
The Greedy method & Dynamic Programming

The Greedy method

Topics to be covered:-

- Greedy Technique : the general method
- Knapsack Problem
- Job Sequencing With Deadlines
- Minimum Spanning Tree Problem
 - ✓ Prim's Algorithm
 - ✓ Kruskal's Algorithm

Greedy Technique

- These problems have N inputs and a set of constraints
- Any subset that satisfies the constraints is called *feasible solution*
- Optimal solution is a feasible solution that maximize or minimize the given problem
- An **optimization problem** is the **problem** of finding the best solution from all feasible solutions
- Greedy technique is for solving *optimization problems*
- Greedy approach suggests constructing a solution through a sequence of steps called *decision steps*

Greedy Technique

- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
 - You take the best you can get right now, without regard for future consequences
 - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Greedy Algorithms

- A greedy algorithm makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

The choice made at each step must be:

- Feasible :Satisfy the problem's constraints
- locally optimal : Be the best local choice among all feasible choices
- Irrevocable : Once made, the choice can't be changed on subsequent steps.

Applications of the Greedy Strategy

- Optimal solutions:
 - change making
 - Minimum Spanning Tree (MST)
 - Single-source shortest paths
 - Huffman codes
- Approximations:
 - Traveling Salesman Problem (TSP)
 - Knapsack problem
 - other optimization problems

Control abstraction – greedy method

Greedy Algorithm (a, n)

//a[1: n] contains the n inputs

```
{  
    solution =  $\emptyset$ ; // initialization of the solution  
    for i = 1 to n do  
    {  
        x = select (a);  
        if Feasible ( solution, x ) then  
            solution = Union (solution , x);  
    }  
    return solution;  
}
```


Control abstraction – greedy method

- At each stage , a decision is made regarding whether a particular input is an optimal solution
- The function '**select**' select an input from $a[]$ & remove it
- The selected input value is assigned to x .
- The function '**Feasible**' is a Boolean valued function which determines whether x can be included into the solution.
- The function '**Union**' combines x with the solution and update the objective function

Limitations of Greedy Strategy

- Greedy technique *does not* give the optimal solution for all problems
- There are problems for which a sequence of locally optimal choices does not yield an optimal solution
- Produces approximate solution

Knapsack Problem

knapsack problem Definition

- The **knapsack problem** or rucksack **problem** is a **problem** is an optimization problem.
- Given a set of items, each with a weight and a value, determine the number of each item 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.

Knapsack Problem

Given :

- ✓ n : number of object
- ✓ w_i : weight of object i
- ✓ m : capacity of knapsack
- ✓ p_i : profit of all of i is taken

Find:

- ✓ x_i : fraction of i taken

Feasibility:
$$\sum_{i=1}^n x_i w_i \leq m$$

Optimality:
$$\text{maximize } \sum_{i=1}^n x_i p_i$$

The knapsack problem

n objects, each with a weight $w_i > 0$

a profit $p_i > 0$

capacity of knapsack: M

Maximize $\sum_{1 \leq i \leq n} p_i x_i$

Subject to $\sum_{1 \leq i \leq n} w_i x_i \leq M$ $0 \leq x_i \leq 1, 1 \leq i \leq n$

The Knapsack Problems– Category

The Knapsack Problems –Category

❖ “**0-1 knapsack problem**”

- The typical formulation in practice is the 0/1 knapsack problem, where each item must be put entirely in the knapsack or not included at all.
- Solved with dynamic programming

❖ “**Fractional knapsack problem**”

- Items are divisible: we can take any fraction of an item.
- Solved with a greedy algorithm.

Knapsack Problem

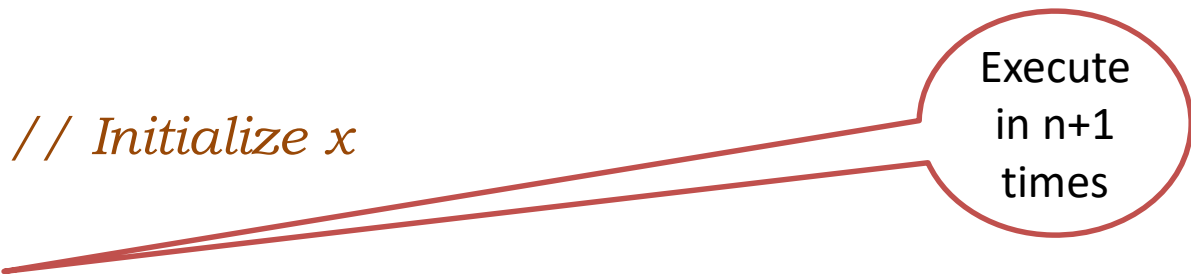
The Knapsack Problems

- A similar problem very often appears in
 - business,
 - complexity theory,
 - cryptography
 - applied mathematics.
- Given a set of items, each with a cost and a value, then determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible

Algorithm GreedyKnapsack(m,n)

// $p[1:n]$ and $w[1:n]$ contain the profits and weights respectively of the n objects ordered such that $p[i] / w[i] \geq p[i+1] / w[i+1]$. M is the knapsack size and $x[1:n]$ is the solution

```
{  
  for i=1 to n do  
     $x[i]=0.0$            // Initialize  $x$   
   $U:=m$ ;  
  for i= 1 to n do  
  {  
    if ( $w[i] > U$ ) then break;  
     $x[i] = 1.0$ ;  
     $U=U-w[i]$ ;  
  }  
  if ( $i \leq n$ ) then  $x[i] = U/w[i]$ ;  
}
```



Execute
in $n+1$
times

Example

$$n = 3,$$

$$M = 20,$$

$$(p_1, p_2, p_3) = (25, 24, 15)$$

$$(w_1, w_2, w_3) = (18, 15, 10)$$

Step 1: Sort p_i/w_i into non increasing order.

Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.

Solution:

$$p_1/w_1 = 25/18 = 1.32$$

$$p_2/w_2 = 24/15 = 1.6$$

$$p_3/w_3 = 15/10 = 1.5$$

So Rearrange the list with p_i/w_i in non increasing order

Object	P2	P3	P1
Profit	24	15	25
Weight	15	10	18
Profit/Weight	1.6	1.5	1.38
X[]	1	5/10=1/2	0

i=1 U=20

$W1 < U \quad \rightarrow 15 < 20$

$X[1] = 1.0$

$U = 20 - 15 = 5$

i=2 U=5

$W2 > U$ BREAK

$X[2] = 5 / 10 = 1/2;$

Solution is : $x_1 = 1, x_2 = 1/2, x_3 = 0$

Profit = $24 + 7.5 = 31.5$

The solution to the original problem is

Optimal solution: $x_1 = 0, x_2 = 1, x_3 = 1/2$

total profit = $24 + 7.5 = 31.5$

Home work

Find an optimal solution to the following knapsack instance

n=5			M=100		
Wi	10	20	30	40	50
Pi	20	30	66	40	60

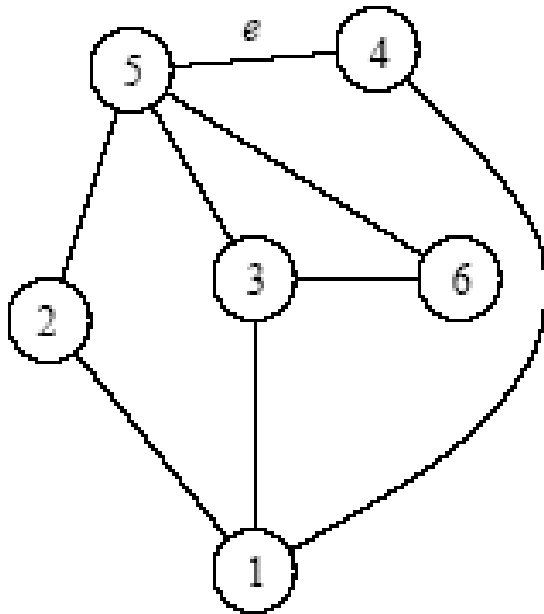
n=7					M=15		
Wi	2	3	5	7	1	4	1
Pi	10	5	15	7	6	18	3

Minimum Cost Spanning Tree

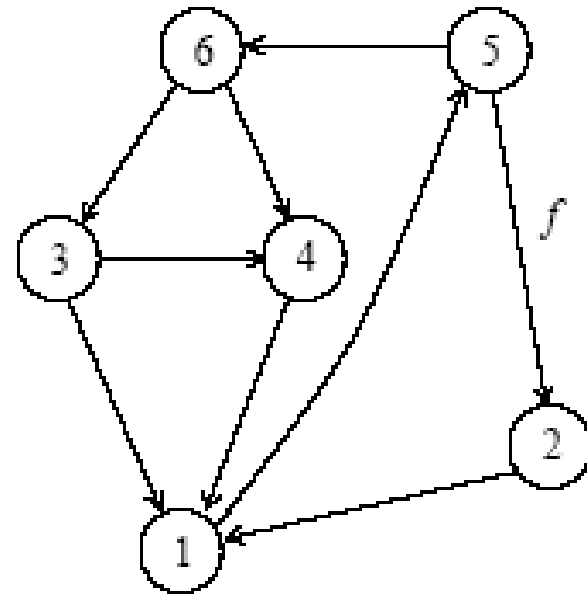
Definitions and Representation

- An undirected graph G is a pair (V, E) , where V is a finite set of points called *vertices* and E is a finite set of *edges*.
- In a directed graph, the edge e is an ordered pair (u, v) . An edge (u, v) is *incident from* vertex u and is *incident to* vertex v .
- The length of a path is defined as the number of edges in the path.

Definitions and Representation



(a)



(b)

a) An undirected graph and (b) a directed graph.

Definitions and Representation

- An undirected graph is *connected* if every pair of vertices is connected by a path.
- A *forest* is an acyclic graph, and a *tree* is a connected acyclic graph.
- A graph that has weights associated with each edge is called a *weighted graph*.

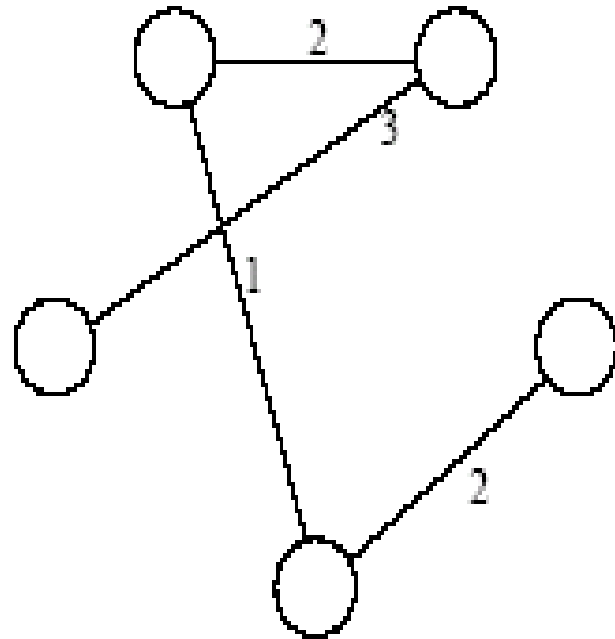
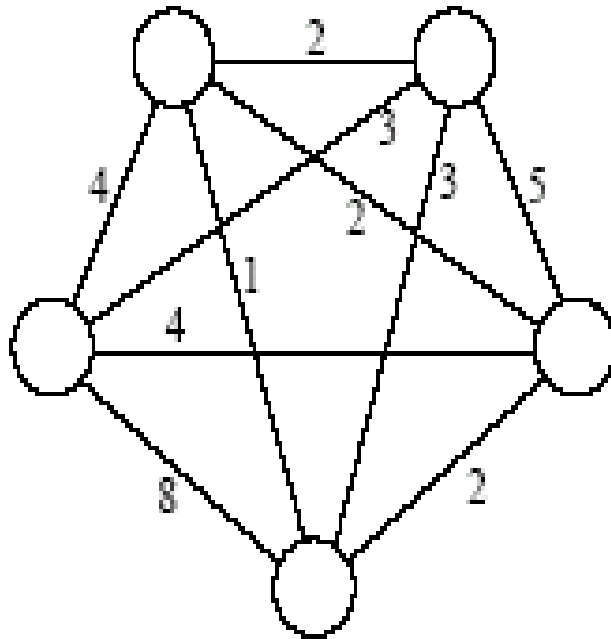
Spanning Tree

- A *spanning tree* of an undirected graph G is a subgraph of G that is a tree containing all the vertices of G .
- In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.
- A minimum spanning tree (MST) for a weighted undirected graph is a spanning tree with minimum weight.

Spanning tree

- The spanning tree is defined as a subgraph of an undirected graph in which all vertices are connected.
- I.e., any tree consisting of all the vertices of a graph is called a spanning tree.
- The weight of a spanning tree for a weighted graph is the sum of the edge weights.
- The spanning tree with minimum total weight is known as the minimum spanning tree- MST

Minimum Spanning Tree

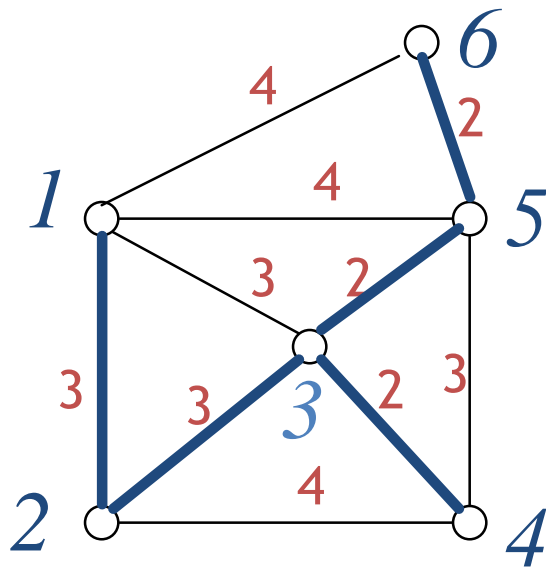


An undirected graph and its minimum spanning tree.

Minimum spanning tree

A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes

- Start by picking any node and adding it to the tree
- Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
- Stop when all nodes have been added to the tree



The result is a least-cost
($3+3+2+2+2=12$) spanning tree

If you think some other edge
should be in the spanning tree:

- Try adding that edge
 - Note that the edge is part of a cycle
 - To break the cycle, you must remove the edge with the greatest cost
- ✓ This will be the edge you just added

Minimum Spanning Trees

Applications

- Problem: Connect a set of nodes by a network of minimal total length
- Some applications:
 - Communication networks
 - Circuit design
 - Layout of highway systems

Formal Definition of MST

- Given a connected, undirected, graph $G = (V, E)$, a *spanning tree* is an *acyclic* subset of edges $T \subseteq E$ that connects all the vertices together.
- Assuming G is weighted, we define the *cost* of a spanning tree T to be the sum of edge weights in the spanning tree

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

- A *minimum spanning tree (MST)* is a spanning tree of minimum weight.

