UNIT III

- Greedy method:
  - ✓ General methodology
- applications
  - ✓ knapsack problem
  - ✓ Minimum cost spanning trees
  - ✓ Single source shortest path problem.

UNIT III
Greedy Method

Greedy Method

- **General Method**:

✓ Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.

✓ This approach never reconsiders the choices taken previously.

✓ This approach is mainly used to solve optimization problems.

✓ Greedy method is easy to implement and quite efficient in most of the cases.

✓ Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

Greedy Method Contd..

- **Components of Greedy Algorithm:**

  Greedy algorithms have the following five components –

✓ A candidate set – A solution is created from this set.

✓ A selection function – Used to choose the best candidate to be added to the solution.

✓ A feasibility function – Used to determine whether a candidate can be used to contribute to the solution.

✓ An objective function – Used to assign a value to a solution or a partial solution.

✓ A solution function – Used to indicate whether a complete solution has been reached.

Greedy Method contd..

- Applications of Greedy method:

✓ Knapsack problem

✓ Greedy approach is used to solve many problems, such as Finding the shortest path between two vertices using Dijkstra's algorithm.

✓ Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

- Where Greedy Approach Fails

In many problems, Greedy algorithm fails to find an optimal solution, moreover it may produce a worst solution. Problems like Travelling Salesperson and Knapsack cannot be solved using this approach.

Job Sequencing with Deadlines

- In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.
- Solution

✓ Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

✓ It may happen that all of the given jobs may not be completed within their deadlines.

✓ Assume, deadline of $\mathbf{i^{th}}$ job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, D(i)>0 for $1 \leqslant i \leqslant n$.

Initially, these jobs are ordered according to profit, i.e. $p1 \geqslant p2 \geqslant p3 \geqslant ... \geqslant pn$.

Job Sequencing with Deadlines example

- Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Job Sequencing with Deadlines example contd..

- Solution:

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	**J_2**	**J_1**	**J_4**	**J_3**	**J_5**
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

Job Sequencing with Deadlines example contd..

- From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.
- Next, J_1 is selected as it gives more profit compared to J_4.
- In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
- The job J_5 is discarded as it cannot be executed within its deadline.
- Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.
- Total profit of this sequence is **100 + 60 + 20 = 180**.

Job Sequencing with Deadlines contd..

- **Algorithm:**

Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0
k := 1
J(1) := 1 // means first job is selected
for i = 2 … n do
     r := k
    while D(J(r)) > D(i) and D(J(r)) ≠ r do
          r := r – 1
   if D(J(r)) ≤ D(i) and D(i) > r then
          for l = k … r + 1 by -1 do
               J(l + 1) := J(l)
               J(r + 1) := I
               k := k + 1
```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$

Knapsack Problem

- Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- The knapsack problem is in combinatorial optimization problem. It appears as a sub problem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.
- The items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.
- Based on the nature of the items, Knapsack problems are categorized as

✓ Fractional Knapsack

✓ Knapsack

Knapsack Problem contd..

- Fractional Knapsack

  In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

- According to the problem statement,

✓ There are **n** items in the store

✓ Weight of **ith** item wi>0

✓ Profit for **ith** item pi>0 and

✓ Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of **ith** item.

$0 \leqslant xi \leqslant 1$

The **ith** item contributes the weight xi.wi to the total weight in the knapsack and profit xi.pi to the total profit.

Hence, the objective of this algorithm is to maximize $\sum_{i=1}^{n}(xi.pi)$ subject to constraint, $\sum_{i=1}^{n}(xi.wi) \leqslant W$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by $\sum_{i=1}^{n}(xi.wi) = W$

In this context, first we need to sort those items according to the value of pi /.wi, so that (pi+1/wi+1) ≤ pi/wi.

Here, ***x*** is an array to store the fraction of items.

Knapsack Problem contd..

- **Algorithm:**

Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n do
      x[i] = 0;
 weight = 0;
 for i = 1 to n
      if weight + w[i] ≤ W then
            x[i] = 1;
            weight = weight + w[i] ;
      else
           x[i] = (W - weight) / w[i] ;
           weight = W ;
           break ;
  return xl
```

- **Analysis**

If the provided items are already sorted into a decreasing order of pi / wi, then the while loop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

