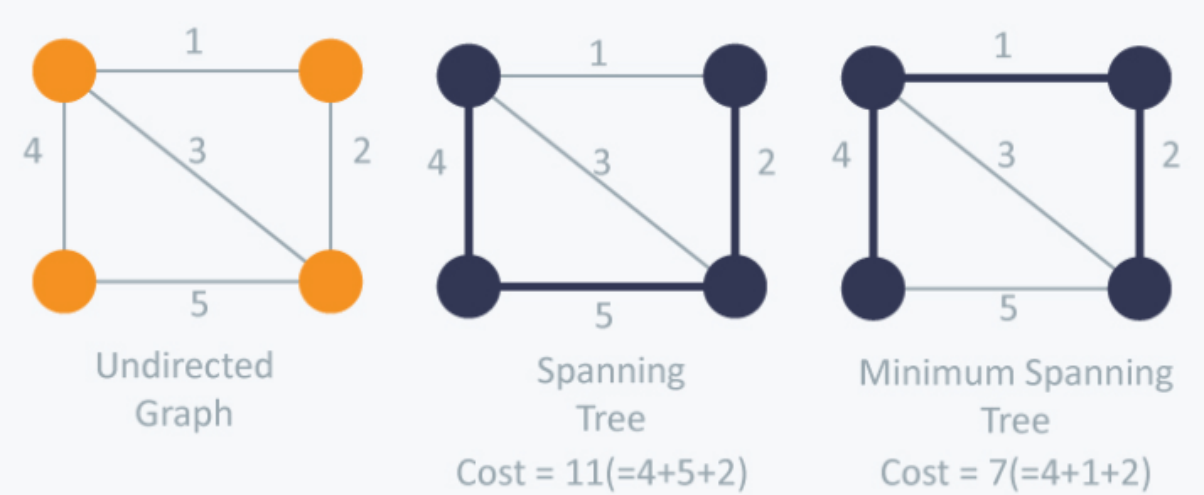
```
    # Next, the hare and tortoise move at same speed until they agree
    mu = 0
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(hare)
        mu += 1
    return lam, mu
```

Applications

Cyclic algorithms are used in message-based distributed systems and large-scale cluster processing systems. It is also mainly used to detect deadlocks in the concurrent system and various cryptographic applications where the keys are used to manage the messages with encrypted values.

Minimum Spanning Tree

A minimum spanning is defined as a subset of edges of a graph having no cycles and is well connected with all the vertices so that the minimum sum is availed through the edge weights. It solely depends on the cost of the spanning tree and the minimum span or least distance the vertex covers. There can be many minimum spanning trees depending on the edge weight and various other factors.



Pseudocode

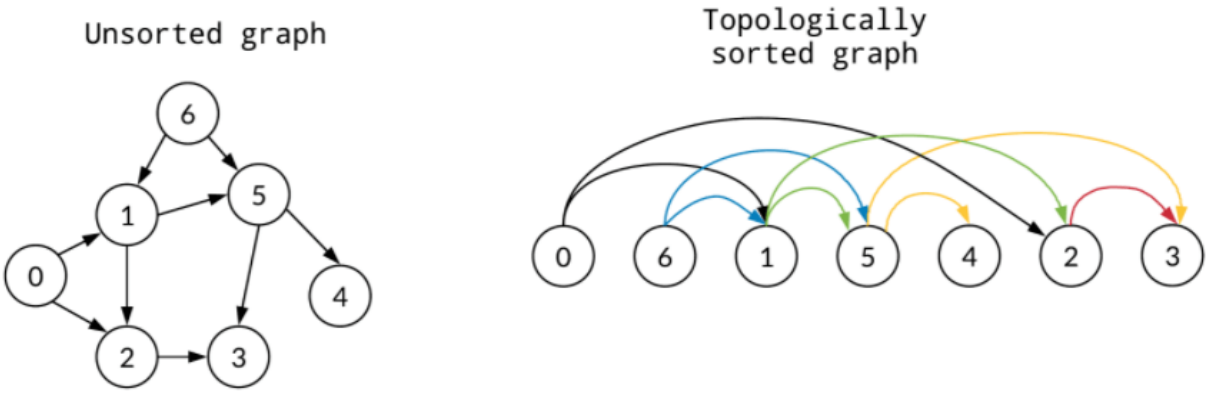
```
1. Prim's Algorithm Example
2. ReachSet = {0};
3.    UnReachSet = {1, 2, ..., N-1};
4.    SpanningTree = {};
5.    while ( UnReachSet ? empty )
6.    {
7.       Find edge e = (x, y) such that:
8.          1. x ? ReachSet
9.       2. y ? UnReachSet
10.            3. e has smallest cost
11.             SpanningTreeSpanningTree = SpanningTree ?
    {e};
12.             ReachSetReachSet  = ReachSet ? {y};
13.             UnReachSetUnReachSet = UnReachSet - {y};
14.          }
```

Applications

Minimum spanning tree finds its application in the network design and is popularly used in **traveling salesman** problems in a data structure. It can also be used to find the minimum-cost weighted perfect matching and multi-terminal minimum cut problems. MST also finds its application in the field of image and handwriting recognition and cluster analysis.

