Unit 3 (Part-I): Greedy Algorithms

Expected Outcomes of topic (CO)

- Understand problem and its formulation to design an algorithm
- Demonstrate the knowledge of basic data structures and their implementation and decide upon use of particular data structure best suited for implementing solution.
- Write efficient algorithms using Greedy, Divide & Conquer to solve the real life problems.
- Trace logic formulation, execution path of particular algorithm and data generated during execution of algorithm.

Basic Characteristics – Greedy algorithms

- Solution is built in small steps
- Decisions on how to build the solution are made to maximize some criterion without looking to the future
  - Want the 'best' current partial solution as if the current step were the last step

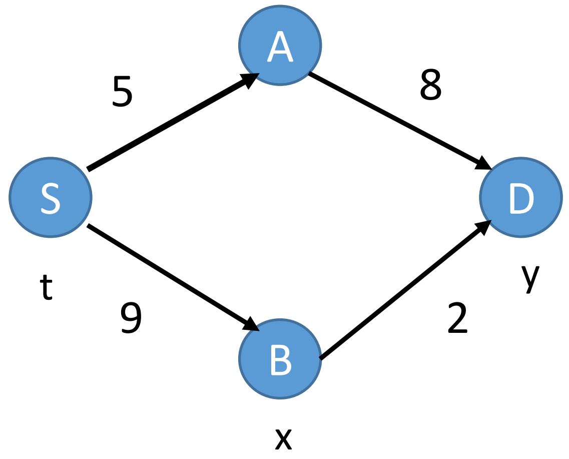
Greedy is a strategy that works well on optimization problems with the following characteristics:

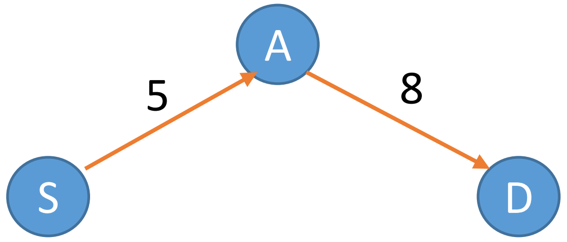
1. **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum.
2. **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblems.

- Greedy algorithms
  - Easy to produce
  - Fast running times
  - Work only on certain classes of problems

Greedy Algorithms

- Characteristics
  - Greedy Algorithms are short – sighted.
  - Greedy Algorithms are most efficient if it works.
  - For every instance of input Greedy Algorithms makes a decision and continues to process further set of input.
  - The other input values at the instance of decision are lost and not used in further processing.

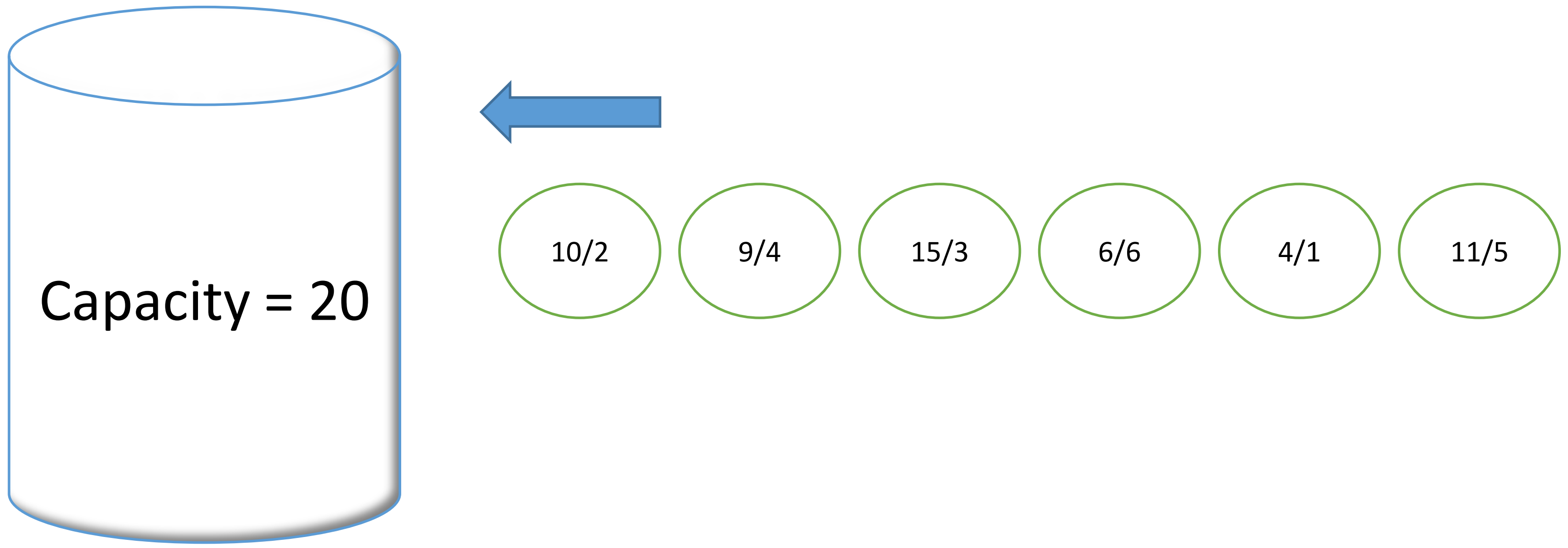




Greedy Algorithm: Knapsack Problem

- Principle: **Knapsack is a bag of fixed capacity.**
- **In this problem it is assumed that “n” objects with profit value and weight are given.**
- **The main objective is to select the objects and place them in knapsack such that the object placed in the bag will generate maximum profit.**
- Two types of Knapsack problems:
  - Fractional Knapsack problem
  - 0/1 Knapsack problem
- The Fractional Knapsack problem can be solved using Greedy approach, where as 0/1 Knapsack problem does not have greedy solution.

