


Greedy Method

- ▶ The greedy method is perhaps the most straight forward design technique.
 - ▶ we can be applied to a wide variety of problems. Most, though not all, of these 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.
- 

```

1  Algorithm Greedy( $a, n$ )
2  //  $a[1 : n]$  contains the  $n$  inputs.
3  {
4       $solution := \emptyset$ ; // Initialize the solution.
5      for  $i := 1$  to  $n$  do
6      {
7           $x := \text{Select}(a)$ ;
8          if Feasible( $solution, x$ ) then
9               $solution := \text{Union}(solution, x)$ ;
10     }
11     return  $solution$ ;
12 }

```

Greedy method control abstraction for the subset paradigm

Advantages of the Greedy Approach:

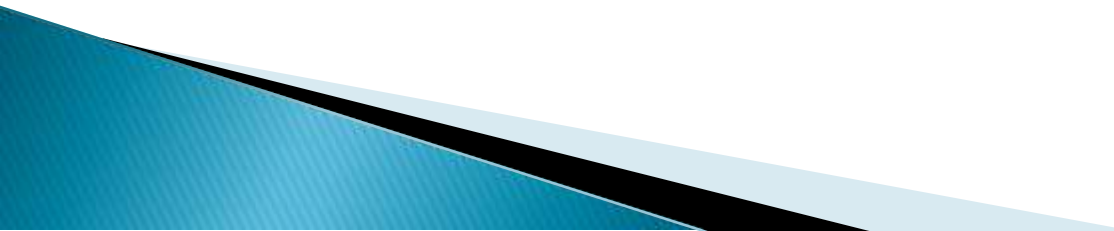
- ▶ The greedy approach is easy to implement.
- ▶ Typically have less time complexity.
- ▶ Greedy algorithms can be used for optimization purposes or finding close to optimization in case of Hard problems.

Disadvantages of the Greedy Approach:

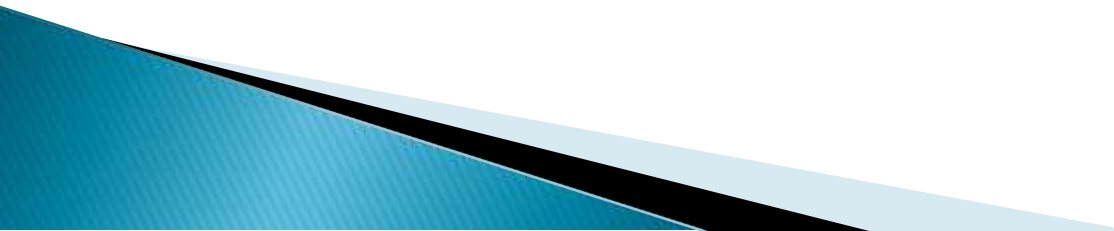
- ▶ The local optimal solution may not always be globally optimal.



Applications of Greedy Algorithm

- ▶ This algorithm is also used to solve the fractional knapsack problem.
 - ▶ It is used in a job sequencing with a deadline.
 - ▶ It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
 - ▶ It is used to Choose Optimal Merge Patterns.
 - ▶ It is used in finding the single source shortest path.
- 

1.KNAPSACK PROBLEM

- ▶ Apply the greedy method to solve the knapsack problem.
 - ▶ We are given n objects and a knapsack or bag.
 - ▶ Objects has a weight w_i and the knapsack has a capacity m .
 - ▶ If a fraction x_i , $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
 - ▶ The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
 - ▶ Since the Knapsack capacity is m , we require the total weight of all chosen objects to be at most m .
- 

- ▶ Formally, the problem can be stated as

$$\text{maximize } \sum_{1 \leq i \leq n} p_i x_i \quad (4.1)$$

$$\text{subject to } \sum_{1 \leq i \leq n} w_i x_i \leq m \quad (4.2)$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \quad (4.3)$$

The profits and weights are positive numbers. A feasible solution (or filling) is any set (x_1, \dots, x_n) satisfying (4.2) and (4.3) above.

