GREEDY ALGORITHMS

AOA Module 2

INTRODUCTION

- The greedy method is one of the strategies like Divide and conquer used to solve the problems.
- The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique.
- Used for solving **optimization problems.**
- Decision is taken based on the currently available information.
- This technique is basically used to determine the feasible solution that may or may not be optimal.
- The **feasible solution** is a subset that satisfies the given criteria.
- The **optimal solution** is the solution which is the best and the most favorable solution in the subset.
- In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

INTRODUCTION

General Characteristics

1. **Greedy choice Property:**

   # Globally optimal solution can be arrived at by making a locally optimal choice.

   # Greedy choices are made one after another, reducing each given problem instance to smaller one.

   # Greedy choice property brings efficiency in solving the problem with help of subproblems

2. **Optimal Substructure:**

   # A problem shows optimal substructure if an optimal solution to the problem contains optimal solution to the subproblems.

INTRODUCTION

- **Candidate function:** The set of all possible elements is known as a configuration set, or a candidate set from which the solution is created.
- **Selection function:** The selection function tells which of the candidates is the most promising.
- **Feasible Solution**: For a given input n, we need to obtain a subset that satisfies some constraints. Then any subset that satisfies these constraints is called feasible solution.
- **Optimum solution:** The best solution from the feasible solution satisfying the objective function.
- **Objective function:** This gives the final value of the solution. It is a constraint that maximizes or minimizes the profit.

NOTE: The greedy choice may not necessarily produce the optimum solution.

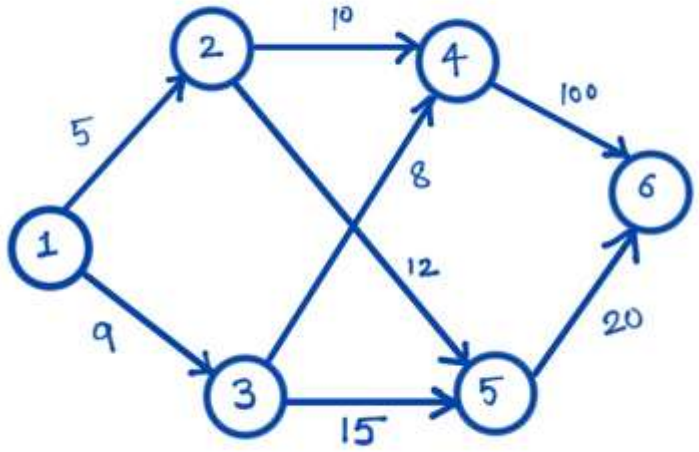
GREEDY ALGORITHM

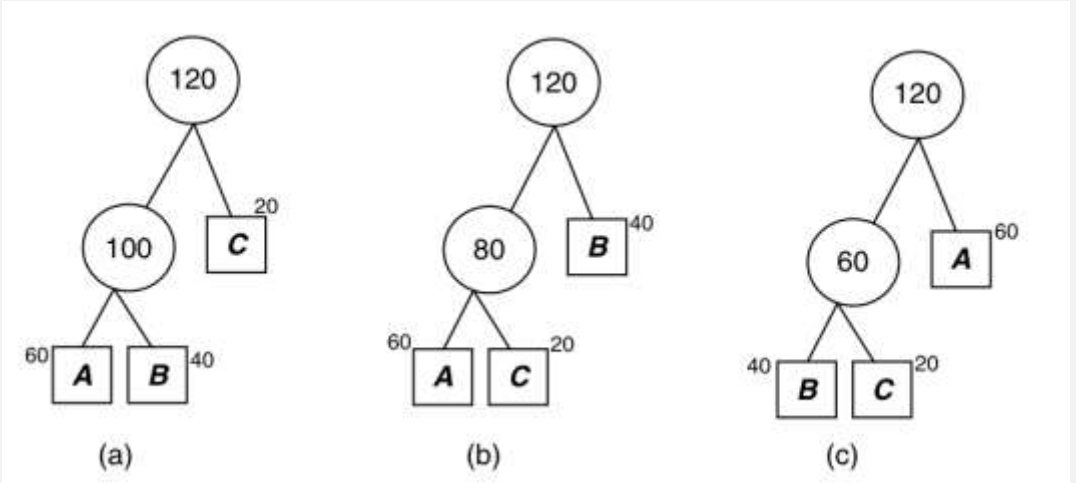
```
Greedy(int a[], int n)
{
/* the array a[1]...a[n] contains the input */
1.   solution=ϕ /* initialise the solution */
2.   for(i=1;i<=n;i++)
3.   {
4.      j=select(a);
5.      if(feasible(solution,j))
6. solution=union(solution,j);/*  extend  the
   solution */
7.   }
8.   return solution;
}/* End of Greedy(...) */
```

EXAMPLES

1. Coin Change Problem
2. Shortest path
3. Optimal merge Patterns

2.





APPLICATIONS

1. Optimal merge pattern.
2. Huffman coding.
3. Knapsack problem.
4. Minimum spanning tree.
5. Single-source shortest path.
6. Job scheduling with deadlines.
7. Travelling salesman problem.