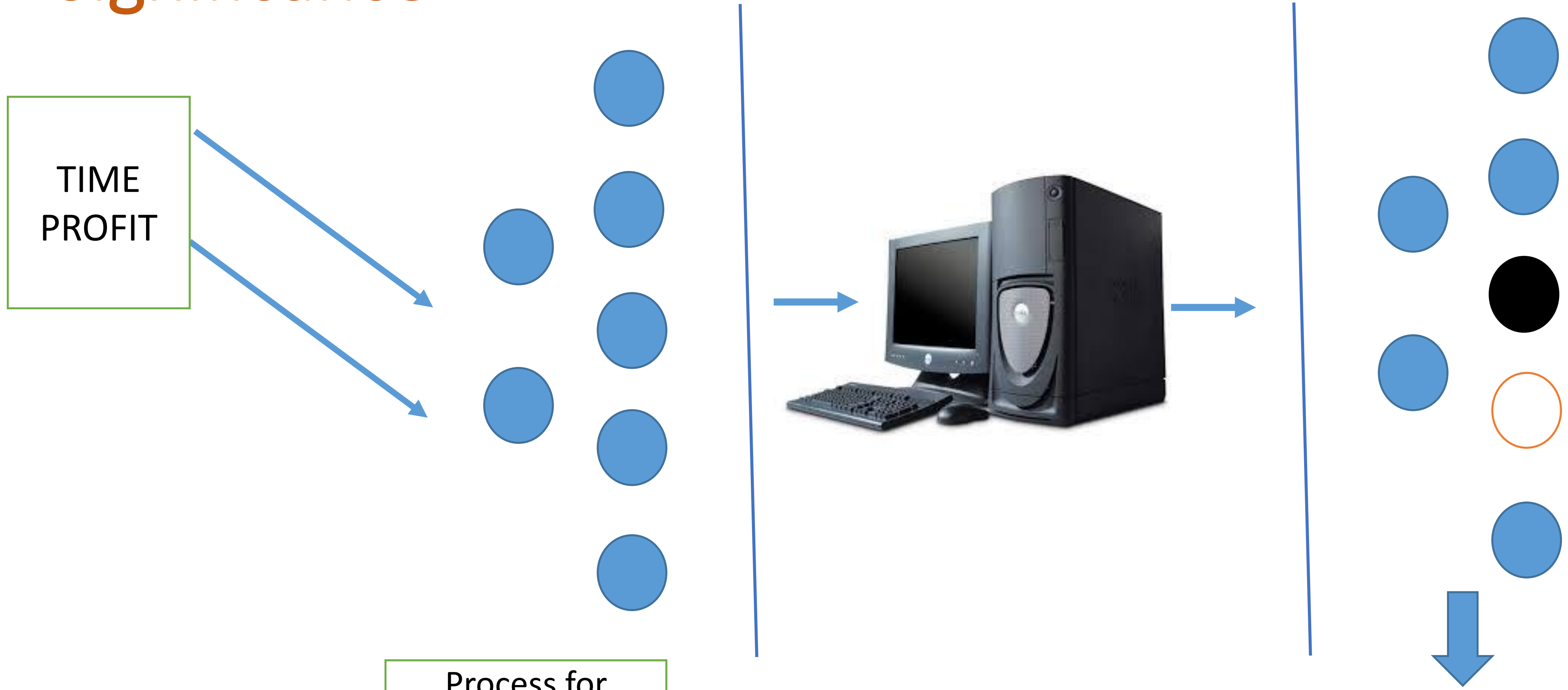
Representation



Greedy Algorithm: Knapsack Problem

- **Approaches:**
- There are three basic approaches to solve knapsack problem:
  - Select the maximum profit objects
  - Select the minimum weight objects
  - Select the objects based on Profit/Weight ratio

Significance



Generate maximum Profit in available time by proper selection of processes

Greedy Algorithm: Knapsack Problem

Example 1: Capacity = 15, number of objects = 7

Profit	10	5	15	7	6	18	3
Weight	2	3	5	7	1	4	1
Index	1	2	3	4	5	6	7

Approach: Maximum Profit				
Object	Weight	Profit	Capacity Remaining	Partial/Complete
O6	4	18	15-4=11	C
O3	5	15	11-5=6	C
O1	2	10	6-2=4	C
O4*	4	4	4-4=0	P
TOTAL	**15**	**47**	**0**	

Greedy Algorithm: Knapsack Problem

Example 1: Capacity = 15, number of objects = 7

Profit	10	5	15	7	6	18	3
Weight	2	3	5	7	1	4	1
Index	1	2	3	4	5	6	7

Approach: Minimum Weight				
Object	Weight	Profit	Capacity Remaining	Partial/Complete
O5	1	6	15-1=14	C
O7	1	3	14-1=13	C
O1	2	10	13-2=11	C
O2	3	5	11-3=8	C
O6	4	18	8-4=4	C
O3*	5(4)	12	4-4=0	P
TOTAL	**15**	**54**	**0**	

Greedy Algorithm: Knapsack Problem

Example 1: Capacity = 15, number of objects = 7

Profit	10	5	15	7	6	18	3
Weight	2	3	5	7	1	4	1
Ratio	**5.0**	**1.6**	**3.0**	**1.0**	**6.0**	**4.5**	**3.0**
Index	1	2	3	4	5	6	7

Approach: Maximum Pi/Wi ratio				
Object	Weight	Profit	Capacity Remaining	Partial/Complete
O5	1	6	15-1=14	C
O1	2	10	14-2=12	C
O6	4	18	12-4=8	C
O3	5	15	8-5=3	C
O7	1	3	3-1=2	C
O2*	3(2)	10/3=3.33	2-2=0	P
TOTAL	**15**	**55.3**	**0**	

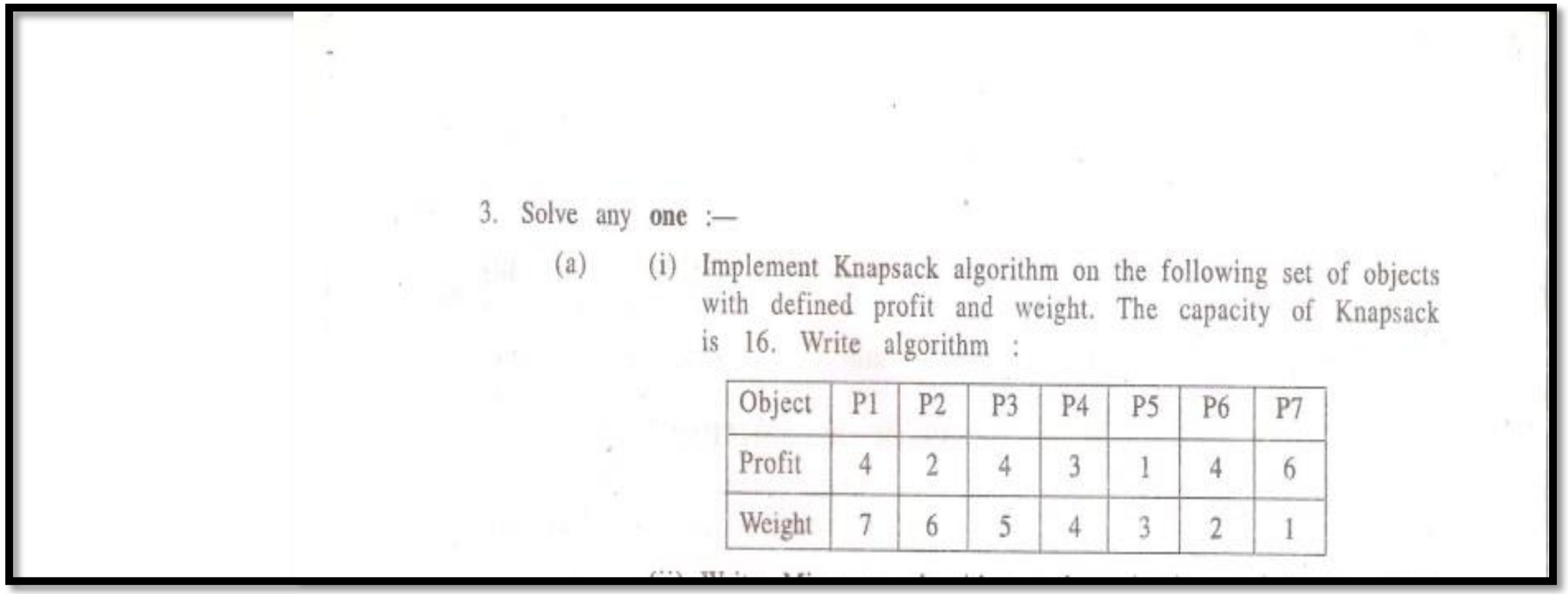
Greedy Algorithm: Knapsack Problem

Example 2: Capacity = 18, number of objects = 7

Pi	**9**	**15**	**12**	**4**	**6**	**16**	**8**
Wi	**2**	**3**	**5**	**4**	**3**	**6**	**3**
Ratio	**4.5**	**5.0**	**2.4**	**1.0**	**2.0**	**2.6**	**2.6**
Index	1	2	3	4	5	6	7

Maximum (Pi/Wi) Ratio				
Object	Weight	Profit	Capacity Remaining	Partial/Complete

Exam Questions



3. Solve any **one** :—

(a) (i) Implement Knapsack algorithm on the following set of objects with defined profit and weight. The capacity of Knapsack is 16. Write algorithm :

Object	P1	P2	P3	P4	P5	P6	P7
Profit	4	2	4	3	1	4	6
Weight	7	6	5	4	3	2	1

Knapsack Problem: Algorithm

- Data Structures:
    - Weight array: W[n] → To store weight values
    - Profit array: P[n] → To store profit values
    - Capacity → Given and will decrease after each step
    - X[n] → To store 0, 1 or fraction depending upon object placement
    - Weight = 0 (initially) and will increase towards capacity
    - Profit = 0 and will be calculated by X[i] and P[i]
- Process: To select the object with best Pi/Wi Ratio. (Sorting)

