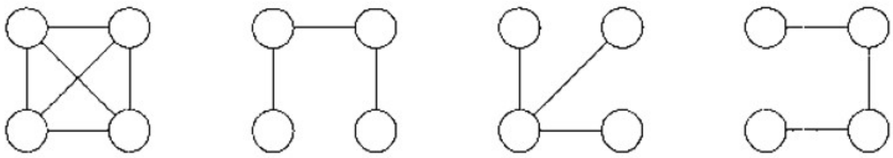
**COURSE HANDOUT**

| | |
|---|---|
| Course Code | ACSC13 |
| Course Name | Design and Analysis of Algorithms |
| Class / Semester | IV SEM |
| Section | A-SECTION |
| Name of the Department | CSE-CYBER SECURITY |
| Employee ID | IARE11023 |
| Employee Name | Dr K RAJENDRA PRASAD |
| Topic Covered | minimum cost spanning trees |
| Course Outcome/s | Construct the minimum spanning trees for the graph using Prim's and Kruskal's algorithm |
| Handout Number | 29 |
| Date | 6 April, 2023 |

## Content about topic covered: Minimum cost spanning

**Spanning tree:** Let G = (V, E) is an undirected connected graph. A sub-graph t = (V, E') of G is a spanning tree of G if and only if t is tree.

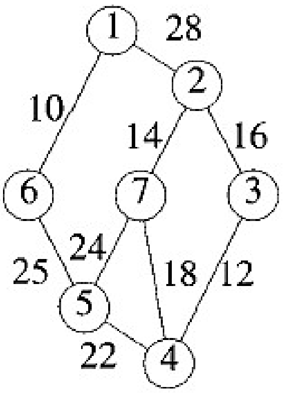
An undirected graph and its spanning trees are shown below:

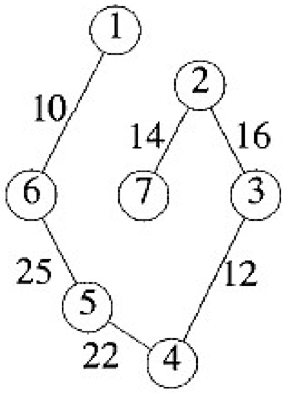


The cost of spanning tree is the sum of the cost of the edges in that tree.

A **spanning tree** with minimum cost is called minimum cost spanning tree.

A graph and its minimum cost spanning tree are shown below:




To find the minimum cost spanning tree, two algorithms are used.