Topological sorting

Topological sorting of a graph follows the algorithm of ordering the vertices linearly so that each directed graph having vertex ordering ensures that the vertex comes before it. Users can understand it more accurately by looking at the sample image given below.



In the above example, you can visualize the ordering of the unsorted graph and topologically sorted graph. The topologically sorted graph ensures to sort vertex that comes in the pathway.

Pseudocode

```
1. topological_sort(N, adj[N][N])
2.         T = []
3.         visited = []
4.         in_degree = []
5.         for i = 0 to N
6.                 in_degree[i] = visited[i] = 0
7.         for i = 0 to N
8.                 for j = 0 to N
9.                         if adj[i][j] is TRUE
10.                                in_degree[j] = in_degree[j] + 1
11.                for i = 0 to N
12.                        if in_degree[i] is 0
13.                                enqueue(Queue, i)
14.                                visited[i] = TRUE
15.                while Queue is not Empty
16.                        vertex = get_front(Queue)
17.                        dequeue(Queue)
18.                        T.append(vertex)
19.                        for j = 0 to N
```

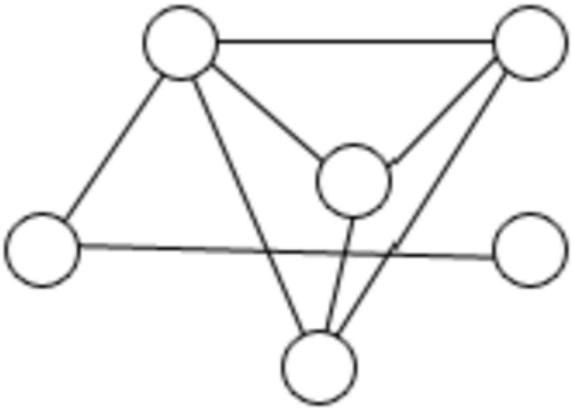
```
20.                     if adj[vertex][j] is TRUE and
   visited[j] is FALSE
21.                         in_degree[j] = in_degree[j] - 1
22.                         if in_degree[j] is 0
23.                             enqueue(Queue, j)
24.                             visited[j] = TRUE
25.             return T
```

Application

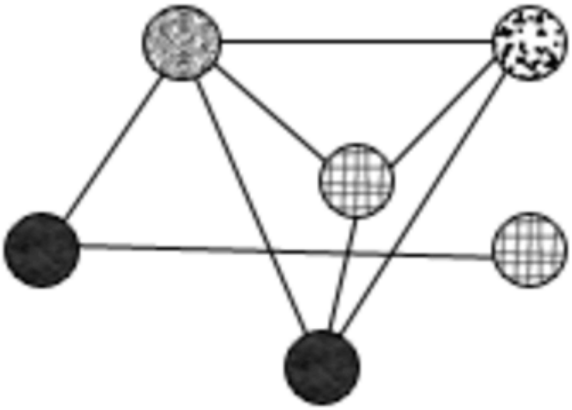
Topological sorting covers the room for application in Kahn's and DFS algorithms. In real-life applications, topological sorting is used in scheduling instructions and serialization of data. It is also popularly used to determine the tasks that are to be compiled and used to resolve dependencies in linkers.

Graph coloring

Graph coloring algorithms follow the approach of assigning colors to the elements present in the graph under certain conditions. The conditions are based on the techniques or algorithms. Hence, vertex coloring is a commonly used coloring technique followed here. First, in this method, you try to color the vertex using k color, ensuring that two adjacent vertexes should not have the same color. Other method includes face coloring and edge coloring. Both of these methods should also ensure that no edge or face should be inconsequent color. The coloring of the graph is determined by knowing the chromatic number, which is also the smaller number of colors needed. Consider the below image to understand how it works.