Knapsack Algorithm

- **<u>Initialization steps:</u>**
  - Profit
  - Weight
  - x[1..n] ➔ to hold data about object placement [partly, completely, rejected]
- **<u>Decision step</u>**
  - While (condition   )
  -     Select object //Call function for selecting best suited object//
  -     Place in bag {Completely / Partially }
  -     *Update x [i]*
- **<u>Final step: Find profit</u>**
- Multiply x[i] and associated profit

```
Algorithm Knapsack (p, w, n, Capacity : profit,x)
{Step 1
    profit = 0;
    for i = 1 to n do
        x[i] = 0;
    weight = 0;
    While (weight < capacity) do
    {
        Select an object with best pi/wi ratio and let the index of
object be "i"
        if (weight + w[i] <= capacity) then
        {
            x[i] = 1;
            weight = weight + w[i];
        }
        else
        {
        x[i] = ( capacity – weight )/w[i];
            weight = capacity;
        }
    } //End of while loop
STEP 2   for  i  = 1 to n do
    profit = profit + (p[i] * x[i])
} //End of algorithm
```

Greedy Algorithms on Graph

Prim’s Algorithm

Reverse Delete Algorithm

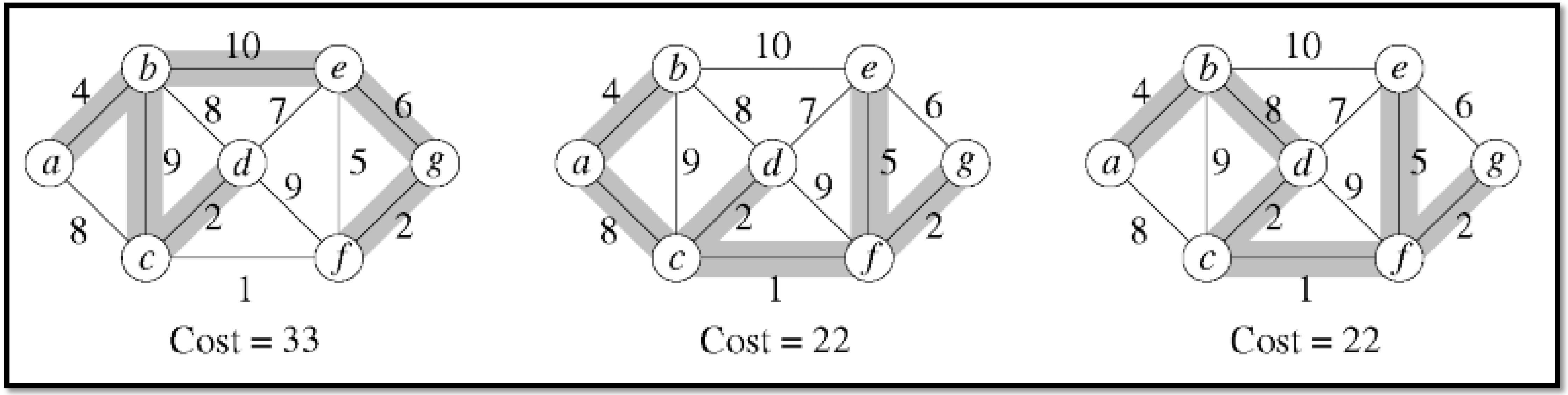
Minimum Cost Spanning Tree

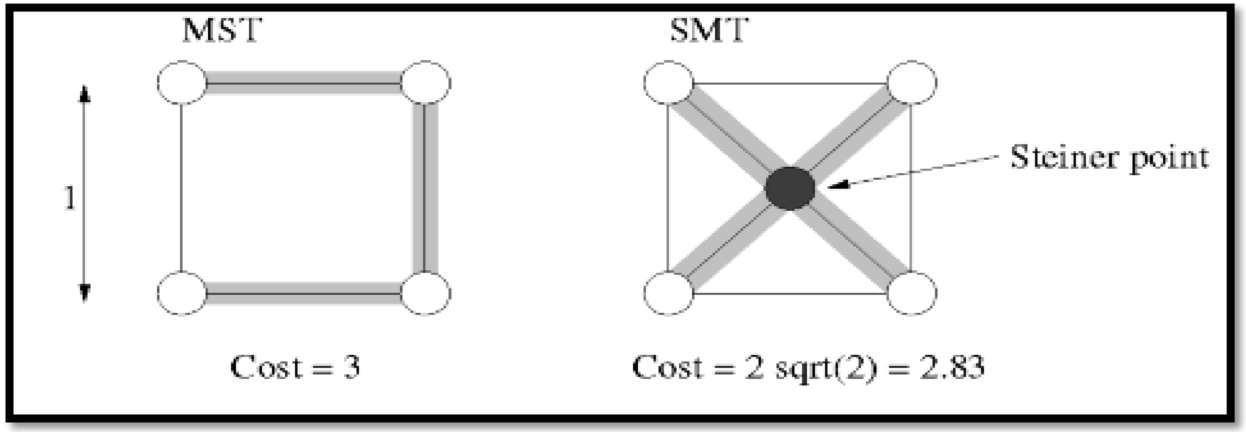
- Given a graph G =(V,E) V=Number of vertices and E=Number of edges, then Spanning tree is tree generated from graph, with characteristics
  - All the vertices present in the tree
  - There is no cycle in the tree i.e., e=v-1
- Subset of edges, that forms a tree, where total cost of edges is minimum.
- *Minimum cost spanning tree is a spanning tree with "edges of minimum cost".*
- **Applications:**
- To transmit the message without broadcasting.
- To generate travel plan in minimum cost.
- Designing network, home electric wiring etc.
- Approaches: Prims' Method and Kruskals' Method.

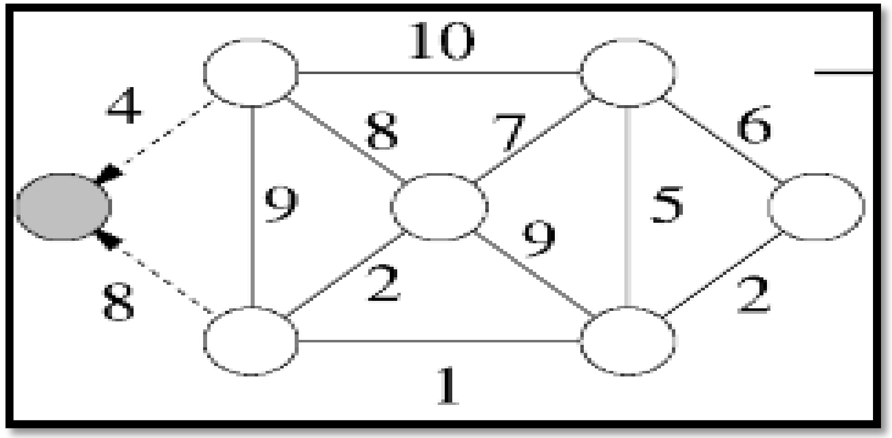
Minimum Cost Spanning Tree

- Principle:
  - Select an edge of minimum cost. The edge will derive two vertices.
  - Continue the process from one of the vertex by selecting the next edge of minimum cost.
  - Once the vertex is visited, then it is marked.
  - The process will attempt to visit unmarked vertices and terminates when all the vertices are visited.
  - Process guarantees no cycle in the tree.
- *"What if the process of tree generation starts from any arbitrary vertex".*
- **Motivation: To join points as cheaply as possible: Applications in clustering and networking**
- **Acyclic graph to connect all nodes in minimum cost.**
- **One of the term in U.S. legal code (AT&T)**

