Advanced Data Structures and Algorithms

GREEDY METHOD

Dr G.Kalyani
Department of Information Technology
Velagapudi Ramakrishna Siddhartha Engineering College

Topics

- General Method
- Knapsack Problem
- Minimum Cost Spanning Trees
- Single Source Shortest Path Problem
- Job Sequencing with Deadlines

Greedy Method

- The **problems have n inputs** and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies those constraints is called a **feasible solution.**
- We need to find a feasible solution that either maximizes or minimizes a given **objective function.**
- A feasible solution that does this is called an **optimal solution**.
- In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

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 algorithm is an algorithmic paradigm based on heuristic that follows **local optimal choice at each step with the hope of finding global optimal solution**.

Control Abstraction of Greedy Method

```
Algorithm Greedy($a, n$)
// $a[1:n]$ contains the $n$ inputs.
{
    $solution := \emptyset$; // Initialize the solution.
    for $i := 1$ to $n$ do
    {
        $x :=$ Select($a$);
        if Feasible($solution, x$) then
            $solution :=$ Union($solution, x$);
    }
    return $solution$;
}
```

Select: selects an input from a[] with an optimal strategy. The selected input's value is assigned to x.

Feasible is a Boolean-valued function that determines whether x can be included into the solution vector or not.

Union combines x with the solution and updates the objective function.

Types of Greedy Method

- Subset Paradigm
  - To solve a problem (or possibly find the optimal/best solution), greedy approach generate subset by selecting one or more available choices.
- Ordering Paradigm
  - In this, greedy approach generate some arrangement/order to get the best solution.

Topics

- General Method
- Knapsack Problem
- Minimum Cost Spanning Trees
- Single Source Shortest Path Problem
- Job Sequencing with Deadlines

Knapsack Problem

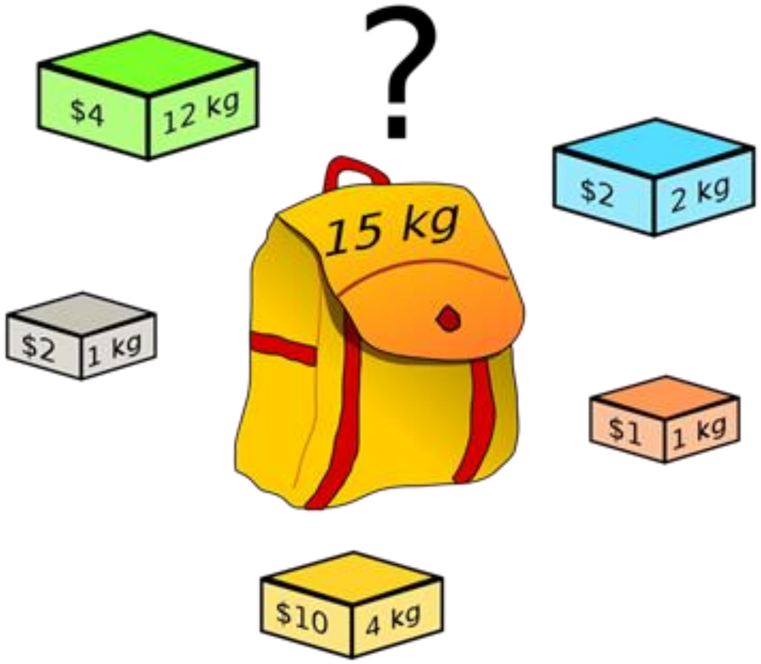
You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value/profit.

The problem states-

Which items should be placed into the knapsack such that-

- The **value or profit** obtained by putting the items into the knapsack is **maximum.**
- the **weight limit** of the knapsack does **not exceed.**



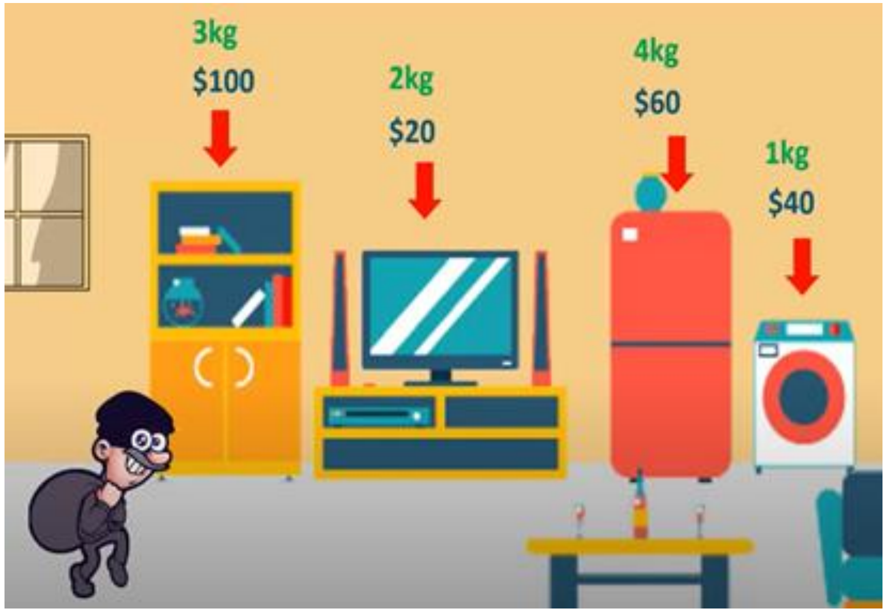
Knapsack Problem Variants & its differences

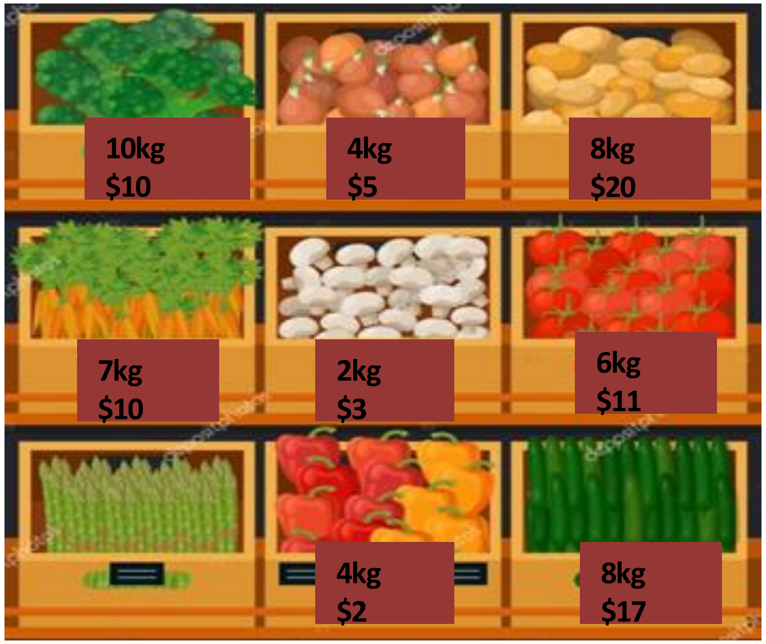
0/1 Knapsack.

- not allowed to take part of the items.
- Either take the whole item or don’t take it.

Fractional Knapsack

- can consider the items based on requirement for maximizing the total profit.





Fractional Knapsack Problem

- N -number of objects

  M -knapsack or bag Capacity

  p_i - a positive profit

  w_i - a positive weight

  x_i – Selection Vector [0 - 1](allows fractional values)

Fractional Knapsack Problem

- Let, we are given n objects and a Knapsack or Bag of capacity M.
- Object i has weight Wi and profit Pi.
- if a fraction (Xi) of object i is placed into Knapsack, then a profit of Pi*Xi is earned.
- The **objective is to obtain a filling of Knapsack that maximizes the total profit earned.**

Fractional Knapsack Problem

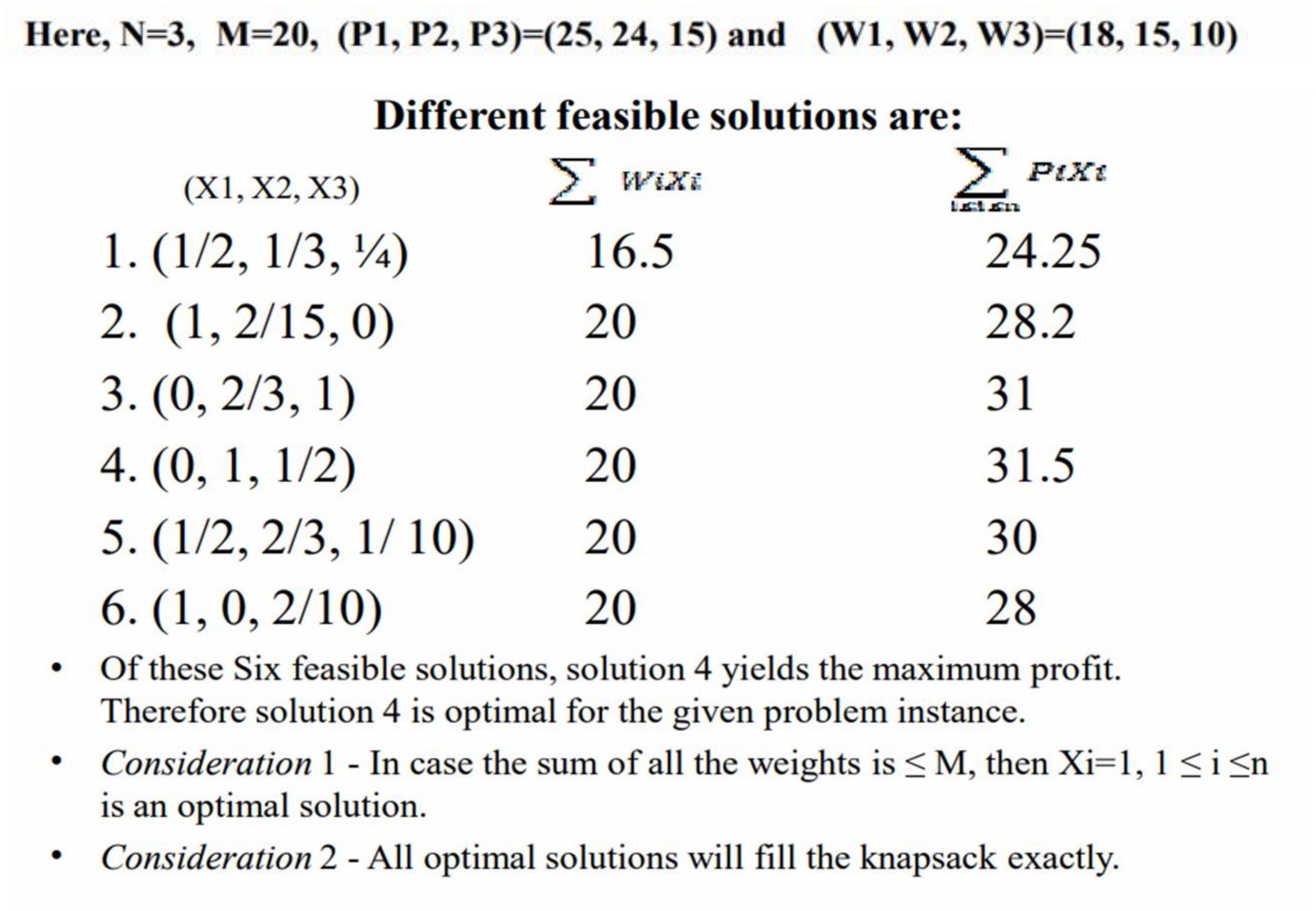
$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \quad \text{----------A}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \text{ -------- B}$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \text{ ------ C}$$

- The profit and weights are the positive numbers.
- Here, A feasible solution is any set (X1, X2, …, Xn) satisfying above rules (B) and (C).
- An optimal solution is feasible solution for which rule (A) is maximized.

Example-1 for Knapsack Problem



Here, N=3, M=20, (P1, P2, P3)=(25, 24, 15) and (W1, W2, W3)=(18, 15, 10)

Different feasible solutions are:

(X1, X2, X3)	$\sum WiXi$	$\sum_{1 \le i \le n} PiXi$
1. (1/2, 1/3, ¼)	16.5	24.25
2. (1, 2/15, 0)	20	28.2
3. (0, 2/3, 1)	20	31
4. (0, 1, 1/2)	20	31.5
5. (1/2, 2/3, 1/ 10)	20	30
6. (1, 0, 2/10)	20	28

- Of these Six feasible solutions, solution 4 yields the maximum profit. Therefore solution 4 is optimal for the given problem instance.
- *Consideration* 1 - In case the sum of all the weights is ≤ M, then Xi=1, 1 ≤ i ≤n is an optimal solution.
- *Consideration* 2 - All optimal solutions will fill the knapsack exactly.

Knapsack Problem – Strategy-1: Random Selection

Example:

Consider the following instance of the knapsack problem

$n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$

Objects = {A, B, C}

Maximum Weight that Knapsack can hold (m) = 20

Objects	Profit (p_i)	Weight (w_i)
A	25	18
B	24	15
C	15	10
Maximum Weight that Knapsack can hold (m) = 20		

Solution:1 **Randomly selected**

Selection Vector (x_1, x_2, x_3) : $(1/2, 1/3, 1/4)$

Weight of Selected Objects ($\sum_{i=1}^{n} w_i x_i$) = (1/2 *18 + 1/3 *15 + 1/4*10) = (9+5+2.5) = 16.5

$Profit\ of\ the\ selected\ Object\ (\sum_{i=1}^{n} \rho_i x_i)$ = (1/2 *25 + 1/3 *24 + 1/4*15) = (12.5+8+3.75) = 24.25

Knapsack Problem – Strategy-2: Decreasing Order of Profits

decreasing order of **profits** : A, B, C

Objects	Profit (p_i)	Weight (w_i)
A	25	18
B	24	15
C	15	10
Maximum Weight that Knapsack can hold (m) = 20		

Selection Vector (x_1,x_2,x_3) : $(1,2/15,0)$

Knapsack Weight = $\sum w_i x_i$ = $(1*18 + 2/15*15 + 0*10) = (18+2+0) = 20$

Profit = $\sum p_i x_i$ = $(1 *25+ 2/15 *24+0*15) = (25+3.2+0) = 28.2$

Knapsack Problem – Strategy-3: Increasing Order of Weights

Objects are arranged in

increasing order of **weights** : C, B, A

Objects	Profit (p_i)	Weight (w_i)
C	15	10
B	24	15
A	25	18
Maximum Weight that Knapsack can hold (m) = 20		

Selection Vector (x_1,x_2,x_3) : (0,2/3,1)

Knapsack Weight = $\sum w_i x_i$ = (0*18+ 2/3*15+1*10) = (0+10+10) = 20

Profit = $\sum p_i x_i$ = (0 *25+ 2/3 *24+1*15) = (0+16+15) = 31

Knapsack Problem – Strategy-4: Decreasing Order of P_i / W_i Ratio

- **The greedy principle:**
  - Put the objects into the knapsack according to the highest profit per unit weight until the knapsack filled completely.
- **Ex:** n=3,M=20,(p1,p2,p3)=(25,24,15)

  (w1,w2,w3)=(18,15,10)

Sol: p1/w1=25/18=1.39

p2/w2=24/15=1.6

p3/w3=15/10=1.5

Knapsack Problem – Strategy-4: Decreasing Order of P_i /W_i Ratio

$p_1/w_1 = 25/18 = 1.39$
$p_2/w_2 = 24/15 = 1.6$
$p_3/w_3 = 15/10 = 1.5$

- because w2<=M ➔ **X2=1**
- remaining M= 20-15= 5

- because w3>M, place only part of the object i.e., M/w3=5/10=½ ➔ **X3= ½**
- remaining M=5-5 = 0

- since remaining M=0 ➔ **X1=0**
- Hence the solution **X = (0, 1, ½)**

Knapsack Weight =$\sum w_ix_i$ =(0*18+ 1*15+1/2*10)=(0+15+5) = 20

Total Profit = $\sum$ p_ix_i =(0*25+ 1*24+1/2*15)=(0+24+7.5)= 31.5

Example-2 on Knapsack Problem

- N=7
- M=15
- (p1,p2...p7)=(10,5,15,7,6,18,3)
- (w1,w2..w7)=(2,3,5,7,1,4,1)

- P1/w1=5; p2/w2=1.66; p3/w3=3; p4/w4=1 p5/w5=6; p6/w6=4.5; p7/w7=3

- Descending order of Pi/Wi: x5,x1,x6,x3,x7,x2,x4

Example-2 Contd…

- Descending order of Pi/Wi: x5,x1,x6,x3,x7,x2,x4
- X5=1 (M= 15-1 = 14)
- X1=1 (M= 14-2 = 12)
- X6=1 (M= 12-4 = 8)
- X3=1 (M= 8-5 = 3)
- X7=1 (M=3-1 = 2)
- X2= w2>M hence X2=2/3

- **solution vector is (1,2/3,1,0,1,1,1)**

- **Weight is:** 1*2+(2/3)*3+1*5+0*7+1*1+1*4+1*1=15
- **Profit is:** 1*10+(2/3)*5+1*15+0*7+1*6+1*18+1*3 =55.34

Algorithm for Knapsack Problem

```
• Algorithm  GreedyKnapsack(m,n)
//order the n objects such that p[i]/w[i]≥p[i+1]≥w[i+1]
{
for i:=1 to n do x[i]:=0.0;
    U:=m;
for i:=1 to n do
 {
     if(w[i] > U) then break;
    x[i]:=1.0; U:=U-w[i];
}
if( i ≤ n) then x[i]:=U/w[i];
}
```

Algorithm for Knapsack Problem

```
Algorithm FractionalKnapsack(items[], n, capacity)
{
    Initialize totalValue = 0.0;
    Initialize used[] = [0, 0, ..., 0] ; / / n elements
    remainingCapacity = capacity;
    while (remainingCapacity > 0)
    {
        bestIndex = -1;    bestRatio = 0.0
        for i = 0 to n-1
        {
            if (used[i] == 0) then
            {
                ratio = items[i].value /items[i].weight
                if (ratio > bestRatio) then
                {
                    bestRatio = ratio;
                    bestIndex = I;
                }
            }
        }
        if bestIndex == -1 then  break; // No more items to consider
        used[bestIndex] = 1;
        if (items[bestIndex].weight <= remainingCapacity) then
            totalValue += items[bestIndex].value;
            remainingCapacity - = items[bestIndex].weight;
        else
            totalValue += items[bestIndex].value *
                (remainingCapacity / items[bestIndex].weight);
            remainingCapacity = 0 // Knapsack is full
    }                              // end of while

    return totalValue;
}
```

Examples for Practice

PROBLEM-1:

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24

PROBLEM-2:

For example, there are 10 different items and the weight limit is 67. So,

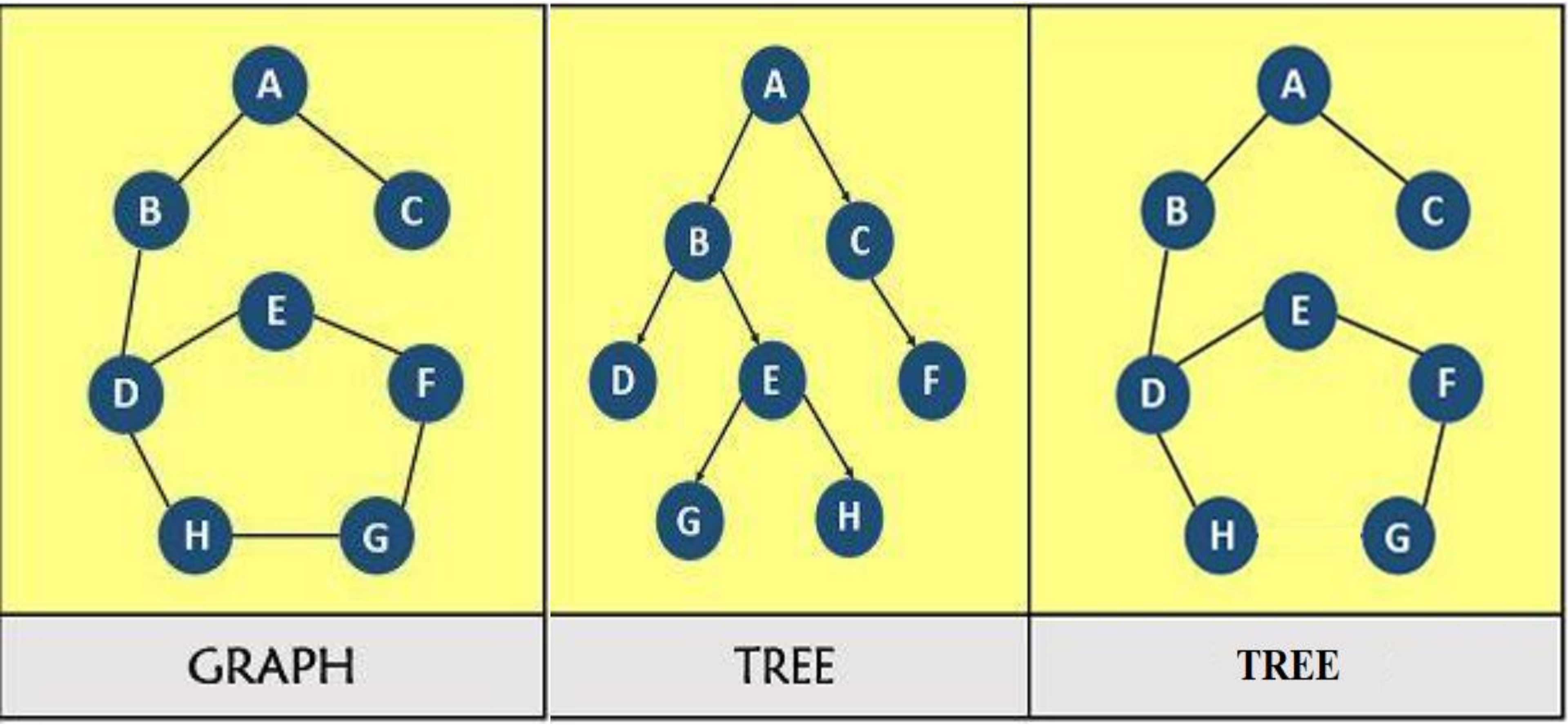
$$w[1] = 23, w[2] = 26, w[3] = 20, w[4] = 18, w[5] = 32, w[6] = 27, w[7] = 29, w[8] = 26, w[9] = 30, w[10] = 27$$

$$v[1] = 505, v[2] = 352, v[3] = 458, v[4] = 220, v[5] = 354, v[6] = 414, v[7] = 498, v[8] = 545, v[9] = 473, v[10] = 543$$