An optimal solution is a feasible solution for which (4.1) is maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$.

Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	$(1/2, 1/3, 1/4)$	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

Of these four feasible solutions, solution 4 yields the maximum profit. we shall soon see, this solution is optimal for the given problem instance.

2. Consider the following instance of the knapsack problem:
 $n = 7, m = 15, (p_1, p_2, p_3, \dots, p_7) = (10, 5, 15, 7, 6, 18, 3)$ and,
 $(w_1, w_2, w_3, \dots, w_7) = (2, 3, 5, 7, 1, 4, 1)$.

3. Objects:

	1	2	3	4	5	6	7
Profit (P):	5	10	15	7	8	9	4
Weight(w):	1	3	5	4	1	3	2

W (Weight of the knapsack): 15
n (no of items): 7

4. For the given set of items and knapsack capacity = 60 kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

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 vector.

{

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**;

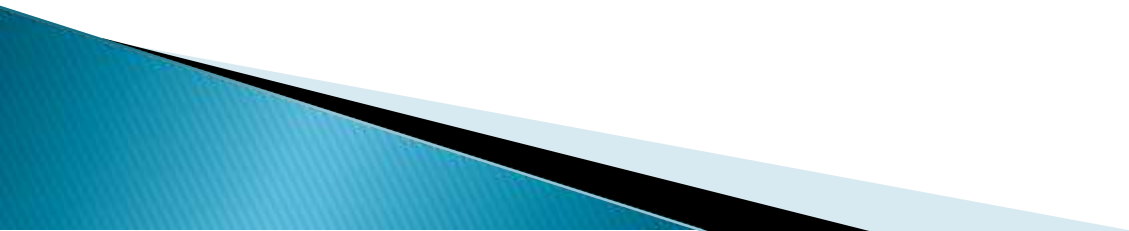
$x[i] := 1.0$; $U := U - w[i]$;

 }

if ($i \leq n$) **then** $x[i] := U/w[i]$;

}

The Time Complexity of knapsack problem is
 $O(n \log n)$



2. [0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem


$$\max \sum_{i=1}^n p_i x_i$$

$$\text{subject to } \sum_{i=1}^n w_i x_i \leq m$$

$$\text{and } x_i = 0 \text{ or } 1, \ 1 \leq i \leq n$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits. Show that this strategy doesn't necessarily yield an optimal solution.

2. Job Sequencing with Deadlines

- ▶ Given a set of n jobs. Associated with job i is an integer deadline $d_i > 0$ and a profit $p_i > 0$.
 - ▶ For any job i the profit p_i is earned iff the job is completed by its deadline.
 - ▶ 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 deadline.
 - ▶ The value of a feasible solution J is the sum of the profits of the jobs in J , or $\sum_{i \in J} p_i$
 - ▶ An optimal solution is a feasible solution with maximum value.
 - ▶ Here again, since the problem involves the identification of a subset, it fits the subset paradigm.
- 

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 solution	processing 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

Example 4.3 Let $n = 5$, $(p_1, \dots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \dots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
\emptyset	none	1	assign to $[1, 2]$	0
$\{1\}$	$[1, 2]$	2	assign to $[0, 1]$	20
$\{1, 2\}$	$[0, 1], [1, 2]$	3	cannot fit; reject	35
$\{1, 2\}$	$[0, 1], [1, 2]$	4	assign to $[2, 3]$	35
$\{1, 2, 4\}$	$[0, 1], [1, 2], [2, 3]$	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40.

□

Below is the greedy algorithm that is always supposed to give an optimal solution to the job sequencing problem.