non-colored graph



colored graph

Pseudocode

```
1. #include <iostream>
2. #include <list>
3. using namespace std;
4. // A class that represents an undirected graph
5. class Graph
6. {
7.     int V;    // No. of vertices
8.     list<int> *adj;    // A dynamic array of adjacency lists
9. public:
10.         // Constructor and destructor
11.         Graph(int V)    { this->VV = V; adj = new
    list<int>[V]; }
12.         ~Graph()      { delete [] adj; }
13.         // function to add an edge to graph
14.         void addEdge(int v, int w);
15.         // Prints greedy coloring of the vertices
16.         void greedyColoring();
17.     };
18.     void Graph::addEdge(int v, int w)
19.     {
20.         adj[v].push_back(w);
```

```
21.         adj[w].push_back(v); // Note: the graph is
  undirected
22.     }
23.     // Assigns colors (starting from 0) to all vertices and
  prints
24.     // the assignment of colors
25.     void Graph::greedyColoring()
26.     {
27.         int result[V];
28.         // Assign the first color to first vertex
29.         result[0]  = 0;
30.         // Initialize remaining V-1 vertices as unassigned
31.         for (int u = 1; u < V; u++)
32.             result[u] = -1;  // no color is assigned to u
33.         // A temporary array to store the available colors.
  True
34.         // value of available[cr] would mean that the color
  cr is
35.         // assigned to one of its adjacent vertices
36.         bool available[V];
37.         for (int cr = 0; cr < V; cr++)
38.             available[cr] = false;
39.         // Assign colors to remaining V-1 vertices
40.         for (int u = 1; u < V; u++)
41.         {
42.             // Process all adjacent vertices and flag their
  colors
43.             // as unavailable
44.             list<int>::iterator i;
45.             for (i = adj[u].begin(); i != adj[u].end(); ++i)
46.                 if (result[*i] != -1)
47.                     available[result[*i]] = true;
48.             // Find the first available color
```

```
            int cr;
            for (cr = 0; cr < V; cr++)
                if (available[cr] == false)
                    break;
            result[u] = cr; // Assign the found color
            // Reset the values back to false for the next
  iteration
            for (i = adj[u].begin(); i != adj[u].end(); ++i)
                if (result[*i] != -1)
                    available[result[*i]] = false;
        }
        // print the result
        for (int u = 0; u < V; u++)
            cout << "Vertex " << u << " --->  Color "
                 << result[u] << endl;
    }
    // Driver program to test above function
    int main()
    {
        Graph g1(5);
        g1.addEdge(0, 1);
        g1.addEdge(0, 2);
        g1.addEdge(1, 2);
        g1.addEdge(1, 3);
        g1.addEdge(2, 3);
        g1.addEdge(3, 4);
        cout << "Coloring of graph 1 \n";
        g1.greedyColoring();
        Graph g2(5);
        g2.addEdge(0, 1);
        g2.addEdge(0, 2);
        g2.addEdge(1, 2);
        g2.addEdge(1, 4);
```

```
81.         g2.addEdge(2, 4);
82.         g2.addEdge(4, 3);
83.         cout << "\nColoring of graph 2 \n";
84.         g2.greedyColoring();
85.         return 0;
86.     }
```

Application

Graph coloring has vast applications in data structures as well as in solving real-life problems. For example, it is used in timetable scheduling and assigning radio frequencies for mobile. It is also used in Sudoko and to check if the given graph is bipartite. Graph coloring can also be used in geographical maps to mark countries and states in different colors.

Maximum flow

The maximum flow algorithm is usually treated as a problem-solving algorithm where the graph is modeled like a network flow infrastructure. Hence, the maximum flow is determined by finding the path of the flow that has the **maximum flow rate.** The maximum flow rate is determined by augmenting paths which is the total flow-based out of source node equal to the flow in the sink node. Below is the illustration for the same.