Topics

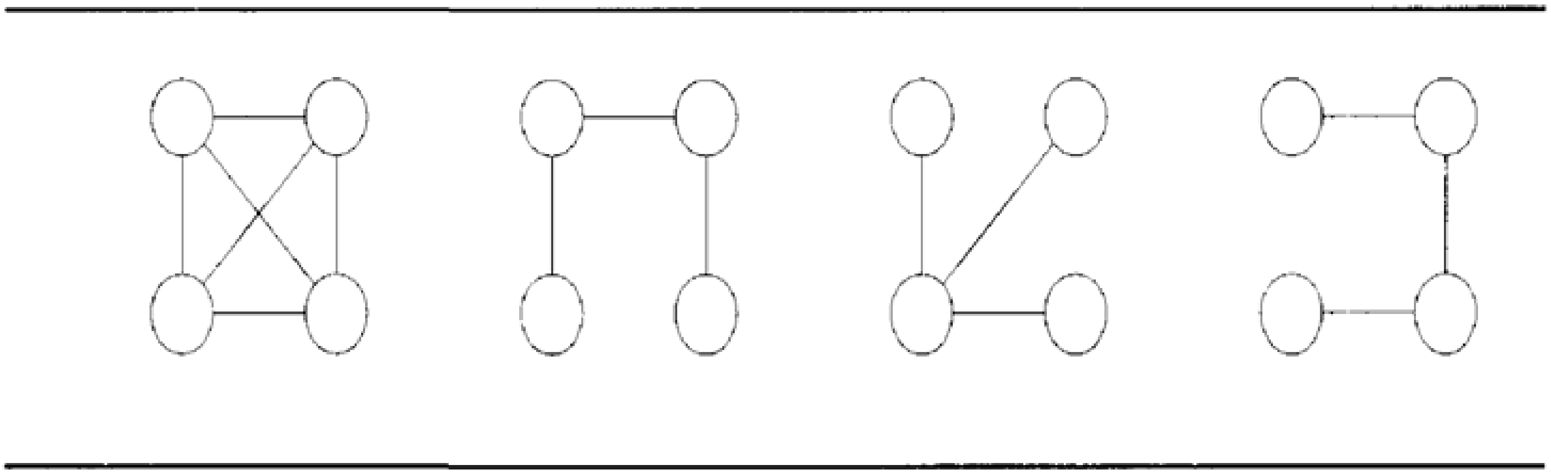
- General method
- Knapsack problem
- Minimum Cost Spanning Trees
- Single source shortest path problem
- Job Sequencing with deadlines

Graph vs Tree



Spanning Tree

Definition 4.1 Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a *spanning tree* of G iff t is a tree. $\square$

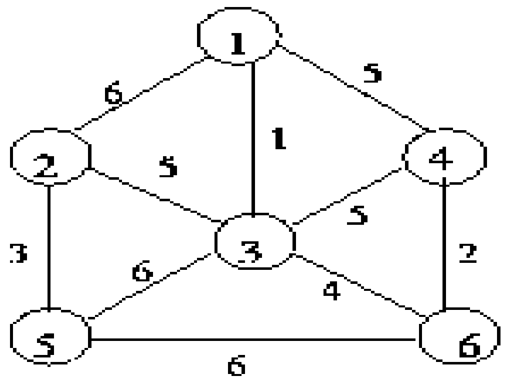


An undirected graph and three of its spanning trees

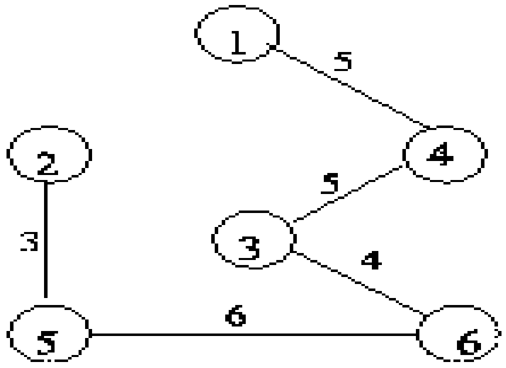
Minimum Cost Spanning Tree

- In practical situations, the **edges have weights** assigned to them. These weights may represent the cost of construction, the length of the link, and so on.
- **Given such a weighted graph,** one would then wish to **select spanning tree which have minimum total cost** or minimum total length.
- **Minimum Cost Spanning Tree problem is finding a spanning tree of G with minimum cost.** The cost of a spanning tree is the sum of the costs of the edges in that tree.

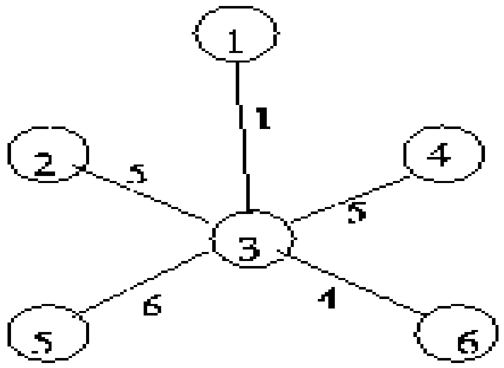
Example for Minimum Cost Spanning Tree



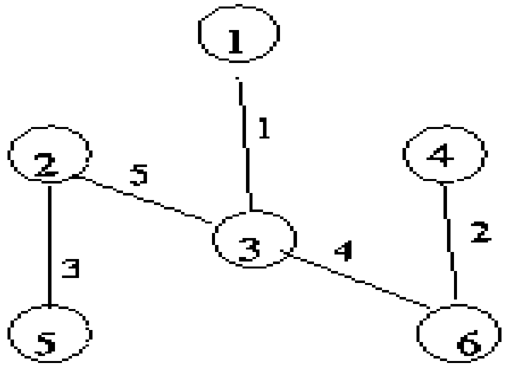
A connected graph.



A spanning tree with cost = 23



Another spanning tree with cost 21



MST, cost = 15

Minimum Cost Spanning Trees

- A greedy method to obtain a minimum-cost spanning tree builds this tree Edge by edge.
- The next edge to include is chosen according to some criterion.
- The simplest such criterion is to **choose an edge that results in a minimum increase in the sum of the costs of the edges** so far included.
- There are **two possible ways** to interpret this criterion.

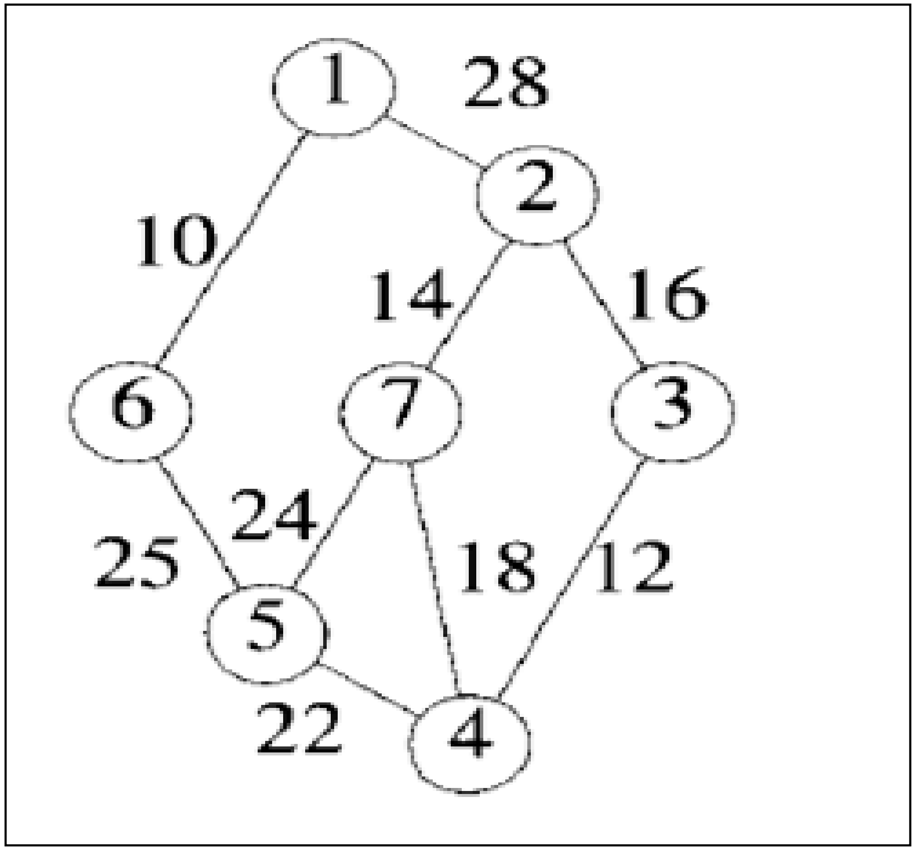
Minimum Cost Spanning Trees

- **Prims Algorithm:**
  - The set of edges (S) so far selected form a tree.
  - The next edge(u,v) to be included in S is a minimum-cost edge not in S with the property that A U {(u,v)} is also a tree.

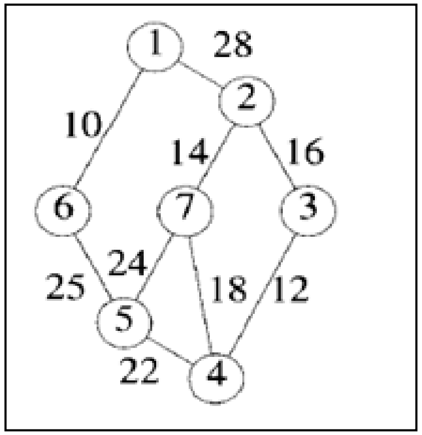
- **Kruskals Algorithm:**
  - The set of edges (S) so far selected may not form a tree.
  - The next edge(u,v) to be included in S is a minimum-cost edge not in S with the property that A U {(u,v)} not forms a cycle.

An Example for Prims Algorithm

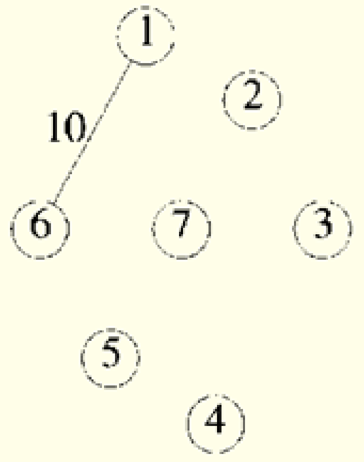
Given Graph

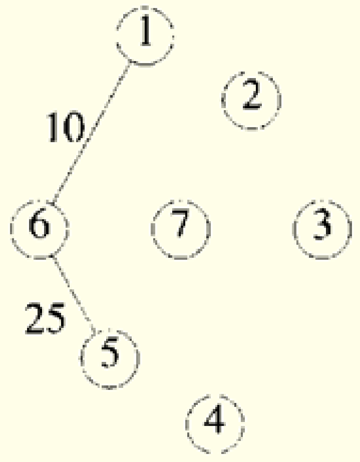


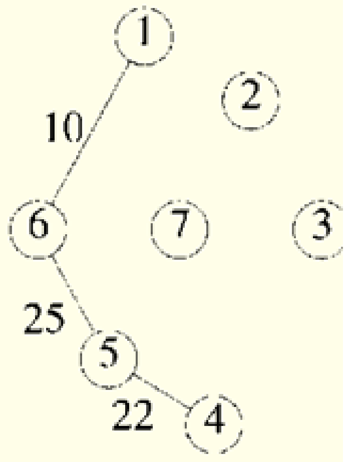
An Example for Prims Algorithm

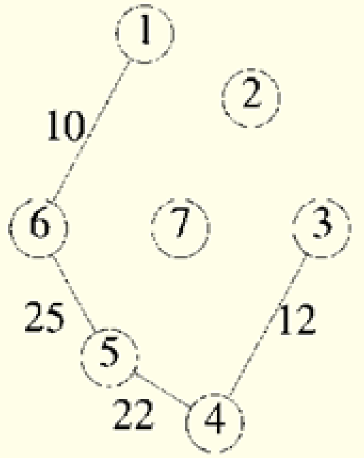


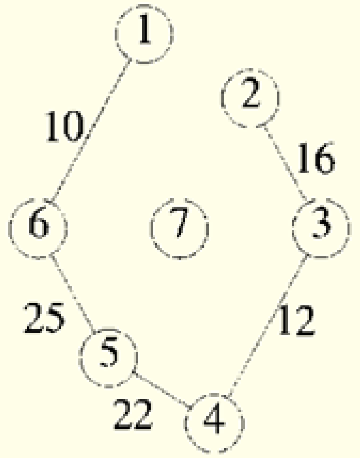
Given Graph

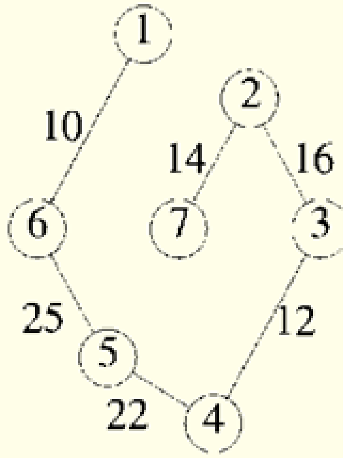

(a)


(b)

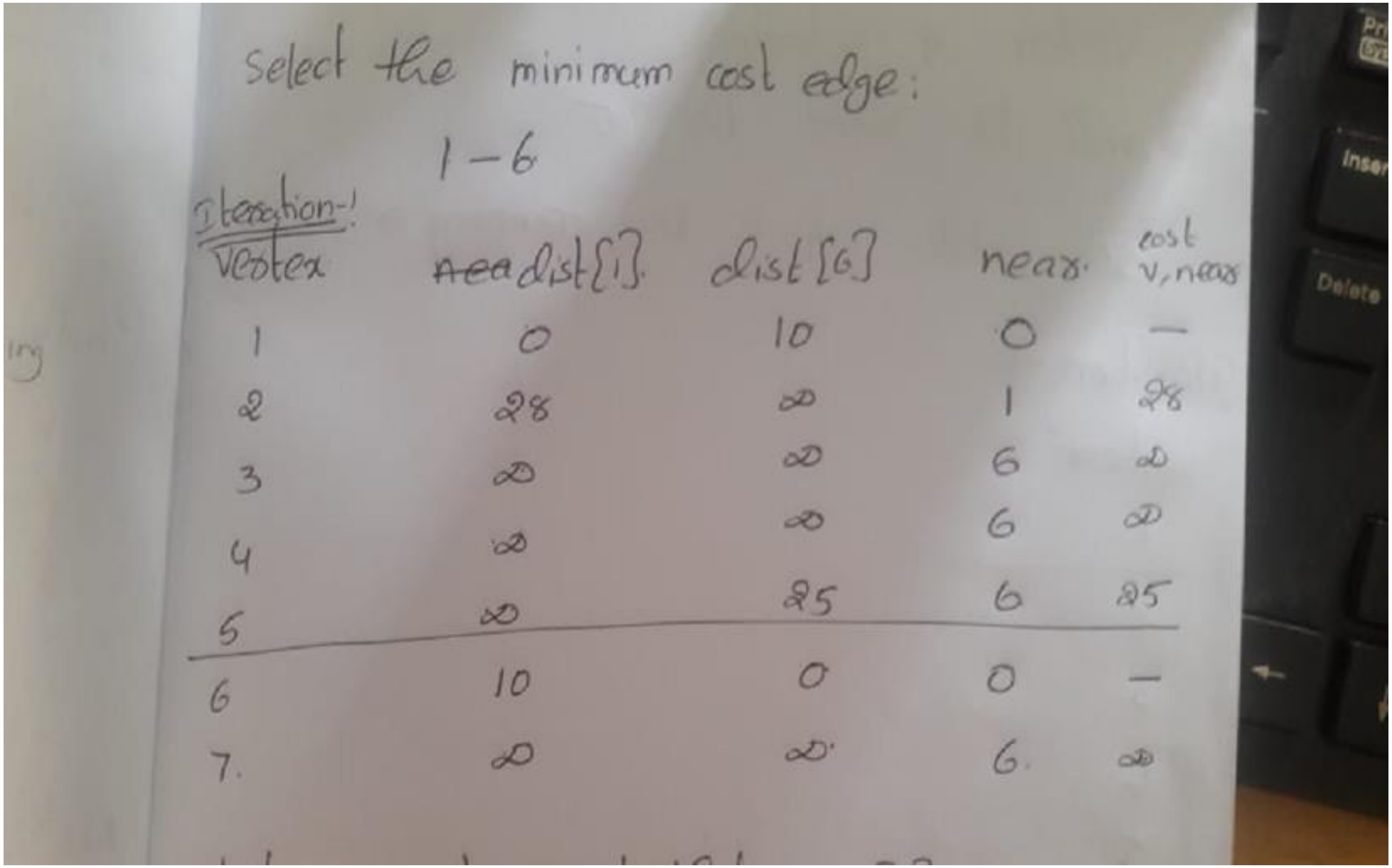

(c)


(d)


(e)


(f)

An Example for Prims Algorithm



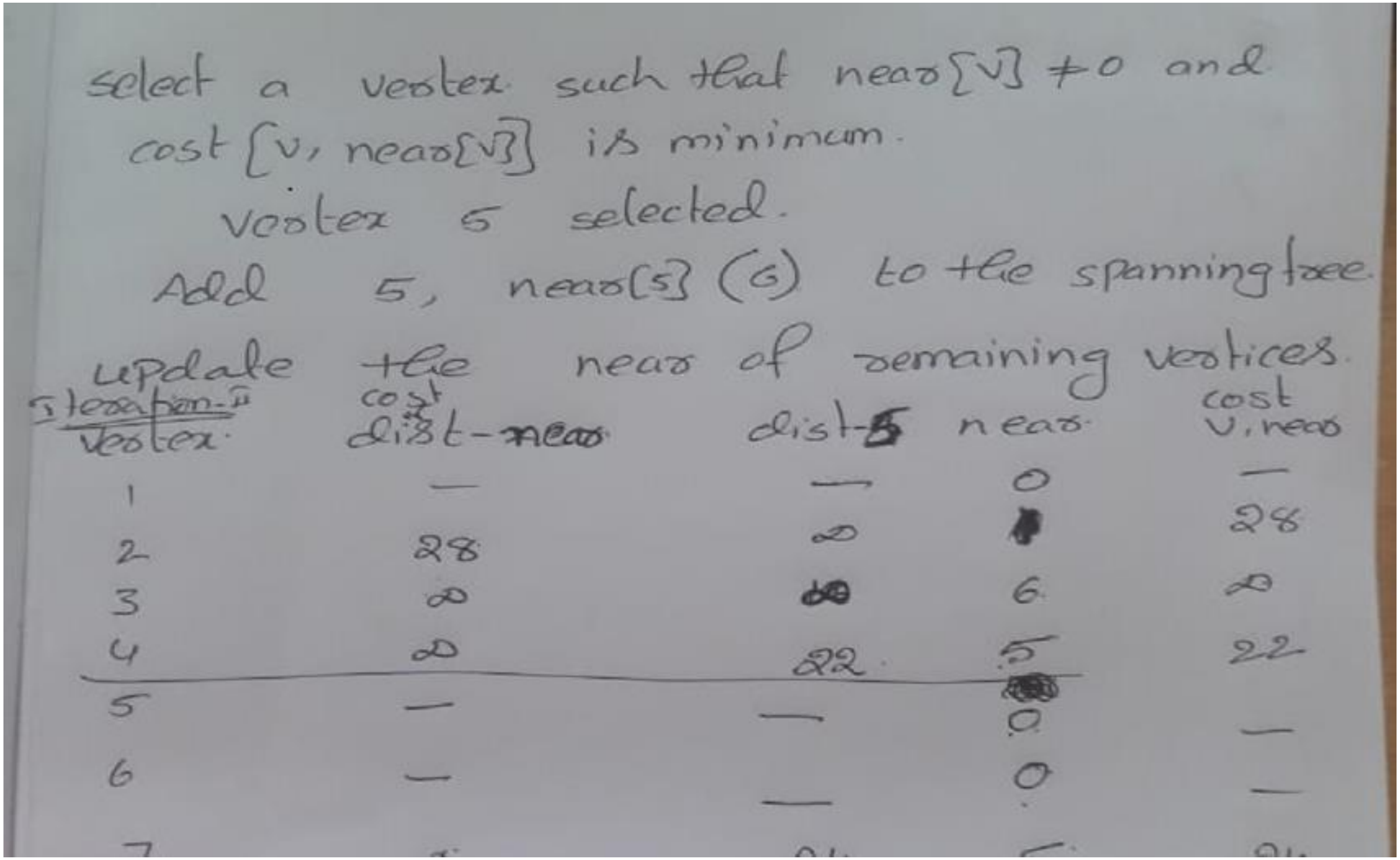
Select the minimum cost edge:

1 – 6

Iteration-1

Vertex	~~near~~ dist[i]	dist[6]	near	cost v, near
1	0	10	0	—
2	28	∞	1	28
3	∞	∞	6	∞
4	∞	∞	6	∞
5	∞	25	6	25
6	10	0	0	—
7.	∞	∞	6.	∞

An Example for Prims Algorithm



select a vertex such that near[v] ≠ 0 and cost [v, near[v]] is minimum.