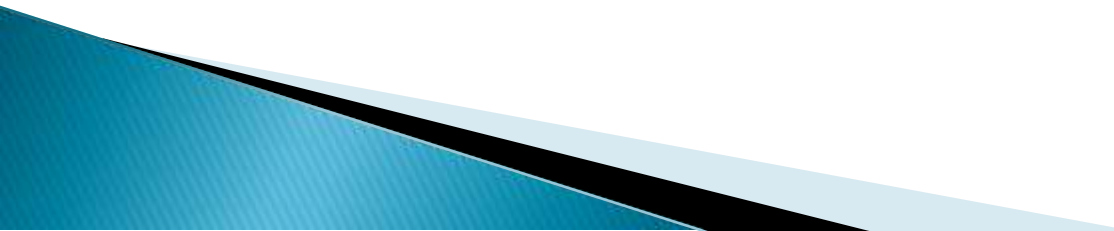
Step-01:

- ▶ Sorting of all the given jobs in the decreasing order of their profit.

Step-02:

- ▶ Checking the value of the maximum deadline.
- ▶ Drawing a Gantt chart such that the maximum time on the Gantt chart is the value of the maximum deadline.

Step-03:

- ▶ Picking up the jobs one after the other.
 - ▶ Adding the jobs on the Gantt chart in such a way that they are as far as possible from 0. This ensures that the job gets completed before the given deadline.
- 

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

Step-01:

Firstly, we need to sort all the given jobs in decreasing order of their profit as follows.

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

Step-02:

For each step, we calculate the value of the maximum deadline.

Here, the value of the maximum deadline is 5.

So, we draw a Gantt chart as follows and assign it with a maximum time on the Gantt chart with 5 units as shown below.



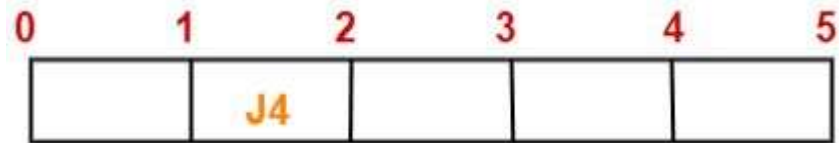
Gantt Chart

Now, We will be considering each job one by one in the same order as they appear in the Step-01.

We are then supposed to place the jobs on the Gantt chart as far as possible from 0.

Step-03:

- ▶ We now consider job4.
- ▶ Since the deadline for job4 is 2, we will be placing it in the first empty cell before deadline 2 as follows.



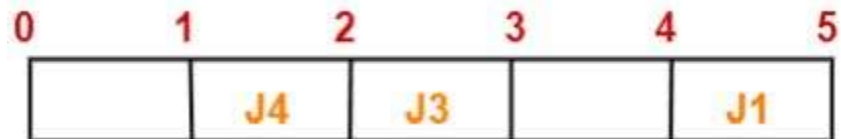
Step-04:

- ▶ Now, we go with job1.
- ▶ Since the deadline for job1 is 5, we will be placing it in the first empty cell before deadline 5 as shown below



Step-05:

- ▶ We now consider job3.
- ▶ Since the deadline for job3 is 3, we will be placing it in the first empty cell before deadline 3 as shown in the following figure.



Step-06:

- ▶ Next, we go with job2.
- ▶ Since the deadline for job2 is 3, we will be placing it in the first empty cell before deadline 3.
- ▶ Since the second cell and third cell are already filled, so we place job2 in the first cell as shown below.



Step-07:

- ▶ Now, we consider job5.
- ▶ Since the deadline for job5 is 4, we will be placing it in the first empty cell before deadline 4 as shown in the following figure.



- ▶ Now, We can observe that the only job left is job6 whose deadline is 2.
- ▶ Since all the slots before deadline 2 are already occupied, job6 cannot be completed.

Part-01:

- ▶ The optimal schedule is–**Job2, Job4, Job3, Job5, Job1**
- ▶ In order to obtain the maximum profit this is the required order in which the jobs must be completed.

Part-02:

- ▶ As we can observe, all jobs are not completed on the optimal schedule.
- ▶ This is because job6 was not completed within the given deadline.