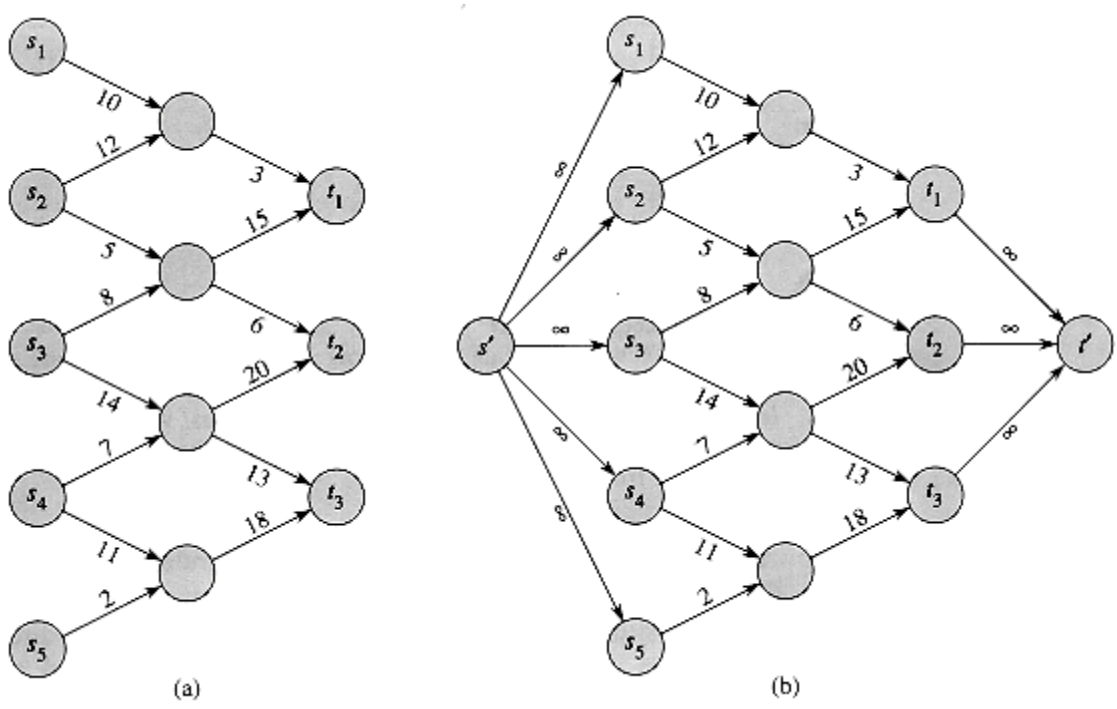
```
1. function: DinicMaxFlow(Graph G,Node S,Node T):
2.     Initialize flow in all edges to 0, F = 0
3.     Construct level graph
4.     while (there exists an augmenting path in level graph):
5.         find blocking flow f in level graph
6.         FF = F + f
7.         Update level graph
8.     return F
```

Applications

Like you, the maximum flow problem covers applications of popular algorithms like the Ford-Fulkerson algorithm, Edmonds-Karp algorithm, and Dinic's algorithm, like you saw in the pseudocode given above. In real life, it finds its applications in scheduling crews in flights and image segmentation for foreground and background. It is also used in games like basketball, where the score is set to a maximum estimated value having the current division leader.

Matching

A matching algorithm or technique in the graph is defined as the edges that no common vertices at all. Matching can be termed maximum matching if the most significant number of edges possibly matches with as many vertices as possible. It follows a specific approach for determining full matches, as shown in the below image.



Applications

Matching is used in an algorithm like the Hopcroft-Karp algorithm and Blossom algorithm. It can also be used to solve problems using a Hungarian algorithm that covers concepts of

matching. In real-life examples, matching can be used resource allocation and travel optimization and some problems like stable marriage and vertex cover problem.

Conclusion

In this article, you came across plenty of graph coloring algorithms and techniques that find their day-to-day applications in all instances of real life. You learned how to implement them according to situations, and hence the pseudo code helped you process the information strategically and efficiently. Graph algorithms are considered an essential aspect in the field confined not only to solve problems using data structures but also in general tasks like Google Maps and Apple Maps. However, a beginner might find it hard to implement Graph algorithms because of their complex nature. Hence, it is highly recommended to go through this article since it covers everything from scratch

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.

Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

1. $A \leftarrow \emptyset$
2. for each vertex $v \in V$ [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) $\in$ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) $\neq$ if FIND-SET (v)
7. then $A \leftarrow A \cup \{(u, v)\}$
8. UNION (u, v)
9. return A

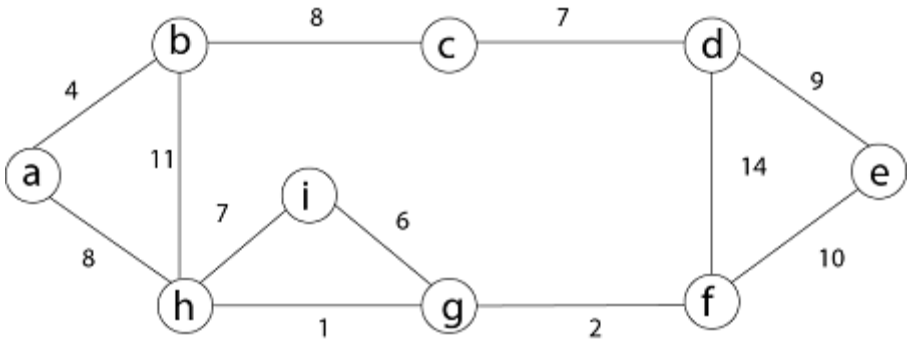
Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, $V \leq 2$ E, so log V is O (log E).