KNAPSACK PROBLEM

- Based on the concept of maximizing the benefit.
- Knapsack is an empty bag.
- Size of the Knapsack is M.
- There are n items with weights $w_1, w_2, w_3, \ldots\ldots w_n$ respectively.
- Profit of each weight is $p_1, p_2, p_3, \ldots\ldots p_n$.
- Let $x_1, x_2, x_3, \ldots\ldots x_n$ be the fractions of items

- The objective is to fill the Knapsack that maximizes the total profit earned.
- Since Knapsack capacity is 'M', we require total weights of chosen items to be atmost 'M'

KNAPSACK PROBLEM

- The Knapsack problem has two versions:

1. **Fractional Knapsack**
   - List of items are divisible i.e. any fraction of item can be considered.
   - Let $x_j , j = 1,2,3, \ldots n$ be the fractions of item j taken into knapsack.
   - Where $0 \leq x_j \leq 1$
   - Mathematically, we can write as

$$\sum_{i=1}^{n} x_i w_i \leq M \quad \text{and} \quad \sum_{i=1}^{n} x_i p_i \text{ is maximized}$$

The greedy method to solve the fractional knapsack problem is summarized as follows:

1. **Compute the profit value ratio of each item x_i.**
2. **Arrange each item in ascending/descending order of their profit value ratio.**
3. **Select an item in order from the sorted list provided the selected item does not break the knapsack.**

KNAPSACK PROBLEM

2. **0/1 Knapsack**
   - Here, list of Items are indivisible i.e. item is either accepted or discarded.
   - $\sum_{i=1}^{n} x_i p_i$ is maximized, subjected to constraints,

$$\sum_{i=1}^{n} x_i w_i \leq M \text{ and } x_j \in \{0,1\} \text{ for } j = 1,2,3,\ldots n$$

KNAPSACK PROBLEM

- **There three greedy choices:**
  1. Arrange the items in descending order of their profit and then pick them in order.
  2. Arrange the items in ascending order of their weight or size and then pick them in order.
  3. Arrange the items in descending order of the ratio (profit/weight)and then pick them in order.

KNAPSACK PROBLEM

- **Example**

Find the solution of the knapsack problem for n=4, M=120

(p1,p2,p3,p4) = (40,20,35,50) and (w1,w2,w3,w4)=(25,30,40,45)

Item	1	2	3	4
w	25	30	40	45
p	40	20	35	50
p/w	1.6	0.66	0.875	1.11
x	1	1/3	1	1

FRACTIONAL KNAPSACK ALGORITHM

Algorithm 1 `FractionalKnapsack(S, W)`

Input: Set $S = \{i_1, i_2, i_3, \ldots, i_n\}$ of items. Each item is assigned with the value v_i and weight w_i. The maximum weight a knapsack can handle is W.
Output: Amount x_i of each item i to maximize benefit with weight at most W.

```
Repeat for each item i =1 to n
      set x_i ← 0        (selection part of item i is set to zero for every item initially)
      set p_i ← v_i / w_i        {profit value of each item}
End repeat
      set cw ← 0 and i ← 1        {set current total weight to 0 and start index to 1}

Repeat while cw < W and i<=n
      remove item i with highest profit value p_i from set S
      if (cw+w_i)<=W then
            set x_i ← 1 and cw ← cw +w_i
      else
            set x_i ← {W - cw}/w_i
            cw = W
      endif
      set i← i+1
      endwhile
return(x)
```

FRACTIONAL KNAPSACK ALGORITHM

- **Time Complexity:**

  Time complexity of the sorting + Time complexity of the loop to maximize profit

  = O(NlogN) + O(N) = O(NlogN)

- Space Complexity: O(1)

SINGLE SOURCE SHORTEST PATH

Introduction:

- Algorithm for finding shortest path from a starting node to a target node in a weighted graph
- Algorithm creates a tree of shortest path from starting vertex to all other nodes in the graph.
- Dijkstra's algorithm is generalization of BFS algorithm
- It can be applied on a weighted directed or undirected graph.
- It uses greedy method i.e. it always picks next closest vertex to the source.
- Disadvantage: Does not work with negative weights
- Application: Maps, Computer Networks-Routing and many more.