Graphs for Examples







Spanning tree: Free tree

- A free tree has following properties

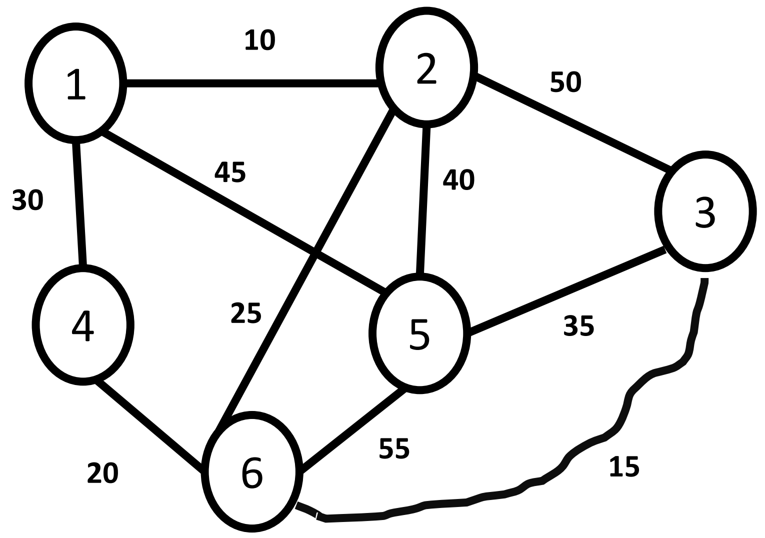
1. Exactly n-1 edges for “n” vertices
2. There exists a unique path between two vertices.
3. By adding an edge, a cycle will be created in free tree. Breaking any edge on the cycle restores the free tree.

- *Greedy: Minimization problem.*

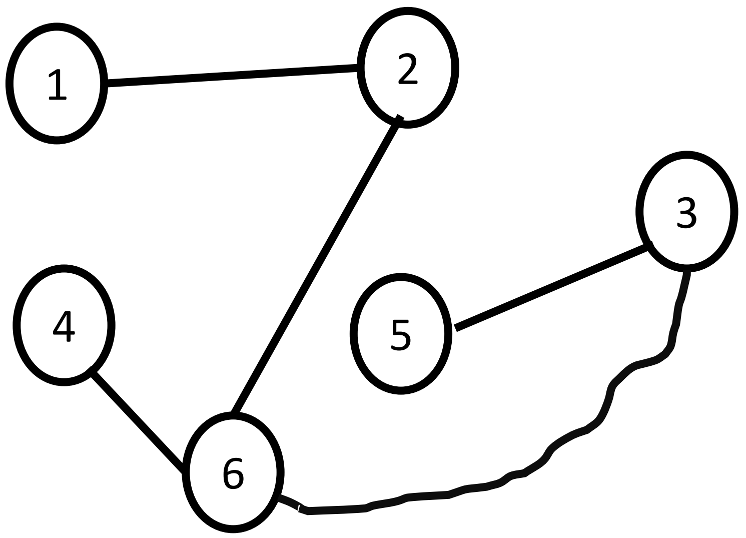
1. Repeated selection: of minimum cost edges with certain test cases.
2. Once decision of adding edge is finalized, it cannot be revoked.
3. Basic idea: To generate subset of “E” connecting all “V”

Minimum Cost Spanning Tree: Prim's method

- Example:
- Consider the following graph:



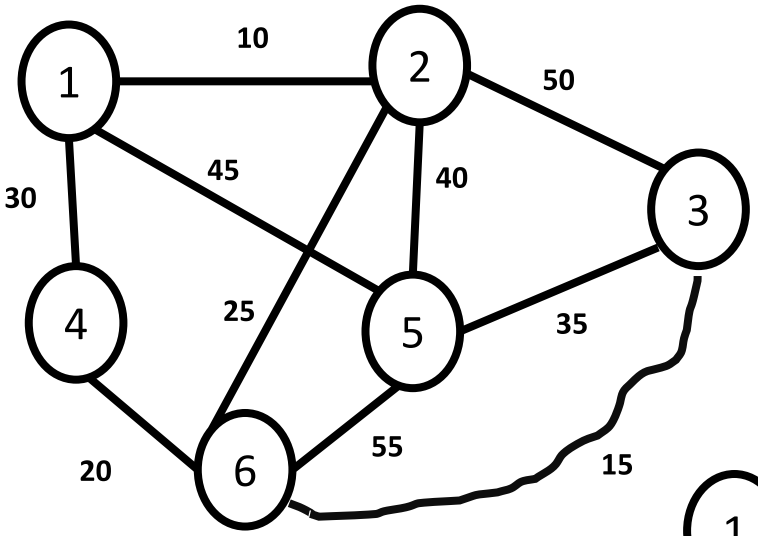
	1	2	3	4	5	6
1	99	10	99	30	45	99
2	10	99	50	99	40	25
3	99	50	99	99	35	15
4	30	99	99	99	99	20
5	45	40	35	99	99	55
6	99	25	15	20	55	99



- ***Select an edge of minimum cost.***
- ***From selected vertex continue selecting edge of minimum cost***

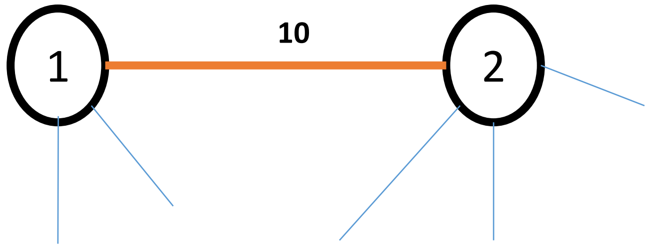
Minimum Cost Spanning Tree: Prim’s method

- Example:
- Consider the following graph:



	1	2	3	4	5	6
1	99	10	99	30	45	99
2	10	99	50	99	40	25
3	99	50	99	99	35	15
4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
6	99	25	15	20	55	99

The next vertex should be the vertex reachable in minimum cost either from vertex 1 or vertex 2



1	2	3	4	5	6
-	-	2	1	2	2

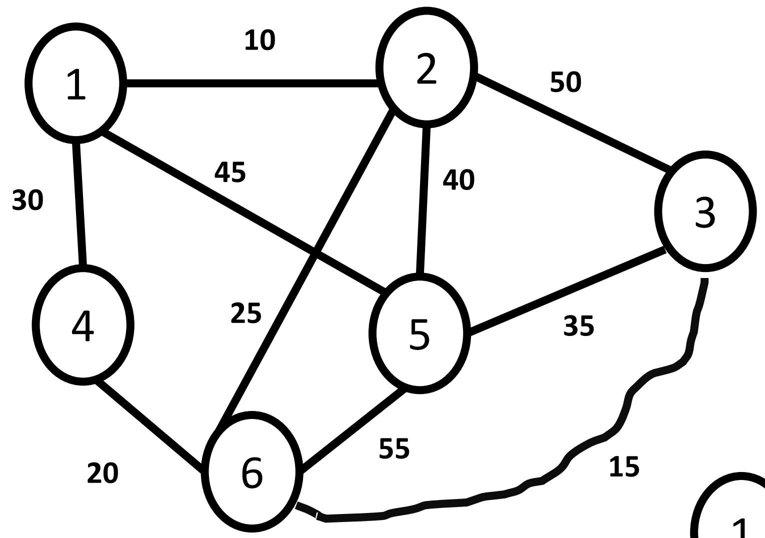
	T	C		F	R
1	1	-	1	2	
2	1	-	2	2	
3	1	<	3	2	F
4	1	<	4	2	T
5	1	<	5	2	F
6	1	<	6	2	F

Out of Four possible options: cost[near[j],j] = minimum
Select the index satisfying above condition: 6

The vertex selected is “6” due to which there may be changes in closeness of vertices.
That the vertices which were close to vertex 1 or vertex 2 may modified due to selection of vertex 6.