Figure 1 : Examples of MST

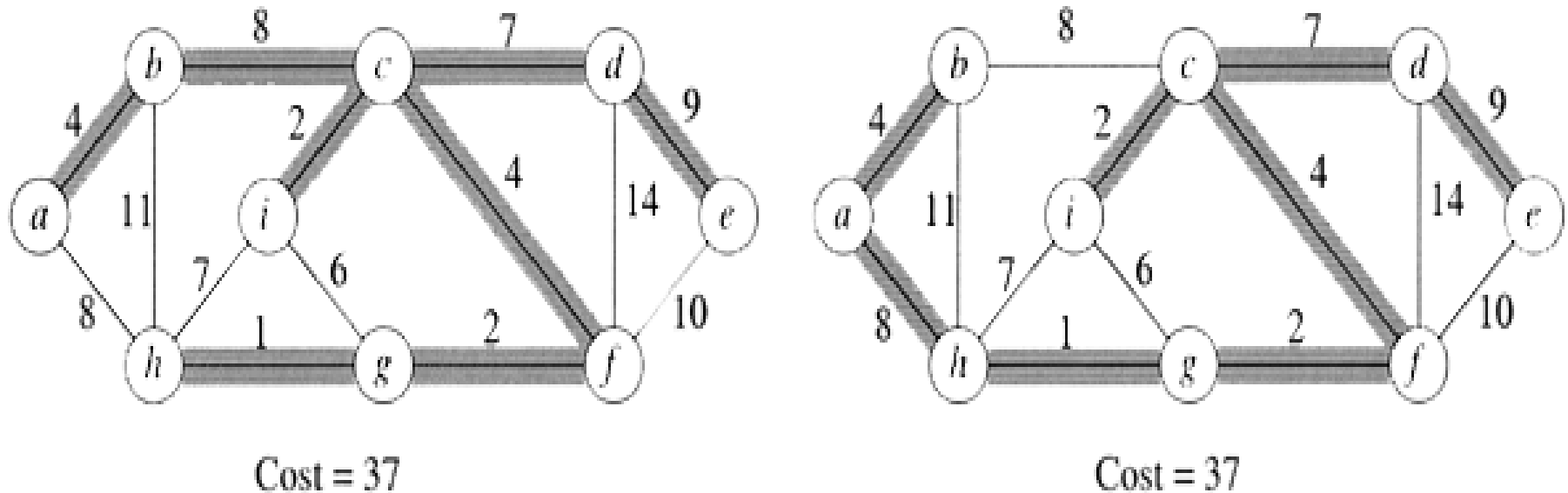
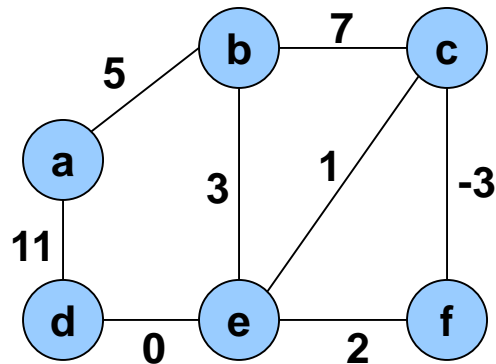


Figure 1: Minimum spanning tree.

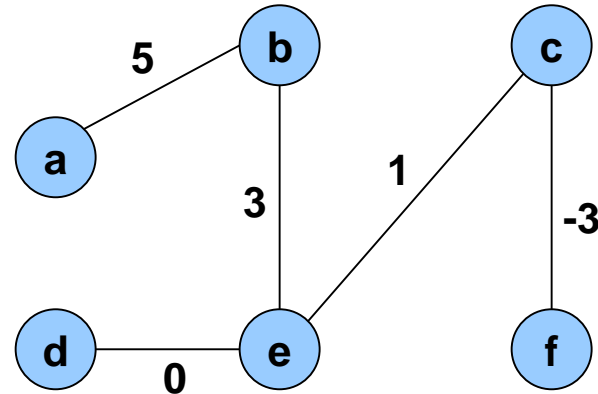
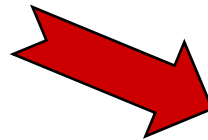
- Not only do the edges sum to the same value, but the same set of edge weights appear in the two MSTs. NOTE: An MST may not be unique.

Minimum Spanning Trees

- **Given:** Connected, undirected, weighted graph, G
- **Find:** Minimum - weight spanning tree, T
- **Example:**



Acyclic subset of edges(E) that connect all vertices of G .



Generic Approaches

- Two greedy algorithms for computing MSTs:
 - Kruskal's Algorithm (similar to connected component)
 - Prim's Algorithm (similar to Dijkstra's Algorithm)

Prim's Algorithm

Minimum Spanning Tree: Prim's Algorithm

- Prim's algorithm for finding an MST is a greedy algorithm.
- Start by selecting an arbitrary vertex, include it into the current MST.
- Grow the current MST by inserting into it the vertex closest to one of the vertices already in current MST.

1. Algorithm Prim(E, cost, n, t)
2. *// E is the set of edges in G . $\text{cost}[1:n, 1:n]$ is the cost adjacency matrix of an n vertex graph such that $\text{cost}[i, j]$ 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.*
3. {
4. Let (k, l) be an edge of minimum cost in E ;
5. $\text{mincost} = \text{cost}[k, l]$;
6. $t[1, 1] = k$; $t[1, 2] = l$;
7. for $i = 1$ to n do
8. if $(\text{cost}[i, l] < \text{cost}[i, k])$ then $\text{near}[i] = l$;
9. else $\text{near}[i] = k$;



$O(E)$

$O(n)$

```
15. near[k] = near[1]=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.}
```

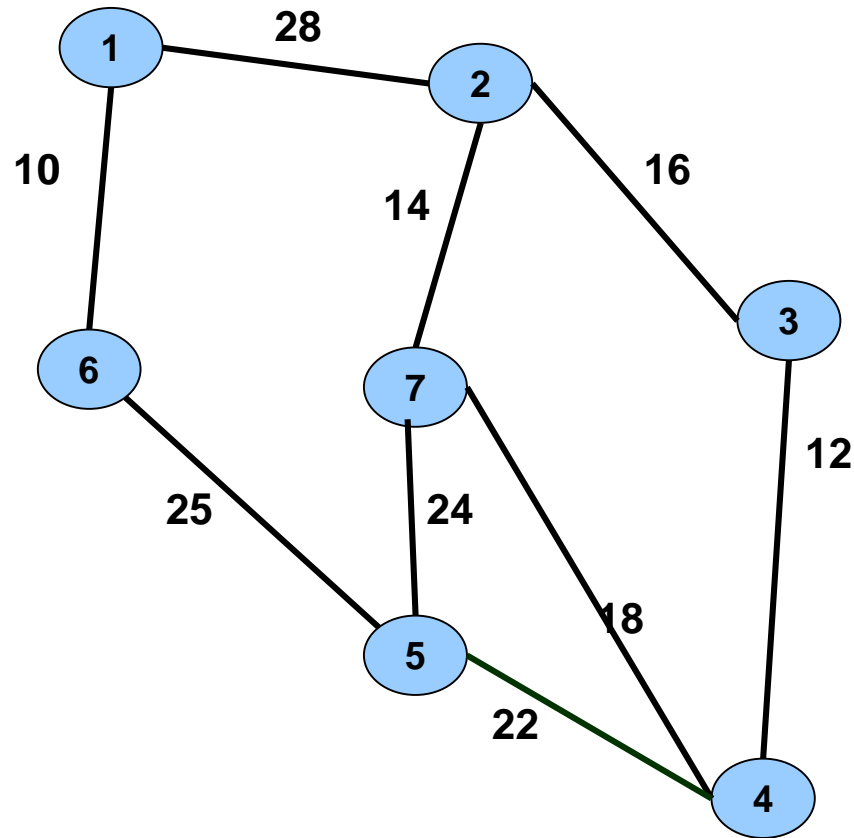
$O(n^2)$

$O(n^2)$

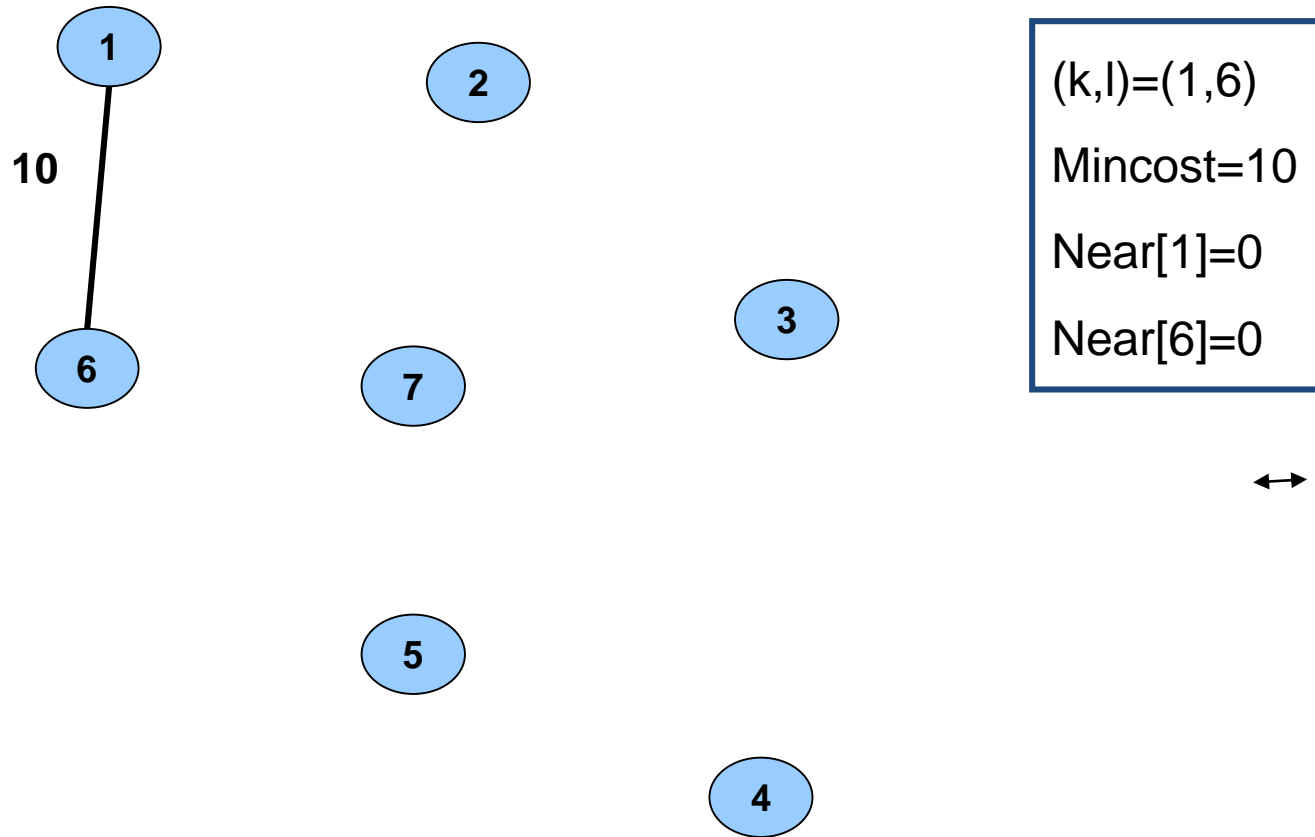
Explanation

- Start with includes a minimum cost edges
- Next edge (i,j) is added ,
 - i is already included in the tree and j not yet included.
 - $\text{Cost}(i,j)$ is minimum
- Line 9: select the minimum edge
- Line 10-15 initialize the variables
- Line 16-17 remainder of the spanning tree is built up edge by edge
- Line 18-19 select the next edge to include
- Line 20-22 calculate and store values
- Line 23-25 update $\text{near}[]$