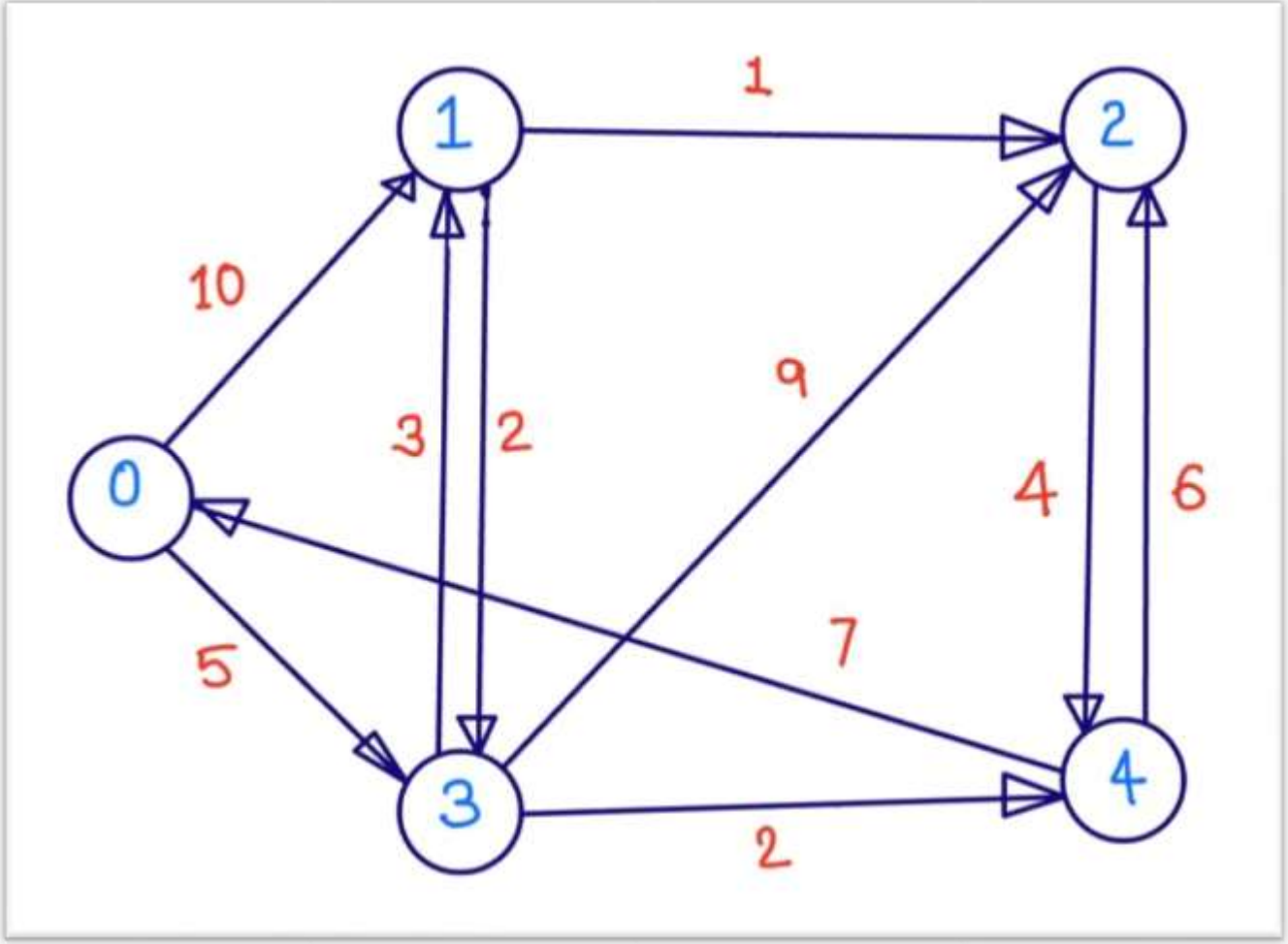
SINGLE SOURCE SHORTEST PATH: DIJKSTRA'S ALGORITHM

Algorithm:

```
Algorithm Dijkstra (v0, W, dist, n)
// v0 is the source vertex, W is the adjacency matrix to store the weights of
the links, dist[k] is the array to store the shortest path to vertex k, n is
the  number of vertices//
{
   for (i = 1 ; i < = n; i ++){
      S[i] = 0;                  // Initialise set S to empty, i.e., i is not
                                    inserted into the set//
      dist[i] = w(v0, i);        //Initialise the distance to each node//
   }
   S[v0] = 1;  dist[v0] = 0;
   for (j = 2; j <= n; j++) {
      Choose a vertex u from those vertices which are not in S such that dist
      [u] is minimum.
      S[u] = 1;
      for (each z adjacent to  u with S[z] = 0) {
        if (dist [z] > dist [u] + w[u, z])
           dist [z] = dist [u] + w[u, z];
      }
   }
}
```

DIJKSTRA'S ALGORITHM

Example:



Adjacency Matrix:

	0	1	2	3	4
0	0	10	∞	5	∞
1	∞	0	1	2	∞
2	∞	∞	0	∞	4
3	∞	3	9	0	2
4	7	∞	6	∞	0

DIJKSTRA'S ALGORITHM

Example:

Relaxation Formula:

If dist[z] > dist[u] + w[u,z], then

dist[z] = dist[u] + w[u,z]