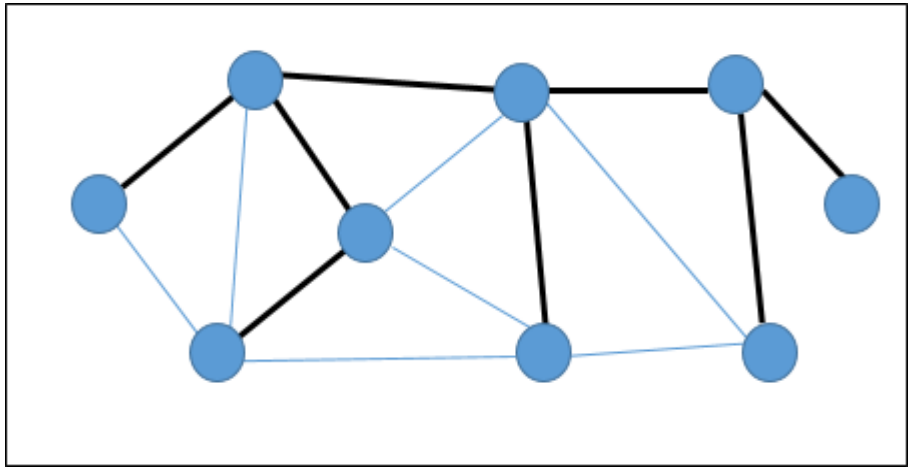
Minimum Cost Spanning Trees

- A spanning tree is a subset of an undirected Graph that has all the vertices connected by minimum number of edges. If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.
- Properties

✓ A spanning tree does not have any cycle.

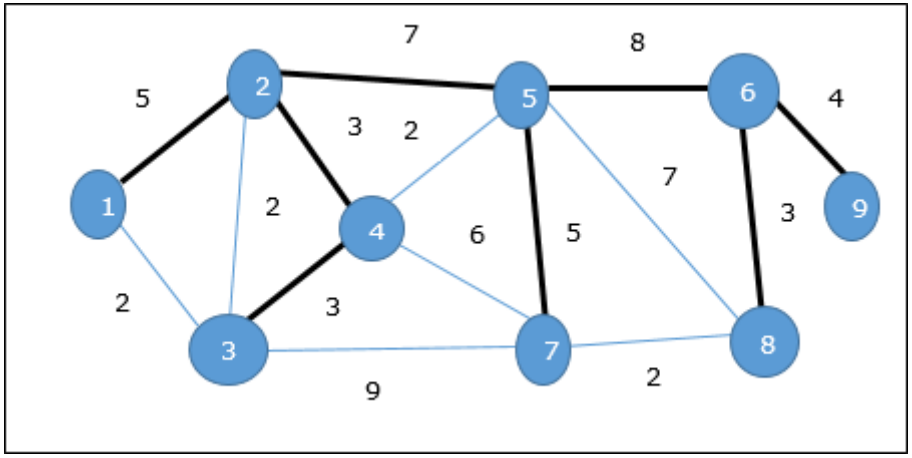
✓ Any vertex can be reached from any other vertex.

- Example:



Minimum Cost Spanning Trees contd..

- A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.
- As we have discussed, one graph may have more than one spanning tree. If there are **n** number of vertices, the spanning tree should have ***n* - 1** number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.
- Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree
- In graph shown below, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is (5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38.



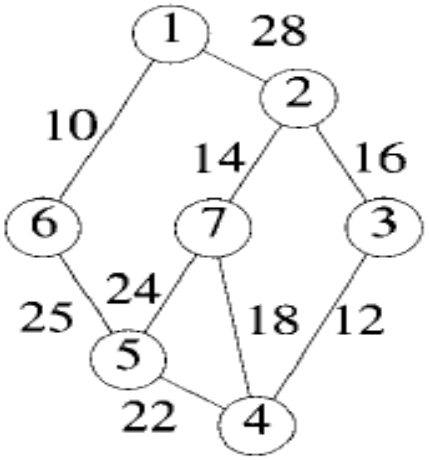
- The minimum cost spanning tree for a given graph is consturcted using two algorithms.

1. Prim's Algorithm
2. Kruskal's Algorthm

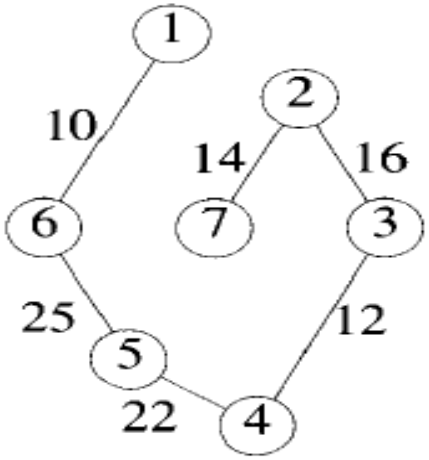
- We will discuss these two algorithms in Unit 3.