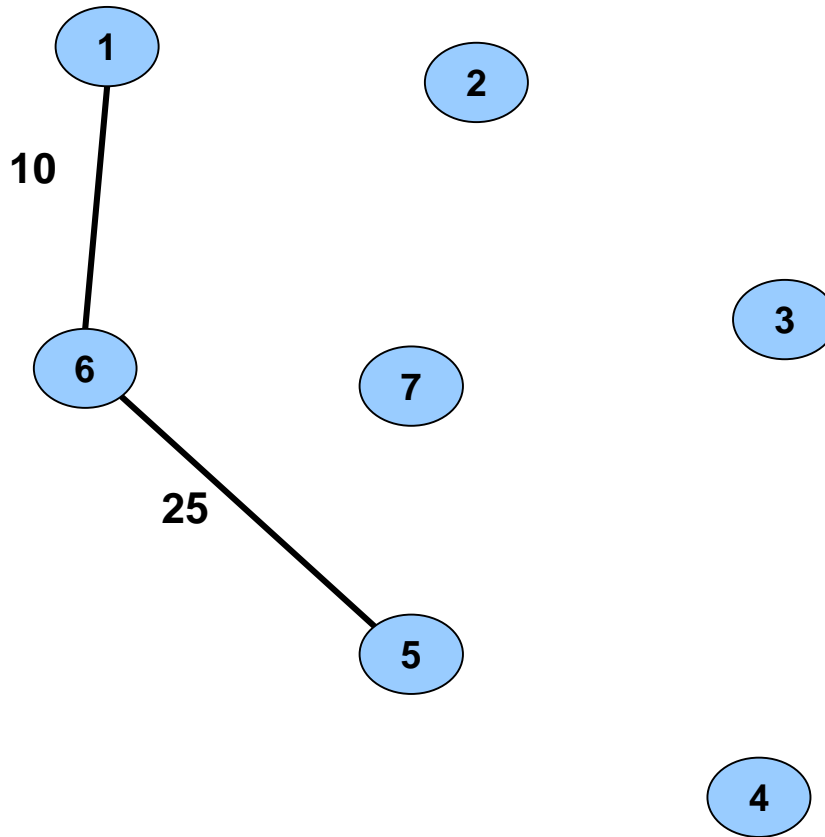
Example of Prim's Algorithm



Example of Prim's Algorithm



Example of Prim's Algorithm



$i=2$

$j=5$

$t(11)=1$ $t(1,2)=6$

$t(21)=6$ $t(2,2)=5$

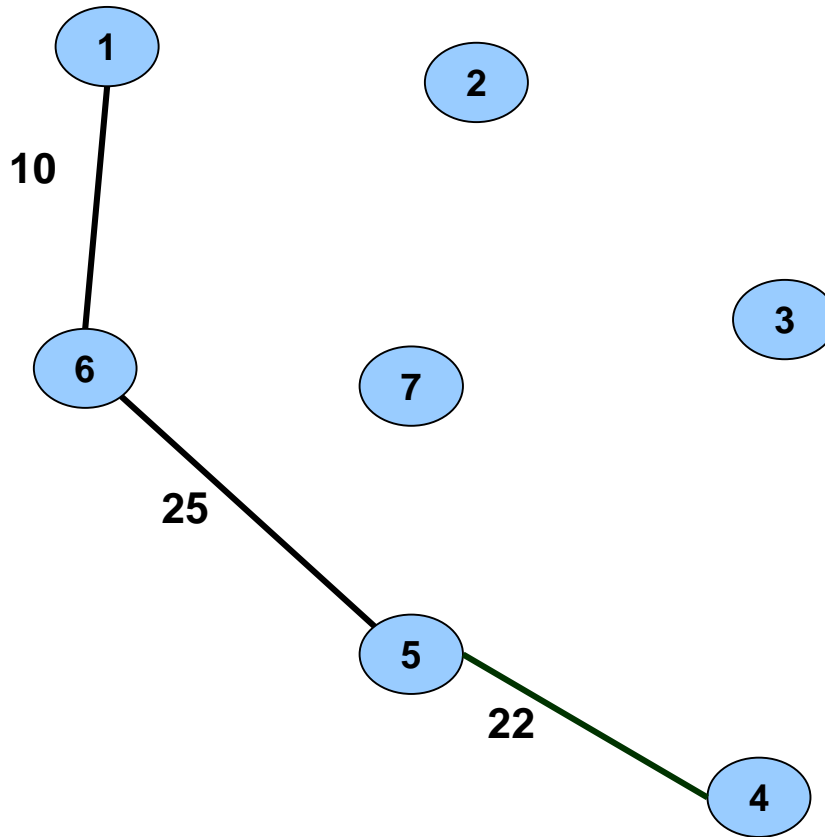
$\text{Mincost}=10+25$

$\text{Near}[1]=0$

$\text{Near}[6]=0$

$\text{Near}[5]=0$

Example of Prim's Algorithm



$i=3$

$j=4$

$t(11)=1$ $t(1,2)=6$

$t(21)=6$ $t(2,2)=5$

$T(31)=5$ $t(3,2)=4$

$\text{Mincost}=35+22=57$

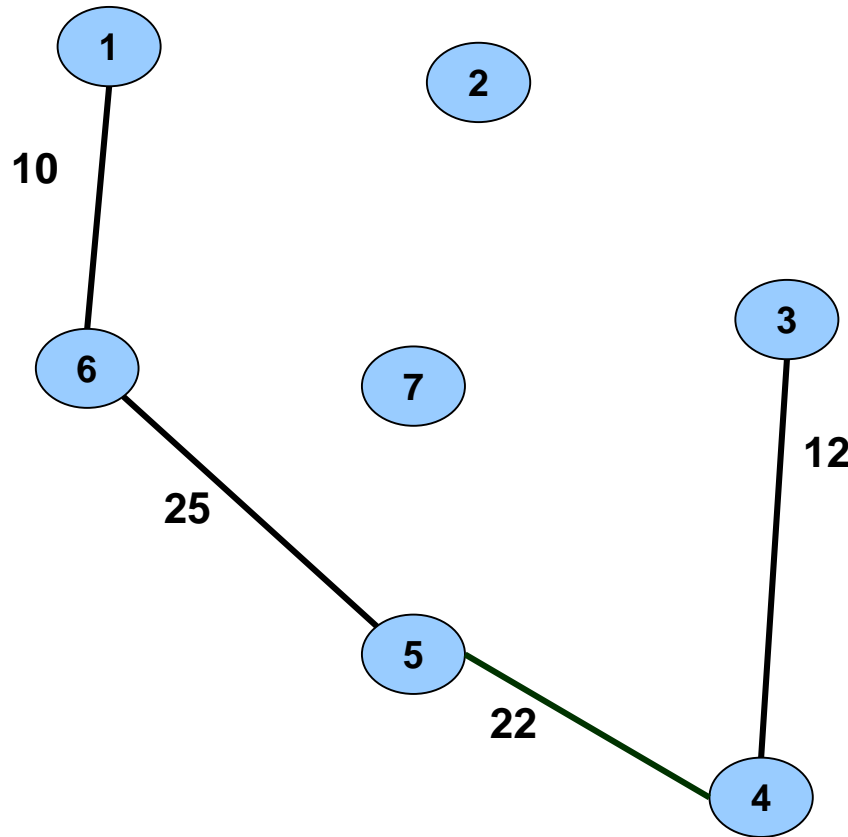
$\text{Near}[1]=0$

$\text{Near}[6]=0$

$\text{Near}[5]=0$

$\text{Near}[4]=0$

Example of Prim's Algorithm



$i=4$

$j=3$

$t(11)=1$ $t(1,2)=6$

$t(21)=6$ $t(2,2)=5$

$t(31)=5$ $t(3,2)=4$

$t(41)=4$ $t(3,2)=3$

$\text{Mincost}=57+12=69$

$\text{Near}[1]=0$

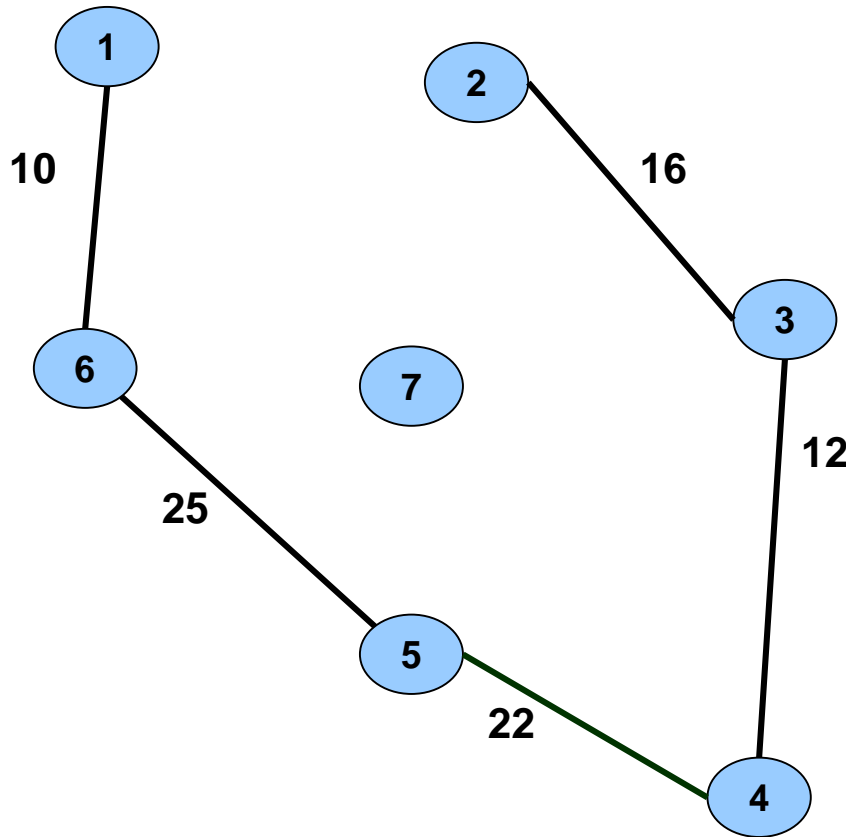
$\text{Near}[6]=0$

$\text{Near}[5]=0$

$\text{Near}[4]=0$

$\text{Near}[3]=0$

Example of Prim's Algorithm



$i=5$

$j=2$

$t(11)=1$ $t(1,2)=6$

$t(21)=6$ $t(2,2)=5$

$t(31)=5$ $t(3,2)=4$

$t(41)=4$ $t(3,2)=3$

$t(51)=3$ $t(5,2)=2$

$\text{Mincost}=69+16=85$

$\text{Near}[1]=0$

$\text{Near}[6]=0$

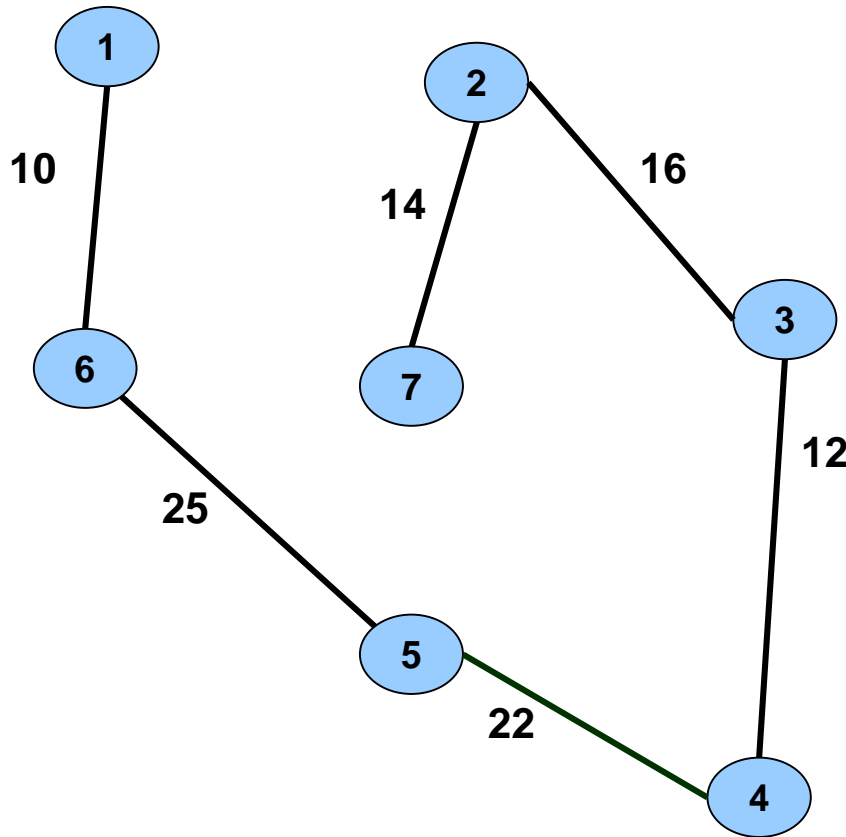
$\text{Near}[5]=0$

$\text{Near}[4]=0$

$\text{Near}[3]=0$