Part-03:

Maximum earned profit = Sum of the profit of all the jobs from the optimal schedule

= Profit of job2 + Profit of job4 + Profit of job3 + Profit of job5 + Profit of job1

= 180 + 300 + 190 + 120 + 200

= 990 units

```

1  Algorithm JS( $d, j, n$ )
2  //  $d[i] \geq 1$ ,  $1 \leq i \leq n$  are the deadlines,  $n \geq 1$ . The jobs
3  // are ordered such that  $p[1] \geq p[2] \geq \dots \geq p[n]$ .  $J[i]$ 
4  // is the  $i$ th job in the optimal solution,  $1 \leq i \leq k$ .
5  // Also, at termination  $d[J[i]] \leq d[J[i + 1]]$ ,  $1 \leq i < k$ .
6  {
7       $d[0] := J[0] := 0$ ; // Initialize.
8       $J[1] := 1$ ; // Include job 1.
9       $k := 1$ ;
10     for  $i := 2$  to  $n$  do
11     {
12         // Consider jobs in nonincreasing order of  $p[i]$ . Find
13         // position for  $i$  and check feasibility of insertion.
14          $r := k$ ;
15         while  $((d[J[r]] > d[i])$  and  $(d[J[r]] \neq r))$  do  $r := r - 1$ ;
16         if  $((d[J[r]] \leq d[i])$  and  $(d[i] > r))$  then
17         {
18             // Insert  $i$  into  $J[ ]$ .
19             for  $q := k$  to  $(r + 1)$  step  $-1$  do  $J[q + 1] := J[q]$ ;
20              $J[r + 1] := i$ ;  $k := k + 1$ ;
21         }
22     }
23     return  $k$ ;
24 }

```

Greedy algorithm for sequencing unit time jobs with deadlines and profits

Analysis of the algorithm:

- ▶ In the job sequencing with deadlines algorithm, we make use of two loops, one loop within another. Hence, the complexity of this algorithm would be $O(n^2)$.



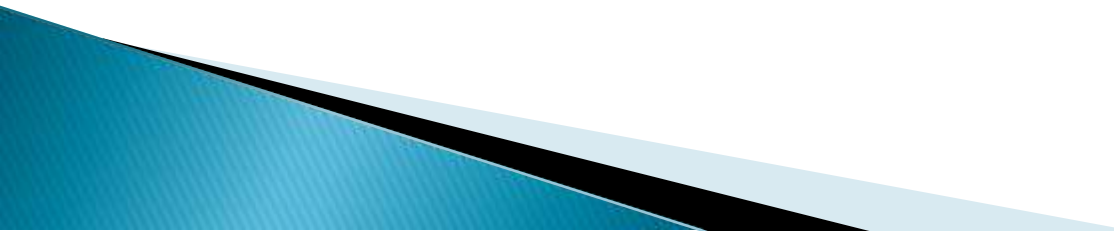
3. Minimum cost Spanning Tree

- ▶ A **spanning tree** is a subset of an undirected Graph that has all the vertices connected by minimum number of edges. For a graph, there may exist more than one spanning tree.