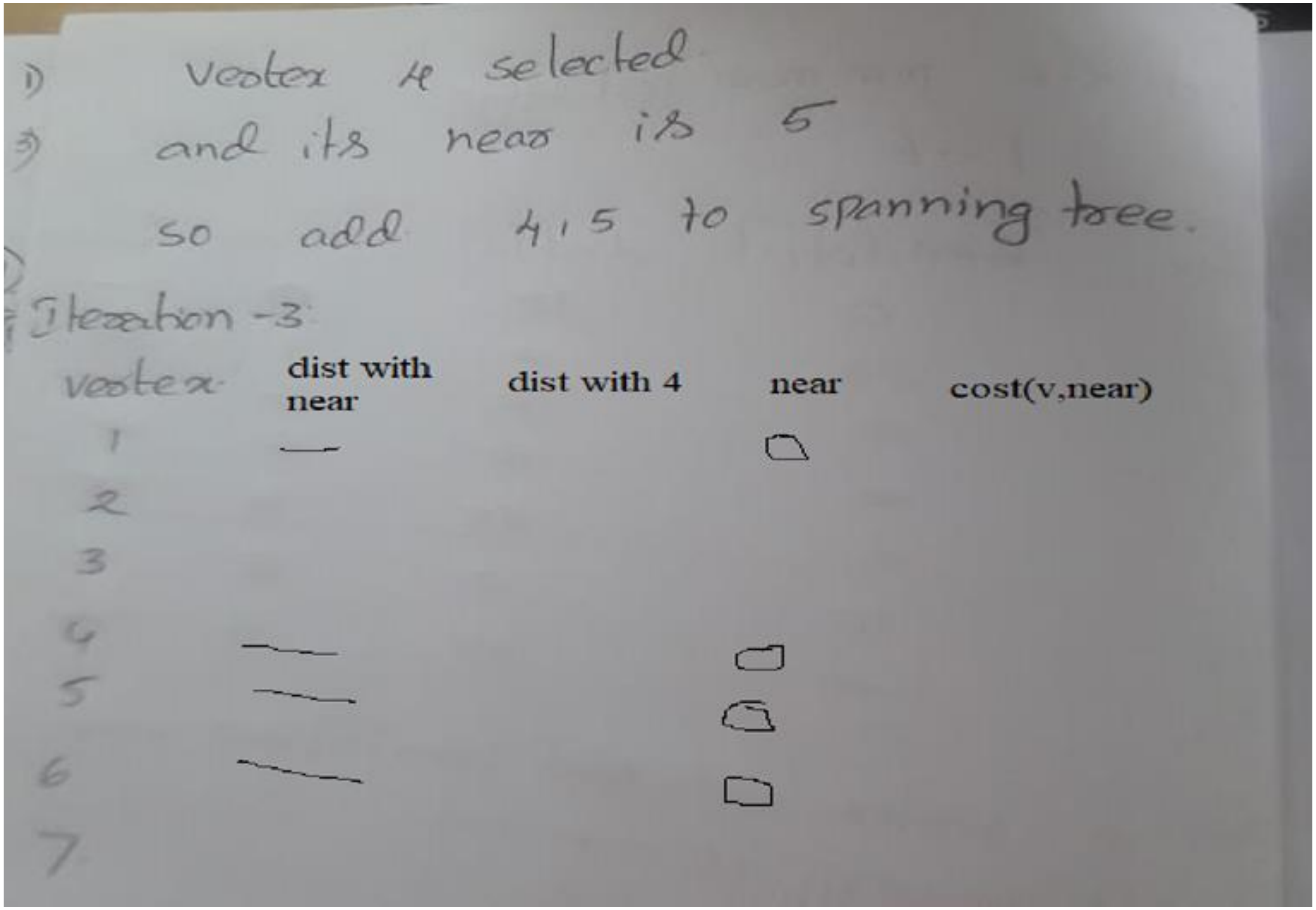
vertex 5 selected.

Add 5, near[5] (6) to the spanning tree.

update the near of remaining vertices.

Iteration-II Vertex	cost dist-~~near~~	dist-5	near	cost v, near
1	—	—	0	—
2	28	∞	[illegible]	28
3	∞	∞	6	∞
4	∞	22	5	22
5	—	—	0	—
6	—	—	0	—
7	[illegible]	[illegible]	[illegible]	[illegible]

An Example for Prims Algorithm

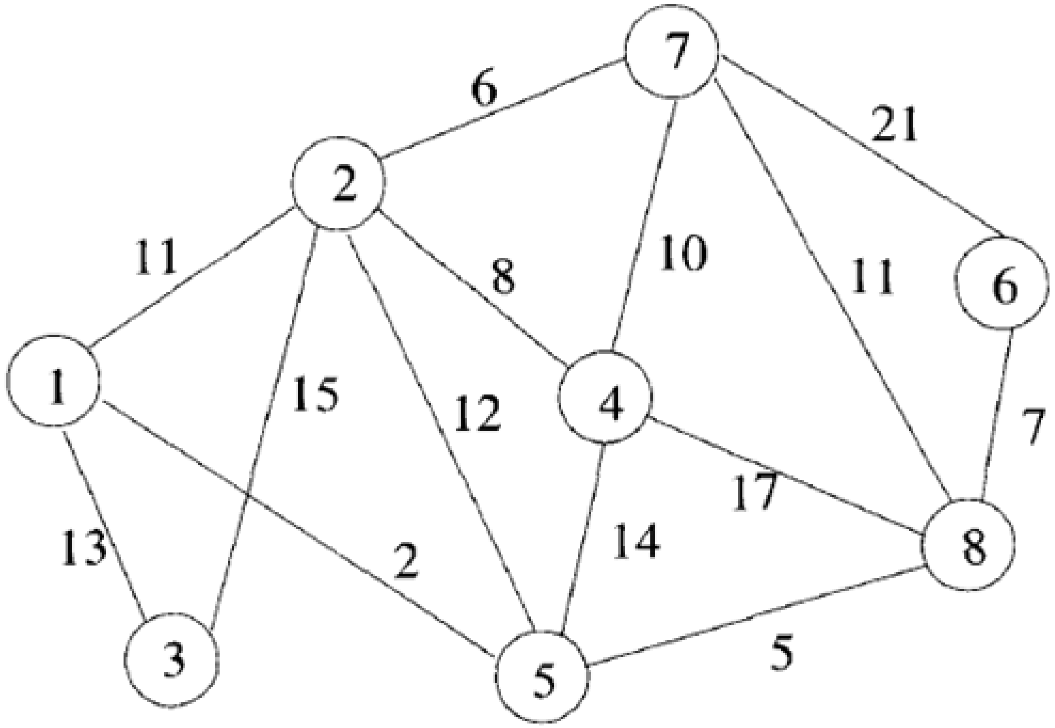


1) vertex 4 selected
2) and its near is 5
so add 4,5 to spanning tree.

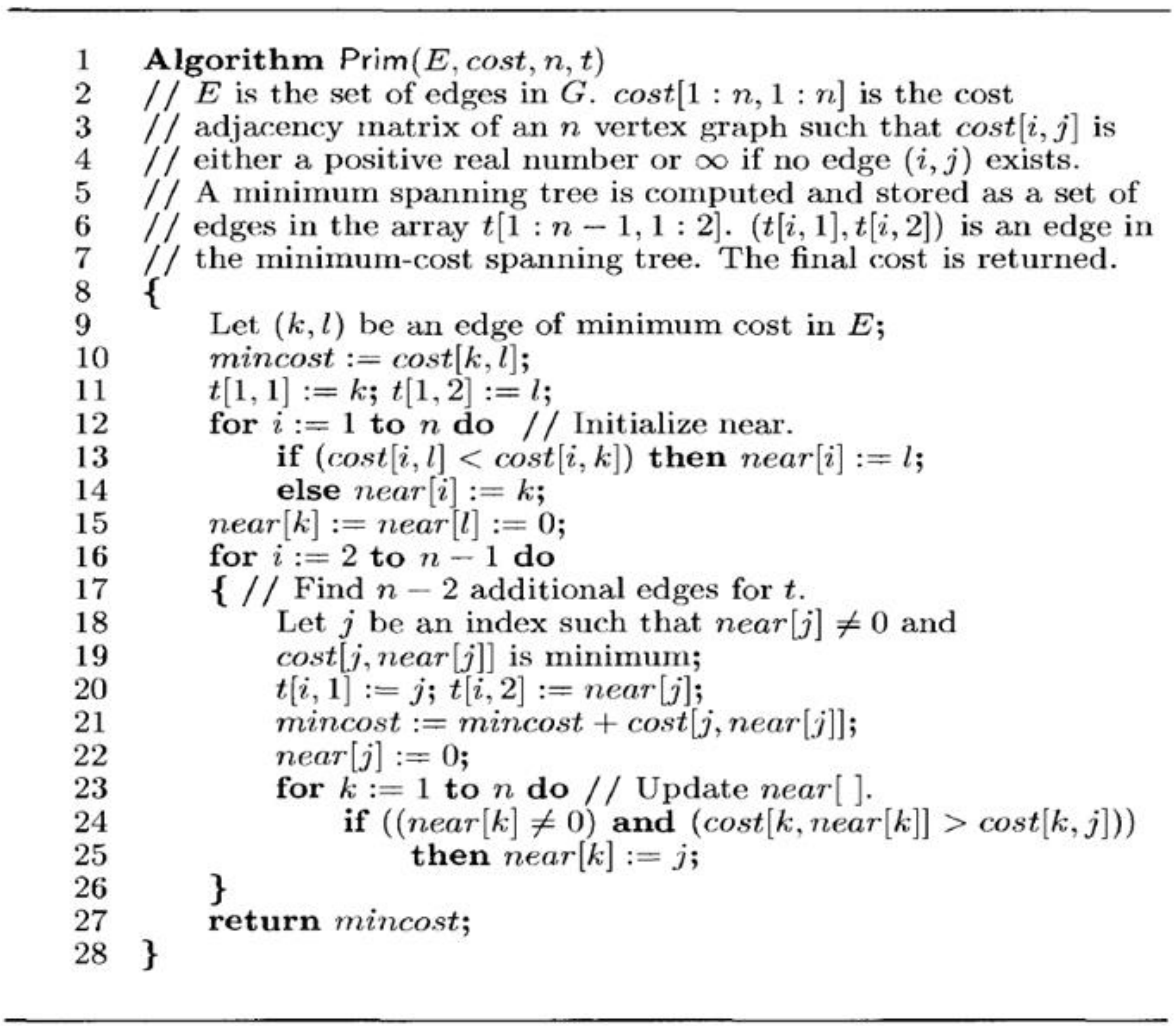
Iteration -3

vertex	dist with near	dist with 4	near	cost(v,near)
1	—		□	
2				
3				
4	—		□	
5	—		□	
6	—		□	
7				

