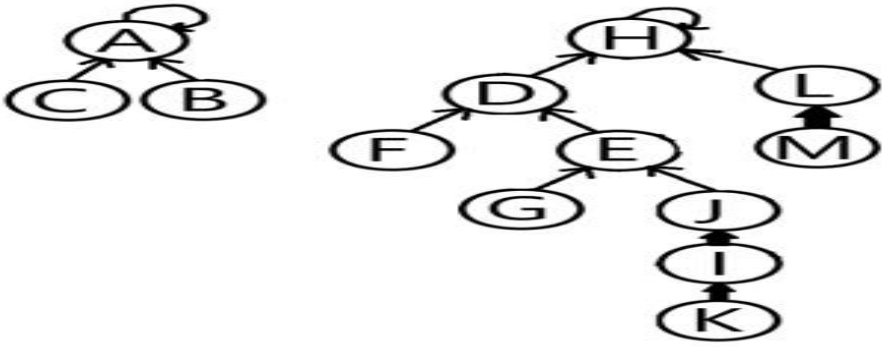
MST_Kruskal -

J-2-M

Sets –

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

Union-Find Partition –



Step 12:

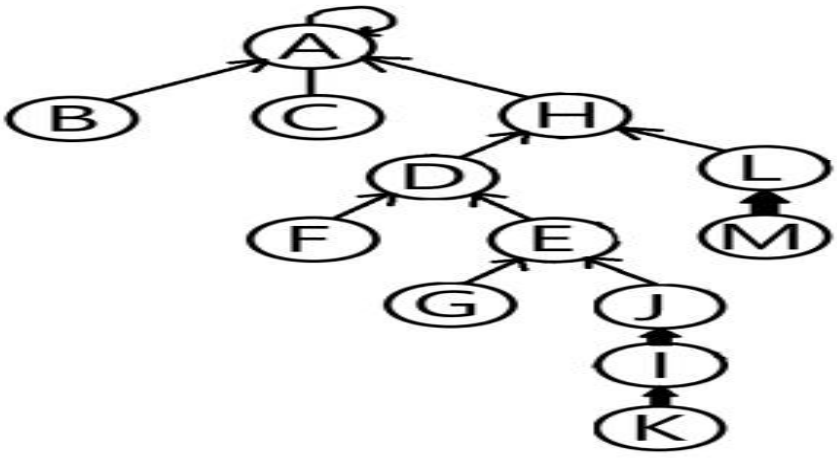
MST_Kruskal -

A-2-F

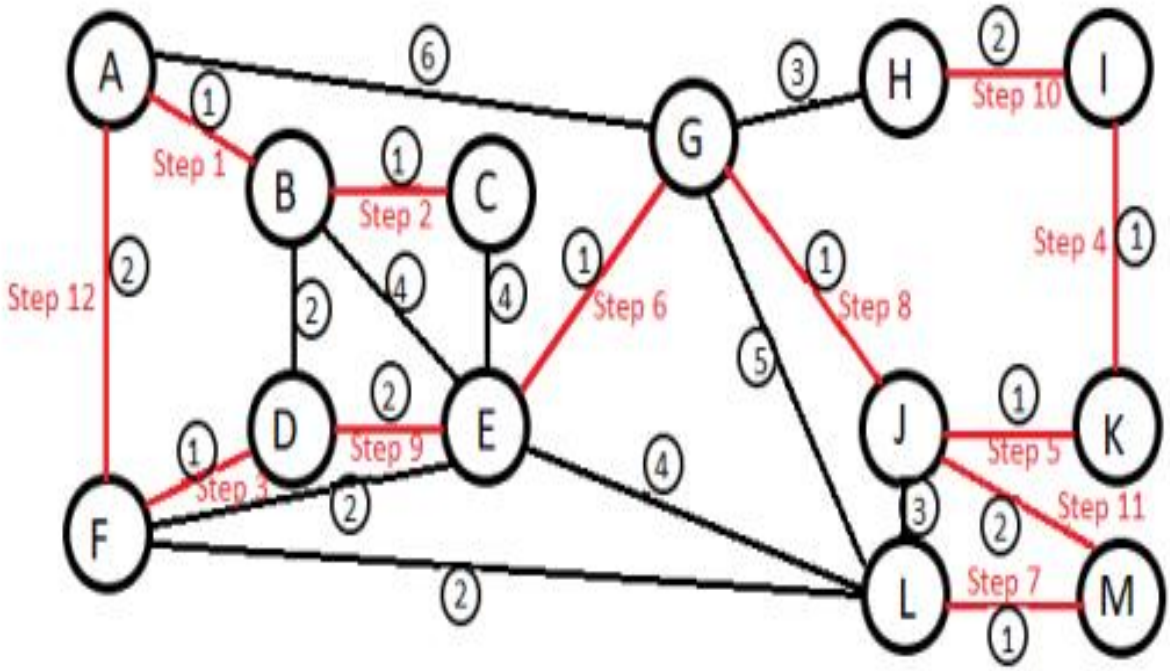
Sets –

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

Union-Find Partition –



Kruskal's MST Graph Diagram:



SCREEN CAPTURES OF PROGRAMS EXECUTING

Output of Prim's Algorithm:

```
Graph Traversal and Shortest Path Algorithms with DFS, BFS, Prim, and Dijkstra

--------------------------------------------------------------------------------

Enter the name of the graph file in .txt extension:
wGraph1.txt

Enter the starting vertex of the graph (as a number):
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

Displaying Adjacency List:

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

```
Depth First Graph Traversal:
Starting with Vertex L

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

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

Adding to MST: Edge

@ <-- (0) --> A
A <-- (1) --> B
B <-- (1) --> C
B <-- (2) --> D
D <-- (2) --> E
D <-- (1) --> F
E <-- (1) --> G
I <-- (2) --> H
K <-- (1) --> I
G <-- (1) --> J
J <-- (1) --> K
F <-- (2) --> L
L <-- (1) --> M
```

```
MST Weight = 16

Minimum Spanning Tree Parent Array:

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

Shortest Path Tree (SPT) using Dijkstra's Algorithm:

Vertex A, distance = 4, path = L->F->A
Vertex B, distance = 5, path = L->F->D->B
Vertex C, distance = 6, path = L->F->D->B->C
Vertex D, distance = 3, path = L->F->D
Vertex E, distance = 4, path = L->E
Vertex F, distance = 2, path = L->F
Vertex G, distance = 4, path = L->J->G
Vertex H, distance = 7, path = L->J->G->H
Vertex I, distance = 5, path = L->J->K->I
Vertex J, distance = 3, path = L->J
Vertex K, distance = 4, path = L->J->K
Vertex M, distance = 1, path = L->M
PS C:\Users\35389\Desktop\TU856 Modules\YEAR 2\Year 2 - Semester 2\Algorithms and Data Structures
```

Output of Kruskal's Algorithm:

```
Kruskal's Minimum Spanning Tree Algorithm

--------------------------------------------------------------------------------

Enter the name of the graph file in .txt extension:
wGraph1.txt

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

Sets Before Kruskal's:

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 }
```

```
Inserting Edge to MST:

Edge A--1--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 }

Tree of Vertices:

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

Inserting Edge to MST:

Edge B--1--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 }

Tree of Vertices:

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
```

```
Inserting Edge to MST:

Edge D--1--F

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

Tree of Vertices:

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

Inserting Edge to MST:

Edge I--1--K

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

Tree of Vertices:

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

Inserting Edge to MST:

Edge J--1--K

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

Tree of Vertices:

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

```
Inserting Edge to MST:

Edge E--1--G

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

Tree of Vertices:

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

Inserting Edge to MST:

Edge L--1--M

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

Tree of Vertices:

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

Inserting Edge to MST:

Edge G--1--J

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

Tree of Vertices:

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

```
Inserting Edge to MST:

Edge D--2--E

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

Tree of Vertices:

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

Inserting Edge to MST:

Edge H--2--I

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

Tree of Vertices:

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

Inserting Edge to MST:

Edge J--2--M

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

Tree of Vertices:

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

Inserting Edge to MST:

Edge A--2--F

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

Tree of Vertices:

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

```
Sets After Kruskal's:

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

Minimum Spanning Tree Built from the 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

Weight of MST = 16

PS C:\Users\35389\Desktop\TU856 Modules\YEAR 2\Year 2 - Semester 2\Algorithms and Data Structures
```

REFLECTION

Through completing this assignment, I have gained valuable knowledge and skills in implementing various algorithms and data structures, specifically Prim's and Kruskal's algorithms for finding the minimum spanning tree, and Dijkstra's shortest path tree algorithm.

I have learned how to represent a graph using an adjacency list data structure and execute various traversal algorithms such as depth-first and breadth-first. I also gained experience in working with priority queues and heaps, as they are essential components of Prim's and Dijkstra's algorithms.

In addition, I have acquired the ability to read and construct a graph from a text file, as well as output the results of the algorithms to the console. This project has helped me develop my time management skills and multitasking abilities, as I had to work on multiple assignments simultaneously while ensuring that the project was completed on time.

Furthermore, this project has significantly improved my programming skills, especially in Java and other object-oriented languages, as well as my understanding of data structures and algorithms. The knowledge and skills I have gained in this project can be applied in future projects and even extended to other areas, such as visualizing how different algorithms work on various graphs.

Overall, this project has been an excellent opportunity for me to gain practical experience and apply theoretical knowledge in a real-world scenario. I am grateful for this opportunity, and I believe that the skills and knowledge I have acquired through this project will serve me well in my future academic and professional pursuits.