Thus the total time is

1. O (E log E) = O (E log V).

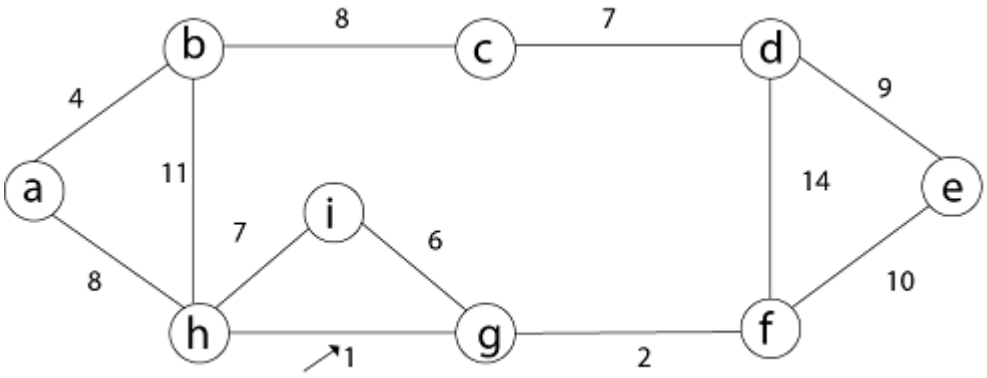
For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



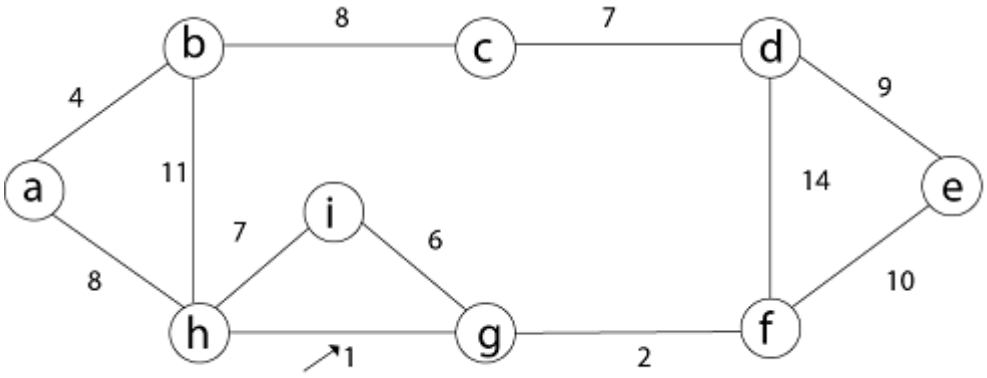
Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.
There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

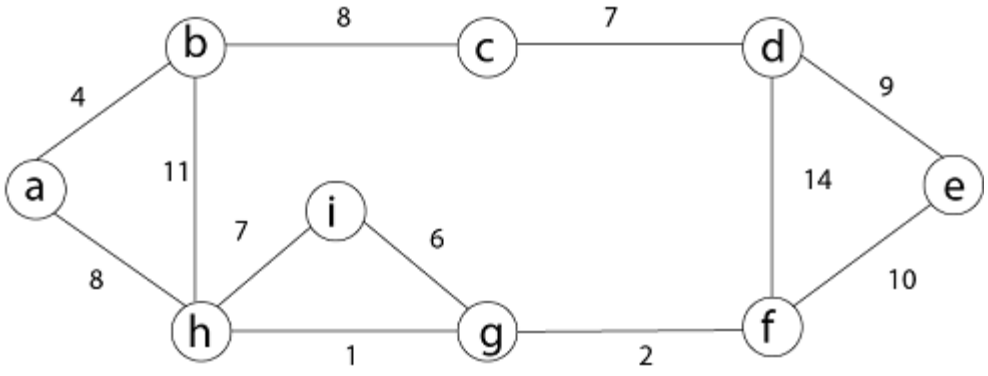
Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