Example 2



Pseudo Code for 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 $\infty$ 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 $near[j] \neq 0$ 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 $((near[k] \neq 0)$ and $(cost[k, near[k]] > cost[k,j]))$
                then $near[k] := j$;
    }
    return $mincost$;
}
```

Time Complexity of Prim's Algorithm

- The time required by algorithm Prim is O(n^2), where n is the number of vertices in the graph G.
- To see this, note that
  - Line 9 takes O(|E|) time and
  - Line10 takes O(1)time.
  - The for loop of line12 takes O(n)time.
  - Lines18 and 19 and the for loop of line require O(n)time.
  - each iteration of the for loop of line 16 takes O(n) time.
  - The total time for the for loop of line16is therefore O(n^2).
  - Hence, Prims algorithm runs in O(n^2)time.

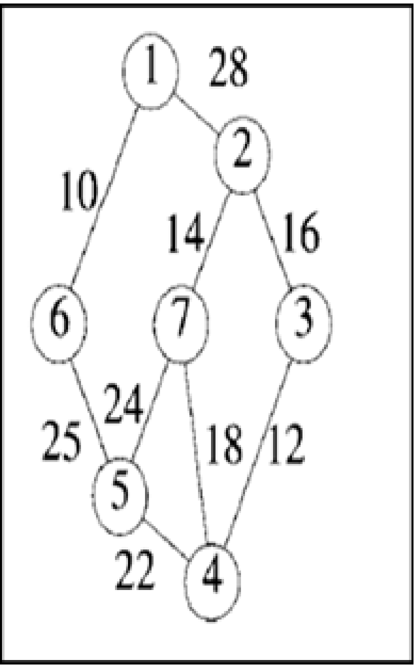
Time Complexity of Prim’s Algorithm

- If we store the nodes not yet included in the tree as a red-black tree lines18 and 19 take O(logn) time.
- Note that a red-black tree supports the following operations in O(logn) time: insert, delete (an arbitrary element),find-min, and search(for an arbitrary element).
- The for loop of line 23 has to examine only the nodes adjacent to j.
- Thus its overall frequency is 0(|E|). Updating in lines 24 and 25 also takes O(logn) time(since an update can be done using a delete and an insertion into the red-black tree).
- Thus the overall run time is 0((n+ |E|) logn).

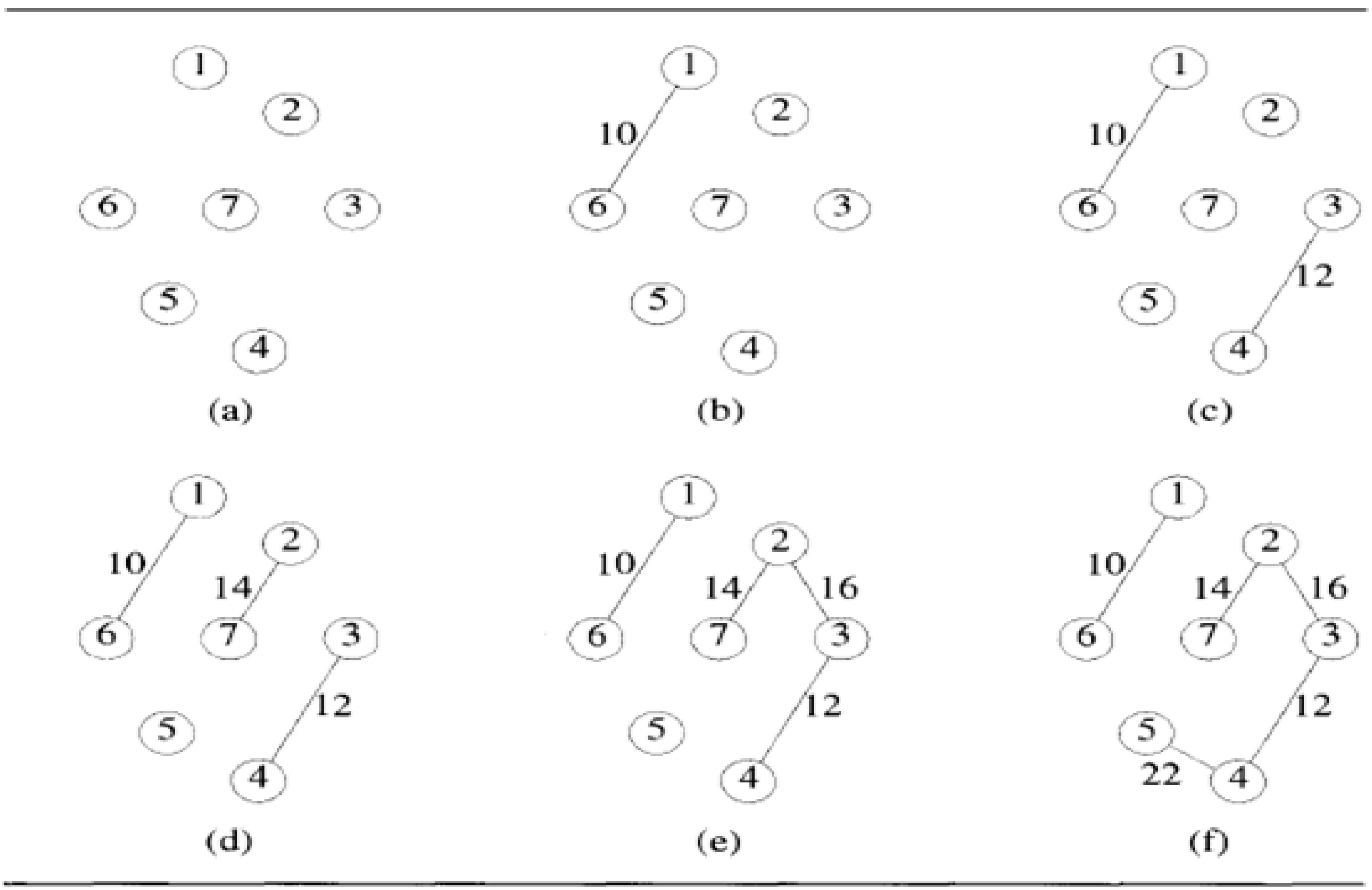
Kruskal’s Algorithm

- There is a second possible Interpretation of the optimization criteria mentioned earlier in which the edges of the graph are considered in increasing order of cost.
- This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to complete t into a tree.
- Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t.

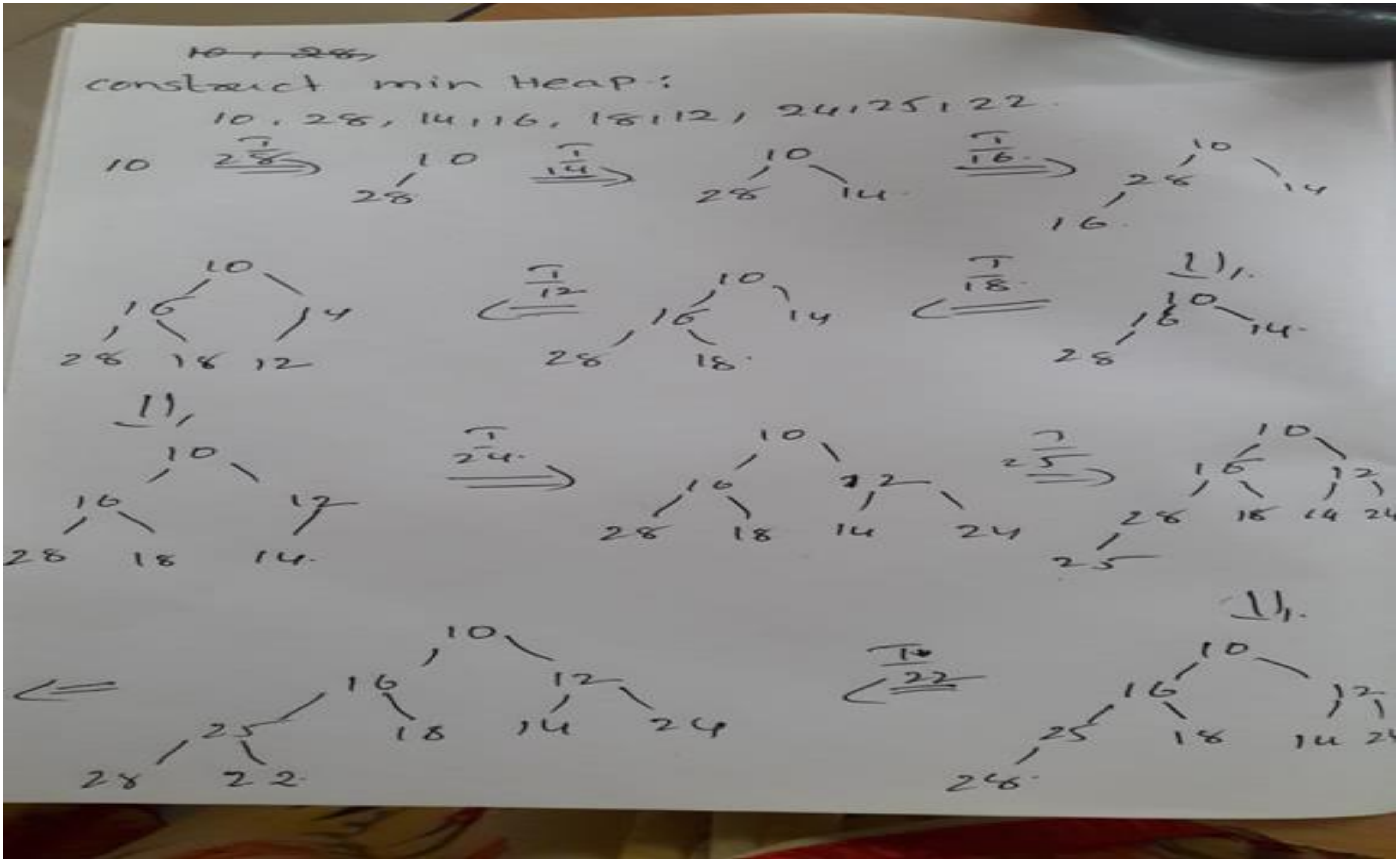
An Example for Kruskal's Algorithm



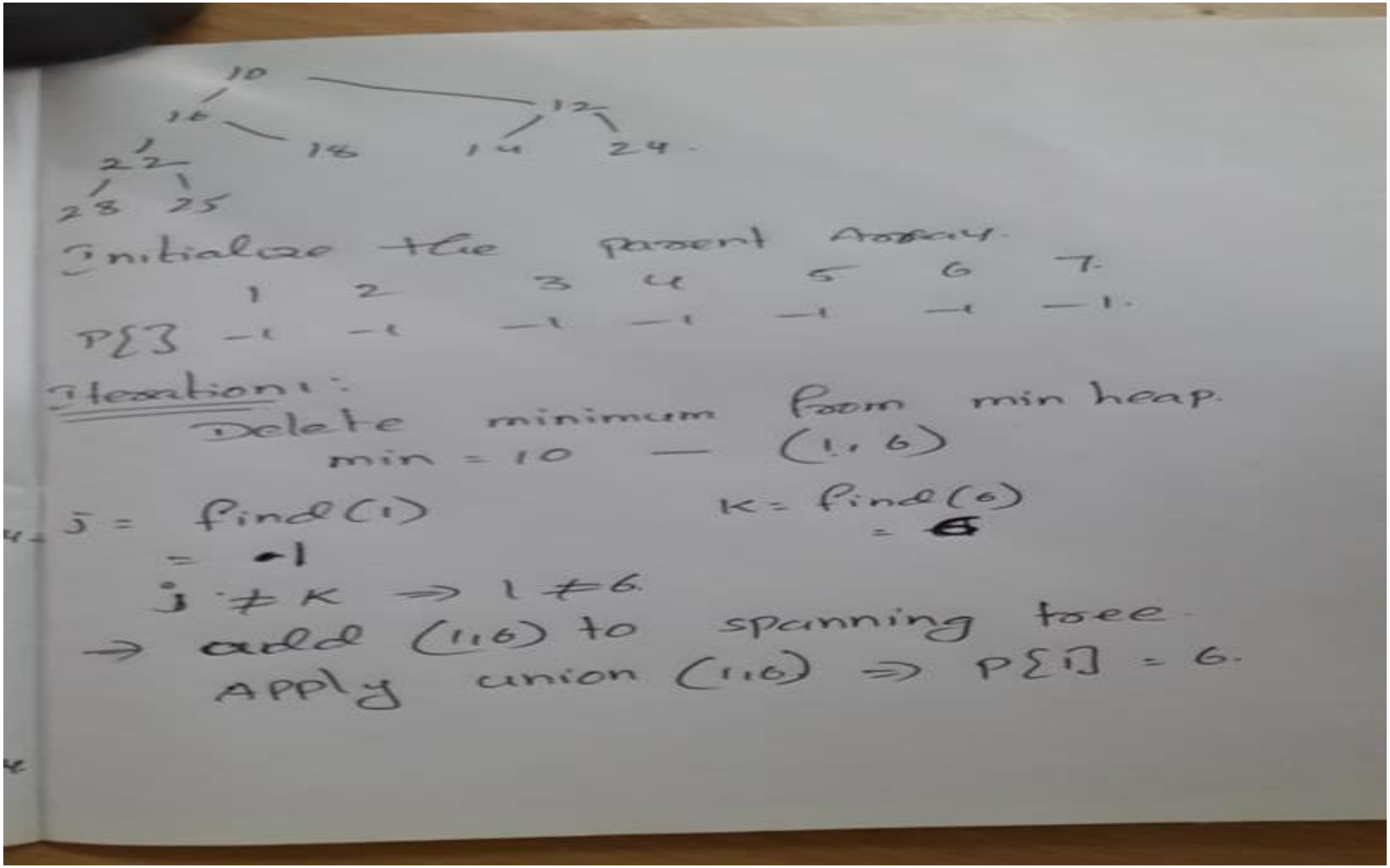
Given Graph



An Example for Kruskal's Algorithm



An Example for Kruskal’s Algorithm



$$\begin{tikzcd}
 & & 10 \arrow[dll, dash] \arrow[drr, dash] & & \\
16 \arrow[d, dash] \arrow[dr, dash] & & & & 12 \arrow[dl, dash] \arrow[d, dash] \\
22 \arrow[d, dash] \arrow[dr, dash] & 18 & & 14 & 24. \\
28 & 25 & & &
\end{tikzcd}$$

Initialize the parent Array.

	1	2	3	4	5	6	7.
P[]	−1	−1	−1	−1	−1	−1	−1.

Iteration 1:

Delete minimum from min heap.

min = 10 — (1, 6)

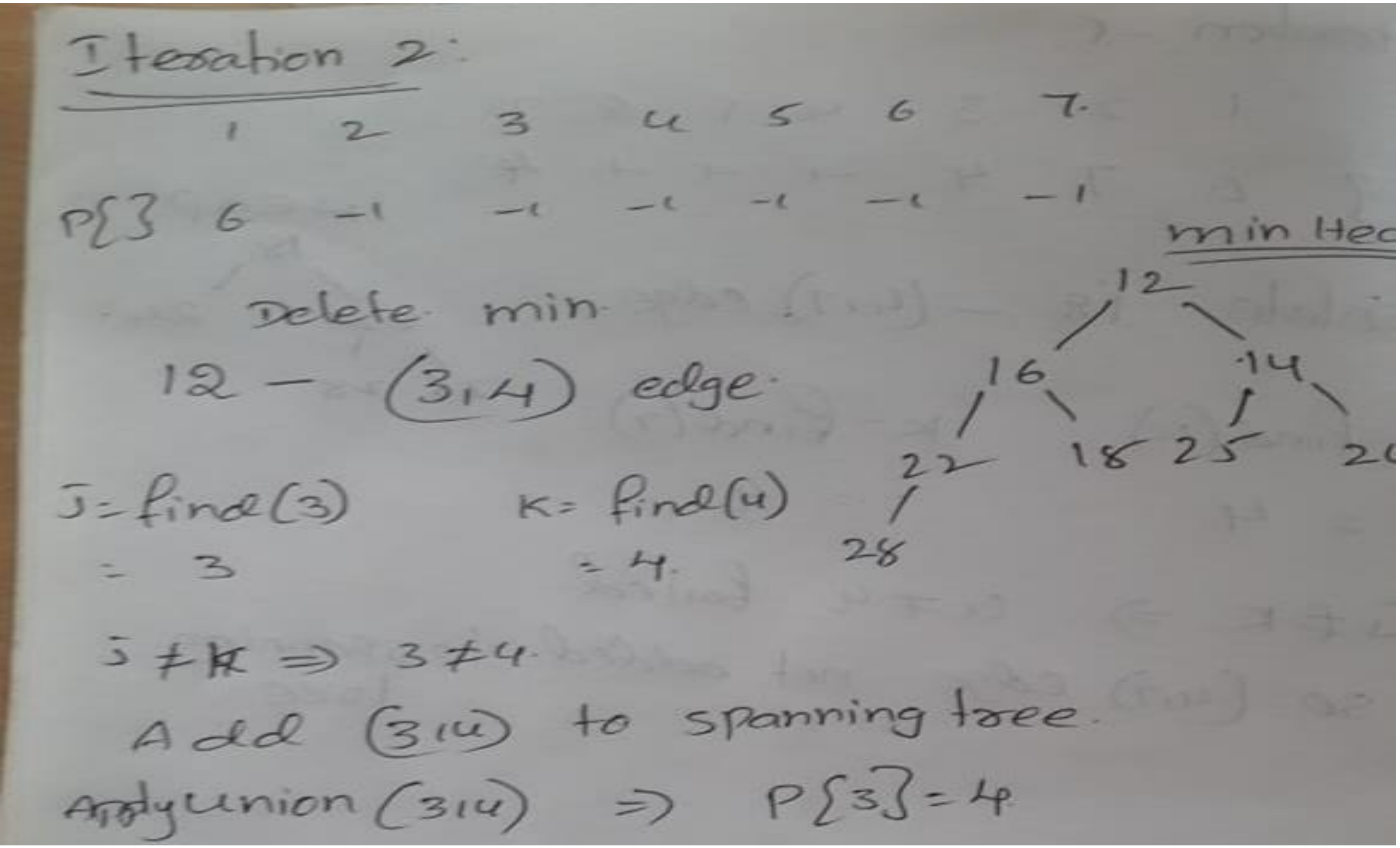
j = find(1) = −1   k = find(6) = 6

$j \neq k \Rightarrow 1 \neq 6.$

→ add (1,6) to spanning tree

Apply union (1,6) ⇒ P[1] = 6.

An Example for Kruskal's Algorithm



Iteration 2:

	1	2	3	4	5	6	7.
P{}	6	−1	−1	−1	−1	−1	−1

Delete min

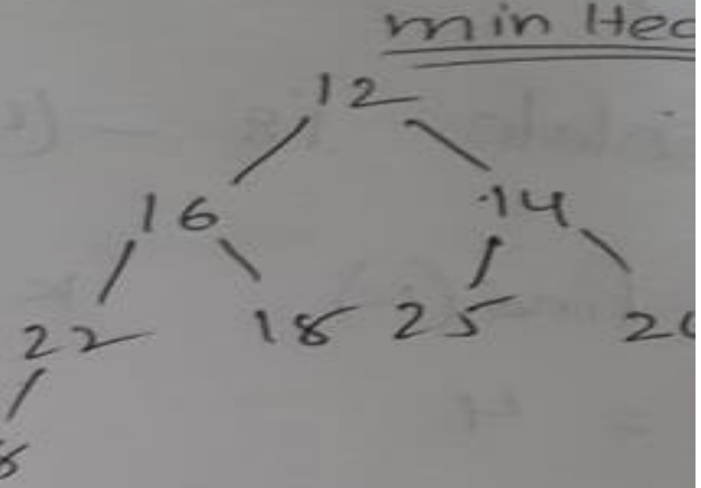
12 — (3,4) edge.

J = find(3)   K = find(4)
= 3   = 4.

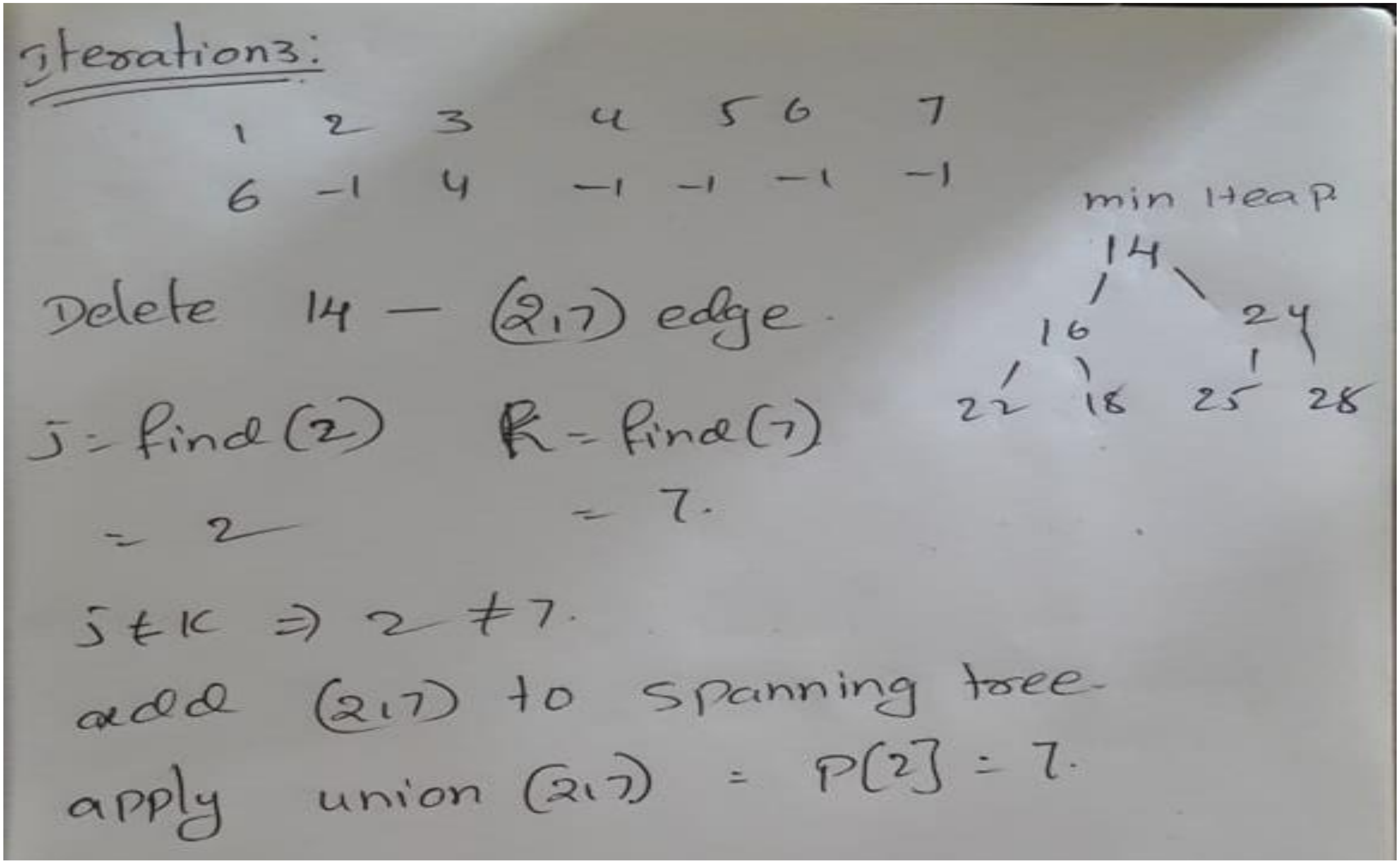
$j \neq k \Rightarrow 3 \neq 4$.

Add (3,4) to spanning tree.

And union (3,4) $\Rightarrow$ P{3} = 4



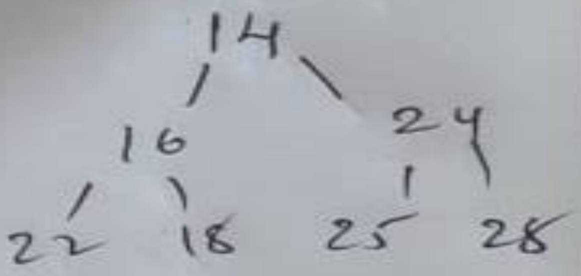
An Example for Kruskal's Algorithm



Iteration3:

1	2	3	4	5	6	7
6	-1	4	-1	-1	-1	-1

min Heap



Delete 14 — (2,7) edge.

j = find(2) k = find(7)

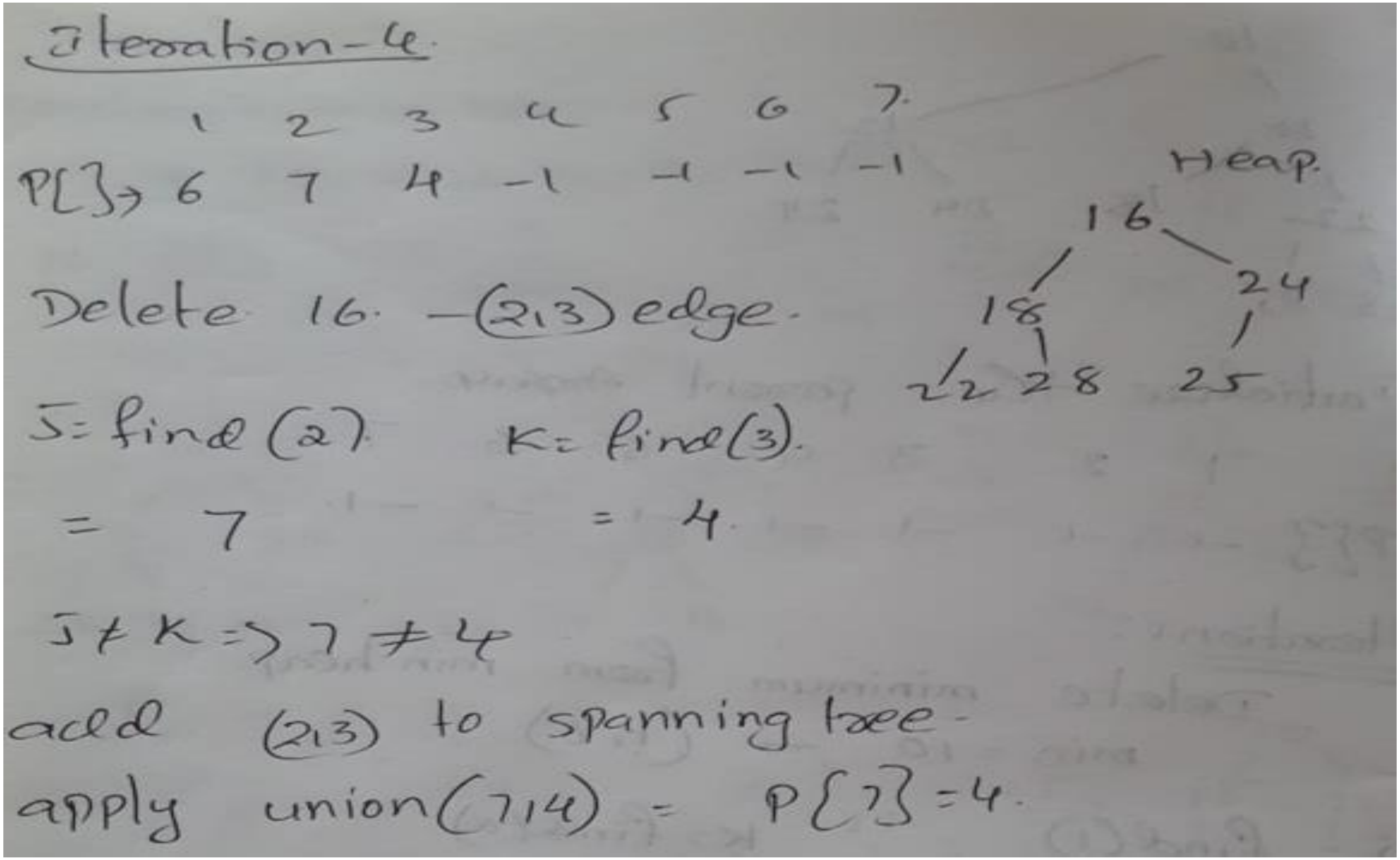
= 2 = 7.

$j \neq k \Rightarrow 2 \neq 7$.

add (2,7) to spanning tree.

apply union (2,7) = P[2] = 7.

An Example for Kruskal's Algorithm



Iteration-4.

	1	2	3	4	5	6	7.
P[] →	6	7	4	-1	-1	-1	-1

Heap.

$$\begin{tikzcd}
 & & 16 \arrow[dl, dash] \arrow[dr, dash] & \\
 & 18 \arrow[dl, dash] \arrow[d, dash] & & 24 \arrow[d, dash] \\
22 & 28 & & 25
\end{tikzcd}$$

Delete 16. —(2,3) edge.

J= find (2). K= find(3).

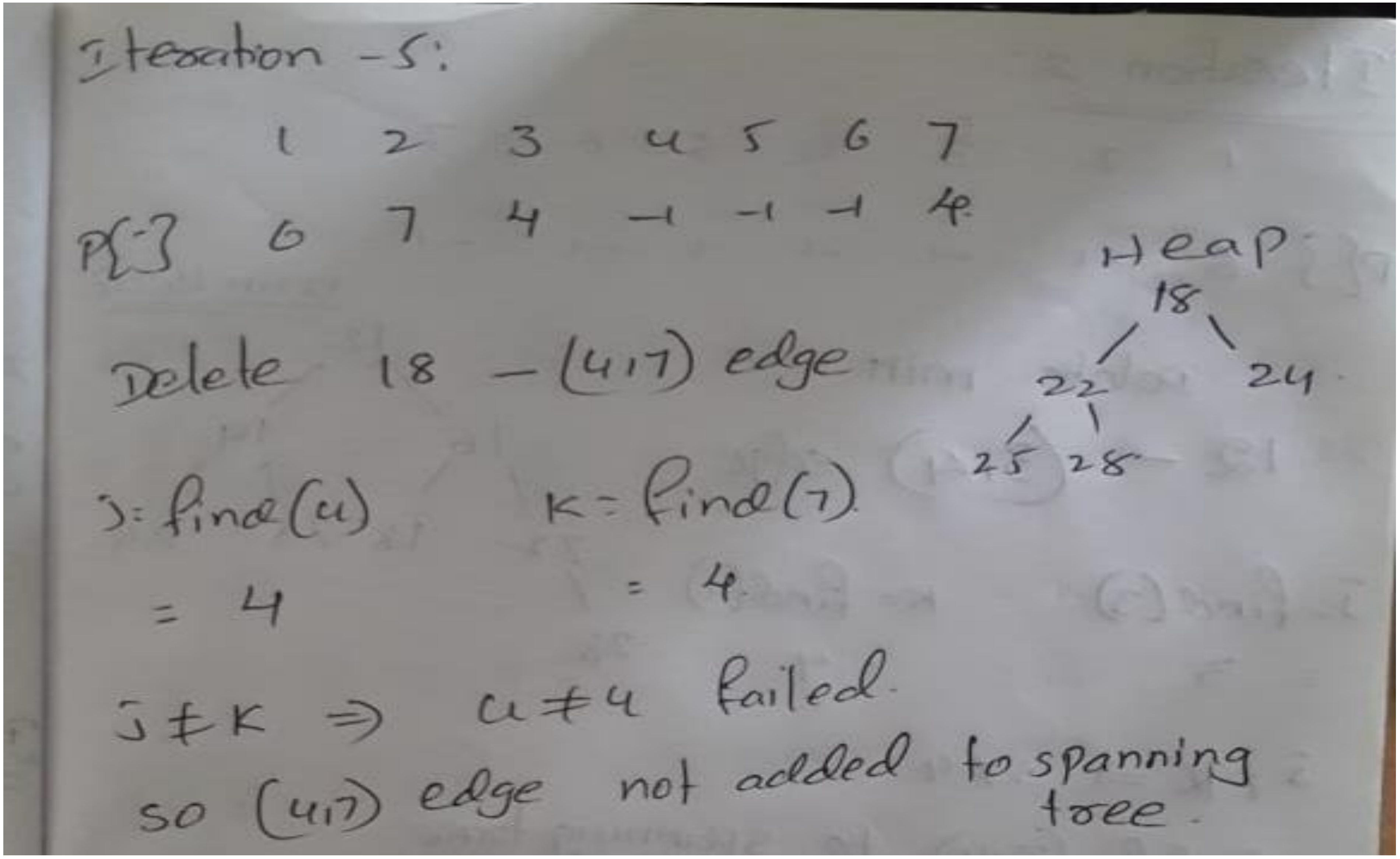
= 7 = 4.

J≠K => 7 ≠ 4

add (2,3) to spanning tree.

apply union(7,4) = P{7}=4.

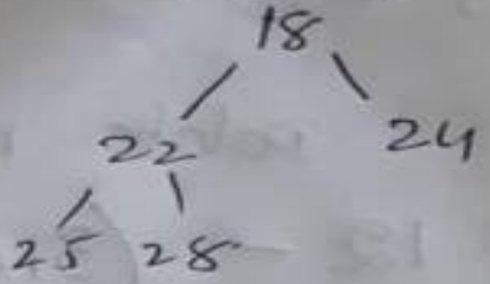
An Example for Kruskal's Algorithm



Iteration -5:

	1	2	3	4	5	6	7
P[·]	6	7	4	-1	-1	-1	4

Heap:



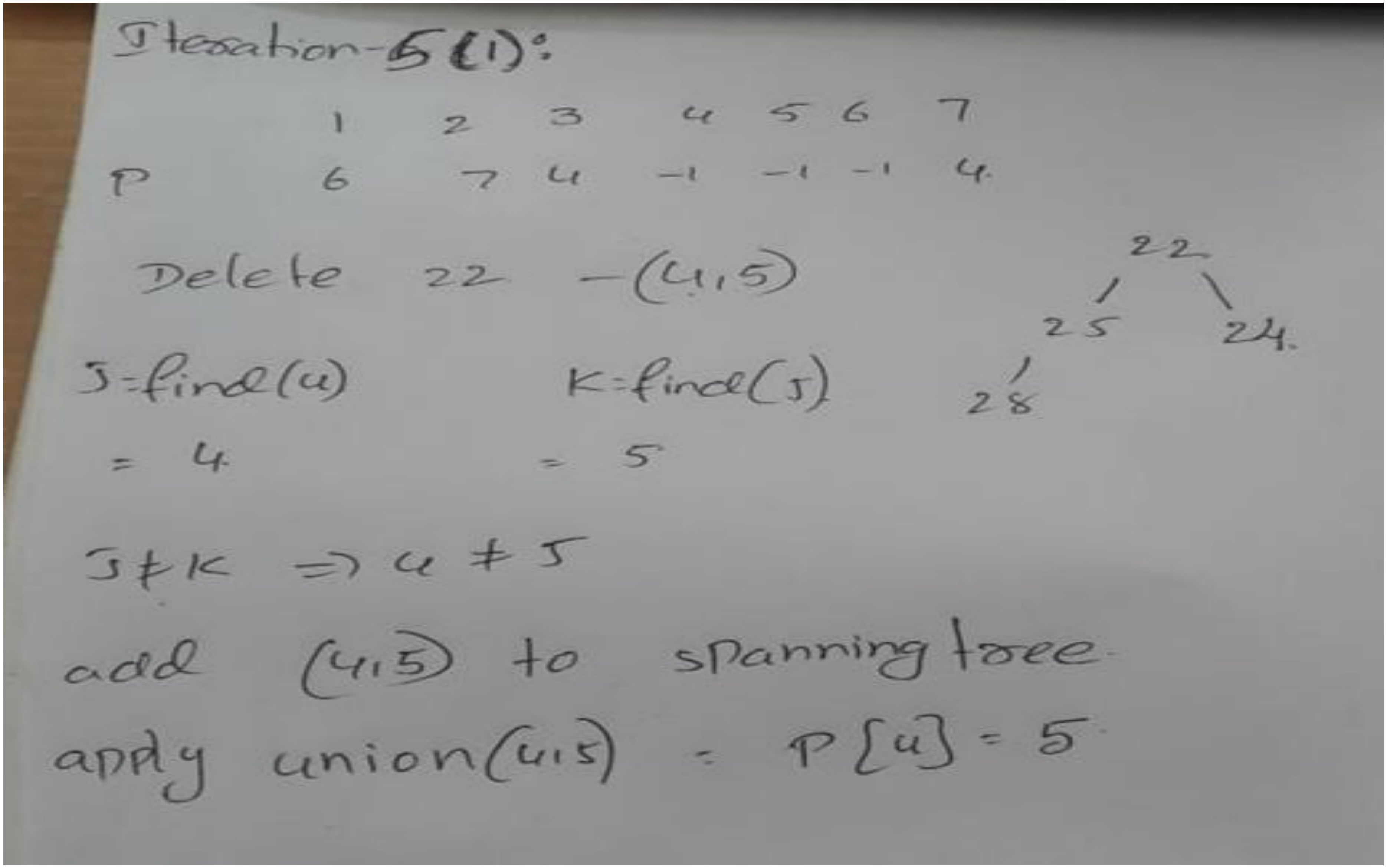
Delete 18 - (4,7) edge

j = find(4) = 4   k = find(7) = 4

$j \neq k \Rightarrow 4 \neq 4$ failed.

so (4,7) edge not added to spanning tree.

An Example for Kruskal's Algorithm



Iteration-5(1):

	1	2	3	4	5	6	7
P	6	7	4	-1	-1	-1	4.

Delete 22 — (4,5)

$$\begin{tikzcd}
 & & 22 \arrow[dl, dash] \arrow[dr, dash] & \\
 & 25 \arrow[dl, dash] & & 24. \\
28 & & &
\end{tikzcd}$$

J=find(4) = 4

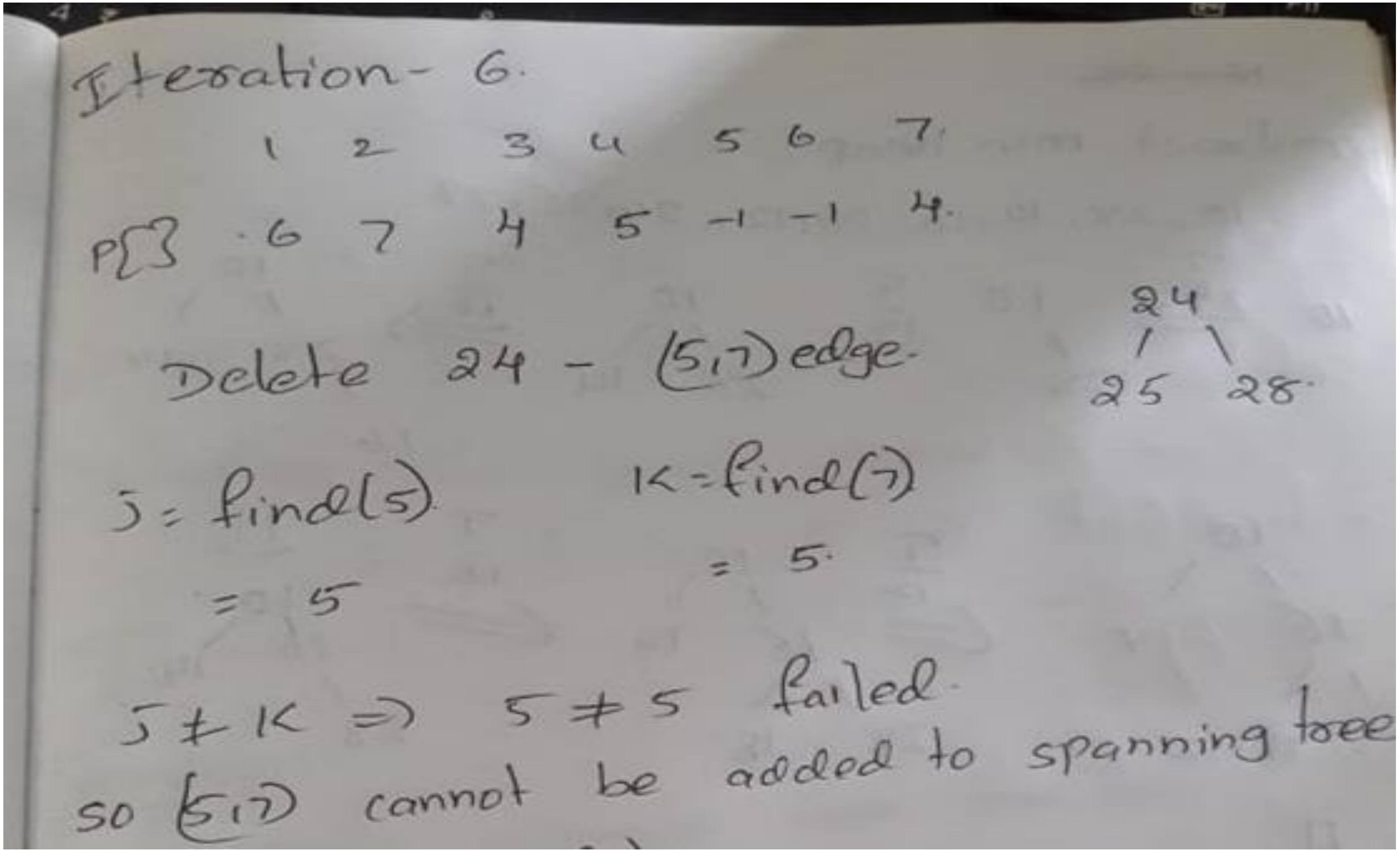
K=find(5) = 5

J≠K ⇒ 4≠5

add (4,5) to spanning tree.

apply union(4,5) : P[4] = 5.

An Example for Kruskal’s Algorithm



Iteration - 6.

	1	2	3	4	5	6	7
P[]	6	7	4	5	-1	-1	4.

Delete 24 - (5,7) edge.

24
/ \
25 28.

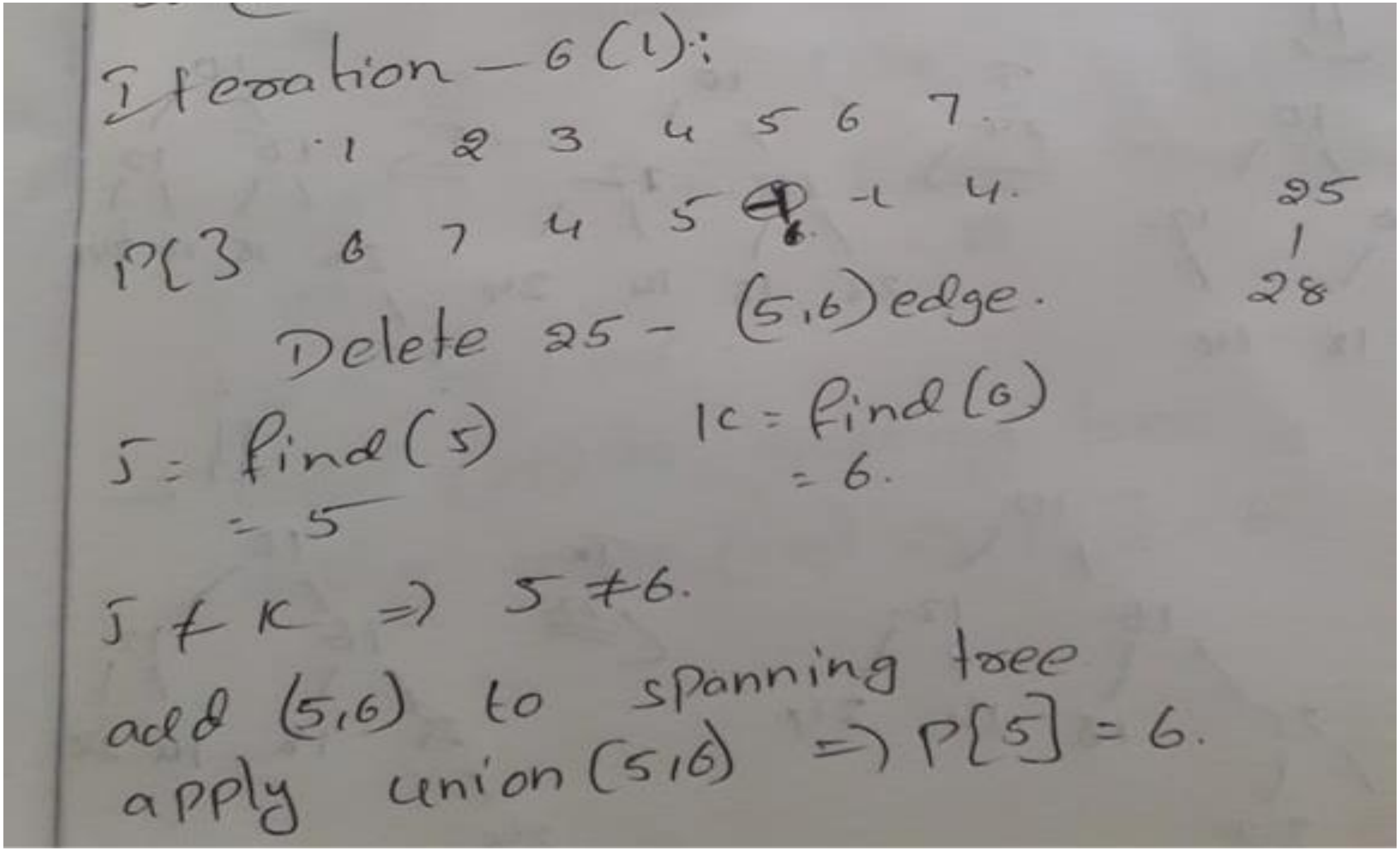
j = find(5)
= 5

K = find(7)
= 5.

j ≠ K ⇒ 5 ≠ 5 failed.

so (5,7) cannot be added to spanning tree

An Example for Kruskal's Algorithm



Iteration – 6(i):

	1	2	3	4	5	6	7
P[]	6	7	4	5	6	-1	4

25
|
28

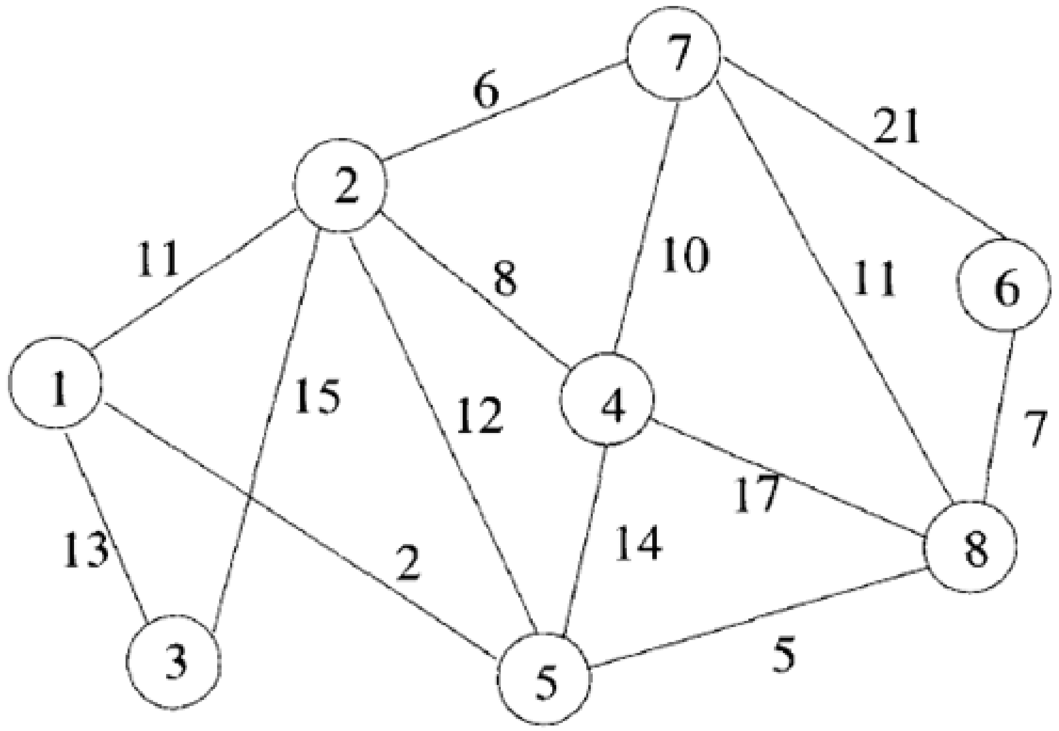
Delete 25 – (5,6) edge.

j = find(5) = 5   k = find(6) = 6.

$j \neq k \Rightarrow 5 \neq 6$.

add (5,6) to spanning tree
apply union(5,6) ⇒ P[5] = 6.

Example 2



Pseudo Code for Kruskal's Algorithm