Table 6.4 Shortest path from source vertex 0

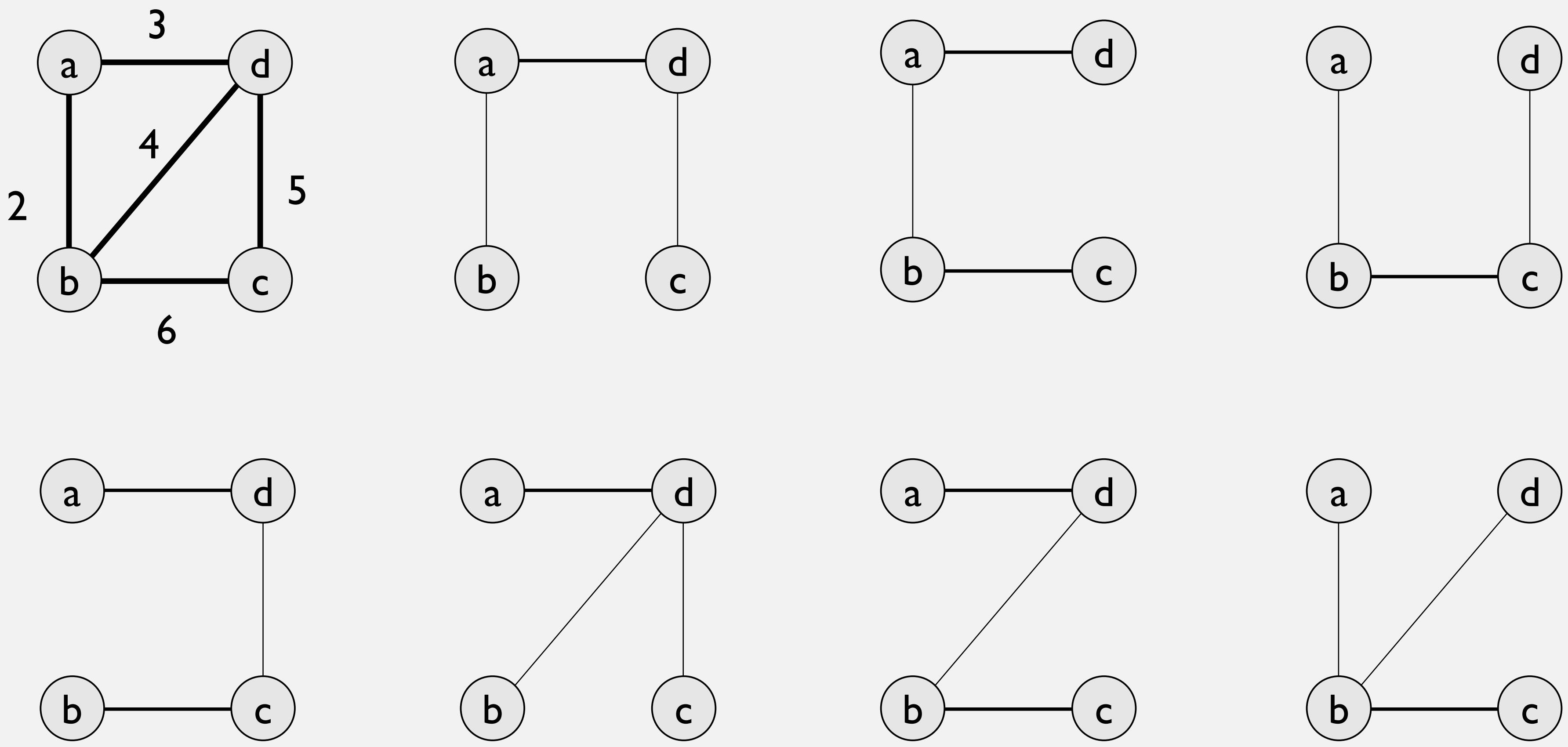
Iteration	*Set of nodes to which the shortest path is found*	*Vertex selected*	*Distance*				
			0	1	2	3	4
Initial	{0}	-	0	10	∞	5	∞
1	{0, 3}	3	0	8	14	5	7
2	{0, 3, 4}	4	0	8	13	5	7
3	{0, 3, 4, 1}	1	0	8	9	5	7
4	{0, 3, 4, 1, 2}	2	0	8	9	5	7

The time complexity of the algorithm is $O(n^2)$.

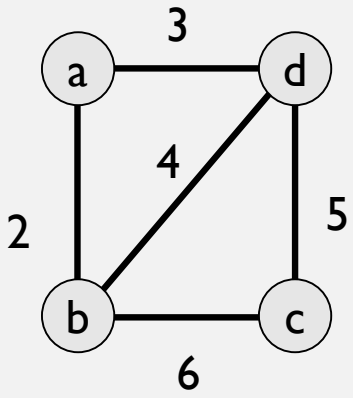
MINIMUM SPANNING TREE

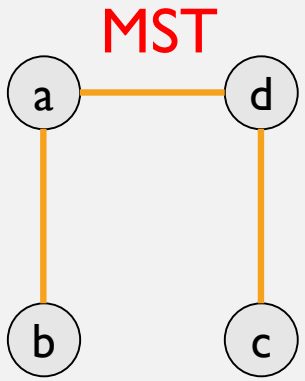
- A spanning tree is defined as a subgraph of an undirected graph in which all the vertices are connected.
- A tree consisting of all the vertices of a graph is called a spanning tree.
- The weight of spanning tree for a weighted graph is the sum of all the edge weights.
- The spanning tree with minimum total weight is called the minimum spanning tree(MST).
- MST is a minimal graph G' of a graph G, such that $V(G') = V(G)$ and $E(G') \subseteq E(G)$
- If there are N vertices in the graph, then the MST contains $(N-1)$ edges.