$\text{Near}[2]=0$

Example of Prim's Algorithm



$i=6; j=7$

$t(11)=1$ $t(1,2)=6$

$t(21)=6$ $t(2,2)=5$

$t(31)=5$ $t(3,2)=4$

$t(41)=4$ $t(3,2)=3$

$t(51)=3$ $t(5,2)=2$

$t(61)=2$ $t(6,2)=7$

$\text{Mincost}=85+14=99$

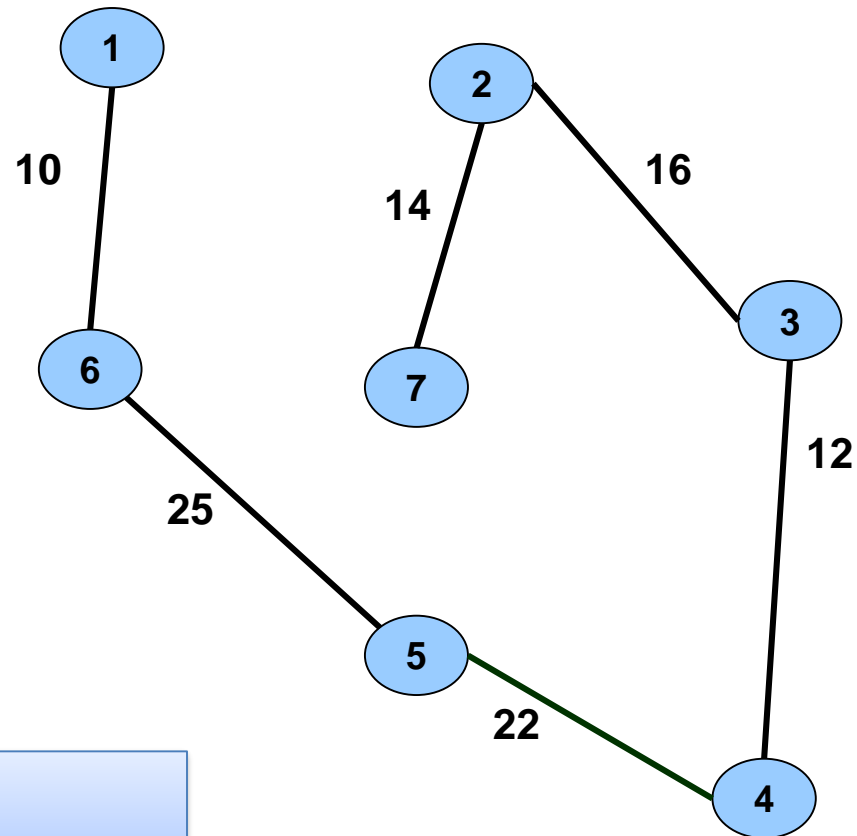
$\text{Near}[1]=0$ $\text{Near}[6]=0$

$\text{Near}[5]=0$ $\text{Near}[4]=0$

$\text{Near}[3]=0$ $\text{Near}[2]=0$

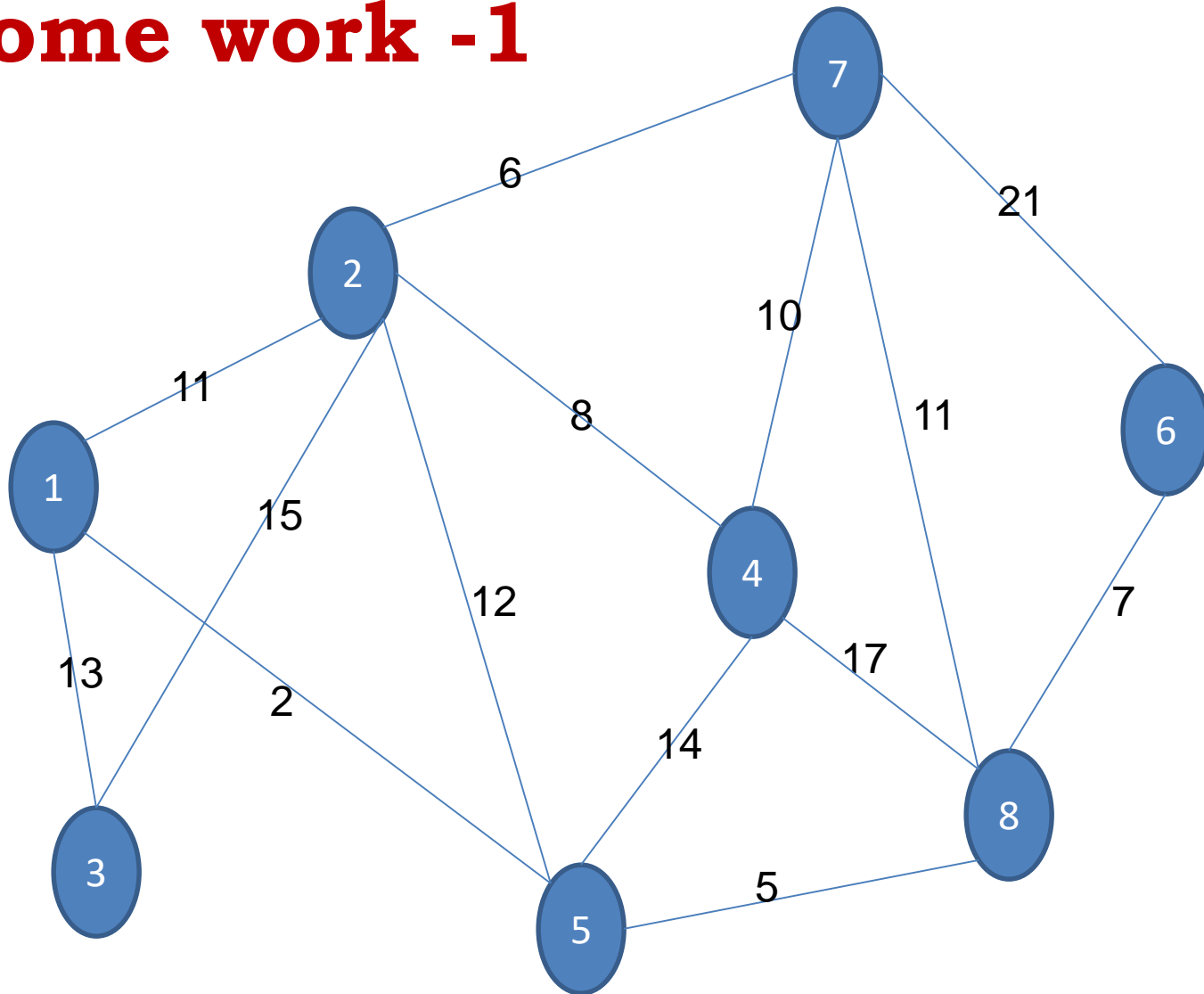
$\text{Near}[7]=0$

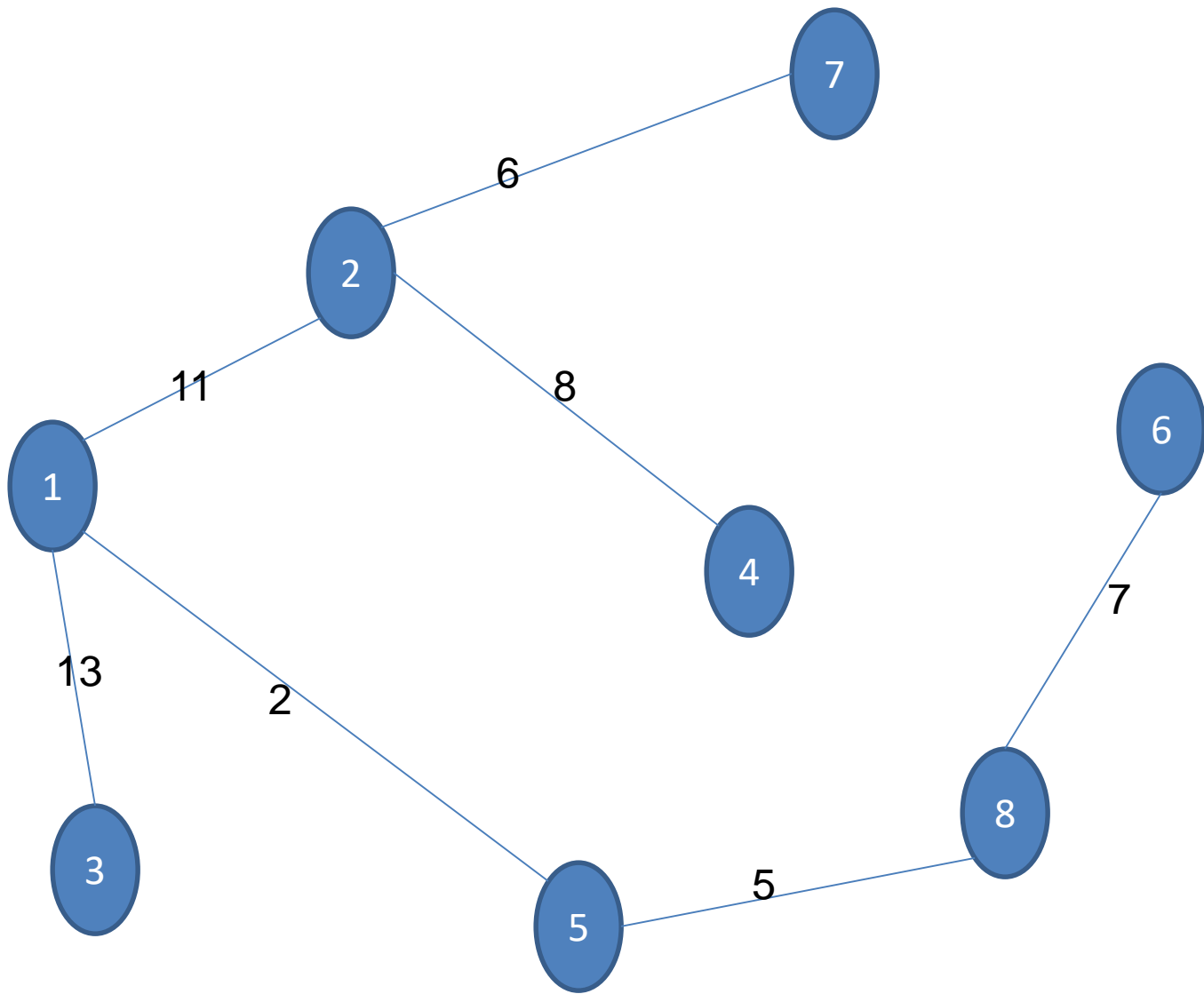
Example of Prim's Algorithm



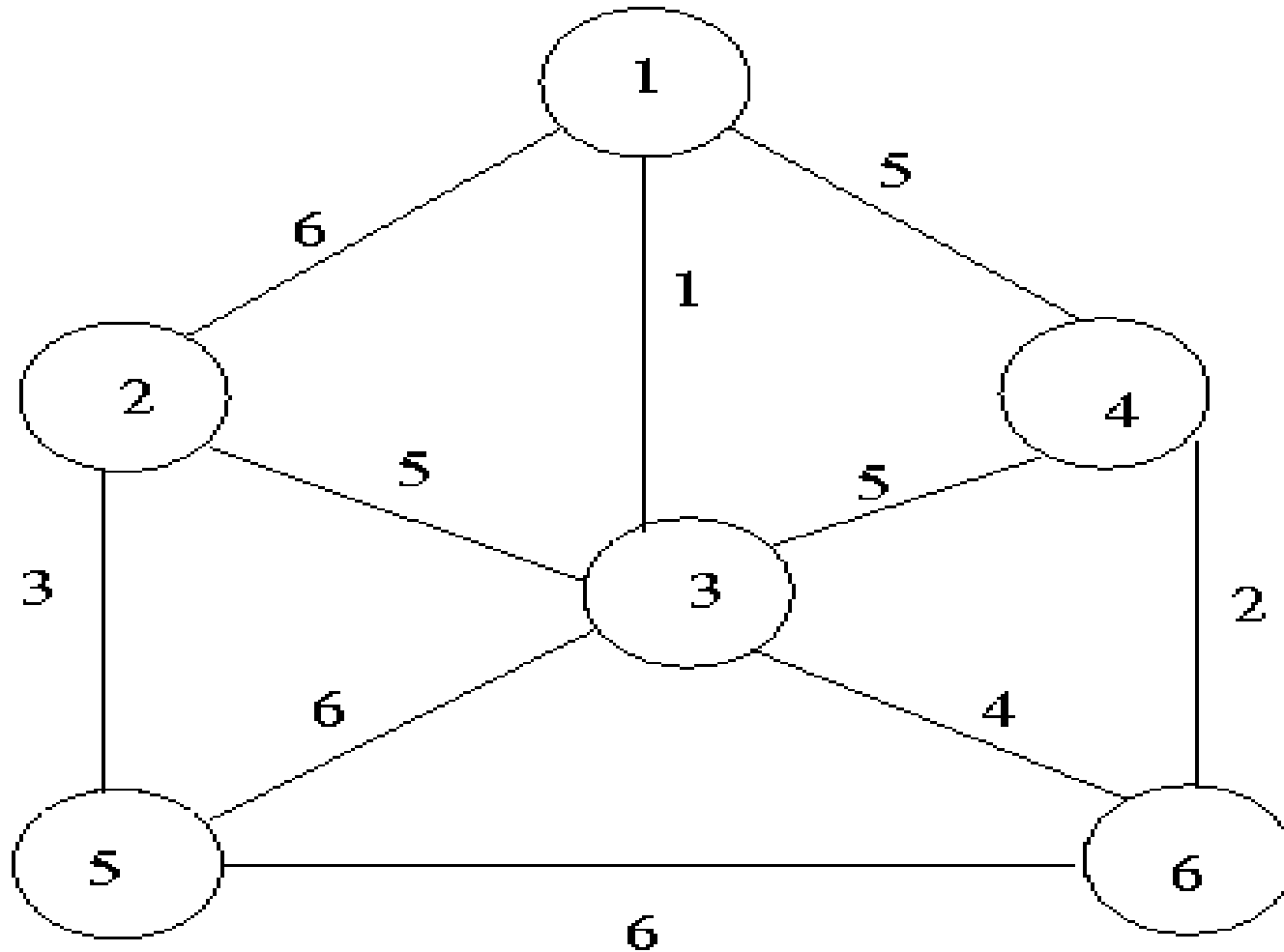
So the MST using Prim's algorithm
Mincost=99

Home work -1

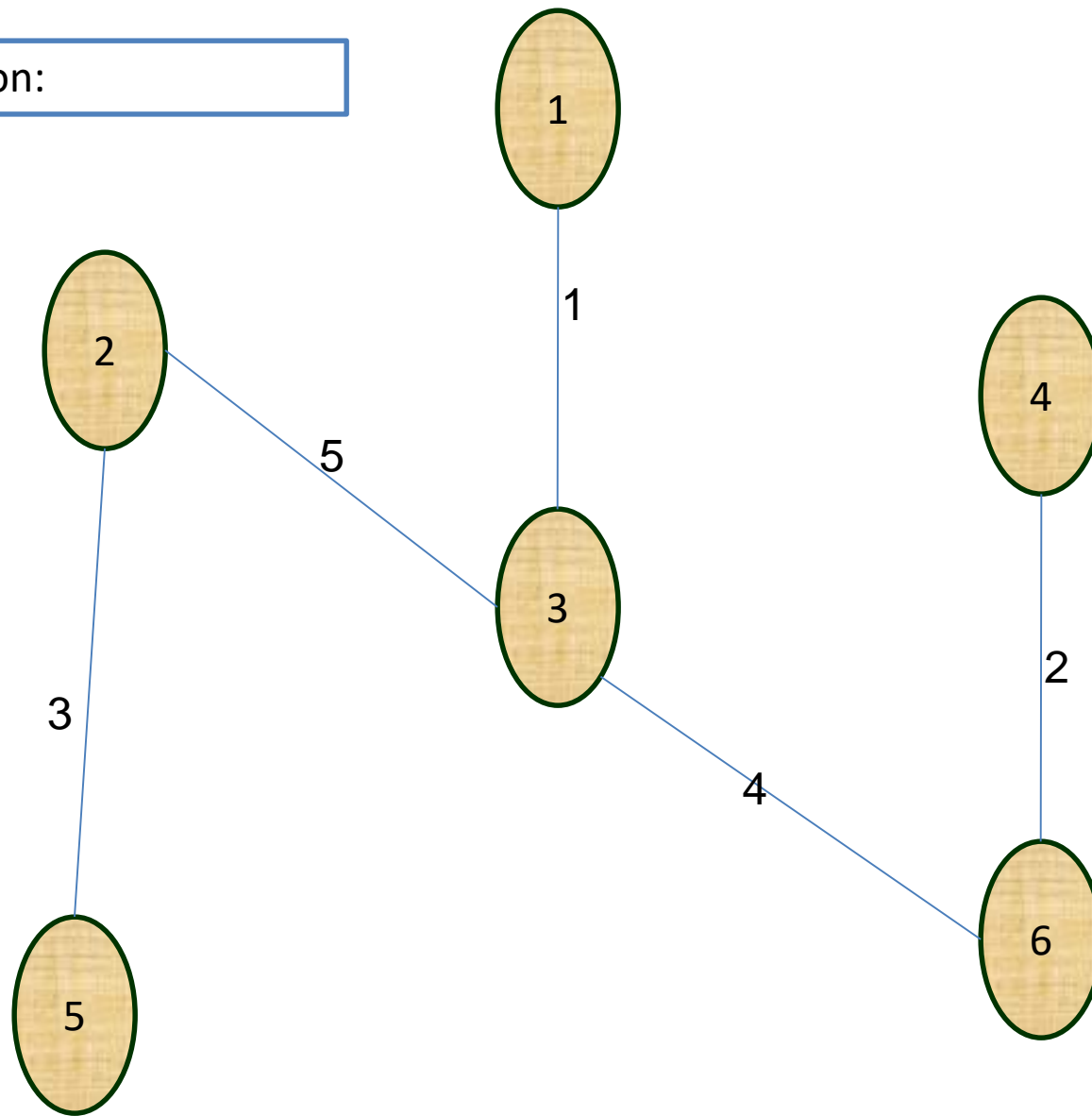




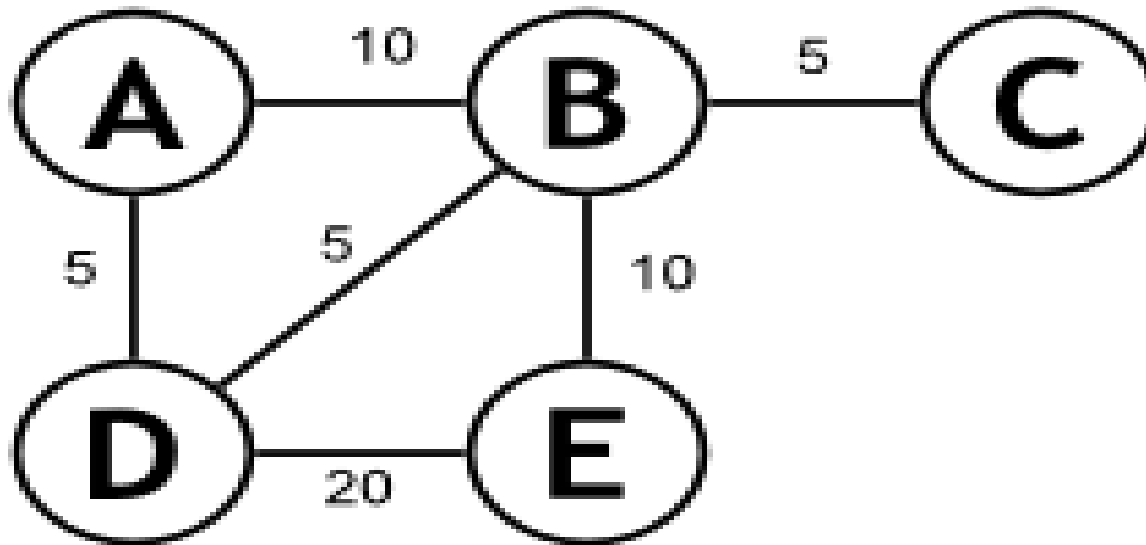
Home work -2



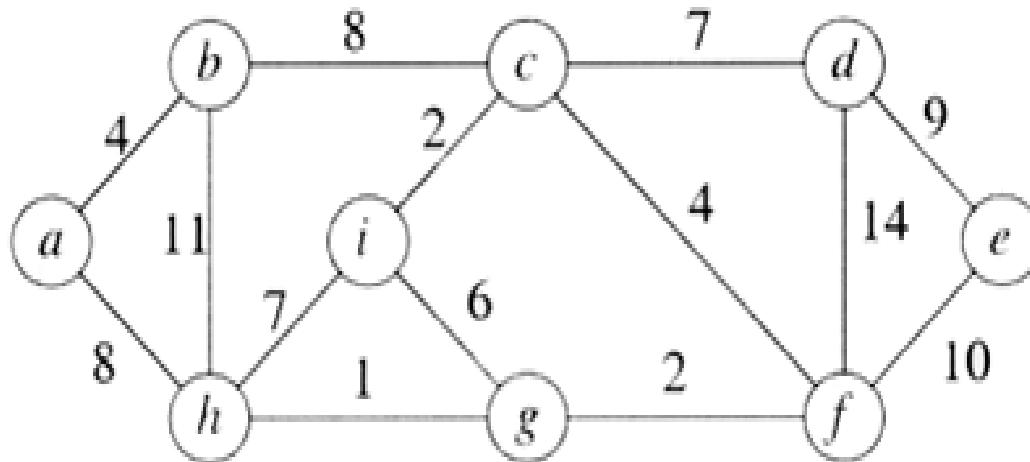
Solution:



Another example



Another Example



Analysis

Let n is the number of vertices in the graph G

- Line 9 select the minimum cost takes $O(E)$
- Line 12 for loop takes $O(n)$
- Line 16 for loop takes $O(n)$
- Line 23 for loop takes $O(n)$
- But line 16 – 26 takes $O(n^2)$

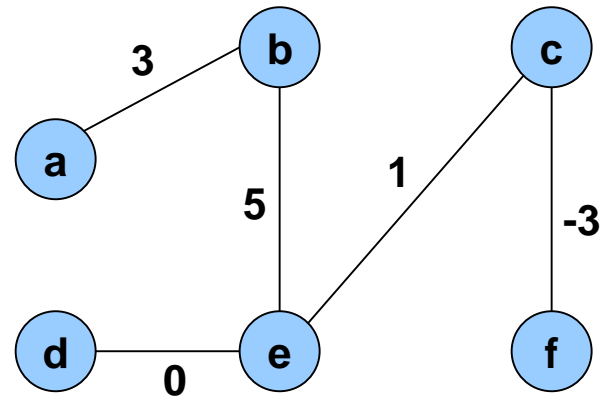
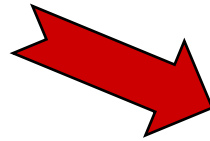
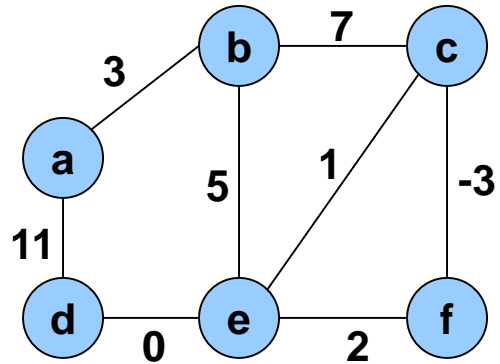
So the *Efficiency of Prim's Algorithm* is $O(n^2)$