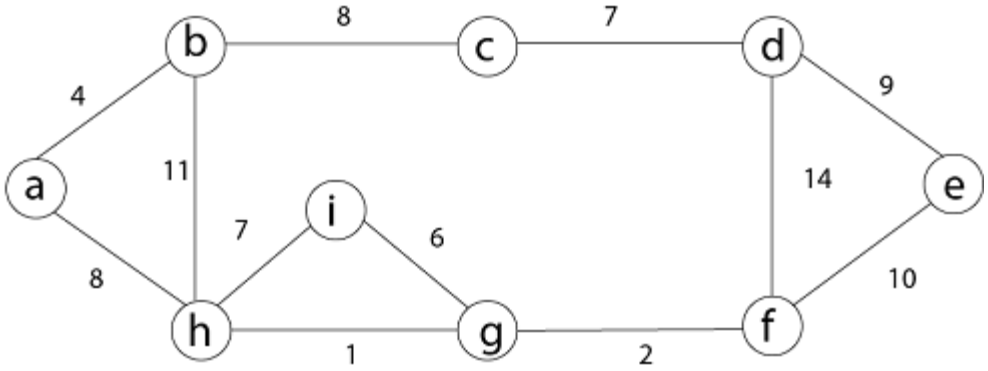


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



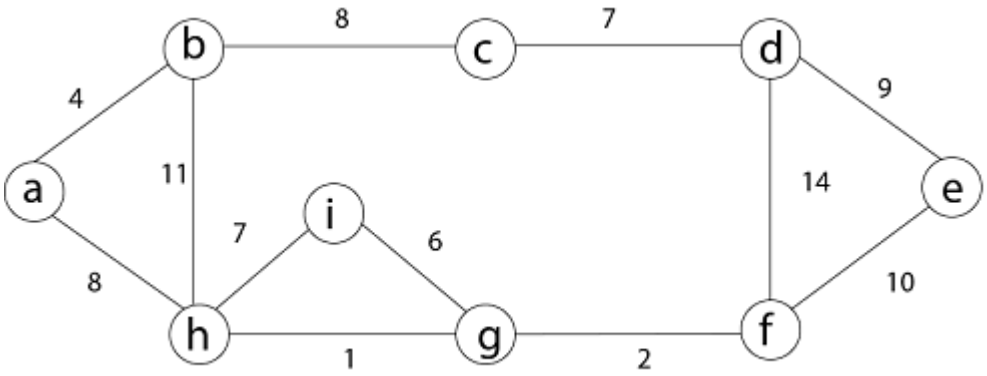
Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



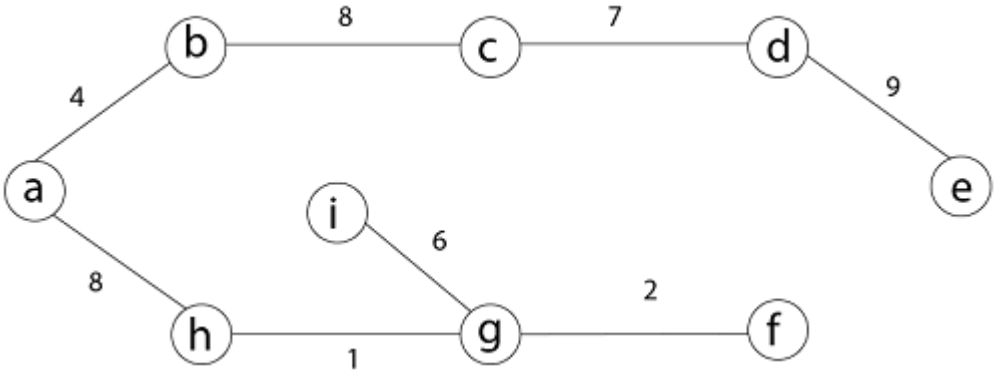
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

Prim's Algorithm:

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.

a. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.

b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

MST-PRIM (G, w, r)

1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0

5. Q ← V [G]

6. While Q ? ∅

7. do u ← EXTRACT - MIN (Q)

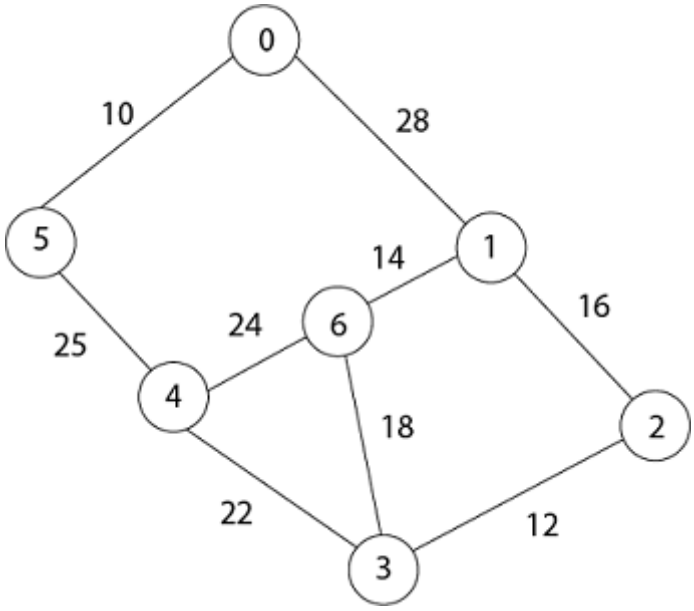
8. for each v ∈ Adj [u]

9. do if v ∈ Q and w (u, v) < key [v]

10. then π [v] ← u

11. key [v] ← w (u, v)

Example: Generate minimum cost spanning tree for the following graph using Prim's algorithm.