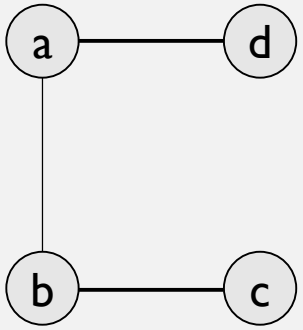
MINIMUM SPANNING TREE

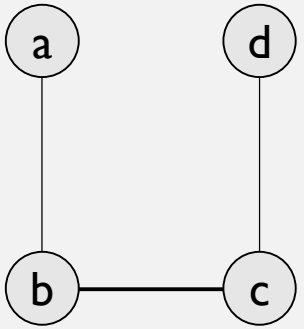


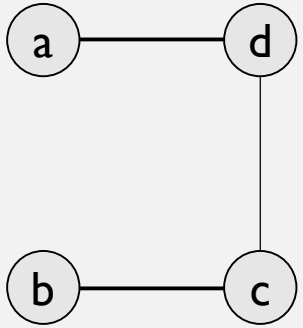
MINIMUM SPANNING TREE

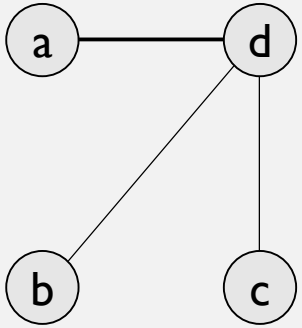


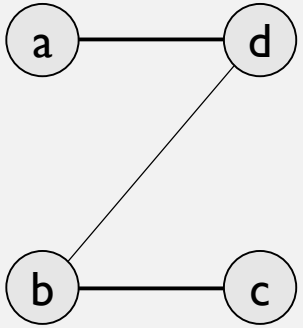


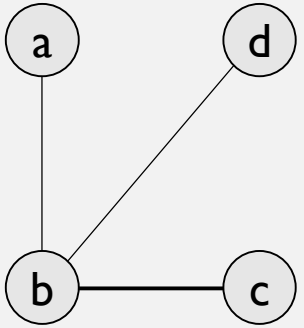












MINIMUM SPANNING TREE

The following are the two different algorithms to find out the MST:

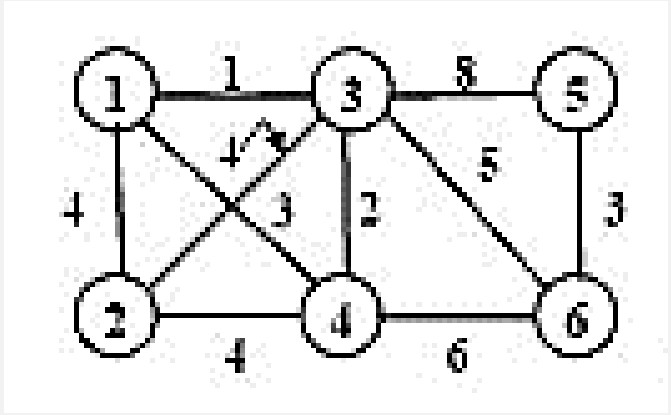
1. Prim's Algorithm
2. Kruskal's Algorithm