```
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 ($j \neq k$) then
        {
            $i := i + 1$;
            $t[i, 1] := u$; $t[i, 2] := v$;
            $mincost := mincost + cost[u, v]$;
            Union($j, k$);
        }
    }
    if ($i \neq n - 1$) then write ("No spanning tree");
    else return $mincost$;
}
```

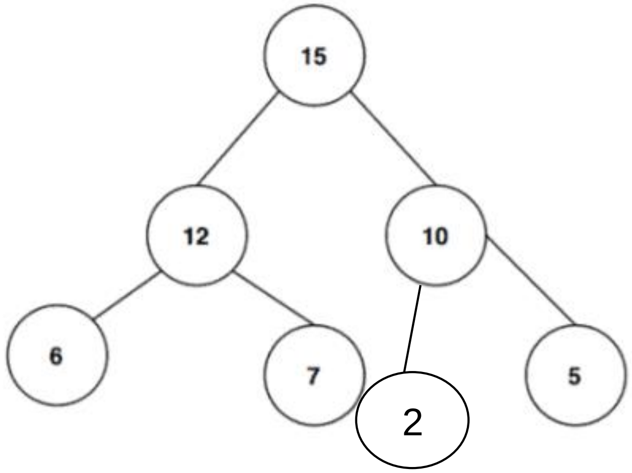
Supporting Functions of Kruskals Algorithm