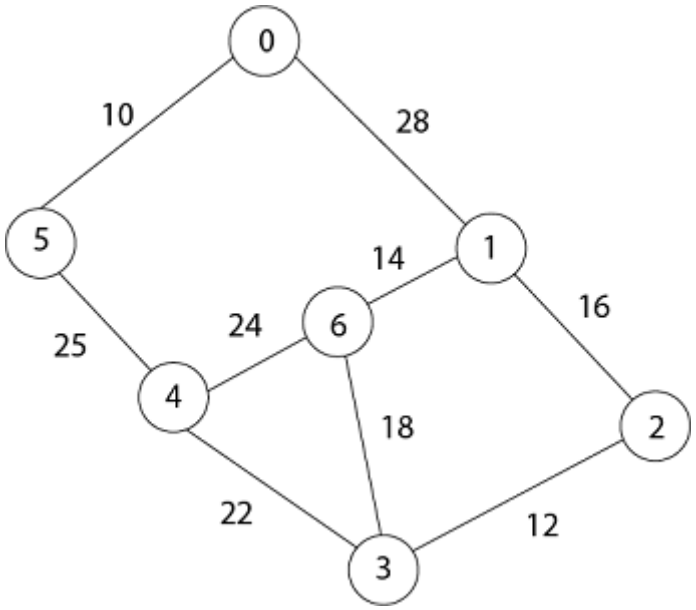
Solution: In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

1. Taking 0 as starting vertex
2. Root = 0
3. Adj [0] = 5, 1
4. Parent, π [5] = 0 and π [1] = 0
5. Key [5] = ∞ and key [1] = ∞
6. w [0, 5) = 10 and w (0,1) = 28
7. w (u, v) < key [5] , w (u, v) < key [1]
8. Key [5] = 10 and key [1] = 28
9. So update key value of 5 and 1 is:

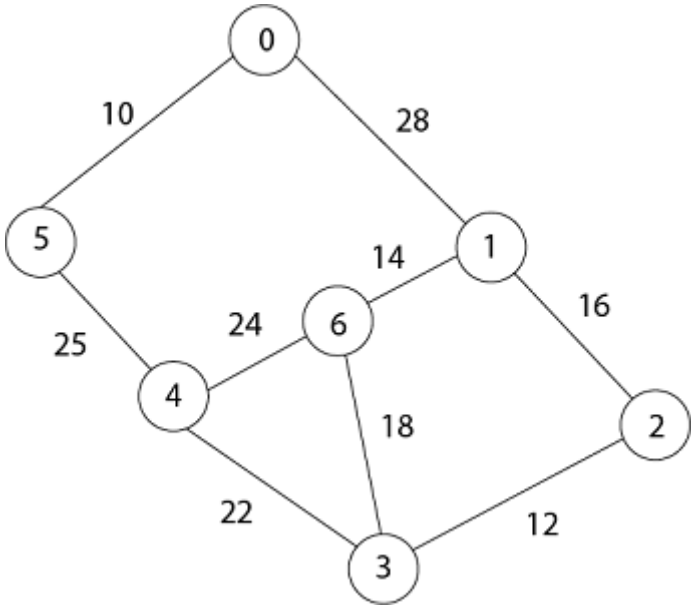
Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

1. Adj [5] = {0, 4} and 0 is already in heap
2. Taking 4, key [4] = ∞ π [4] = 5
3. (u, v) < key [v] then key [4] = 25
4. w (5,4) = 25
5. w (5,4) < key [4]
6. date key value and parent of 4.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u =4

1. Adj [4] = {6, 3}
2. Key [3] = ∞ key [6] = ∞
3. w (4,3) = 22 w (4,6) = 24
4. w (u, v) < key [v] w (u, v) < key [v]
5. w (4,3) < key [3] w (4,6) < key [6]