Properties

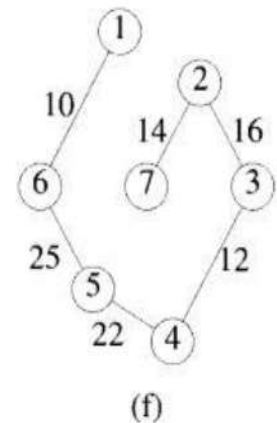
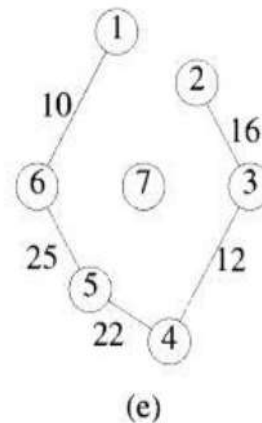
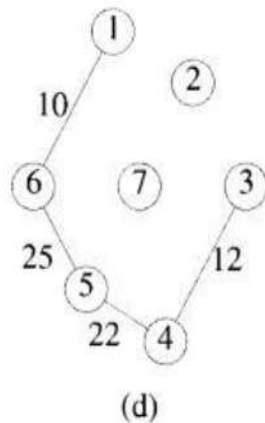
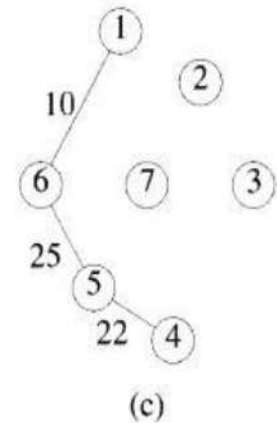
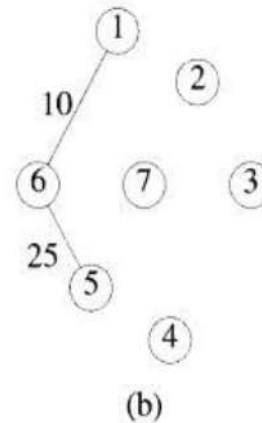
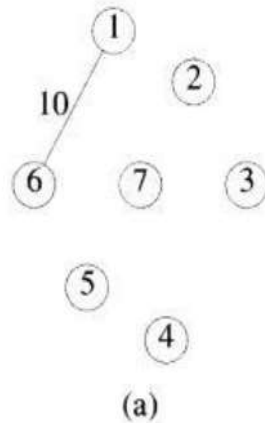
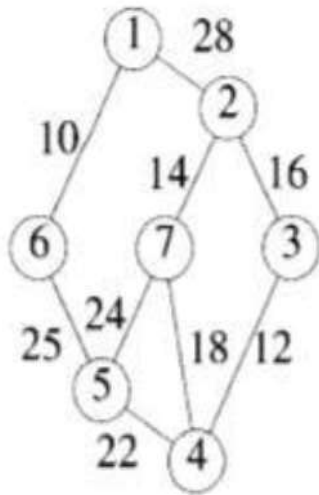
- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.
- ▶ Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a spanning tree of G iff t is a tree.
- ▶ If there are n number of vertices, the spanning tree should have $n - 1$ number of edges.
- ▶ A Minimum Spanning Tree (MST) is a subset of edges that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used.

a.Prim's algorithm

- ▶ Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows –
 - ▶ First, we have to initialize an MST with the randomly chosen vertex.
 - ▶ Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
 - ▶ Repeat step 2 until the minimum spanning tree is formed.
- 

Prim's algorithm

➤ working principle of Prim's Algorithm



```

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

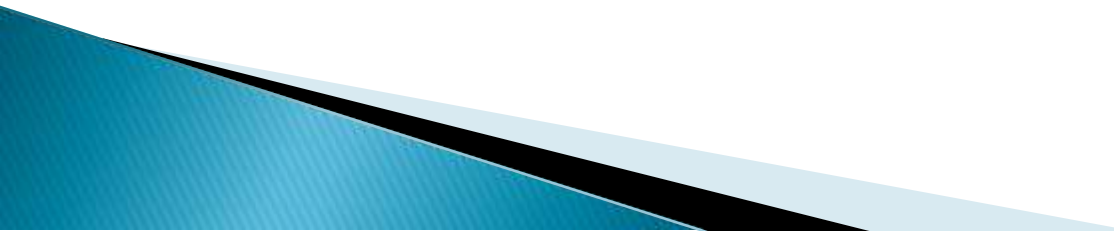
```

- ▶ The time complexity of the prim's algorithm is $O(V^2)$, where V is the no. of vertices.