```
  Algorithm Heapify(E, m)
                  //E is an array of m edge cost values
{
     for i=m/2 to 1 step -1
     {
         adjust(a,i,m);
     }
}
```

Supporting Functions of Kruskals Algorithm

```
Algorithm adjust(E, i, m)
{
    j=2*i ;  // position of the left child
    item=E[i];
    while(j<=m)
    {
        if(E[j] > E[j+1]) then j=j+1;
        if (item <=E[j]) then break;
        E[j/2]=E[j];
        j=2*j;
    }
    E[j/2]=item;
}
```

15	12	10	6	7	2	5



Supporting Functions of Kruskals Algorithm

```
Algorithm Delete(E, m)
{
        if(m==0]) then write “heap empty”;
        else
        {
            x=E[1];    E[1]=E[m];
            adjust(E,1,m-1);
         }
    return x;
}
```

Supporting Functions of Kruskals Algorithm

```
Algorithm Find(k)
{
        while(p[k]>=0)
        {
            k=p[k];
        }
    return k;
}
```

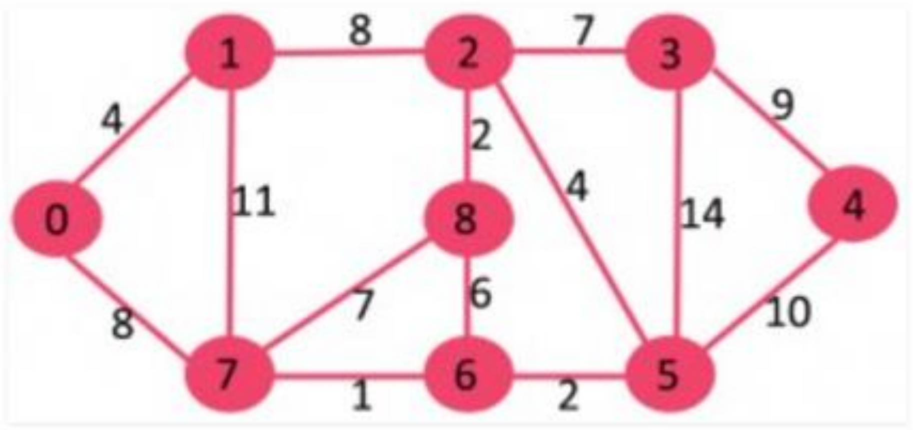
```
Algorithm union(a,b)
{
    p[a]=b;
}
```

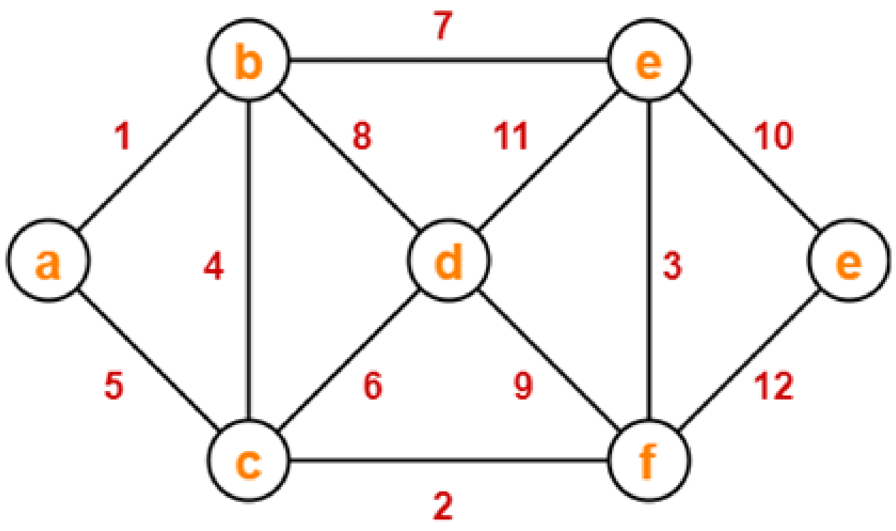
Time Complexity of Kruskals Algorithm

- **The time required by algorithm kruskal's is 0(|E|*log|E|), where E is the number of edges in the graph G.**
- To see this, note that
  - Line 6 takes 0(|E|* log|E|) time
  - Line7 takes 0(n)time.
  - The while loop of line10 takes 0(|E|)time.
  - Lines 12 and 13 require 0(log|E|)time.
  - Line 14 Find() takes 0(log n) time.
  - Hence, Krushkal's algorithm runs in 0(|E|* log|E|) or 0(|E|* logV) time.

Problem for practice

Find the Minimum cost spanning tree using prim's and kruskal's Algorithms





Topics

- General method
- Knapsack problem
- Minimum cost spanning trees:
  - Prim‘s algorithm
  - Kruskal‘s algorithm
- **Single source shortest path problem**
- Job Sequencing with deadlines

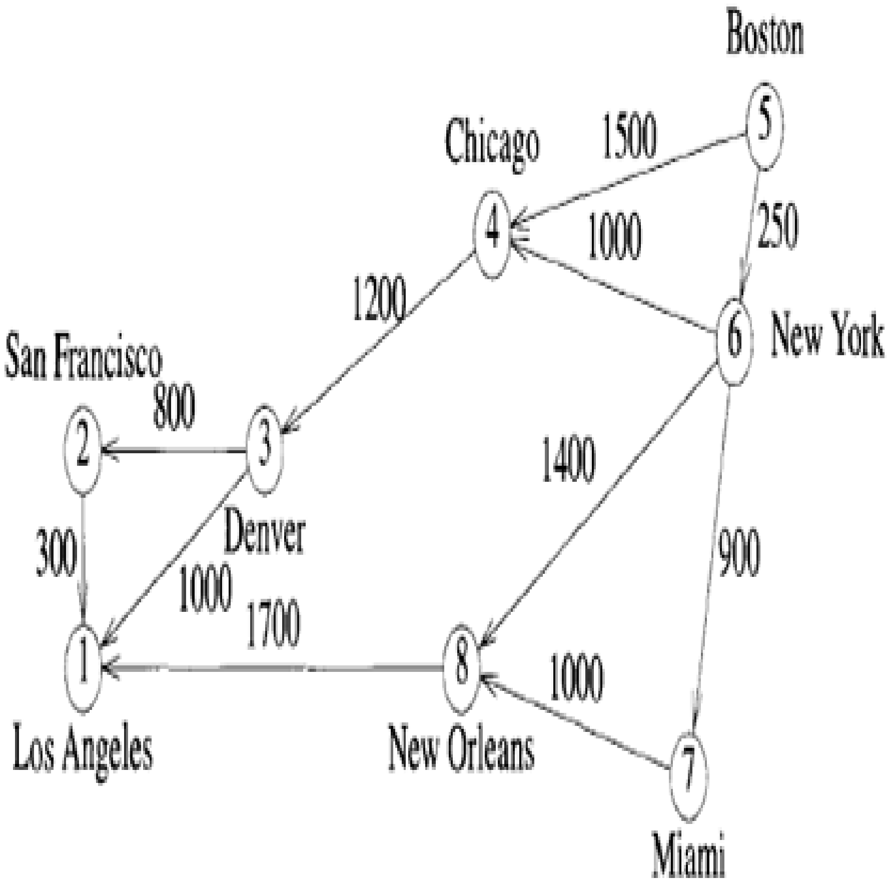
Single source shortest path problem

- Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the time to drive along that section of highway.
- A motorist wishing to drive from city A to B would be interested in answers to the following questions:
  - Is there a path from A to B?
  - If there is more than one path from A to B, which is the shortest path?

Single source shortest path problem

- The length of a path is now defined to be the sum of the weights of the edges on that path.
- The starting vertex of the path is referred to as the source, and the last vertex the destination.
- The graphs are digraphs to allow for one-way streets.
- The problem statement is: we are given a weighted directed graph G = (V,E) and a source vertex S,
  - Determine the shortest paths from S to all the remaining vertices of G.
- It is assumed that all the weights are positive.
- The shortest path between Vo and some other node V is an ordering among a subset of the edges.

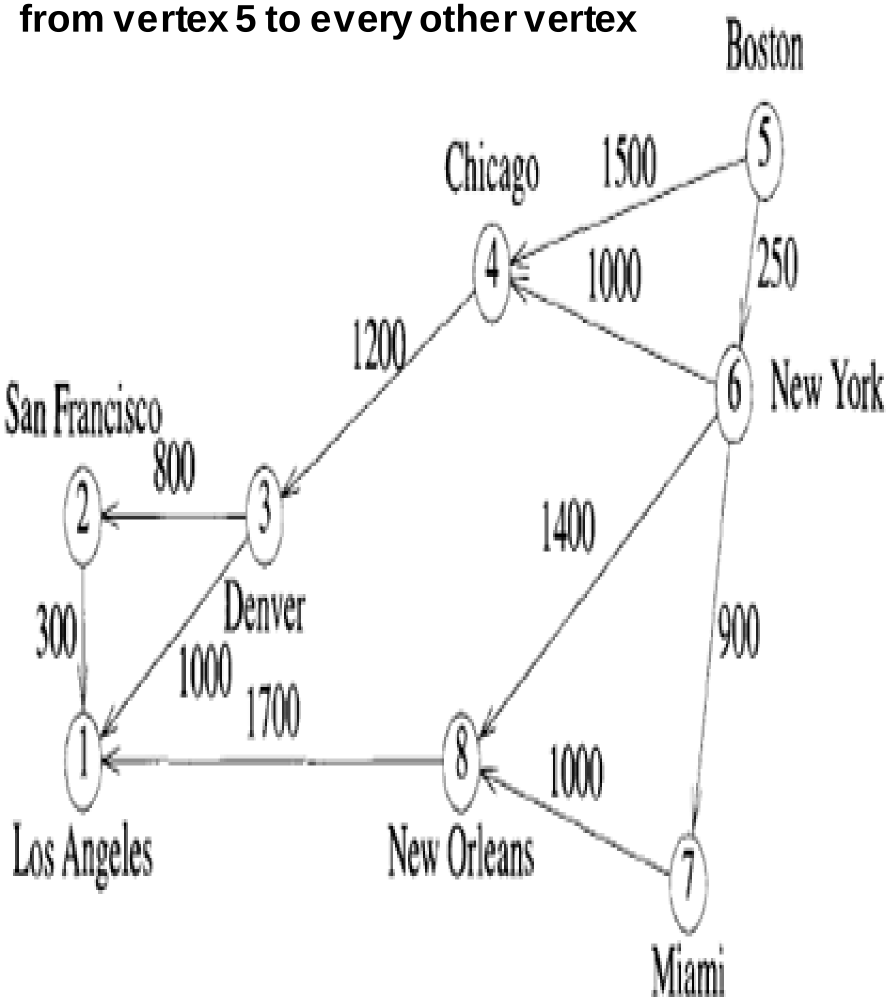
An Example



$$
\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & 0 & & & & & & & \\
2 & 300 & 0 & & & & & & \\
3 & 100 & 800 & 0 & & & & & \\
4 & & & 1200 & 0 & & & & \\
5 & & & & 1500 & 0 & 250 & & \\
6 & & & & 1000 & & 0 & 900 & 1400 \\
7 & & & & & & & 0 & 1000 \\
8 & 1700 & & & & & & & 0
\end{array}
$$

An Example

Find the shortest path distance from vertex 5 to every other vertex



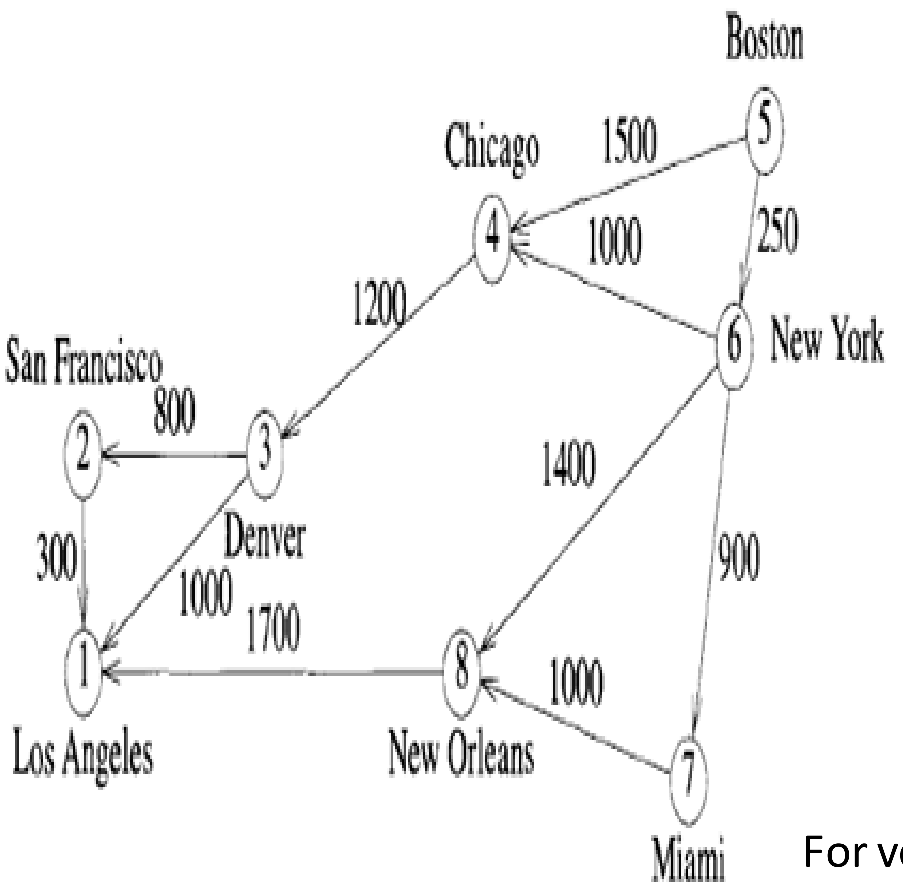
City	Dist
1	
2	
3	
4	
5	0
6	
7	
8	

Update with the distance from 5

Distance from 5

City	Dist
1	∞
2	∞
3	∞
4	1500
5	0
6	250
7	∞
8	∞

An Example



City	Dist
1	∞
2	∞
3	∞
4	1500
5	0
6	250
7	∞
8	∞

Update by considering 6 as intermediate

Distance from 5 by considering **6** as intermediate

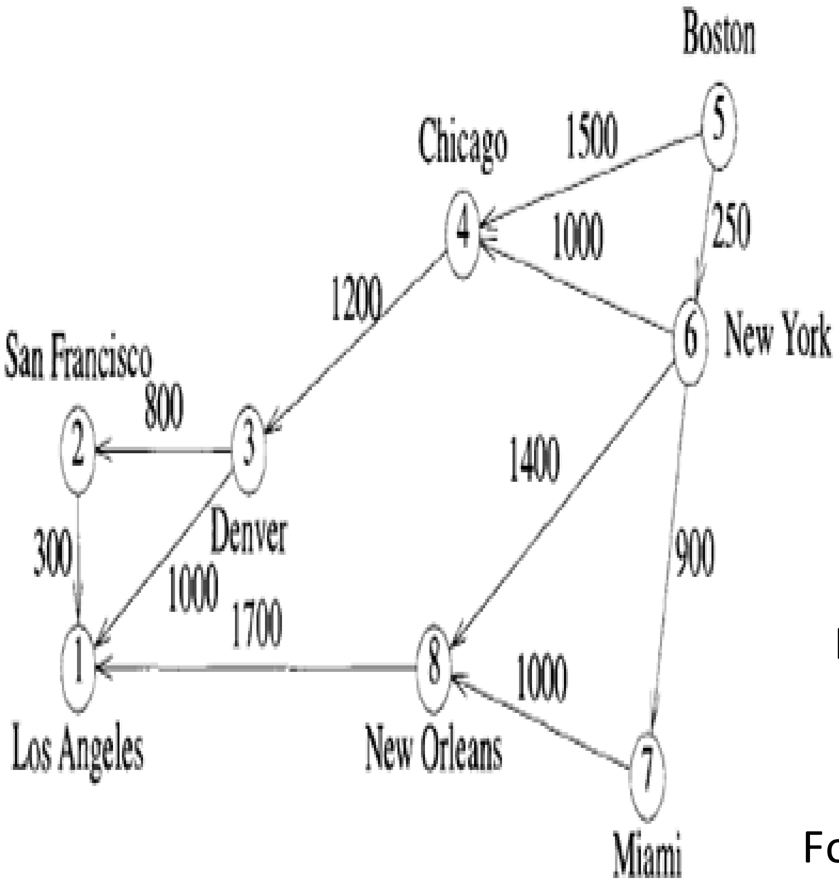
City	Dist
1	∞
2	∞
3	∞
4	1250
5	0
6	250
7	1150
8	1650

For vertex 4: min[dist(4), dist(6)+(6,4)]
min[1500, 250+1000]
min(1500,1250)=1250

Similarly for the other vertices also

An Example

Distance from 5 by considering **7** as intermediate



City	Dist
1	∞
2	∞
3	∞
4	1250
5	0
6	250
7	1150
8	1650

Update by considering 7as intermediate

City	Dist
1	∞
2	∞
3	∞
4	1250
5	0
6	250
7	1150
8	1650

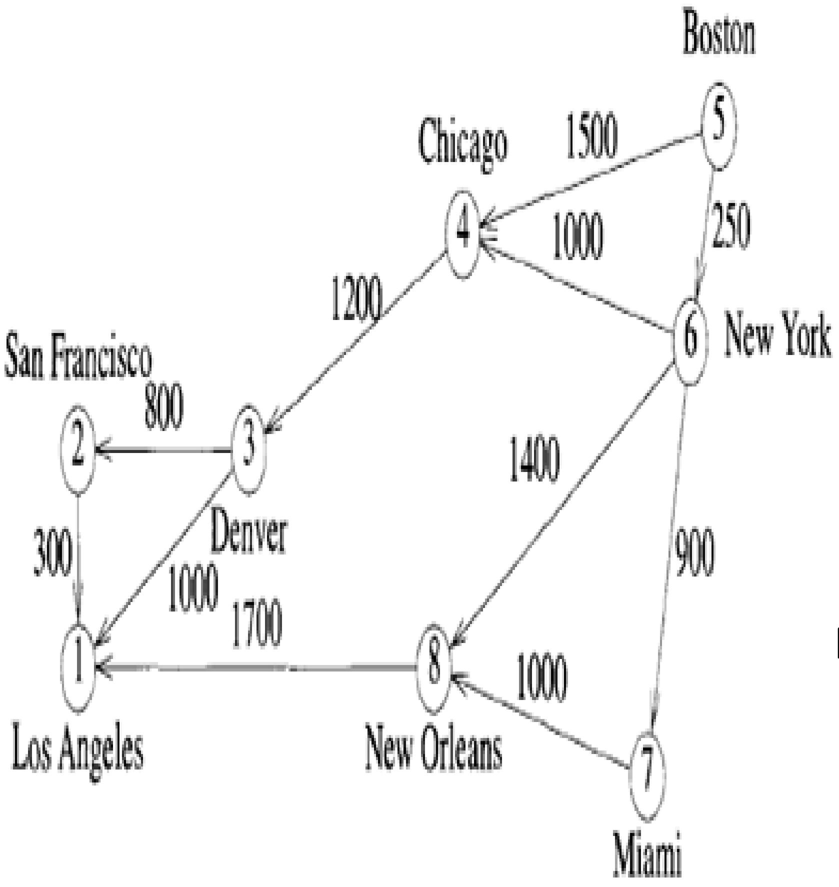
For vertex 4: min[dist(4), dist(7)+(7,4)]
min[1250,1150+∞]
min(1650, ∞)=1650

For vertex 8: min[dist(8), dist(7)+(7,8)]
min[1650,1150+1000]
min(1650,2150)=1650

Similarly for the other vertices also

An Example

Distance from 5 by considering **4** as intermediate



City	Dist
1	∞
2	∞
3	∞
4	1250
5	0
6	250
7	1150
8	1650

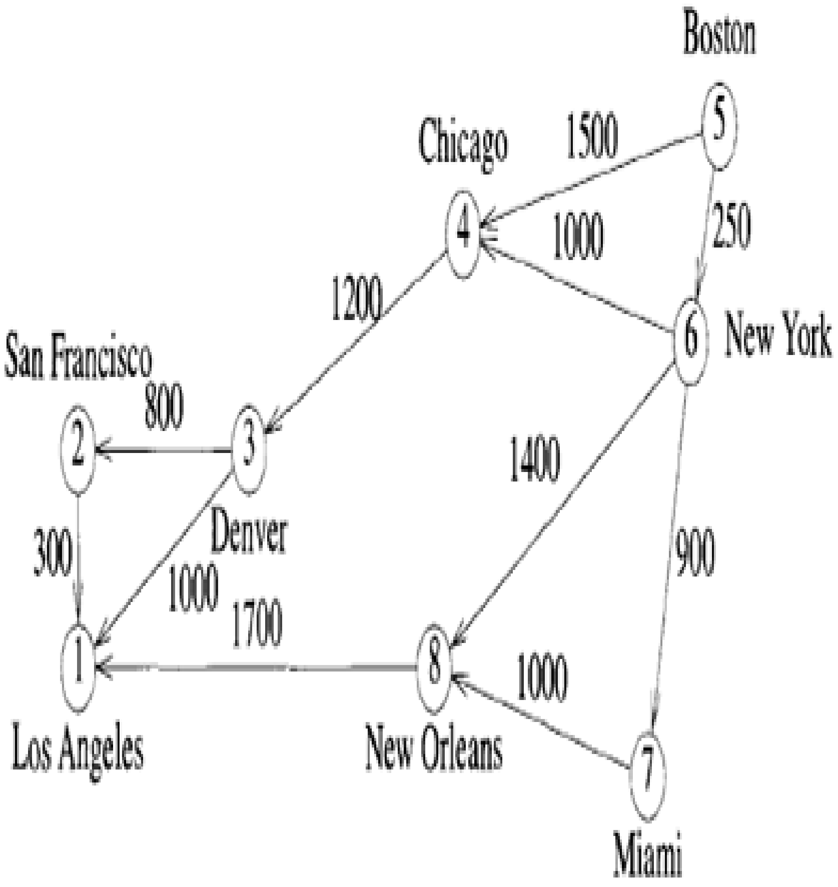
Update by considering 4 as intermediate

City	Dist
1	∞
2	∞
3	2450
4	1250
5	0
6	250
7	1150
8	1650

For vertex 3: min[dist(3), dist(4)+(4,3)]
min[∞,1250+1200]
min(∞, 2450)=2450

Similarly for the other vertices also

An Example



City	Dist
1	∞
2	∞
3	2450
4	1250
5	0
6	250
7	1150
8	1650

Update by considering 8 as intermediate

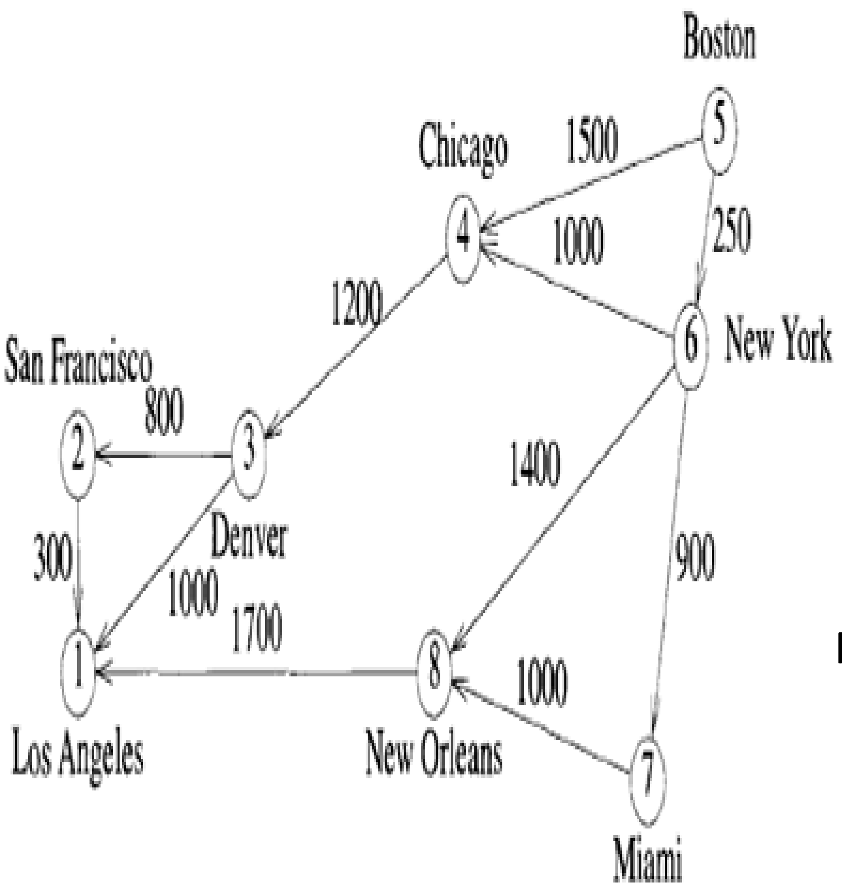
Distance from 5 by considering **8** as intermediate

City	Dist
1	3350
2	∞
3	2450
4	1250
5	0
6	250
7	1150
8	1650

For vertex 1: min[dist(1), dist(8)+(8,1)]
min[∞,1650+1700]
min(∞, 3350)=3350

Similarly for the other vertices also

An Example



City	Dist
1	3350
2	∞
3	2450
4	1250
5	0
6	250
7	1150
8	1650

Update by considering 3 as intermediate

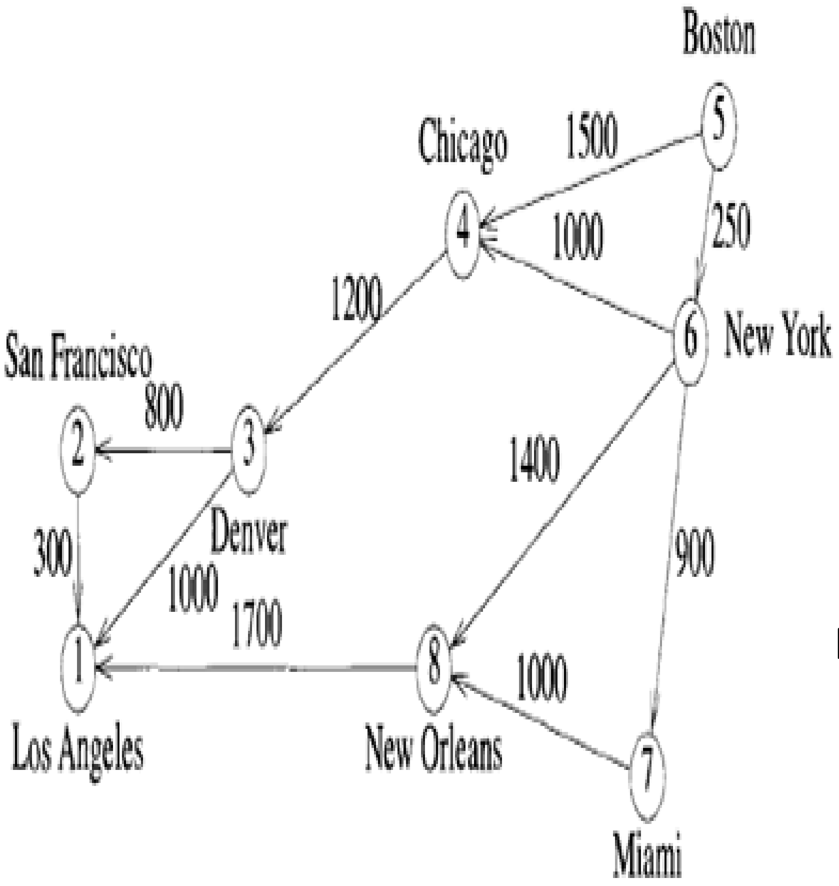
Distance from 5 by considering **3** as intermediate

City	Dist
1	3350
2	3250
3	2450
4	1250
5	0
6	250
7	1150
8	1650

For vertex 1: min[dist(1), dist(3)+(3,1)]
min[3350,2450+1000]
min(3350, 3450)=3350

For vertex 2: min[dist(2), dist(3)+(3,2)]
min[∞,2450+800]
min(∞, 3250)=3250

An Example



City	Dist
1	3350
2	3250
3	2450
4	1250
5	0
6	250
7	1150
8	1650

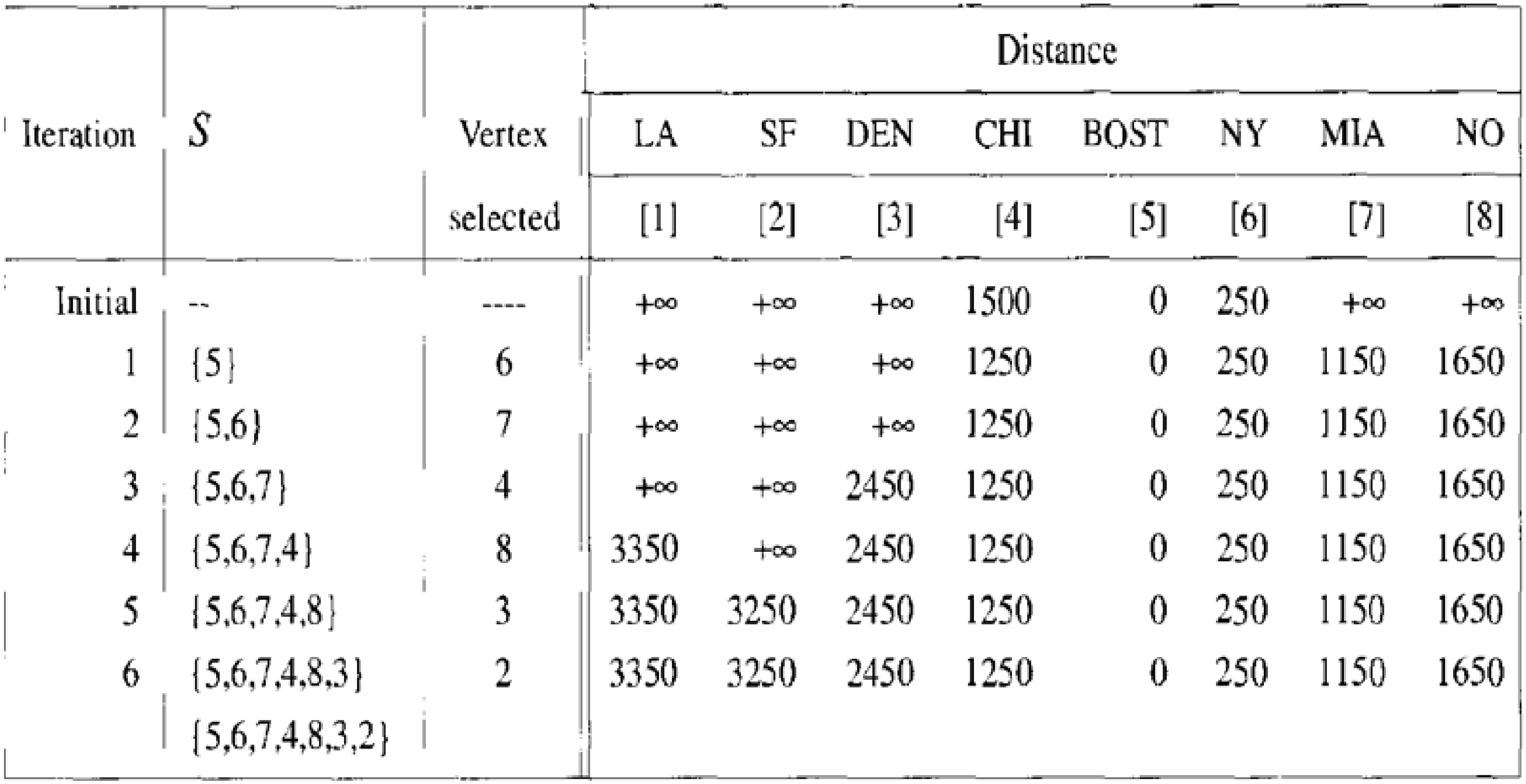
Update by considering 2 as intermediate

Distance from 5 by considering **2** as intermediate

City	Dist
1	3350
2	3250
3	2450
4	1250
5	0
6	250
7	1150
8	1650

For vertex 1: min[dist(1), dist(2)+(2,1)]
min[3350,3250+300]
min(3350, 3550)=3350

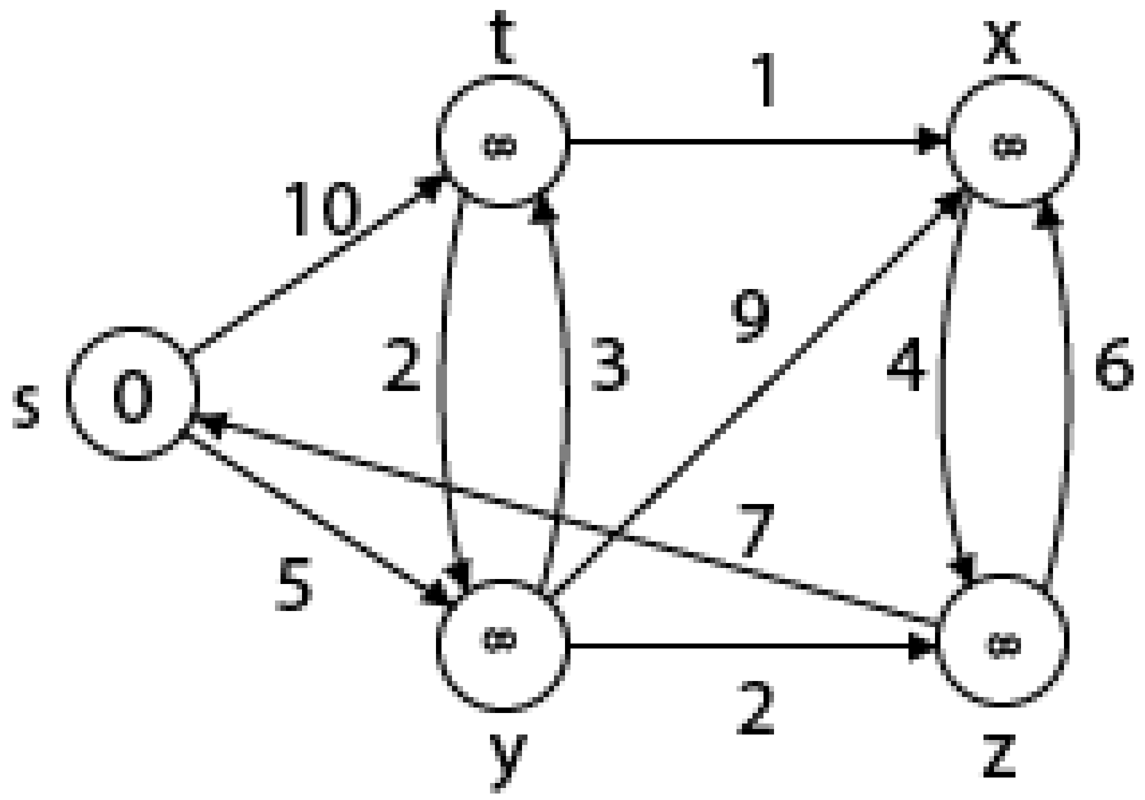
Solution for Example



Iteration	S	Vertex	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
		selected	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Example 2

Find the shortest path from S to all other vertices.