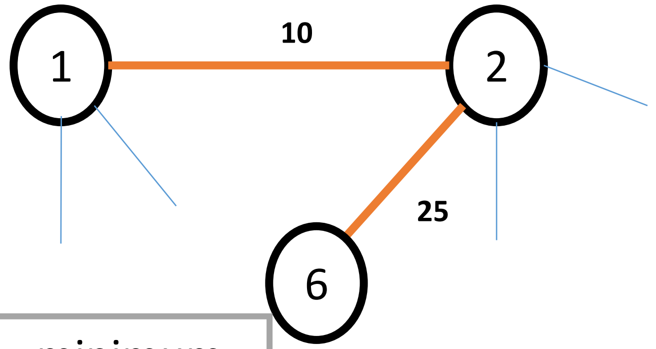
g Tree: Prim’s method

- Consider the following graph:



	1	2	3	4	5	6
1	99	10	99	30	45	99
2	10	99	50	99	40	25
3	99	50	99	99	35	15
4	30	99	99	99	99	20
5	45	40	35	99	99	55
6	99	25	**15**	**20**	55	99

1	2	3	4	5	6
-	-	6	6	2	



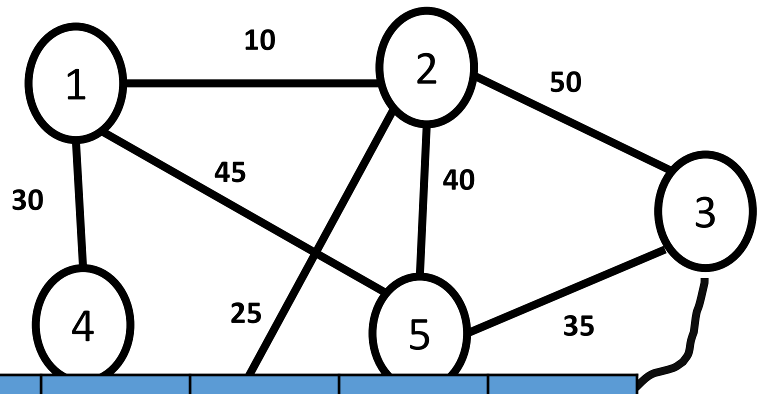
	T	C		F	R
1		-	1		
2		-	2		
3	6	<	3	2	T
4	1	<	4	6	F
5	6	<	5	2	F
6		-	6		

Out of Four possible options: cost[near[j],j] = minimum
Select the index satisfying above condition: 3

g Tree: Prim's method

The vertex selected is "3" due to which there may be changes in closeness of vertices.
That the vertices which were close to vertex 1 or vertex 2 or vertex 6 may modified due to selection of vertex 3.

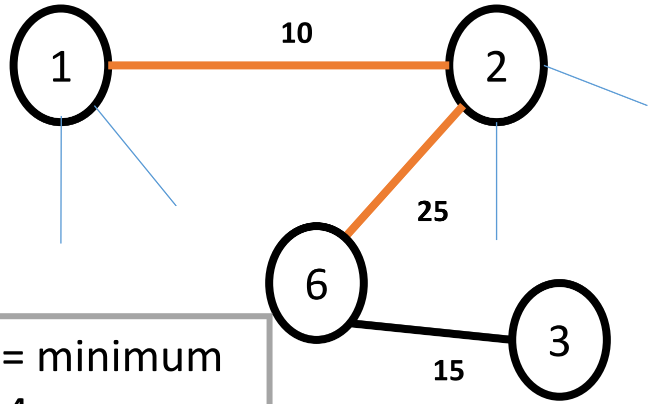
- Consider the following graph:



	1	2	3	4	5	6
1	99	10	99	30	45	99
2	10	99	50	99	40	25
3	99	50	99	99	35	15
4	30	99	99	99	99	20
5	45	40	35	99	99	55
6	99	25	**15**	**20**	55	99

1	2	3	4	5	6
-	-	-	6	3	-

1	2	3	4	5	6
-	-	6	6	2	-



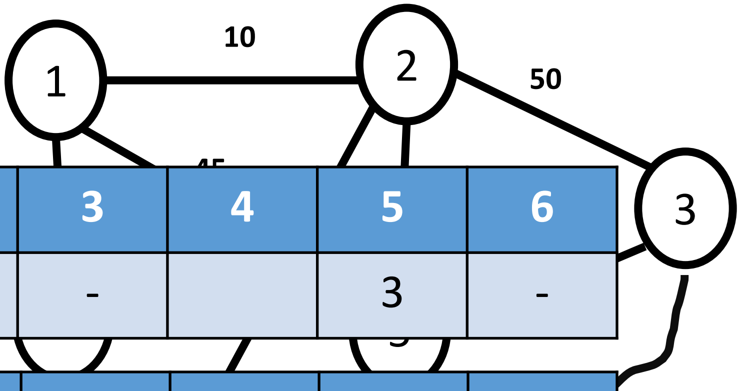
	T	C		F	R
1		-	1		
2		-	2		
3		<	3		
4	3	<	4	6	F
5	3	<	5	2	T
6		-	6		

Out of Four possible options: cost[near[j],j] = minimum
Select the index satisfying above condition: 4

The vertex selected is “4” due to which there may be changes in closeness of vertices.
That the vertices which were close to vertex 1,2,6,3 may modified due to selection of vertex 4.

g Tree: Prim’s method

- Consider the following graph:



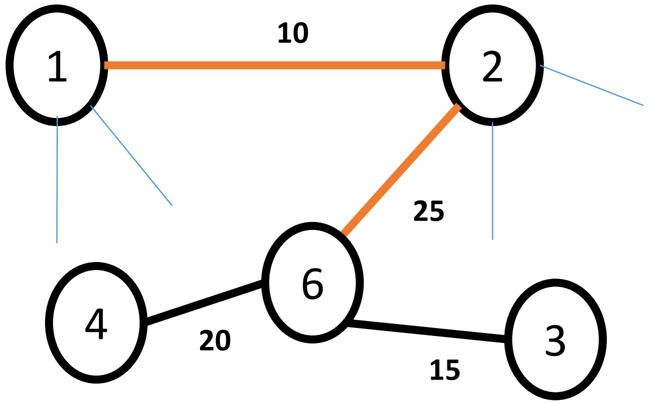
1	2	3	4	5	6
-	-	-		3	-

1	2	3	4	5	6
-	-	-	6	3	-

1	2	3	4	5	6
-	-	6	6	2	-

1	2	3	4	5	6
-	-	2	1	2	2

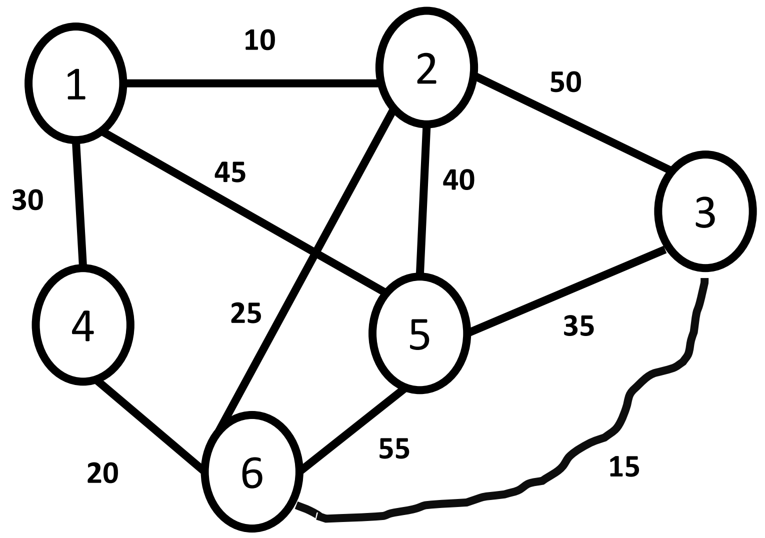
	1	2	3	4	5	6
1	99	10	99	30	45	99
2	10	99	50	99	40	25
3	99	50	99	99	35	15
4	30	99	99	99	99	20
5	45	40	35	99	99	55
6	99	25	**15**	**20**	55	99