Kruskal's Algorithm

Basics of Kruskal's Algorithm

- Edges are initially sorted by increasing weight
- Start with an empty forest ,“grow” MST one edge at a time
 - intermediate stages usually have forest of trees (not connected)
- at each stage add **minimum weight edge** among those not yet used that does not create a cycle
 - at each stage the edge may:
 - ✓ expand an existing tree
 - ✓ combine two existing trees into a single tree
 - ✓ create a new tree
 - need efficient way of detecting/avoiding cycles
- algorithm stops when all vertices are included

Example: Kruskal's Algorithm



Kruskul Algorithm

1. Algorithm Kruskal(E, cost, n, t)

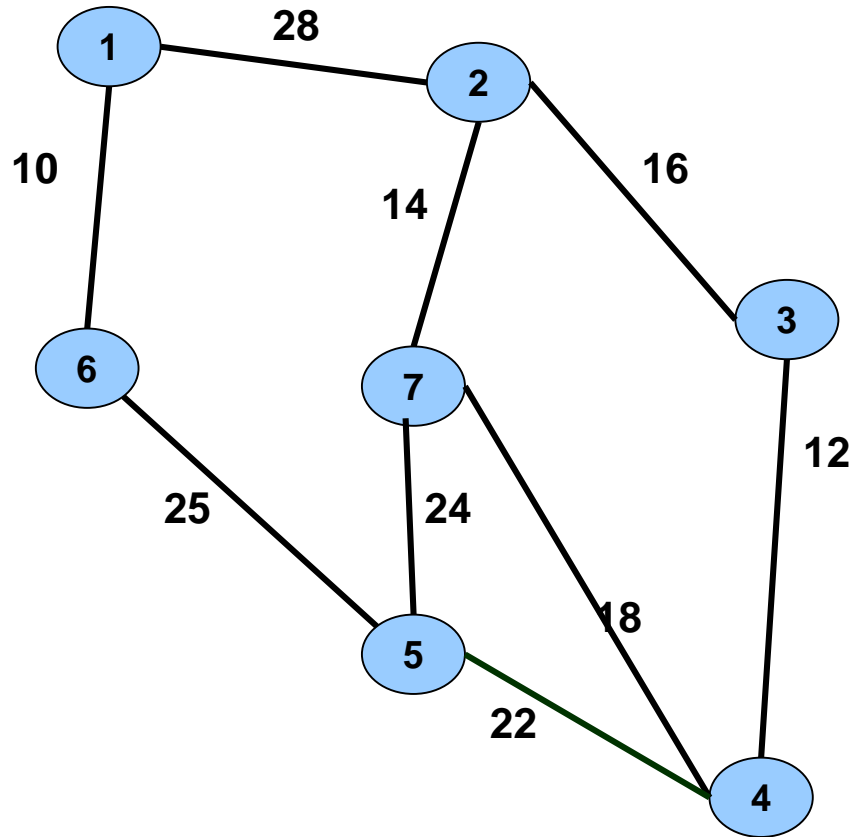
2. // E is the set of edges in G . G has n vertices. $\text{Cost}[u, v]$ is the
3. // cost of edge (u, v) . T is the set of edges in the minimum-cost
4. // spanning tree. The final cost is returned
5. {
6. construct a heap out of the edge costs using heapify;
7. for $i=1$ to n do $\text{parent}[i] = -1$;
8. // each vertex is in a different set.
9. $i=0$; $\text{mincost}=0.0$;
10. while $((i < n-1) \text{ and } (\text{heap not empty}))$ do
11. {

```

13.      Delete a minimum cost edge(u,v) from the heap
14.      and reheapify using Adjust;
15.      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.}