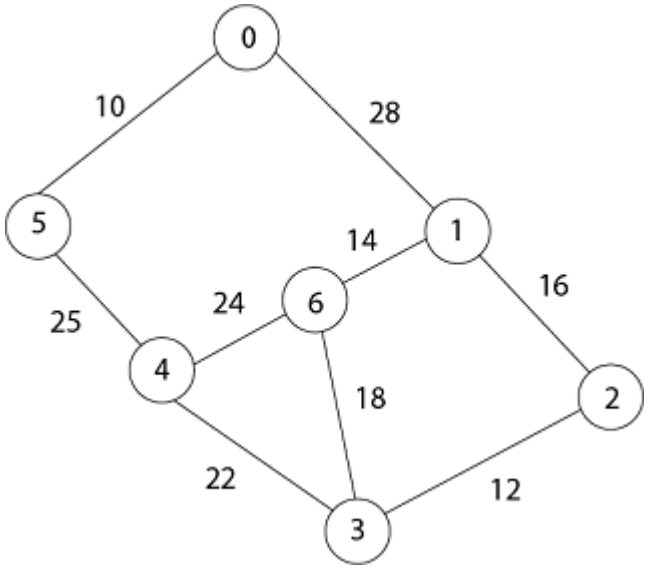
Update key value of key [3] as 22 and key [6] as 24.
And the parent of 3, 6 as 4.

1. $\pi[3]= 4 \qquad \pi[6]= 4$

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	24
Parent	NIL	0	NIL	4	5	0	4

1. u = EXTRACT_MIN (3, 6) [key [3] < key [6]]
2. u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.



1. Adj [3] = {4, 6, 2}
2. 4 is already in heap
3. 4 ≠ Q key [6] = 24 now becomes key [6] = 18
4. Key [2] = ∞ key [6] = 24
5. w (3, 2) = 12 w (3, 6) = 18
6. w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

1. π [2] = 3 π[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

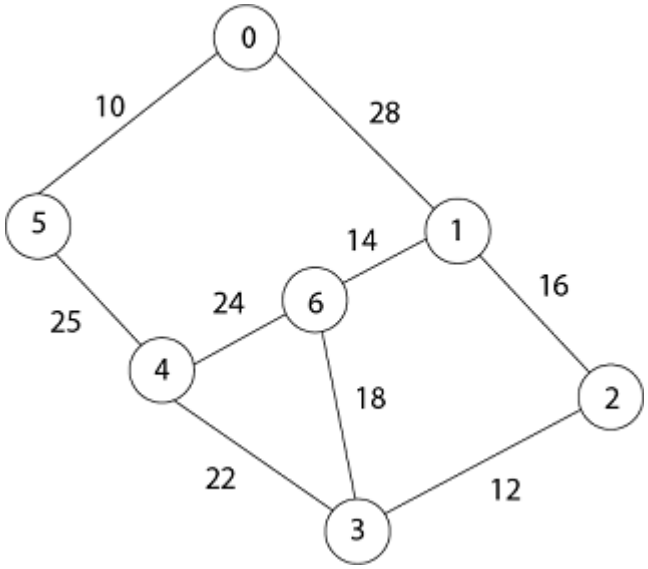
Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

1. u = EXTRACT_MIN (2, 6)
2. u = 2 [key [2] < key [6]]
3. 12 < 18
4. Now the root is 2
5. Adj [2] = {3, 1}
6. 3 is already in a heap
7. Taking 1, key [1] = 28
8. w (2,1) = 16
9. w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

1. π[1]= 2

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.

1. Adj [1] = {0, 6, 2}
2. 0 and 2 are already in heap.
3. Taking 6, key [6] = 18
4. w [1, 6] = 14
5. w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

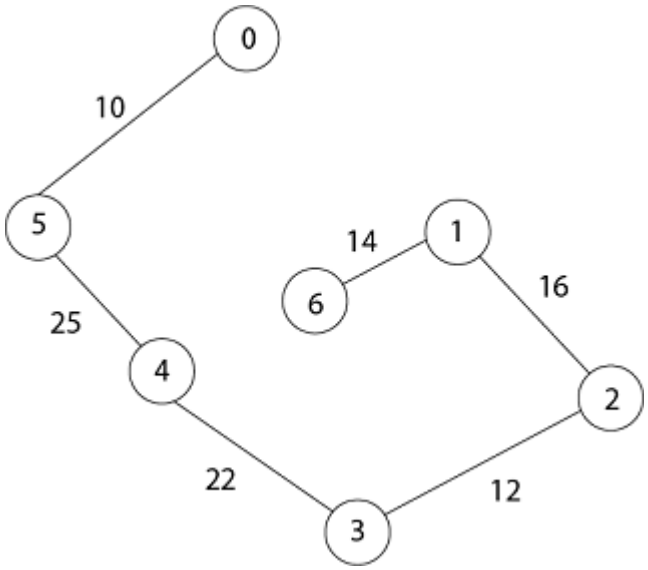
1. Π [6] = 1

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

1. 0 → 5 → 4 → 3 → 2 → 1 → 6
2. [Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

Thus the final spanning Tree is



Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99