	T	C		F	R
1		-	1		
2		-	2		
3		<	3		
4		<	4		F
5	3	<	5	4	T
6		-	6		

Prim’s method: Output

- Example:
- Consider the following graph:



	1	2	3	4	5	6
1	99	10	99	30	45	99
2	10	99	50	99	40	25
3	99	50	99	99	35	15
4	30	99	99	99	99	20
5	45	40	35	99	99	55
6	99	25	15	20	55	99

Vertex 1	Vertex 2	Cost
1	2	10
2	6	25
6	3	15
6	4	20
3	5	35
TOTAL		105

Algorithm: Prim's Algorithm

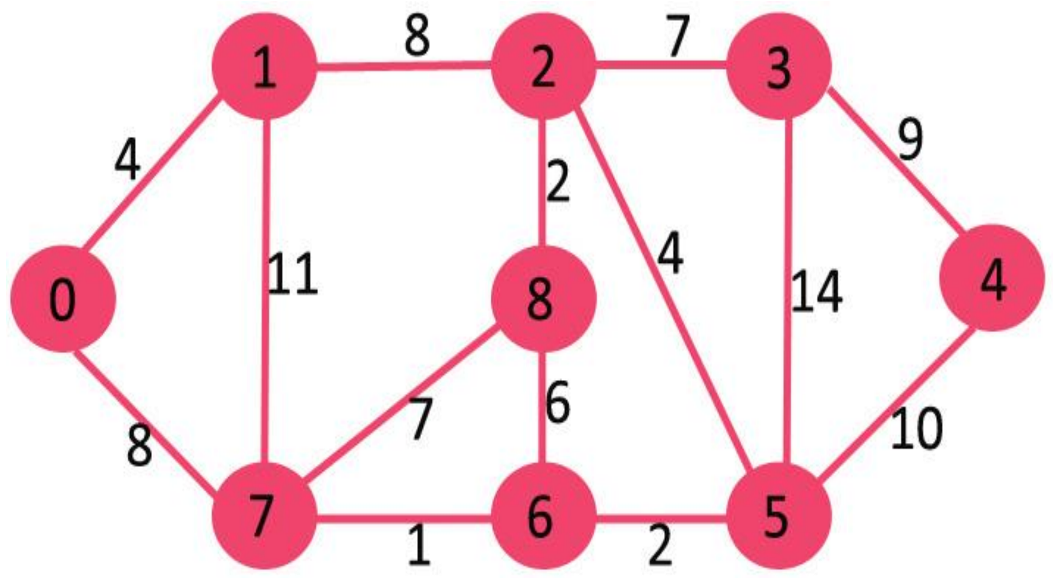
- Data Structures: Input
  - cost[1..n, 1..n] ➔ To store the cost information
- Intermediate:
  - near[1..n] ➔ To store near information for selected vertex
- Output:
  - mincost=0
  - t[1..n-1, 1..3] ➔ To store the spanning tree

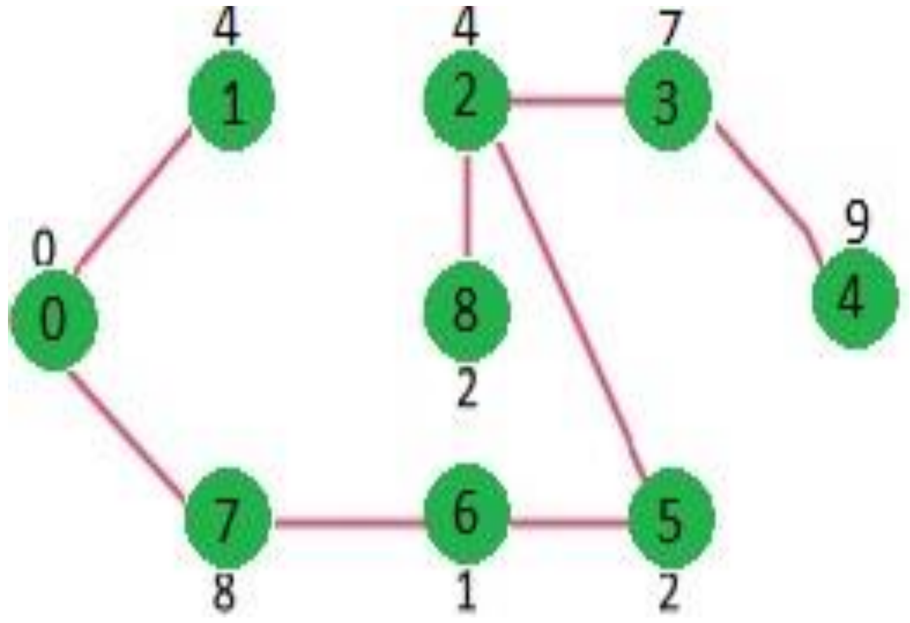
Algorithm

```
Algorithm Prim(cost,n: mincost,t)
{
    Step 1: Select an edge of minimum cost from
    the cost matrix and let it be cost[k,l]
    mincost = cost[k,l];
    t[1,1] = k;   t[1,2] = l;
    t[1,3] = cost[k,l]
    for 1 = 1 to n do
    {
        if cost[i,l] < cost[i,k] then
            near[i] = l;
        else
            near[i] = k;
    } //end of for
for i = 2 to n-1 do
{
let j be an index such that
(near[j] ≠ 0 and cost[near[j], j] =
minimum)
  t[i,1] = j;   t[i.2] = near[j]
  t[i,3] = cost[near[j],j]
   near[j] = 0;
   for k = 1 to n do
       if((near[k] ≠ 0) and
           cost[k,near[k] > cost[k,j]))
             near[k] = j;
 } //End of for
} //End of algorithm
```

Minimum Cost Spanning Tree: Prim’s Algorithm

- Designed by Robert C. Prim in 1957 and was modified by Dijkstra so also called as DJP algorithm.