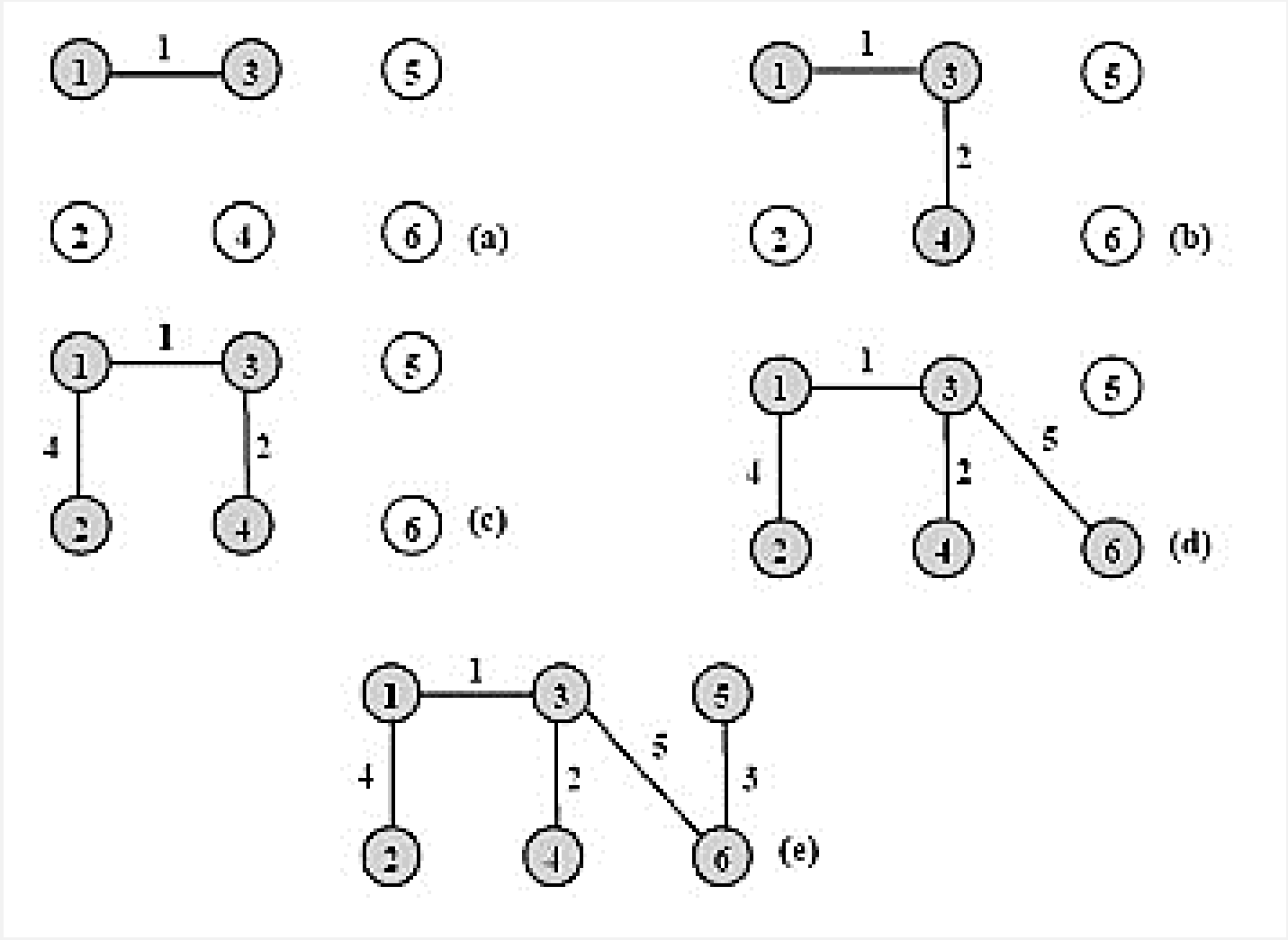
PRIM'S ALGORITHM

- Prim's algorithm starts by selecting a node arbitrarily.
- Prim's algorithm builds the MST by iteratively adding nodes into a working tree.
- A tree on n nodes has n-1 edges.
- Steps for finding MST using Prim's:
  1. Start with a tree which contains only one node.
  2. Identify a node(outside the tree) which is closest to the tree and add the minimum cost edge from that node to some node in the tree and incorporate additional node as a part of the tree.
  3. If there are less than n-1 edges in the tree, go to step 2.

PRIM'S ALGORITHM

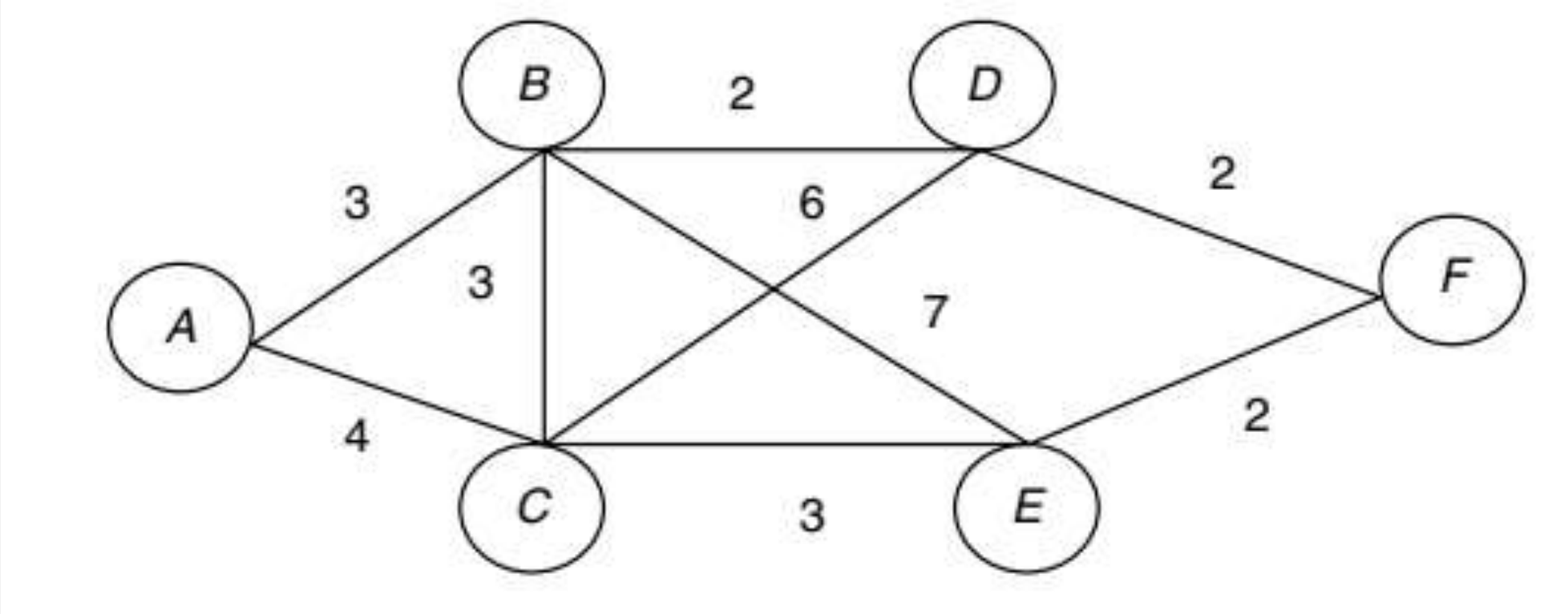
Example 1:





PRIM'S ALGORITHM

Example 2:



PRIM'S ALGORITHM

Algorithm Prim(adjMatrix[1 .. n] [1 .. n])

n *= Number of nodes in the graph*

nEdges *= Current number of edges included in the MST.*

min *= Minimum weight (cost) of edge.*

row, col *indicate the row and column of the edge included in the MST.*

sum *= Total minimum weight (cost) of the edges included in the MST.*

Initialize selected[] with zeros.