1. Prim’s algorithm
2. Kruskals algorithm.

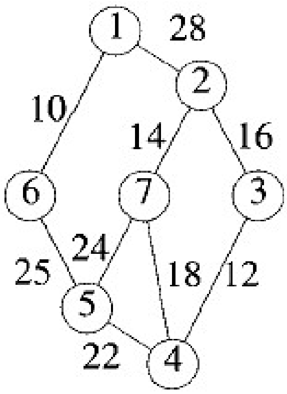
**1. Prim’s Algorithm**

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

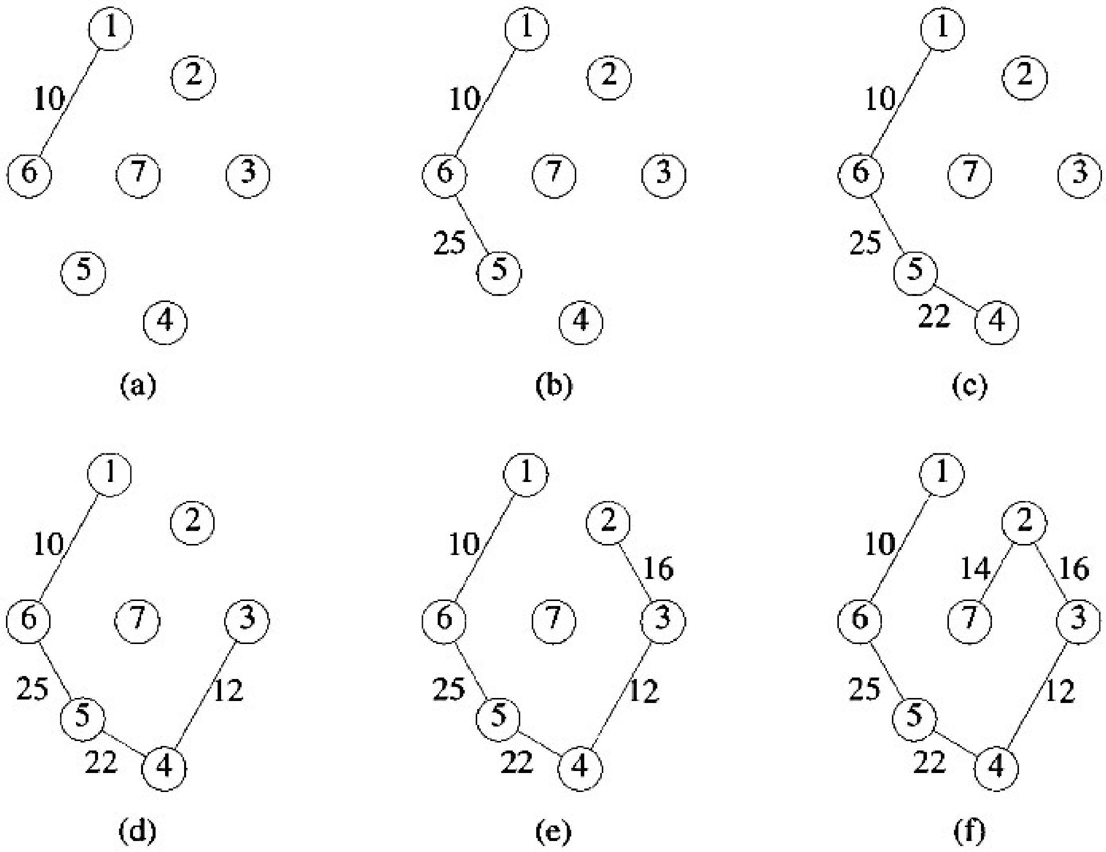
The algorithm will start with a tree that includes only a minimum cost edge of G. Then edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already include in the tree, j is a vertex not yet included, and the cost of (i, j) is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree.

To determine the edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value **near[j]**. The value near[j] is a vertex in the tree such that cost[ j, near[j] ] is minimum among all choices for near[j]. We define near[j] = 0 for all vertices j that are already in the tree. The next edge to include is defined by the vertex j such that near[j] $\neq$ 0 and cost[ j, near[j] ] is minimum.

Eg: Consider the following graph



The stages in the Prim's algorithm are shown below:



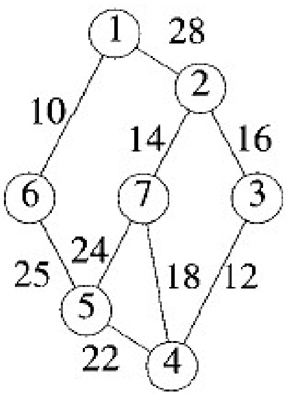

The time required by Prim's algorithm is O($n^2$), where n is the number of vertices in the graph.