```

Example of Kruskal's Algorithm



Need to construct a heap with edge cost

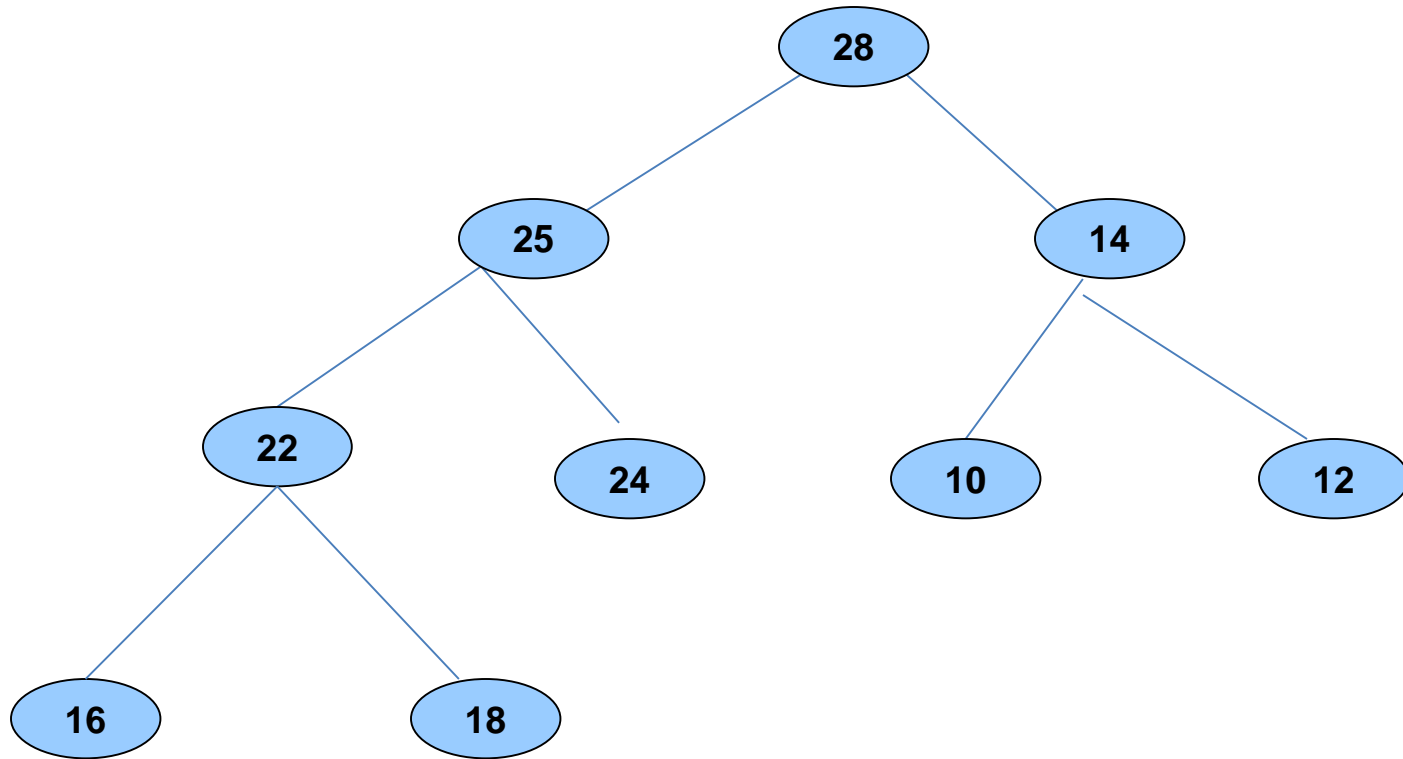
The initial array is

{ 28, 16, 12, 22, 25, 10, 14, 24, 18}

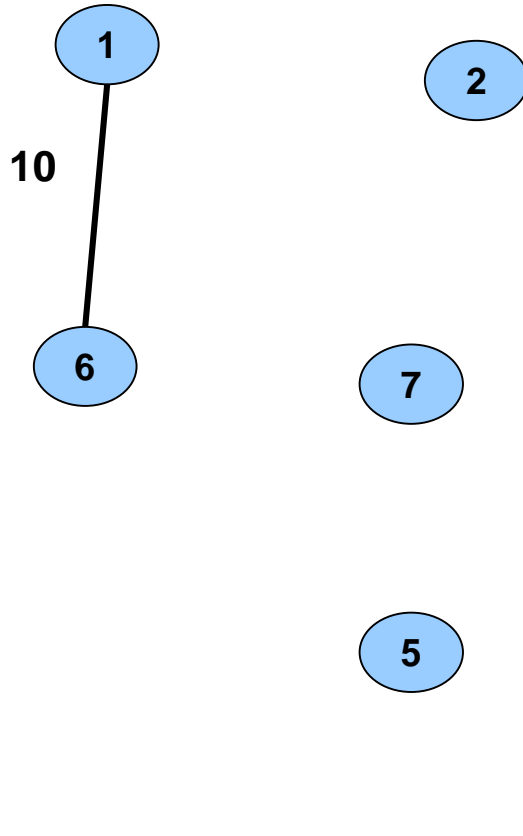
After Heapify the array becomes

A={ 28 25 14 24 22 10 12 16 18}

So The Resulting Heap Is



Example of Kruskal's Algorithm



The Initial Set Is

$\{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\}$

Select the minimum cost edge

$(u,v)=(1,6)$

$i=1$

$J=\{1\} \ k=\{6\}$

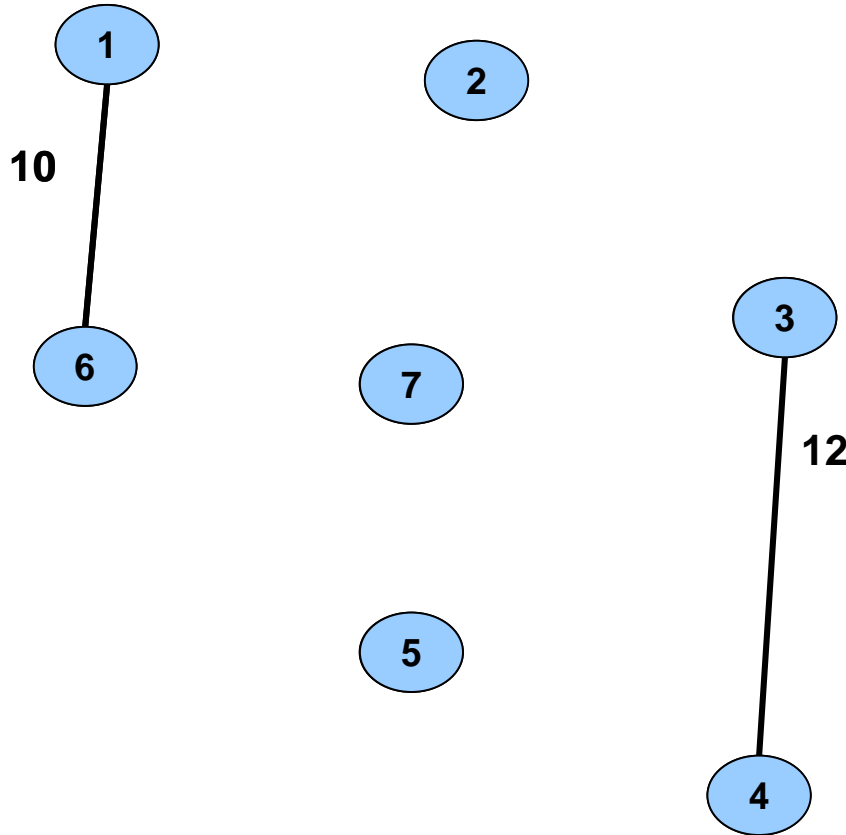
Mincost=10



The set becomes

$\{1\ 6\} \{2\} \{3\} \{4\} \{5\} \{7\}$

Example of Kruskal's Algorithm



Next smallest weight is 12

$(u,v)=(3,4)$

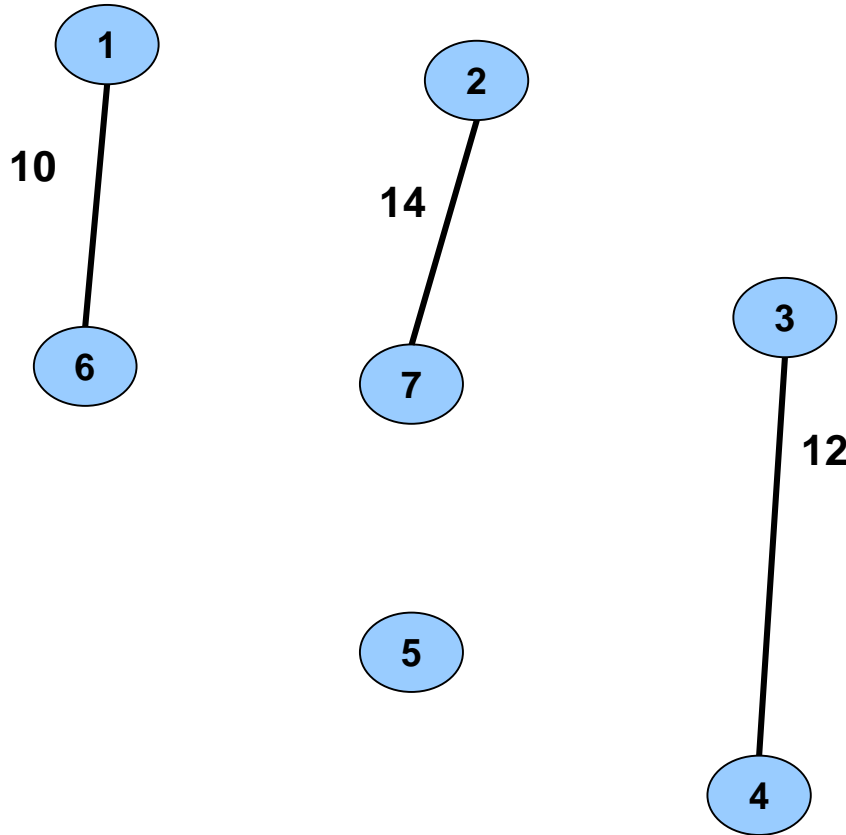
$J=\{3\}$ $k=\{4\}$

$i=2$

$\text{Mincost}=10+12=22$

The set becomes
 $\{1\ 6\} \{2\} \{3\ 4\} \{5\} \{7\}$

Example of Kruskal's Algorithm



Smallest weight is 14

$(u,v)=(2,7)$

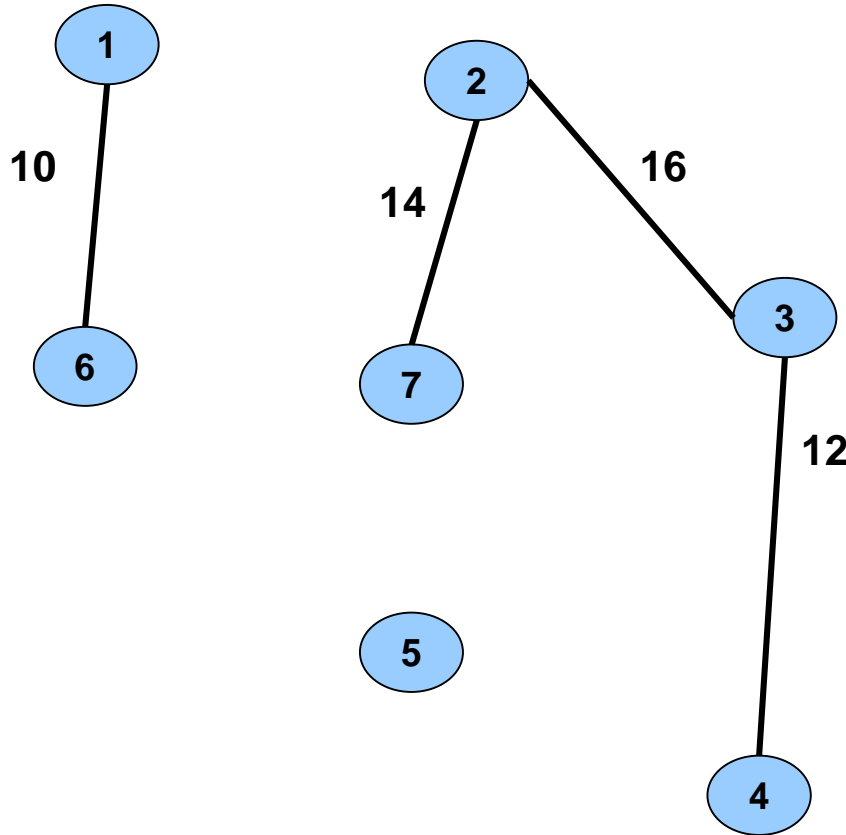
$J=\{2\}$ $k=\{7\}$

$i=3$

$\text{Mincost}=22+14=36$

The set becomes
 $\{1\ 6\} \{2\ 7\} \{3\ 4\} \{5\}$

Example of Kruskal's Algorithm



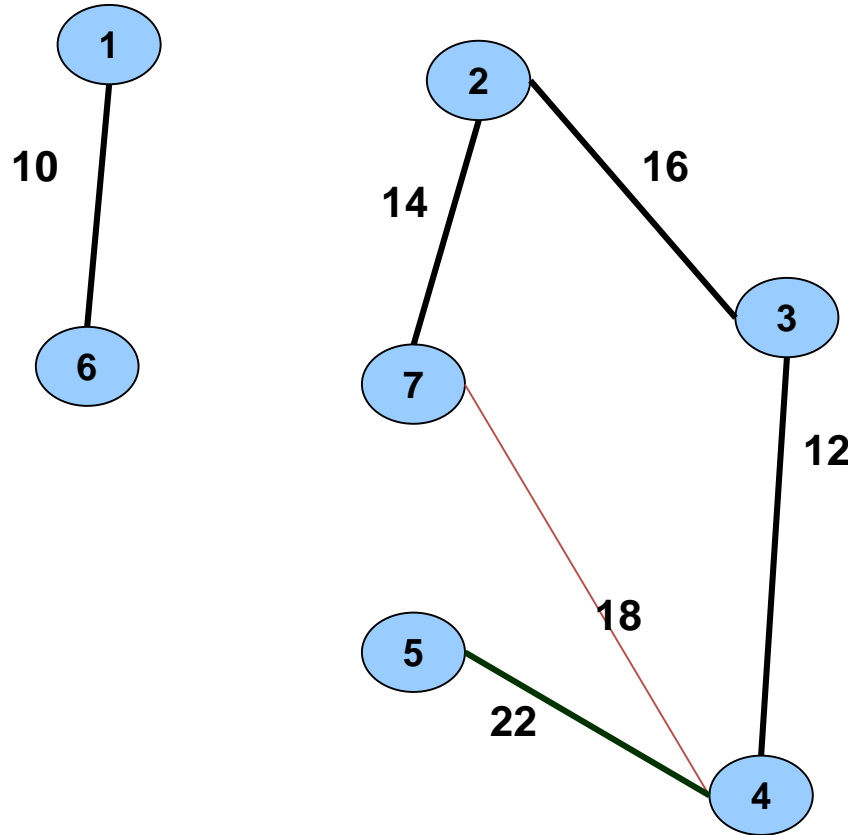
$(u,v)=(2,3)$
 $J=\{2\ 7\}$ $k=\{3\ 4\}$

$i=4$

$\text{Mincost}=36+16=52$

The set becomes
 $\{1\ 6\}$ $\{2\ 3\ 4\ 7\}$ $\{5\}$

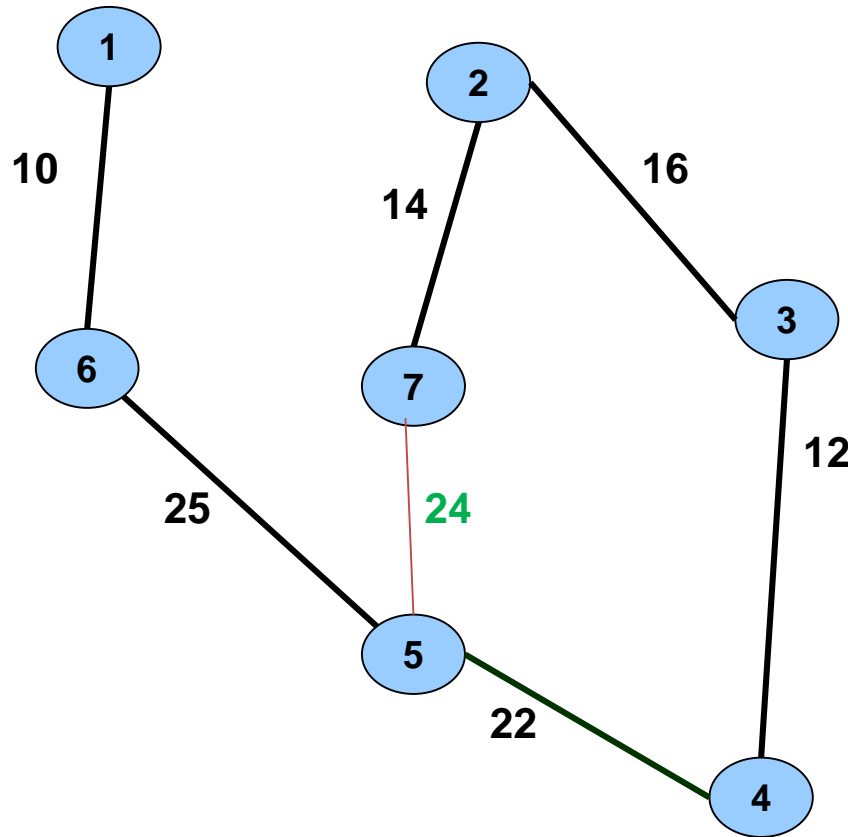
Example of Kruskal's Algorithm



Next smallest cost is 18.
But form a cycle
So we select next smallest 22
 $(u,v)=(4,5)$
 $J=\{2\ 3\ 4\ 7\}$ $k=\{5\}$
 $i=5$
Mincost= $52+22=74$

$\{1\ 6\}$ $\{2\ 3\ 4\ 7\}$ $\{5\}$
Since $(4,7)$ is on the same set we discard it.
So The set becomes $\{1\ 6\}$ $\{2\ 3\ 4\ 5\ 7\}$

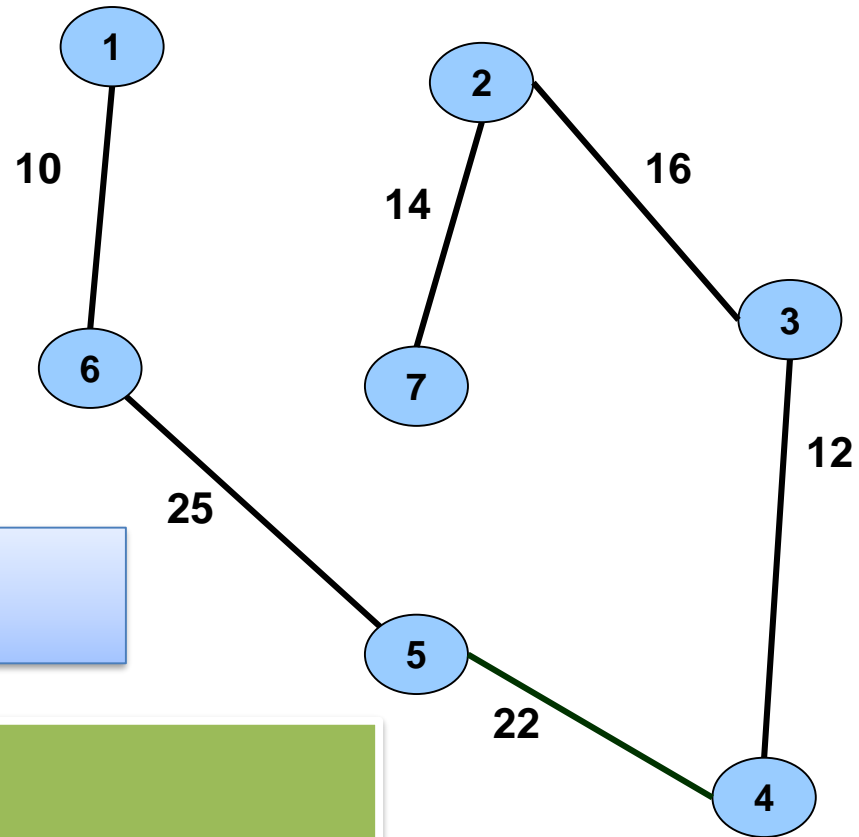
Example of Kruskal's Algorithm



$\{1\ 6\}$ $\{2\ 3\ 4\ 5\ 7\}$
Since (5,7) is on the same set we discard it.
So The set becomes $\{1\ 2\ 3\ 4\ 5\ 6\ 7\}$

Next smallest cost is 24
But form a cycle
So we select next smallest 25
 $(u, v) = (5, 6)$
 $J = \{2\ 3\ 4\ 5\ 7\}$ $k = \{1\ 6\}$
 $i = 6$
Mincost = $74 + 25 = 99$

Example of Kruskal's Algorithm



So the MST using Kruskal's algorithm
Mincost=99

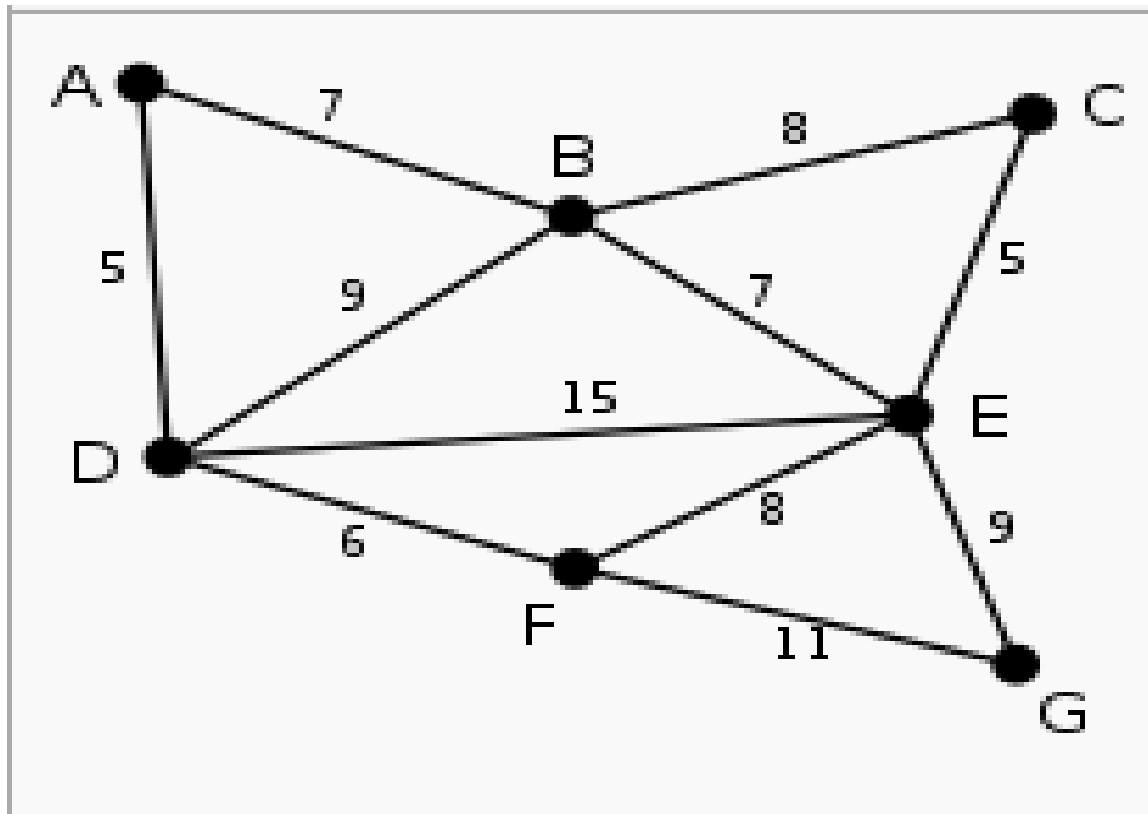
The final set is
 $\{1\ 2\ 3\ 4\ 5\ 6\ 7\}$

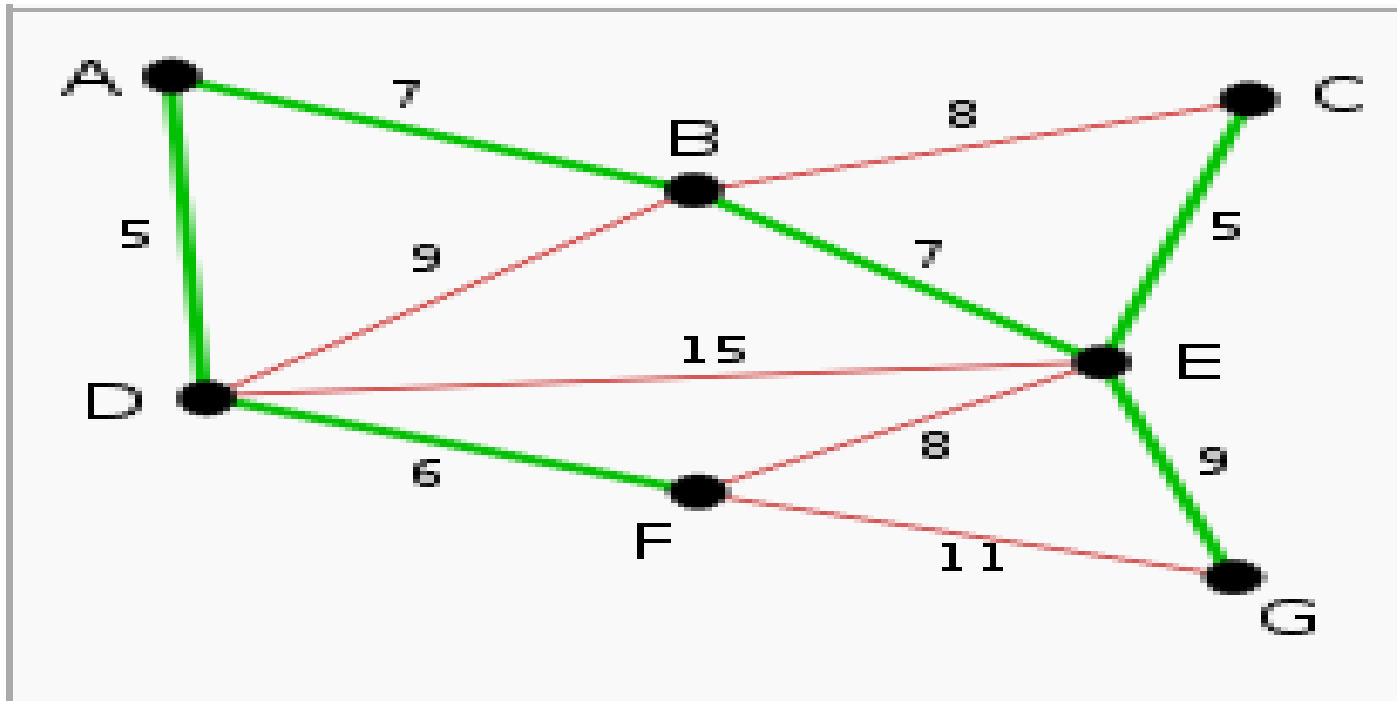
Analysis of Kruskal

- (initialization): $O(V)$
- (sorting): $O(E \log E)$
- (set-operation): $O(E \log E)$