PRIM'S ALGORITHM

Algorithm Prim(adjMatrix[1 .. n] [1 .. n])

n = *Number of nodes in the graph*
nEdges = *Current number of edges included in the MST.*
min = *Minimum weight (cost) of edge.*
row, col *indicate the row and column of the edge included in the MST.*
sum = *Total minimum weight (cost) of the edges included in the MST.*
Initialize selected[] with zeros.

```
selected[1] = 1; // To start with, node 1 is included in MST
nEdges = 1;      // Make one count of edges to be included in MST
sum = 0;
While( nEdges<n)
min = ∞;
/* Repeat following steps to cover all nodes from 1 to n */
For i = 1 to n do
If( selected[i]= 1)
/* If node i is included in MST, then repeat to get min-cost node j to
connect to i */
for j = 1 to n do
If( selected[i]= 0 )     // Select the min-cost edge
If( min > adjMatrix[i][j] )
    min = adjMatrix[i][j];
    row=i; col=j;
    Endif
Endif
Endfor
Print row, col, adjMatrix[row][col];
Endfor
selected[col] = 1;   // Once i and j form a min-edge, mark 1
selected
nEdges++;
sum = sum + adjMatrix[row][col];
Endwhile.
Print sum;           // Total Min-Cost
```

https://somaiya0-my.sharepoint.com/:w:/g/personal/gopal_s_somaiya_edu/EaGquP2a38ROiwUBg1izvZMB-ONtu90-tIjlUuAlMWlMaA?e=WhtPgA

KRUSKAL'S ALGORITHM

- Developed by Joseph Kruskal in 1957.
- The Kruskal's algorithm creates the MST *"T"* by adding the edges with minimum cost one at a time to *T*.
- A minimum cost spanning tree *T* is built edge by edge without forming the cycle.
- Start with the edge of minimum cost.
- If there are several edges with the same minimum cost, then we select any one of them and add it to the spanning tree *T* provided its addition does not form a cycle.
- Then add an edge with the next lowest cost and so on.

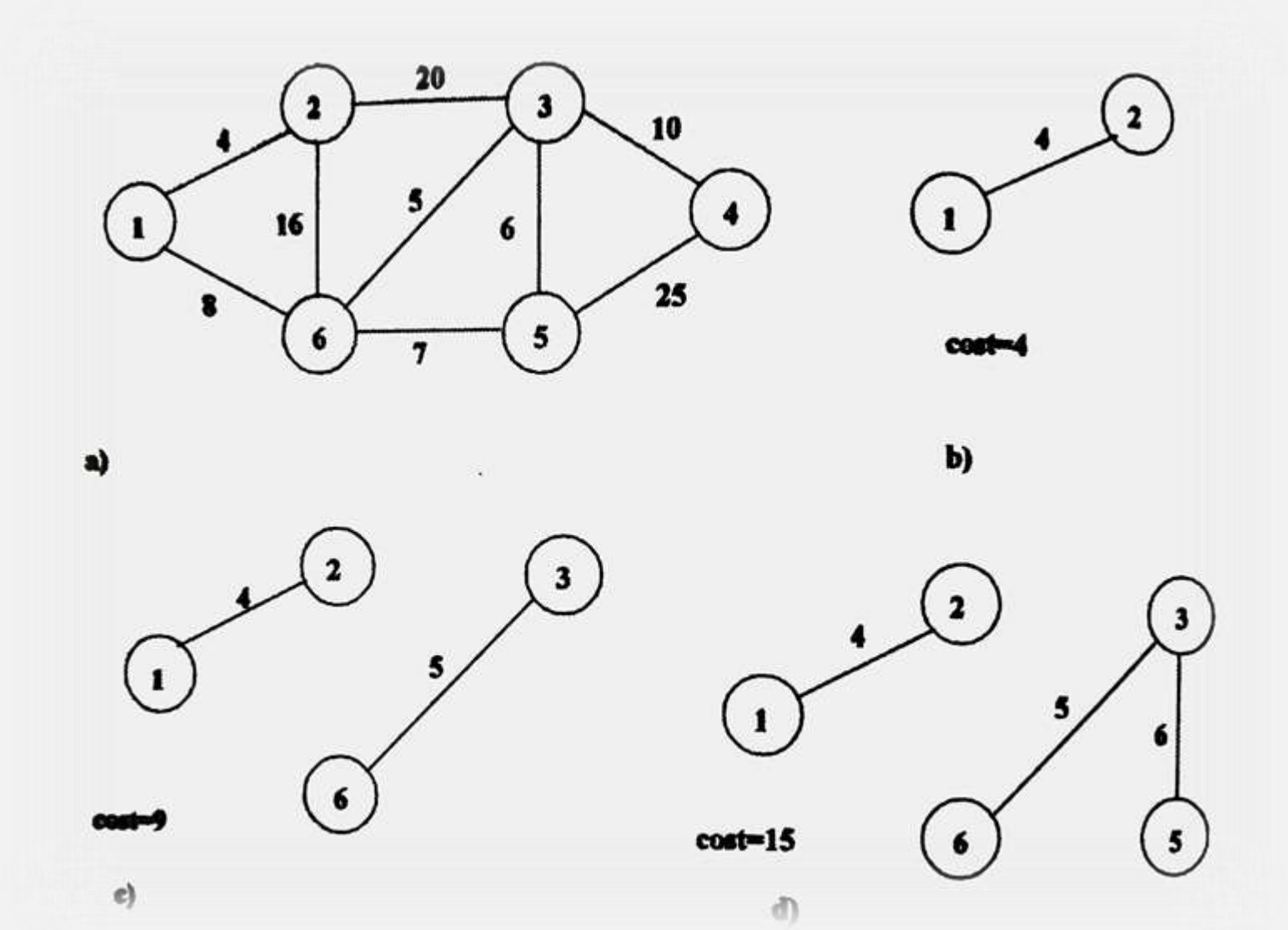
KRUSKAL'S ALGORITHM

- Repeat this process until we have selected $N-1$ edges to form the complete MST.
- Edges can only be added if they do not form a cycle.
- In Kruskal's Algorithm, edges are considered in nondecreasing order of their costs.
- In contrast to Prim's, there may be forest at different stages of the algorithm, which get converted into tree