Pseudo Code for Single Source Shortest path Problem: Dijkstra’s Algorithm

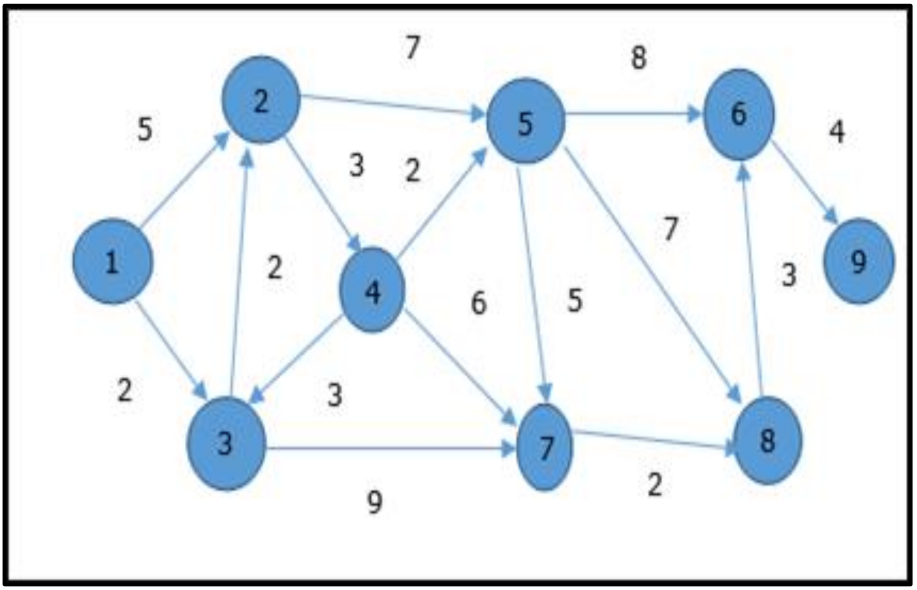
```
Algorithm ShortestPaths($v, cost, dist, n$)
// $dist[j], 1 \le j \le n$, is set to the length of the shortest
// path from vertex $v$ to vertex $j$ in a digraph $G$ with $n$
// vertices. $dist[v]$ is set to zero. $G$ is represented by its
// cost adjacency matrix $cost[1 : n, 1 : n]$.
{
    for $i := 1$ to $n$ do
    { // Initialize $S$.
        $S[i]$ := false; $dist[i] := cost[v, i]$;
    }
    $S[v]$ := true; $dist[v] := 0.0$; // Put $v$ in $S$.
    for $num := 2$ to $n - 1$ do
    {
        // Determine $n - 1$ paths from $v$.
        Choose $u$ from among those vertices not
        in $S$ such that $dist[u]$ is minimum;
        $S[u]$ := true; // Put $u$ in $S$.
        for (each $w$ adjacent to $u$ with $S[w]$ = false) do
            // Update distances.
            if ($dist[w] > dist[u] + cost[u, w]$)) then
                    $dist[w] := dist[u] + cost[u, w]$;
    }
}
```

Time Complexity

- The time taken by the algorithm on a graph with n vertices is $O(n^2)$ if graph is maintained as adjacency matrix.
  - for loop of line 7 takes O(n) time.
  - The for loop of line 12 is executed n-2 times.
  - Each execution of this loop requires O(n) time at lines 15and 16 to select the next vertex
  - the for loop of line 18 also takes O(n) to updated list.
  - So the total time for this loop is $O(n^2)$.

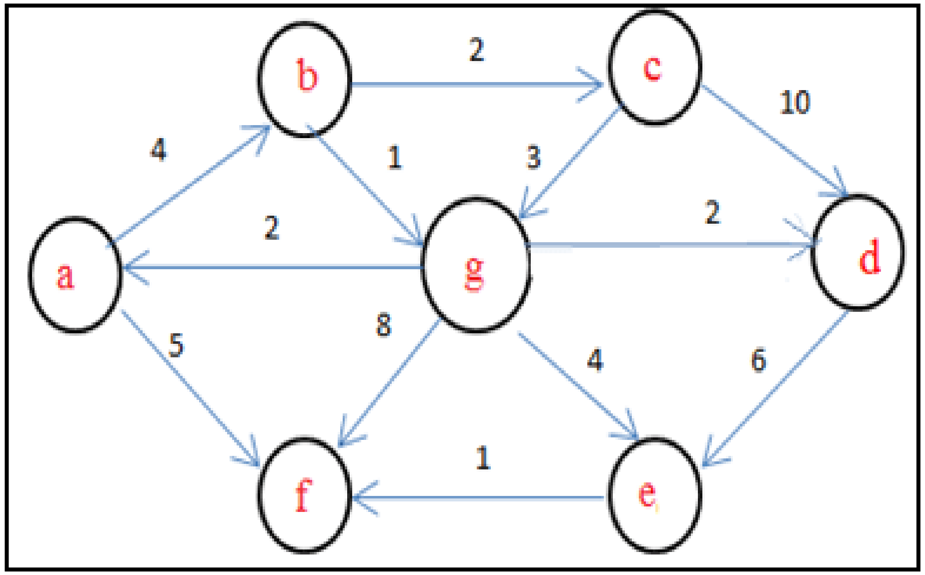
- If a change to adjacency lists is made, the overall frequency of the for loop of line18 can be brought down to O(|E|)(since dist can change only for vertices adjacent from u).
- If V - Sis maintained as a red-black tree,
  - Execution of lines 15and 16 takes O(logn) time. Note that a red-black
  - Each update in line21 takes O(logn) time as well (since an update can be done using a delete and an insertion into the red-black tree).
- Thus the overall run time is O((n+ |E|) logn).

Problems for Practice



Find the shortest path from source 1 to all other vertices

Find the shortest path from source a to all other vertices



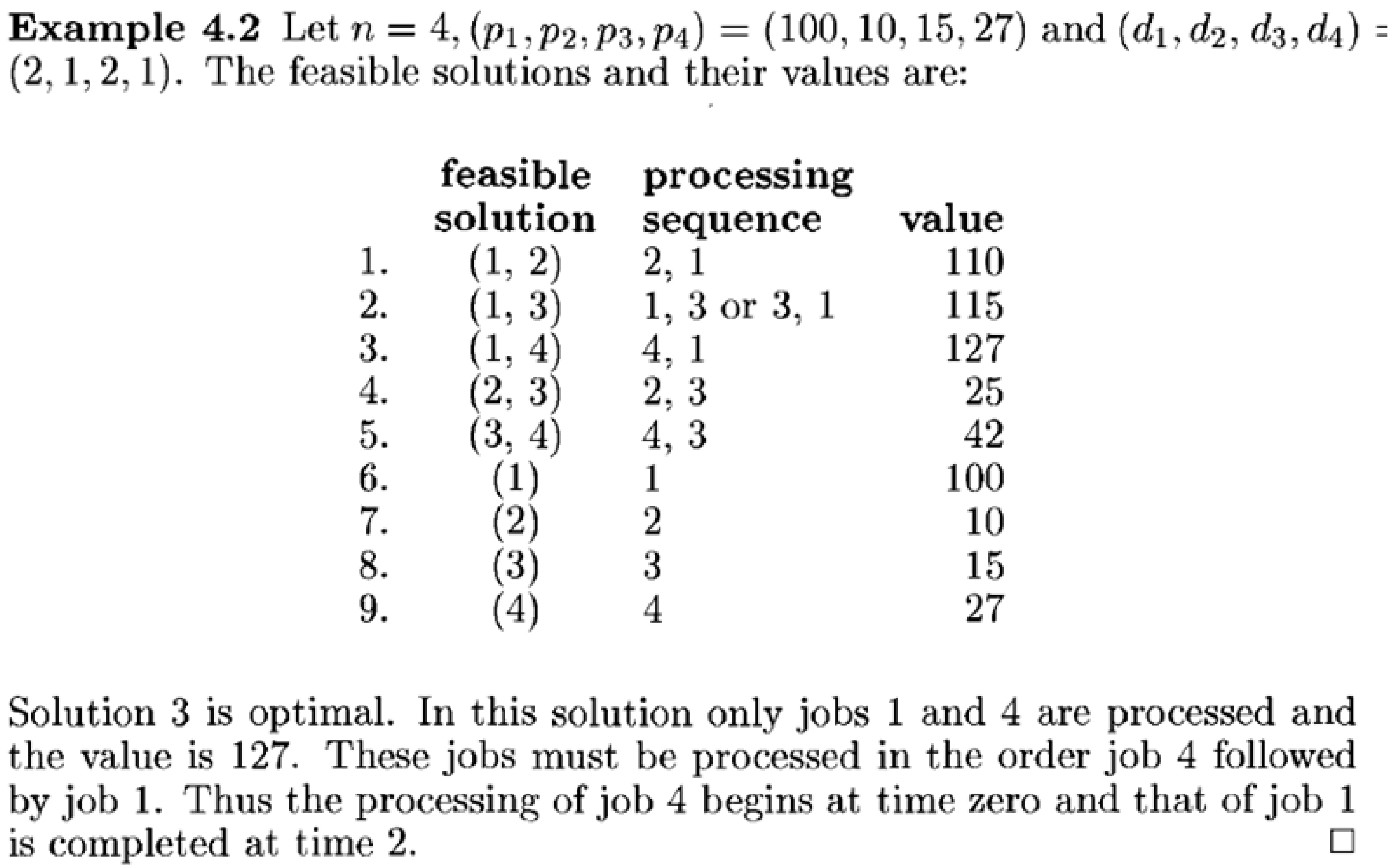
Topics

- General method
- Knapsack problem
- Minimum cost spanning trees:
  - Prim's algorithm
  - Kruskal's algorithm
- Single source shortest path problem
- **Job Sequencing with deadlines**

Job Sequencing with Deadlines

- We are given a set of n jobs.
- Associated with job i, there is an integer dead line d[i] >0 and a profit P[i] >0.
- For any job i the profit P[i] is earned if the job is completed by its dead line.
- To complete a job, one has to process the job on a machine for one unit of 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 dead line.
- An optimal solution is the feasible solution J for which sum of the profits of the jobs in J, is maximum.
- Here again, since the problem involves the identification of subset, it fits the subset paradigm

An Example



Example 4.2 Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible	**processing**	
	solution	**sequence**	**value**
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

Greedy Strategy

- **<u>Step-01:</u>**
  - Sort all the given jobs in decreasing order of their profit.

<u>Step-02:</u>

  - Check the value of maximum deadline.
  - Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.
  - Pick up the jobs one by one.
  - Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

<u>Step-03:</u>

  - Calculate the maximum earned profit

Example

Problem-

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule? |

Example

Solution-

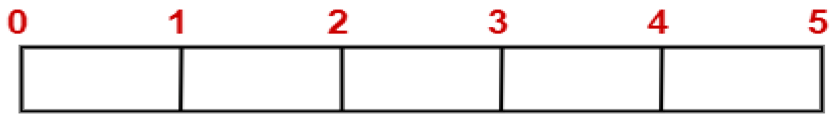
Step-01:

Sort all the given jobs in decreasing order of their profit-

Job	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.
So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



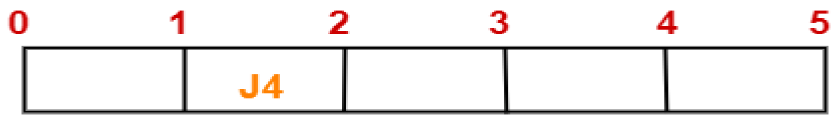
Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-

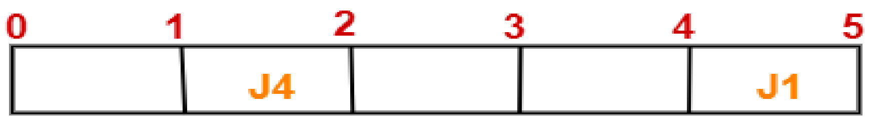


Example

Step-04:

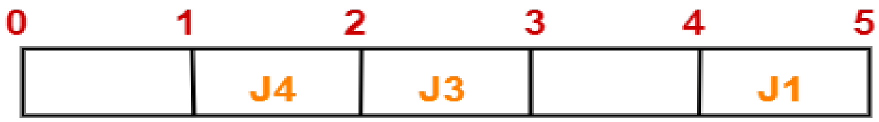
We take job J1.

- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Step-05:

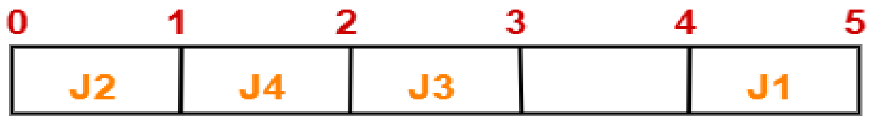
- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



Step-06:

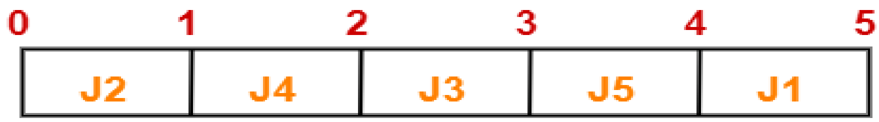
- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.

Since the second and third cells are already filled, so we place job J2 in the first cell as-



Step-07:

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



The optimal schedule is-

J2 , J4 , J3 , J5 , J1 **With a total profit of 990**

Example-Alternative way of doing

Solution-

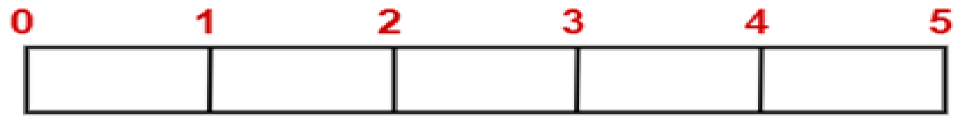
Step-01:

Sort all the given jobs in decreasing order of their profit-

Job	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.
So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



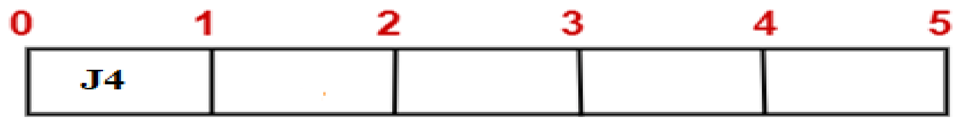
Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-

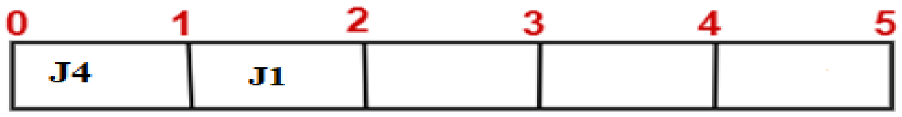


Example-Alternative way of doing

Step-04:

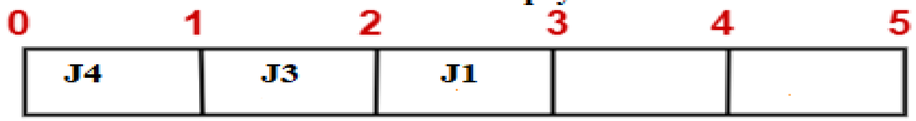
We take job J1.

- Since its deadline is 5, since its deadline is greater than previous job J4, place it in next location



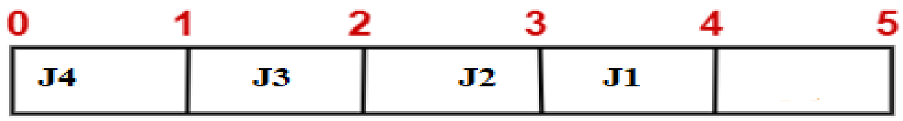
Step-05:

- We take job J3. Compare the deadline of J3 with previous jobs
- Since its deadline is 3, which is less than previous job J1 and greater than J4.so shift J1 to the next location and insert J3 in the empty location



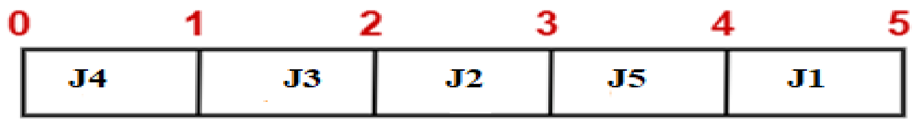
Step-06:

- We take job J2. compare the deadline with the previous jobs
- Since its deadline is 3, which is less than J1 and same as J3. so shift only J1 to the next location and insert J2



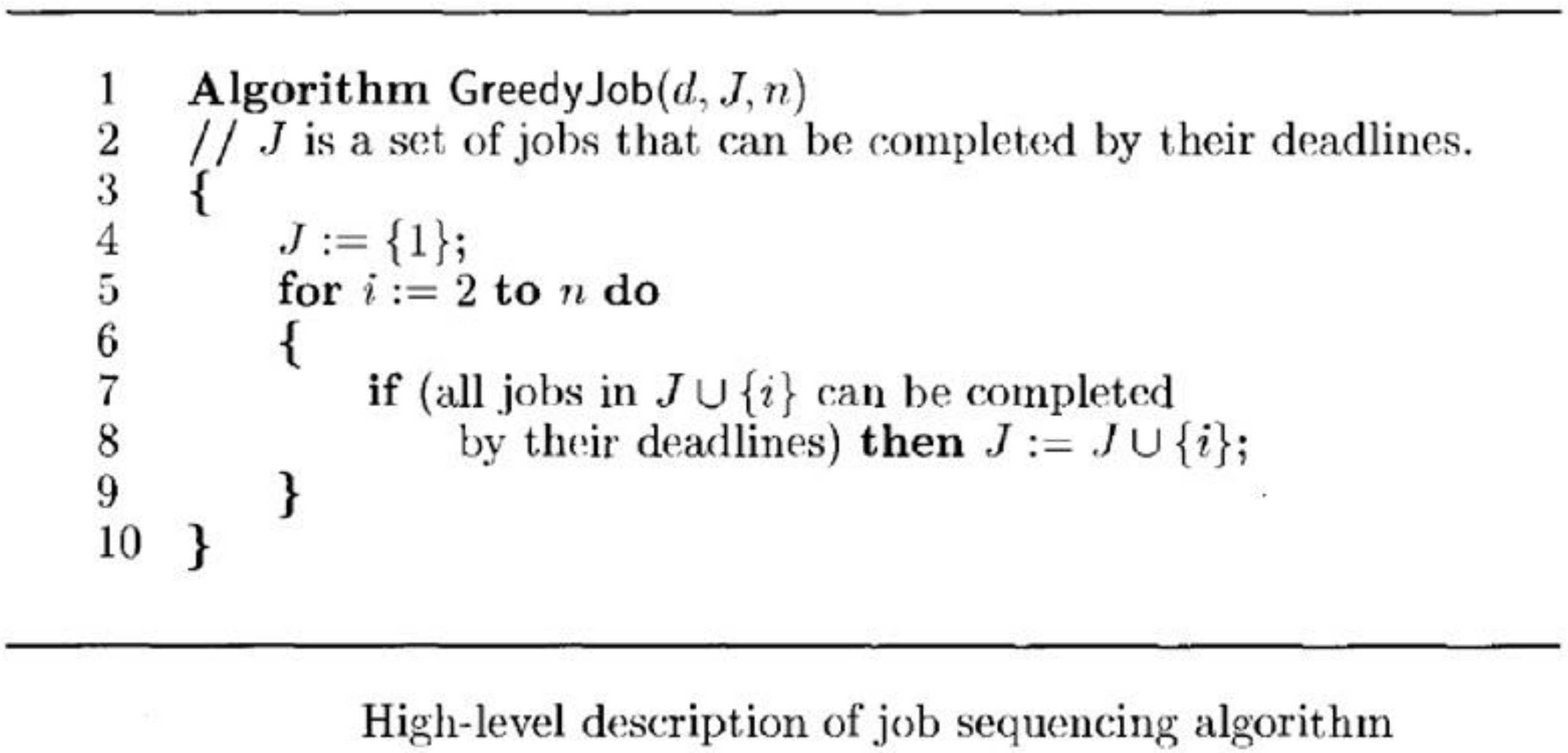
Step-07:

- Now, we take job J5.
- Since its deadline is 4, which is less than J1 and greater than J2, only J1 will be shifted to next location and J5 inserted.



The optimal schedule is- J4, J3, J2, J5, J1 WITH A TOTAL PROFIT OF 990

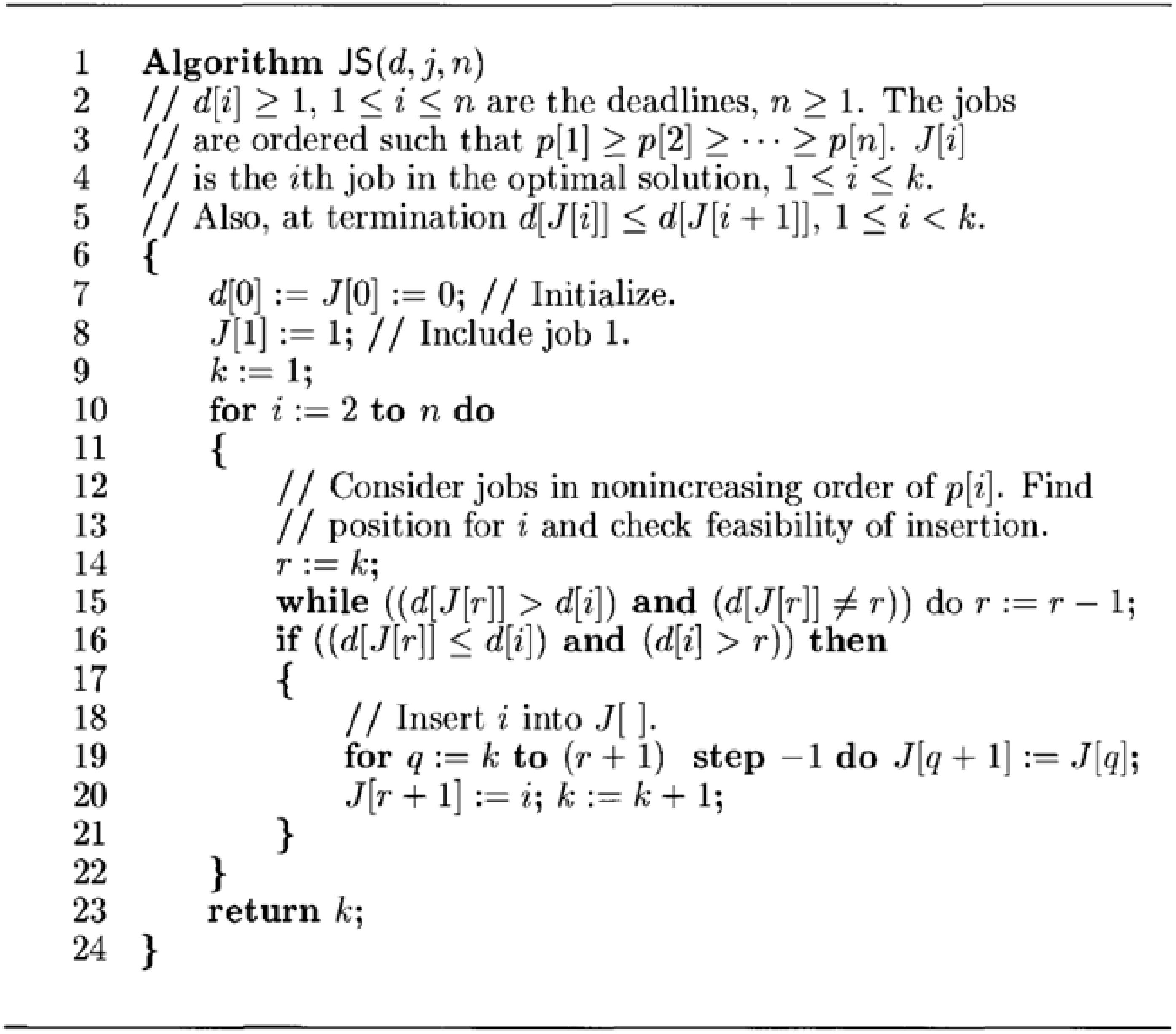
Algorithm for Job Sequencing