Total: $O(E \log E)$

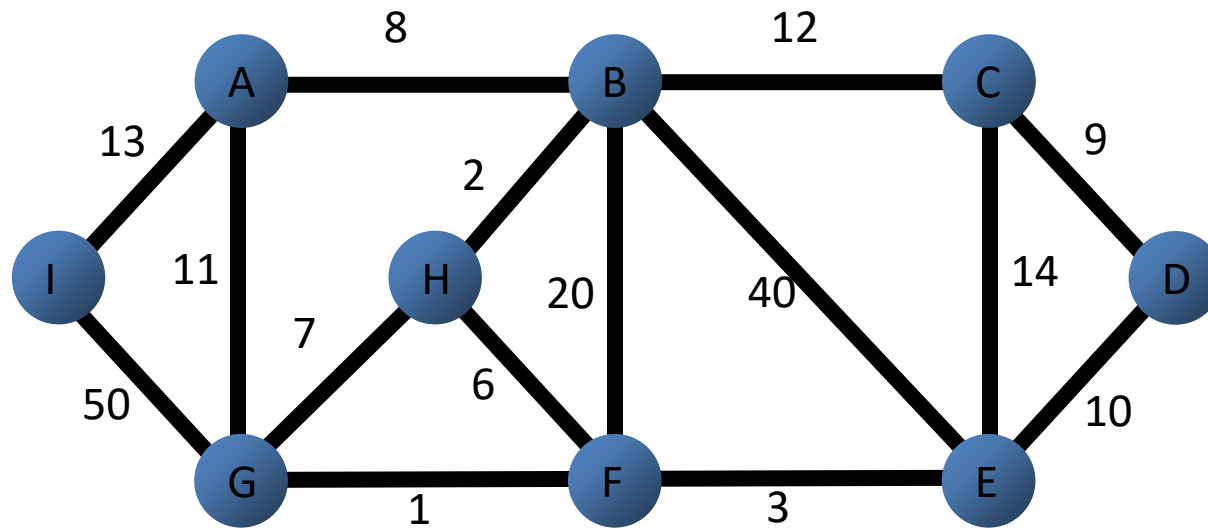
Home Work 1

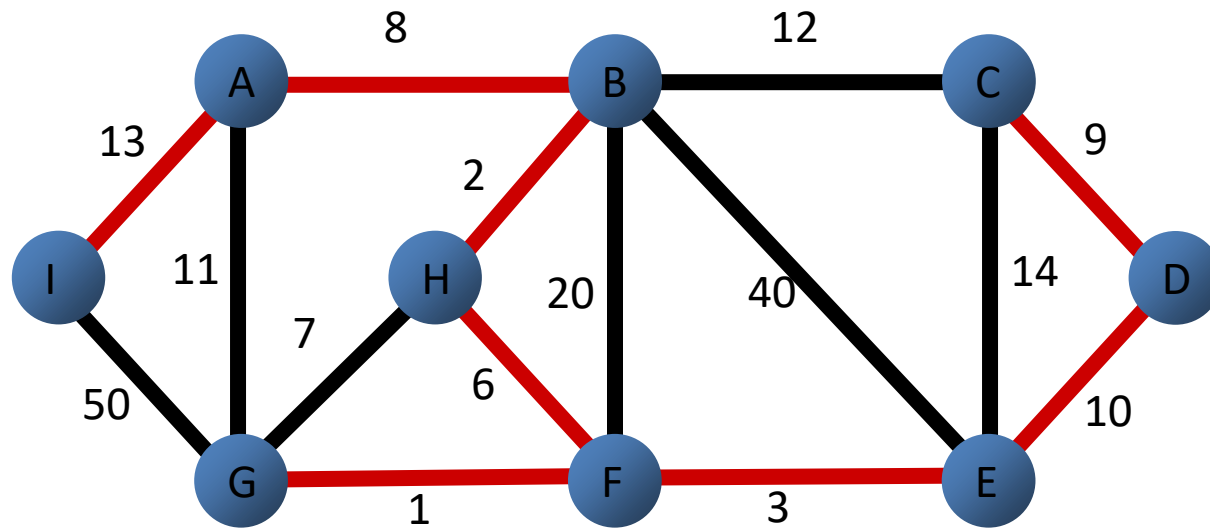




SOLUTION
MINCOST=34

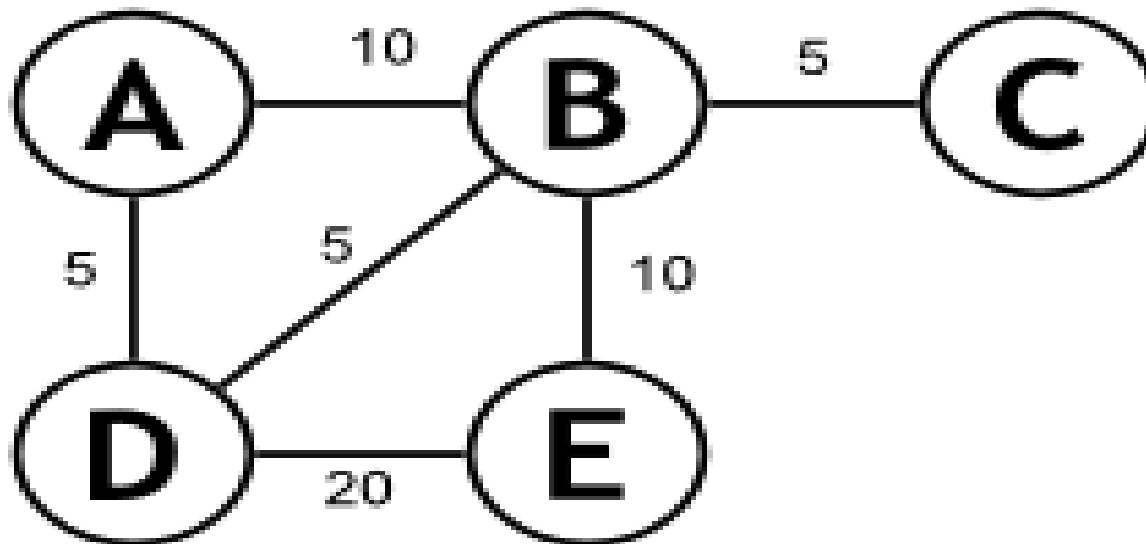
Home Work 2



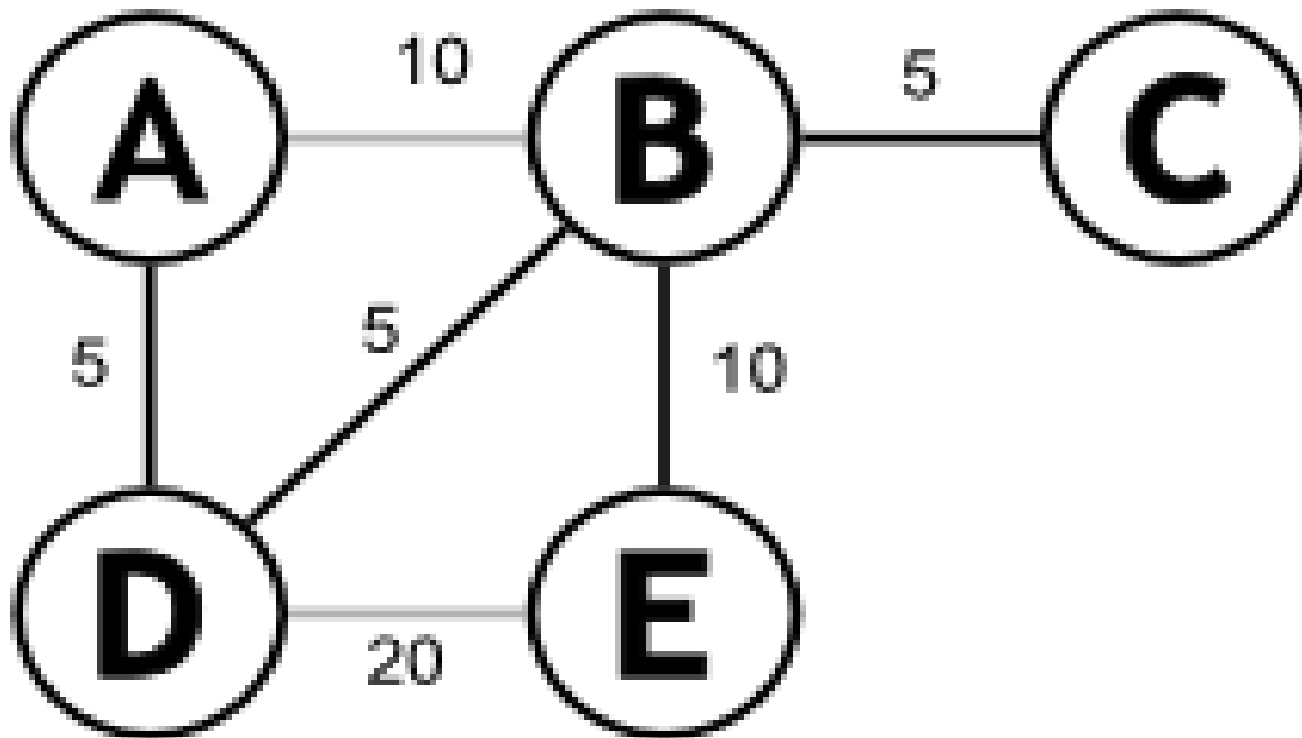


SOLUTION
MINCOST=52

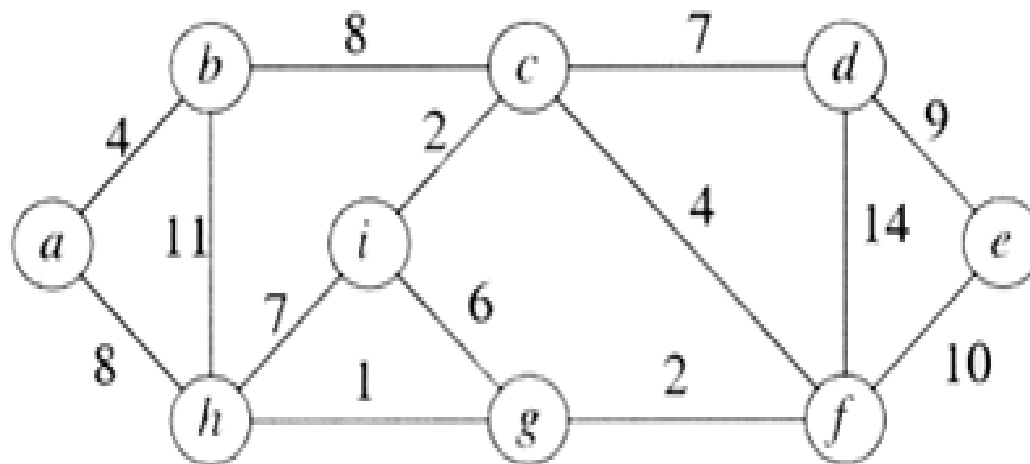
Home Work 3



SOLUTION
MINCOST=25

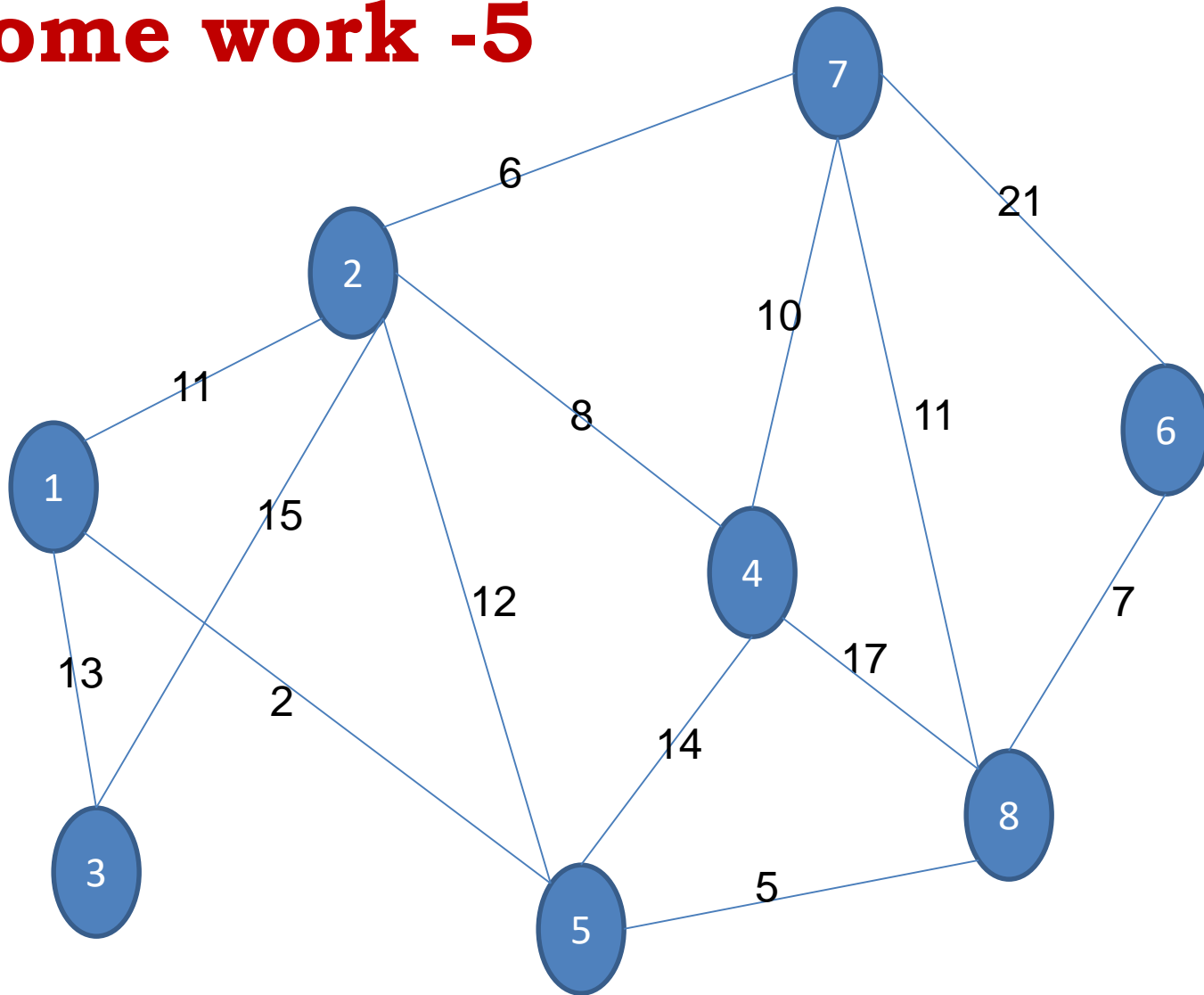


Home Work 4

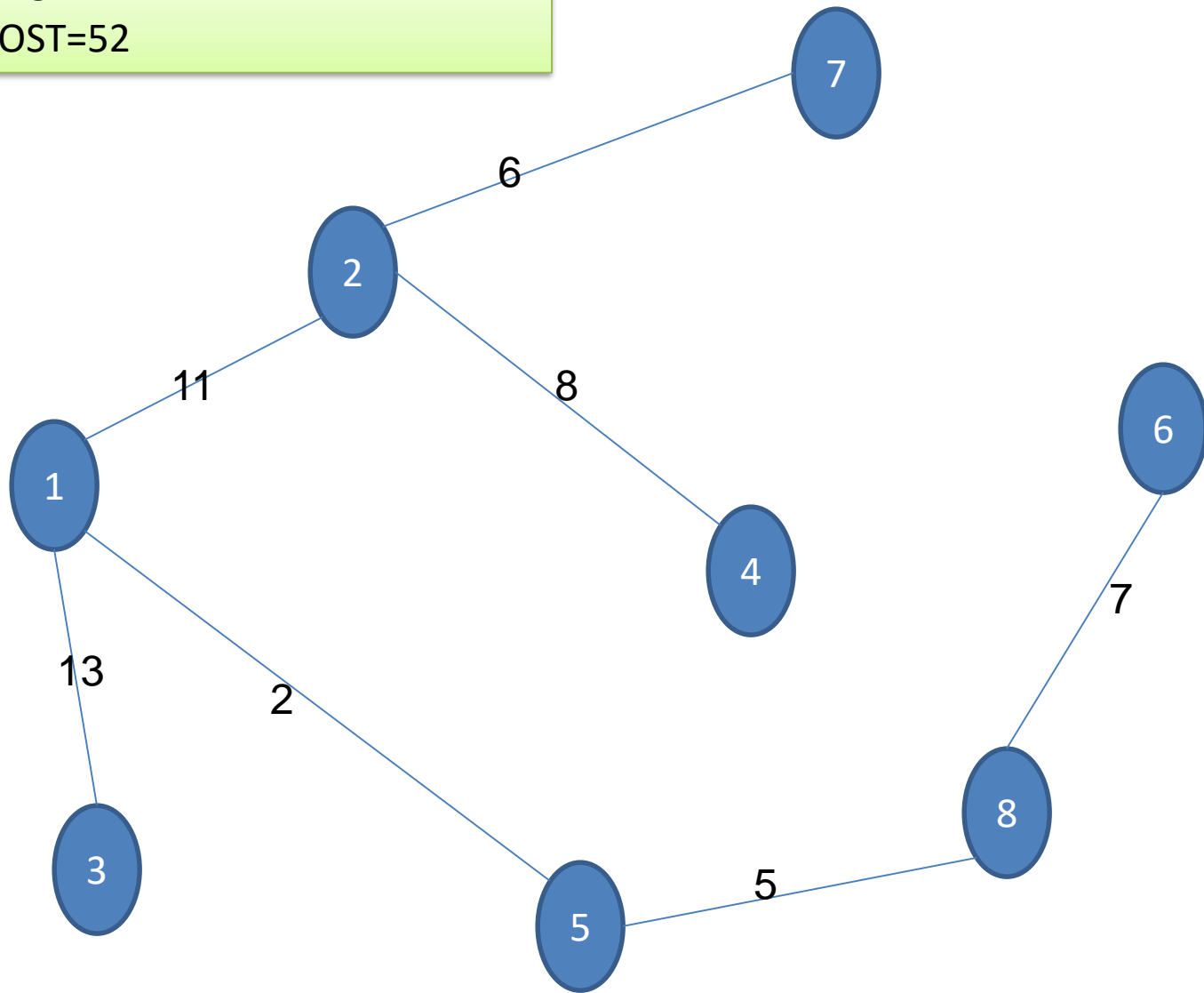


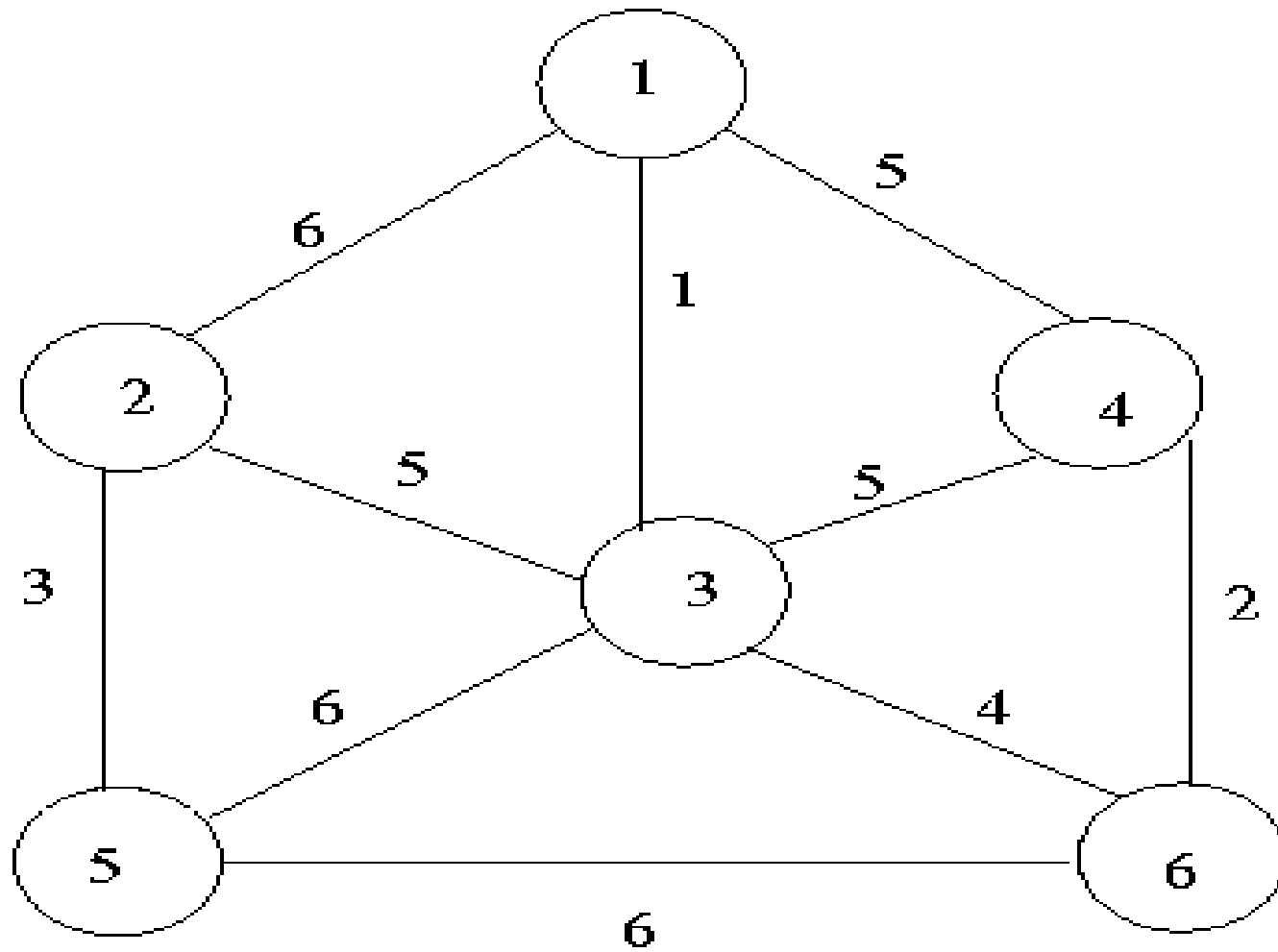


Home work -5

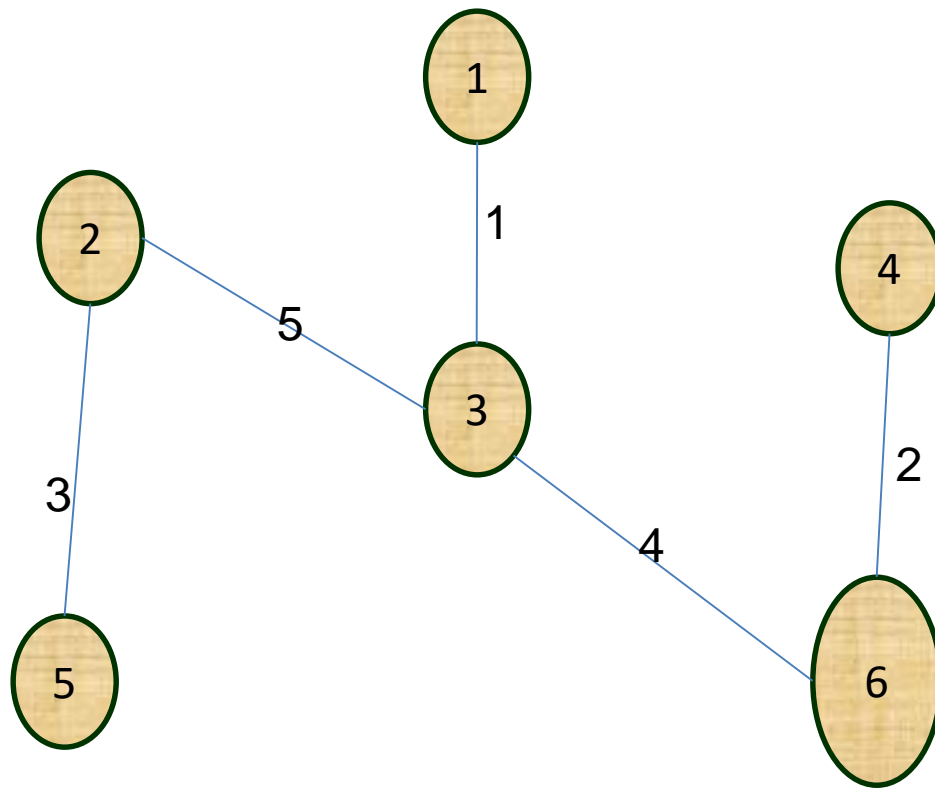


SOLUTION
MINCOST=52





SOLUTION
MINCOST=15



Comparison between prims and kruskal algorithm

- A tree is always connected when the MST is build using the prim's method. The tree is not necessary always connected when the MST is built using the kruskal method
- In prims method, the edge with minimum weight is selected from the vertex already in MST, but in kruskal method, the edge selected with the minimum cost need not be connected with the vertex in in MST.
- Both methods avoid formation of cycle when adding the edge.

Job sequencing with Deadlines

Problem

- n jobs, $S=\{1, 2, \dots, n\}$,
- each job i has a deadline $d_i \geq 0$ and a profit $p_i \geq 0$.
- For any job i profit p_i is earned if and only if the job is completed by the deadline.
- To complete a job one has to process the job on a machine for one unit time .
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.
- The value of a feasible solution J is the sum of the profits of the jobs in J .
- An optimal solution is a feasible solution with maximum value.

Job Sequencing With Deadlines

Given :

- n jobs $1, 2, \dots, n$
- deadline $d_i > 0$ each job taken 1 unit time
- profit $p_i > 0$ 1 machine available

Find

$$J_i \{1, 2, \dots, n\}$$

Feasibility:

The jobs in J can be completed before their deadlines

Optimality:

$$\text{maximize } \sum_{i \in J} P_i$$

Example:

Let $n = 4$,

$(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

The feasible solutions and their values are

Feasible solution	Processing sequence	Value or Profit
(1 , 2)	2,1	$100+10=110$
(1,3)	1,3 or 3,1	$100+15=115$
(1 , 4)	4,1	$100+27=127$
(2,3)	2,3	$10+15=25$
(3,4)	4,3	$15+27=42$
1	1	100
2	2	10
3	3	15
4	4	27

General Method

Algorithm GreedyJob(d, J, n)

// J is the set of jobs that can be completed by their deadline

```
{  
     $J := \{1\};$   
     $i := 2$  to  $n$  do  
    {  