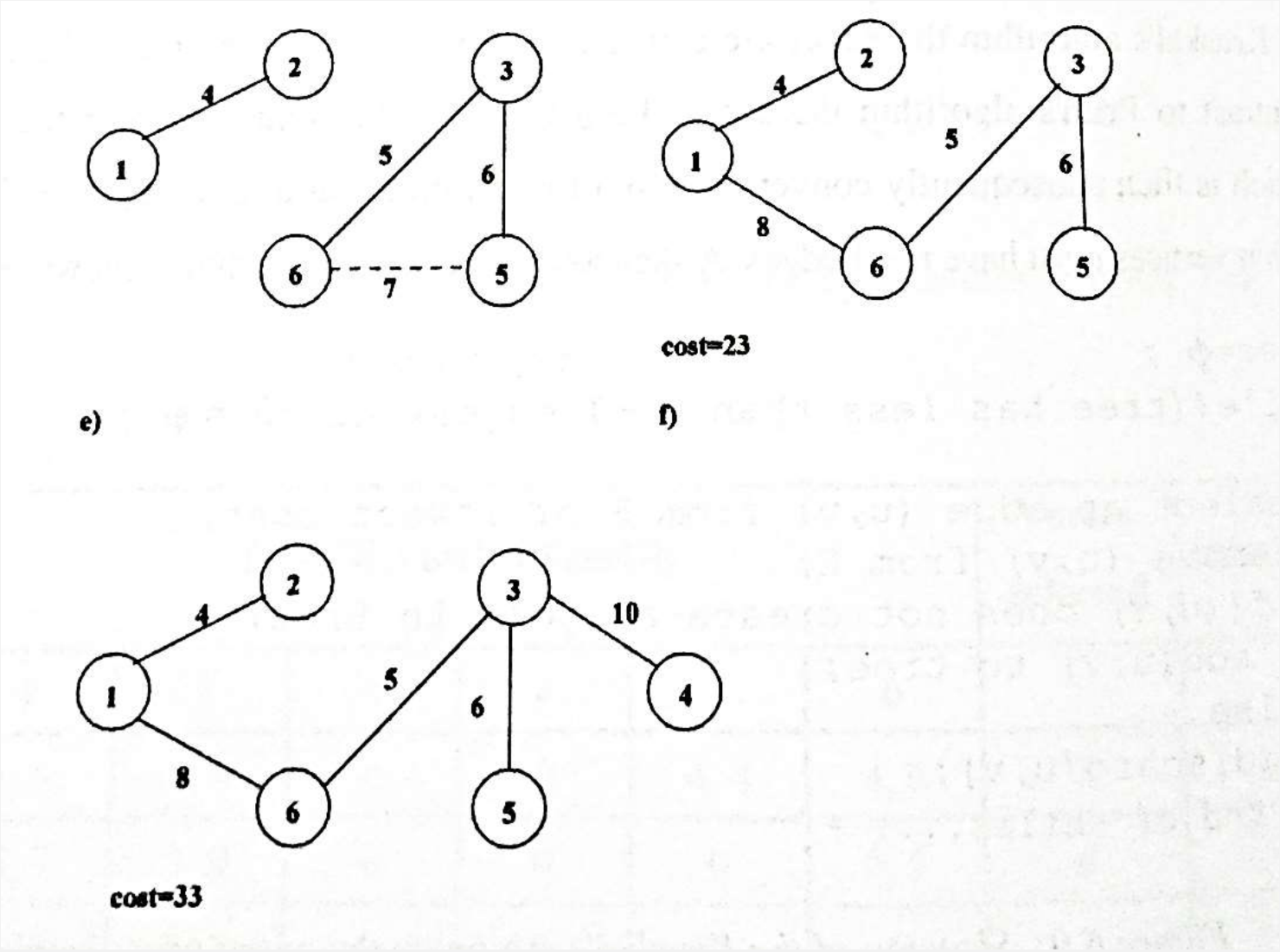
KRUSKAL'S ALGORITHM

```
Algorithm Kruskal
   tree = ∅;
   While((tree has less than n-1 edges) && (E≠ ∅))
   {
      select an edge (u,v) from E of lowest cost;
      remove(u,v) from E;
      if((u,v) does not create a cycle in tree)
         add(u,v) to tree;
      else
         discard(u,v);
   }
```

KRUSKAL'S ALGORITHM



KRUSKAL'S ALGORITHM



COIN CHANGE PROBLEM

- Coin change is a classic example of the greedy approach.
- let us assume that we have coins of different denominations 1000, 500, 100, 50, 20, 10, 5, 2, and 1. The problem is to minimize the number of denominations.
- One can observe the following components that are inherent in the problem:
- **Objective function** This is to minimize the number of coins for a request.
- **Candidate solutions** These involve all possible solutions involving coins of different denominations.
- **Selection procedure** Like a greedy man, the selection procedure should look for coins of the largest denomination to fulfil the request and thereby to minimize the number of coins.