b.Kruskal's algorithm

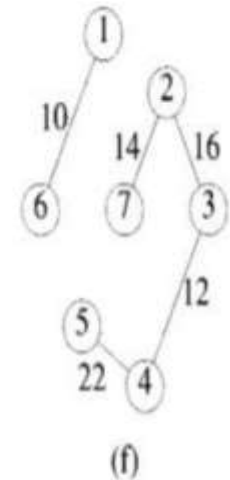
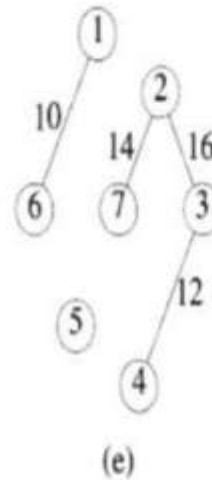
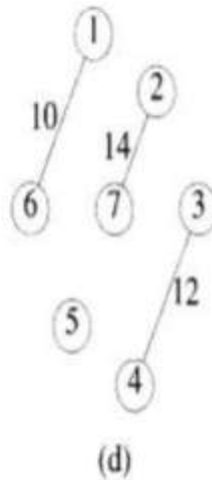
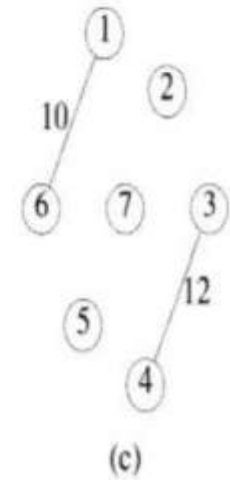
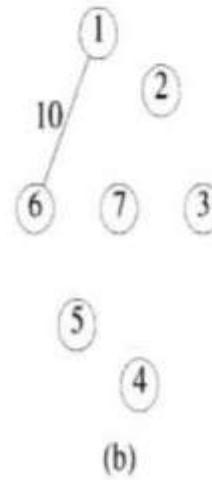
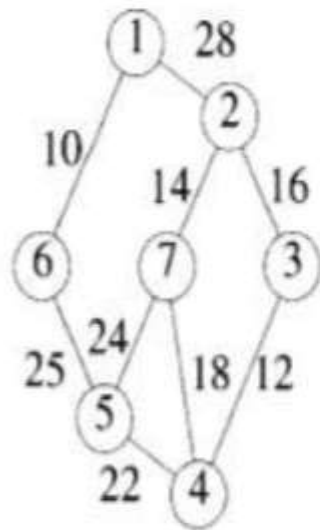
- ▶ In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached.

The steps to implement Kruskal's algorithm are listed as follows –

- ▶ First, sort all the edges from low weight to high.
 - ▶ Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
 - ▶ Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.
- 