        If (all jobs of  $J \cup \{i\}$  can be completed by their deadlines)  
  
        then  
             $J := J \cup \{i\};$   
    }  
}
```

Algorithm JS(d, j, n)

// $d[i] \geq 1, 1 \leq i \leq n$ are deadlines, $n \geq 1$. the jobs are ordered
such that $p[1] \geq p[2] \geq \dots \geq p[n]$. $J[i]$ is the i th job in an
optimal // solution, $1 \leq i \leq k$

{


$d[0] := J[0] := 0$; // initialization

$j[1] := 1$;

$k := 1$;

For $i := 2$ to n do

{



Execute
in $n-1$
times

//consider jobs in non increasing order of $p[i]$. Find position for i and check feasibility of insertion

$r:=k;$

While($(d[j[r]]>d[i])$ and $(d[j[r]] \neq r)$) do $r:=r-1;$

If($(d[j[r]]\leq d[i])$ and $(d[i]>r)$ then

{

// Insert i into j [].

For $q:=k$ to $(r+1)$ to step-1 do $j[q+1]:=j[q];$

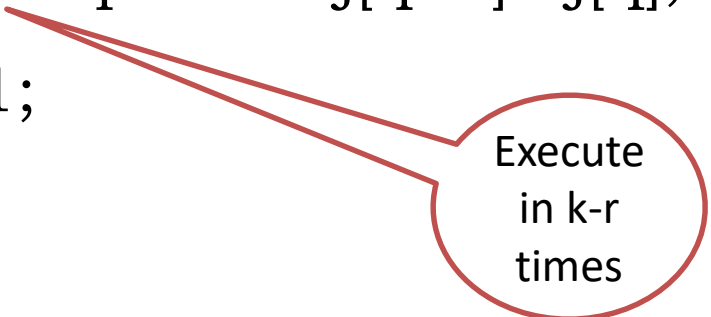
$J[r+1]:=i;$ $k:=k+1;$

}

}

return $k;$

}



Execute
in $k-r$
times

Algorithm:

Step 1: Sort p_i into nonincreasing order. After sorting $p_1 \geq p_2 \geq p_3 \geq \dots \geq p_i$.

Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot $[r-1, r]$, where r is the largest integer such that $1 \leq r \leq d_i$ and $[r-1, r]$ is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

Time complexity: $O(n^2)$

Example:

Let $n = 4$,

$(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

After the first step the problem becomes

$(P_1, P_2, P_3, P_4) = (100, 27, 15, 10)$

$(W_1, W_2, W_3, W_4) = (2, 1, 2, 1)$

Initially $d[0] = J[0] = 0$

$J[1] = 1; k = 1$

$i = 2$

$r = 1$

While $(d(1) > d(2) \ \& \ d(1) \neq 1)$ **true**

$r = 0$

$J[2] = 1; J[1] = 2; K = 2$

$i = 3$

While $(d(1) > d(3) \ \& \ d(1) \neq 2)$ **false**

$i = 4$

While $(d(1) > d(4) \ \& \ d(1) \neq 2)$ **false**

Return $k = 2$

Solution table

Job J[]	d_i	p_i	Total	Sequence
0	0	0		0
{1}	2	100	100	1,0
{2,1}	1	27	127	2,1,0
{3,2,1}	2	15	Not feasible or reject	
{4,2,1}	1	10	Not feasible or reject	

Complexity is $O(n^2)$

What is the solution generated by the function JS when $n=5$,

$(p_1, p_2, \dots, p_5) = (20, 15, 10, 5, 1)$

$(d_1, d_2, \dots, d_5) = (2, 2, 1, 3, 3)$

What is the solution generated by the function JS when $n=5$,

$(p_1, p_2, p_3, p_4) = (50, 10, 15, 30)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

What is the solution generated by the function JS when $n=5$,

$(p_1, p_2, \dots, p_6) = (20, 15, 10, 7, 5, 3)$

$(d_1, d_2, \dots, d_6) = (3, 1, 1, 3, 1, 3)$

Thank
you
😊