Prim's Algorithm Example

- Consider a graph shown in fig: a
- Spanning tree of the given graph in fig: b

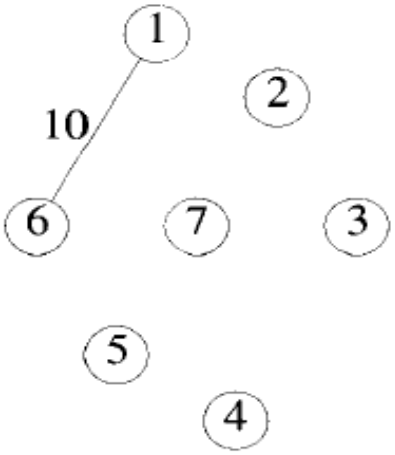


(a)

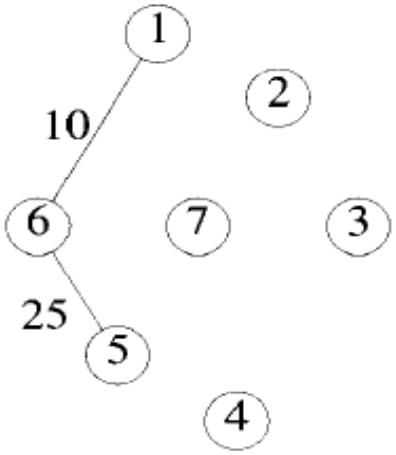


(b)

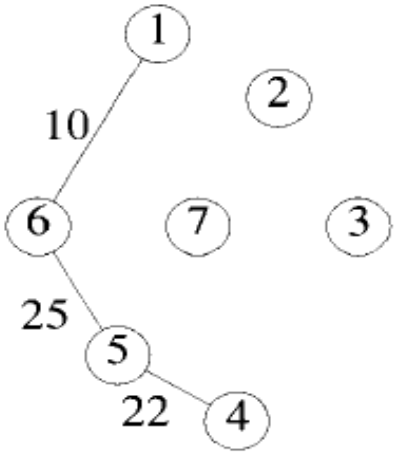
Prim's Algorithm Example (Stages step by step)



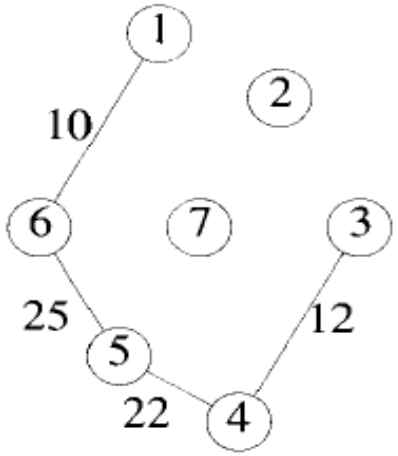
(a)



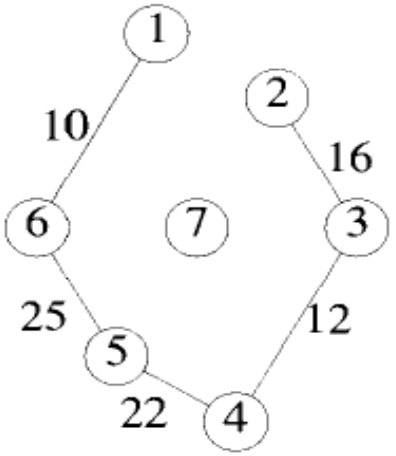
(b)



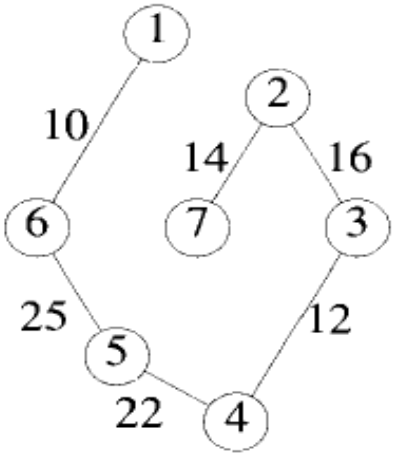
(c)



(d)



(e)



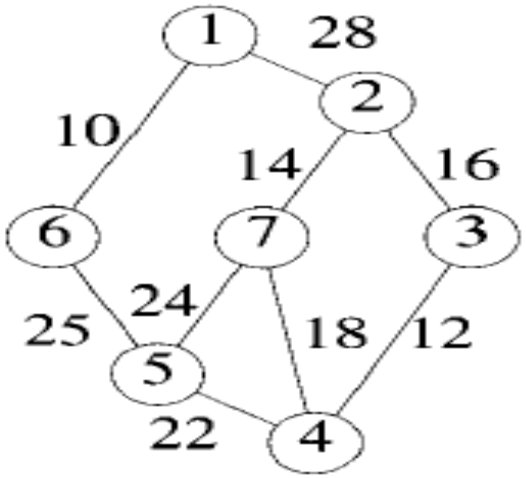
(f)