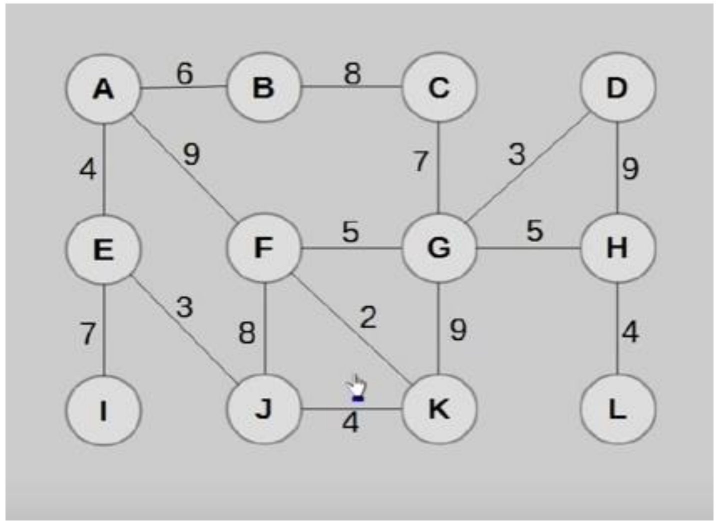
Proof of Correctness: Prims' Algorithm

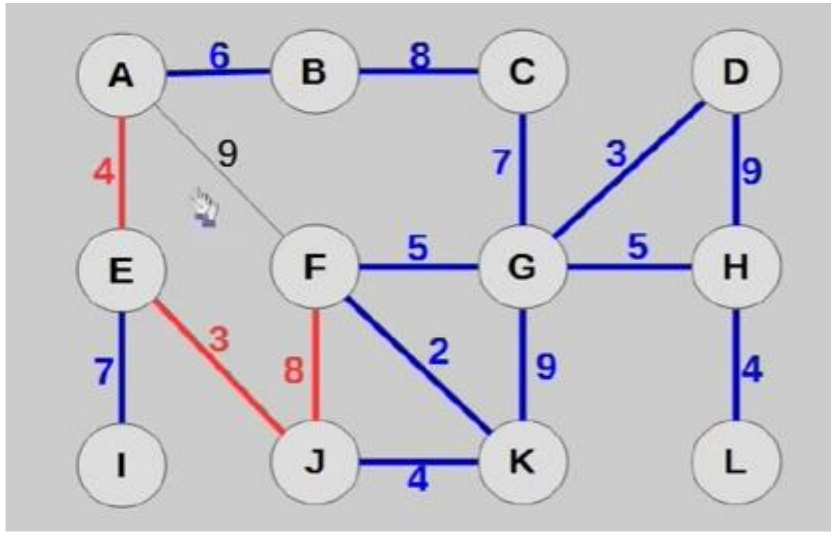
- The proof of prims' algorithms is described using CUT property of a graph.
- Let G=(V,E) be the graph, then CUT property will divide the graph into two sets of vertices (X and V-X), such that there exists three types of edges.
  - *Edges with both vertices in "X"*
  - *Edges with both vertices in "V-X"*
  - *Edges with one vertex in "X" and one vertex in "V-X"* ➔ *Crossing edge*
- If there are "n" vertices in graph, then number of cuts will be $\mathbf{2^n}$
- Empty cut lemma:
- A graph is said to be not connected, if cut has no crossing edges. This is denoted as empty cut.
- ***The correctness is proved using straight forward induction method.***

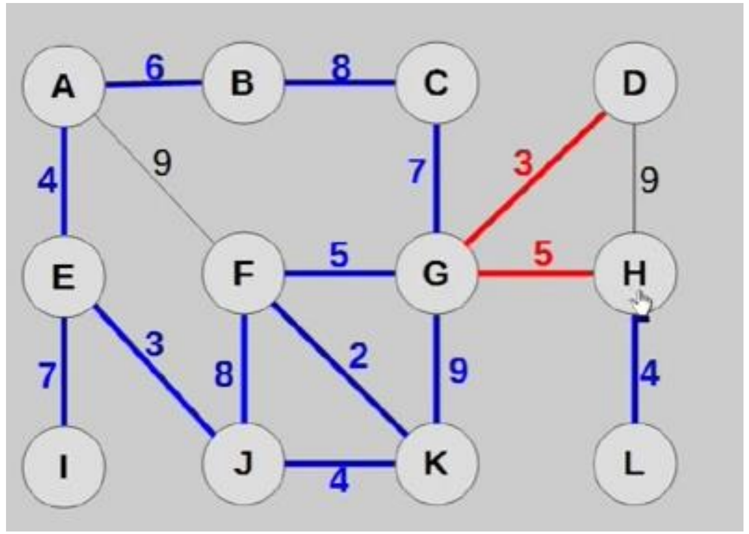
Reverse Delete Algorithm: application of DFS

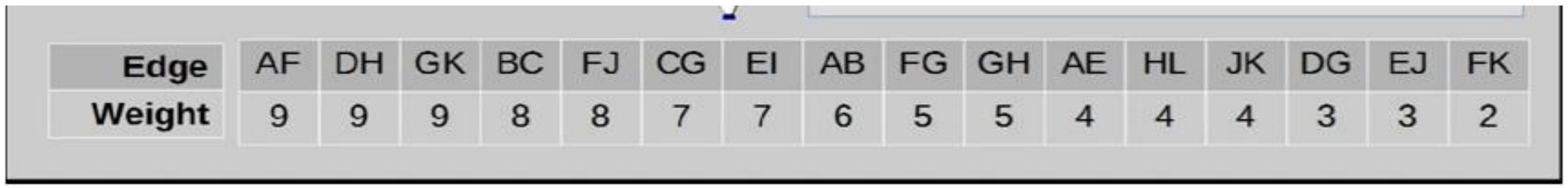
Basic Idea

- Let *A* be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from *A* as long as all vertices are connected by *A*