**2. Kruskal's Method**

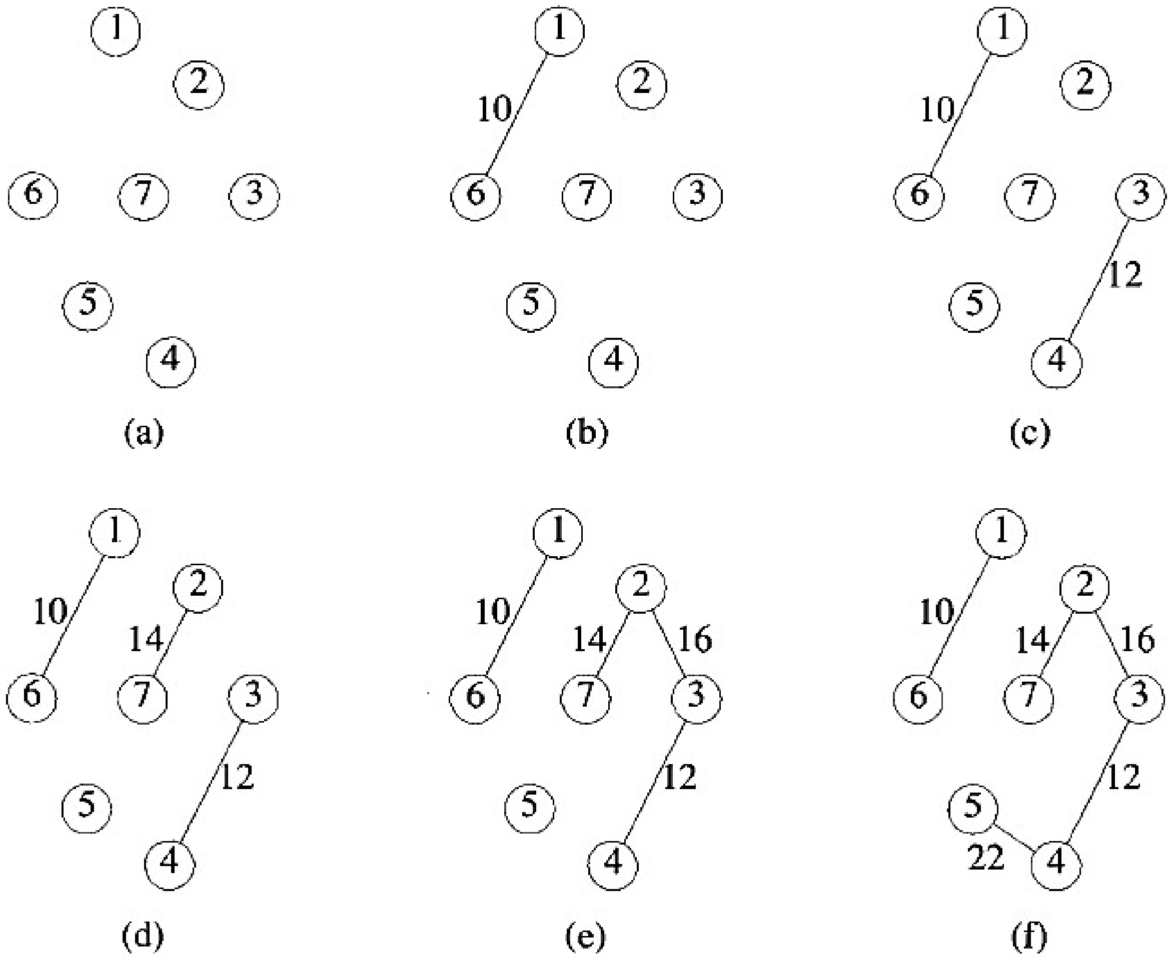
```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

In Kruskal's method, the edges of the graph are considered in non-decreasing order of cost. In this method, the set t of edges so far selected for the spanning tree be such that it is possible to complete t in to a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will be only a forest since the set of edges t can be completed in to a tree if and only if there are no cycles in t.

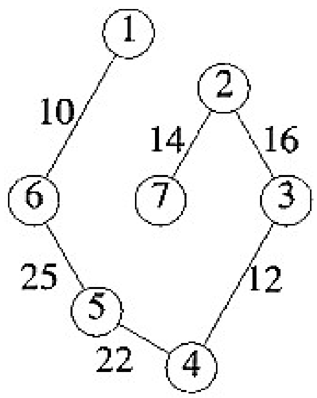
Eg: Consider the following graph




The stages in kruskal's algorithm are:




And then finally



The time complexity is O(|E| log |E| ) where E is the edge set of G.