```
Algorithm GreedyJob($d, J, n$)
// $J$ is a set of jobs that can be completed by their deadlines.
{
    $J := \{1\}$;
    for $i := 2$ to $n$ do
    {
        if (all jobs in $J \cup \{i\}$ can be completed
            by their deadlines) then $J := J \cup \{i\}$;
    }
}
```

High-level description of job sequencing algorithm

Algorithm for Job Sequencing



```
Algorithm JS($d, j, n$)
// $d[i] \geq 1$, $1 \leq i \leq n$ are the deadlines, $n \geq 1$. The jobs
// are ordered such that $p[1] \geq p[2] \geq \cdots \geq p[n]$. $J[i]$
// is the $i$th job in the optimal solution, $1 \leq i \leq k$.
// Also, at termination $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$.
{
    $d[0] := J[0] := 0$; // Initialize.
    $J[1] := 1$; // Include job 1.
    $k := 1$;
    for $i := 2$ to $n$ do
    {
        // Consider jobs in nonincreasing 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 $(r + 1)$ step $-1$ do $J[q + 1] := J[q]$;
            $J[r + 1] := i$; $k := k + 1$;
        }
    }
    return $k$;
}
```

Time Complexity

- The while loop of line15 in Algorithm is iterated at most k times. Each iteration takes O(1)time.
- If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require O(k-r) time to insert job i.
- Hence, the total time for each iteration of the for loop of line 10 is O(k). This loopis iterated n -1times.
- If s is the final value of k, that is,s is the number of jobs in the final solution, then the total time needed by algorithm JS is O(sn).
- Since s < n, the worst-case time,as a function of n alone is $O(n^2)$.

Topics

- General method
- Knapsack problem
- Minimum cost spanning trees:
  - Prim‘s algorithm
  - Kruskal‘s algorithm
- Single source shortest path problem
- Job Sequencing with deadlines