Kruskal's algorithm

➤ working principle of Kruskal's Algorithm



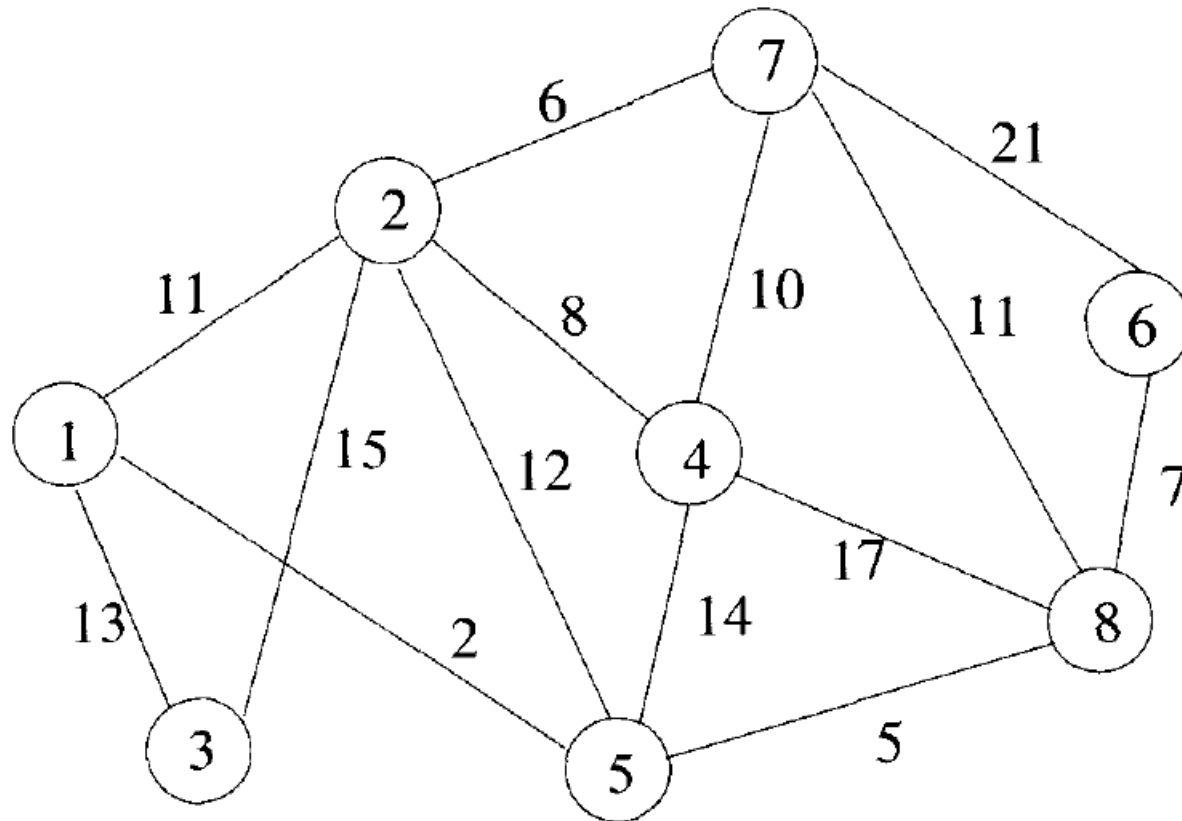
```

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

► Time Complexity

The time complexity of Kruskal's algorithm is $O(E \log V)$ where E is the no. of edges, and V is the no. of vertices.

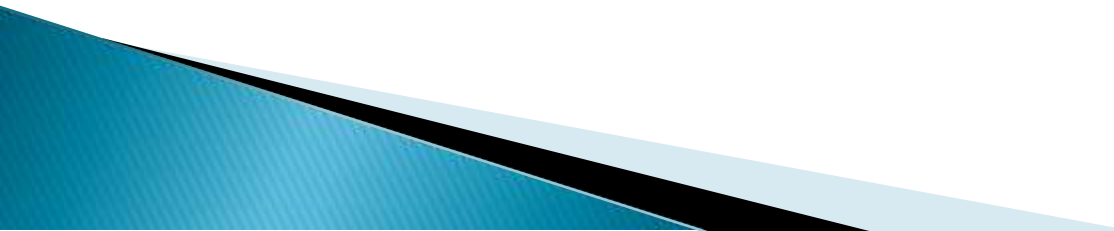
1. Compute a minimum cost spanning tree for the graph of Figure using (a) Prim's algorithm and (b) Kruskal's algorithm.



Difference between Prim's and Kruskal Algorithm

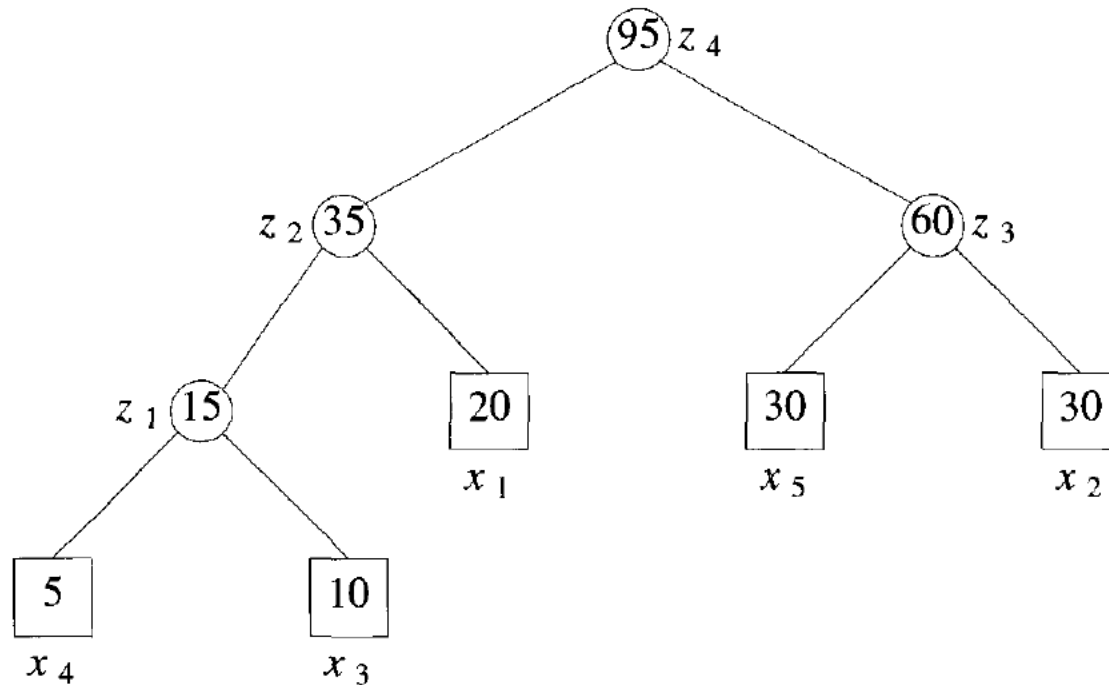
S.No.	Prim's Algorithm	Kruskal's Algorithm
1	This algorithm begins to construct the shortest spanning tree from any vertex in the graph.	This algorithm begins to construct the shortest spanning tree from the vertex having the lowest weight in the graph.
2	To obtain the minimum distance, it traverses one node more than one time.	It crosses one node only one time.
3	The time complexity of Prim's algorithm is $O(V^2)$.	The time complexity of Kruskal's algorithm is $O(E \log V)$.
4	In Prim's algorithm, all the graph elements must be connected.	Kruskal's algorithm may have disconnected graphs.
5	When it comes to dense graphs, the Prim's algorithm runs faster.	When it comes to sparse graphs, Kruskal's algorithm runs faster.
6	It prefers list data structure.	It prefers the heap data structure.

4.Optimal Merge Patterns

- ▶ Given n sorted files, there are many ways in which to pair wise merge them into a single sorted file. Different pairings require differing amounts of computing time.
 - ▶ The problem we address ourselves to now is that of determining an optimal way(one requiring the fewest comparisons) pairwise merge n sorted files.
 - ▶ Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.
- 

- ▶ Greedy attempt to obtain an optimal merge pattern is easy to formulate.