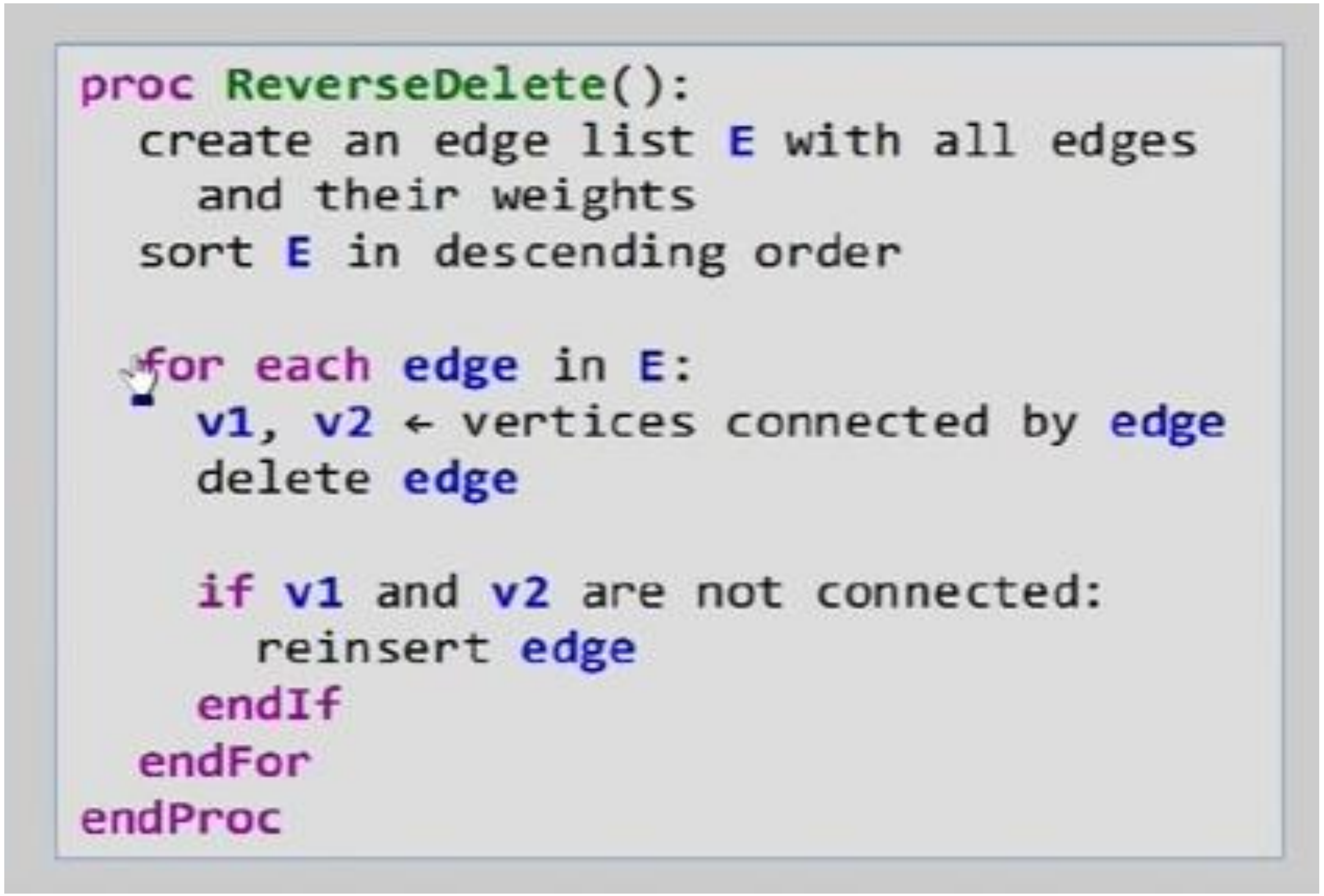




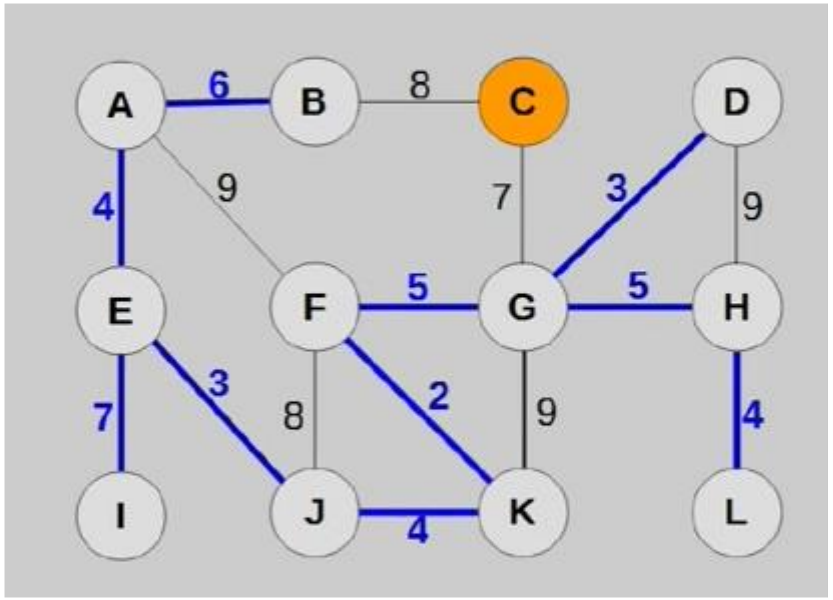


Edge	AF	DH	GK	BC	FJ	CG	EI	AB	FG	GH	AE	HL	JK	DG	EJ	FK
Weight	9	9	9	8	8	7	7	6	5	5	4	4	4	3	3	2

Algorithm: RD method

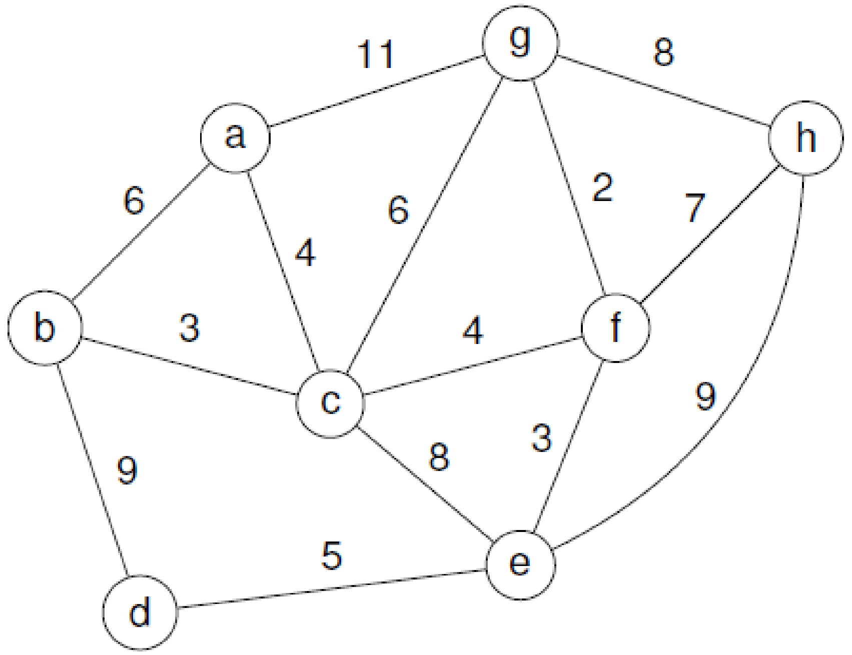


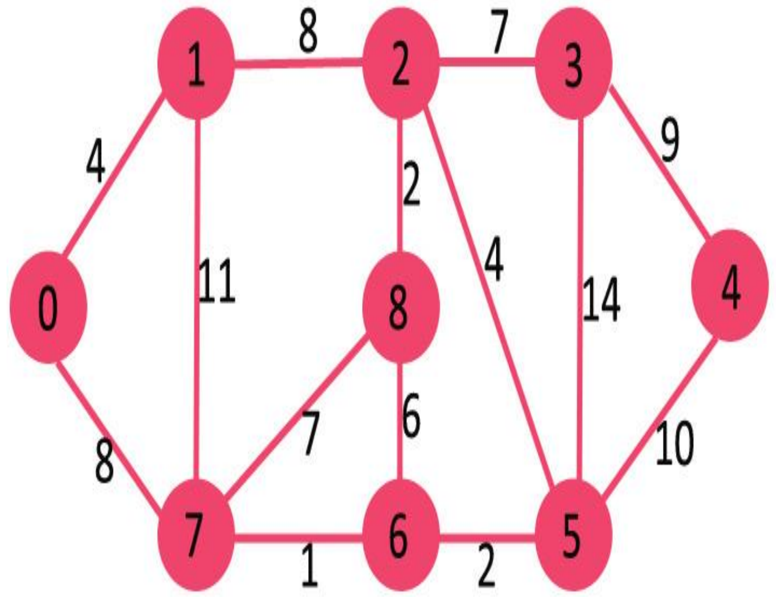
```
proc ReverseDelete():
  create an edge list E with all edges
    and their weights
  sort E in descending order

  for each edge in E:
    v1, v2 ← vertices connected by edge
    delete edge

    if v1 and v2 are not connected:
      reinsert edge
    endIf
  endFor
endProc
```



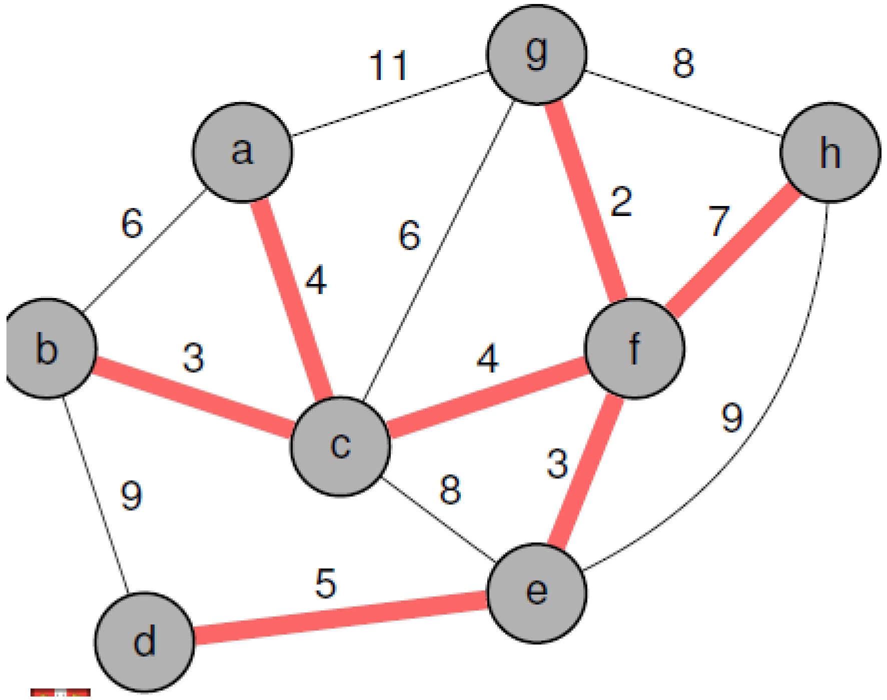
EDGE CG cannot be deleted, as there is no Alternative path.
If from vertex C or Vertex G, DFS is performed It is not possible to reach to other vertex.

Practice examples: Prims & RD algorithm





Solution



g
a
h
b
c
f
e
d
11
8
6
4
6
2
7
3
4
9
8
3
9
5