Prim's Algorithm

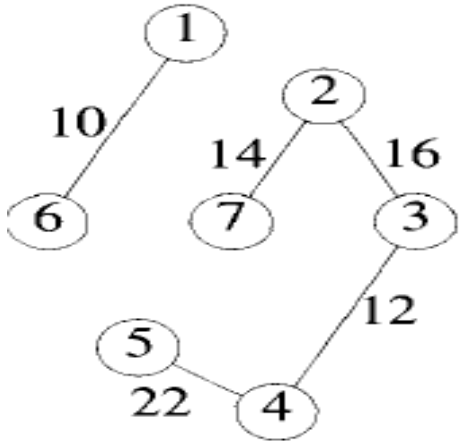
```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Kruskal's Algorithm Example

- Consider a graph shown in fig: a
- Spanning tree of the given graph in fig: b

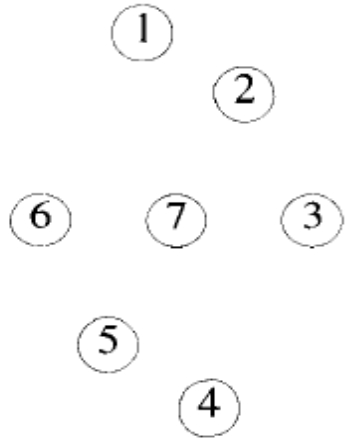


(a)

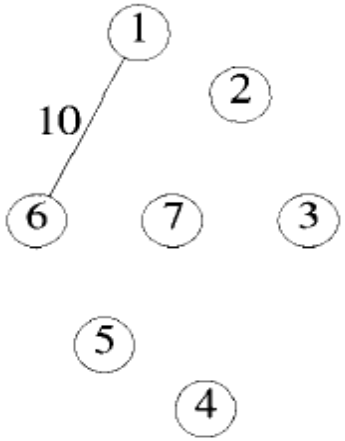


(b)

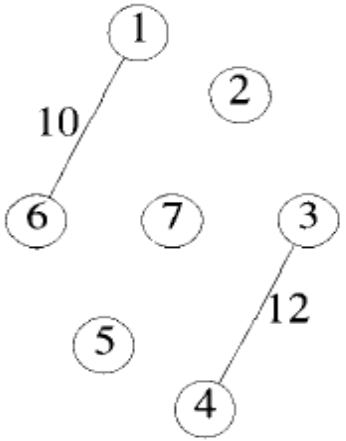
Kruskal's Algorithm Example (Stages step by step)



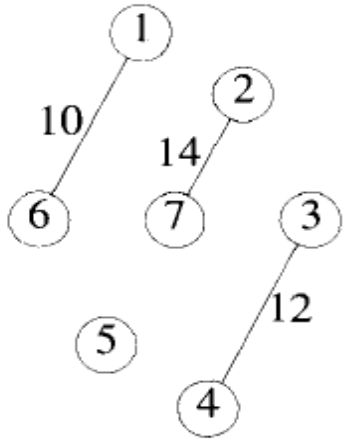
(a)



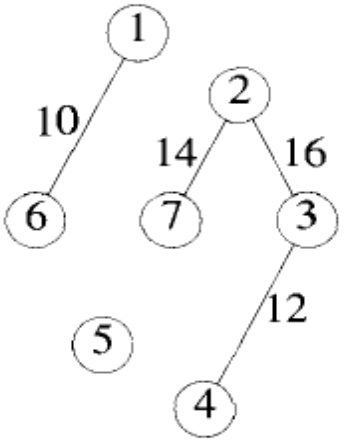
(b)



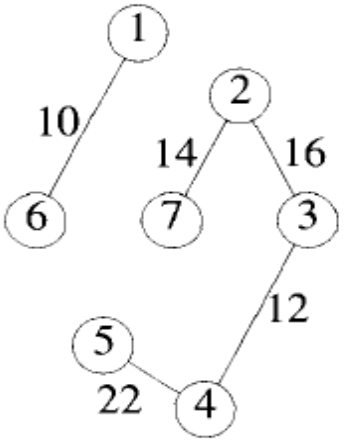
(c)



(d)



(e)



(f)

Kruskal's Algorithm

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```