- ▶ Since merging an n -record file and an m -record file requires possibly $n + m$ record moves, the obvious choice for a selection criterion is at each Step merge the two smallest size files together.
- ▶ The merge pattern such as the one just described will be referred to as a two-way merge pattern (each merge step involves the merging of two files).
- ▶ The two-way merge patterns can be represented by binary merge trees.
- ▶ The leaf nodes are drawn as squares and represent the given files. These nodes are called external nodes.
- ▶ The remaining nodes are drawn as circles and are called internal nodes. Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children.
- ▶ The number in each node is the length(i.e. the number of records)of the file represented by that node.

- ▶ we have five files (x_1, x_2, x_3, x_4, x_5) with sizes (20, 30, 10, 5, 30), our greedy rule would generate the following merge pattern.



If d_i is the distance from the root to the external node for file x_i and q_i , the length of x_i is then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^n d_i q_i$$

- ▶ This sum is called the weighted external path length of the tree.

```

treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

```

```

1  Algorithm Tree(n)
2  // list is a global list of n single node
3  // binary trees as described above.
4  {
5      for i := 1 to n - 1 do
6      {
7          pt := new treenode; // Get a new tree node.
8          (pt → lchild) := Least(list); // Merge two trees with
9          (pt → rchild) := Least(list); // smallest lengths.
10         (pt → weight) := ((pt → lchild) → weight)
11                     + ((pt → rchild) → weight);
12         Insert(list, pt);
13     }
14     return Least(list); // Tree left in list is the merge tree.
15 }

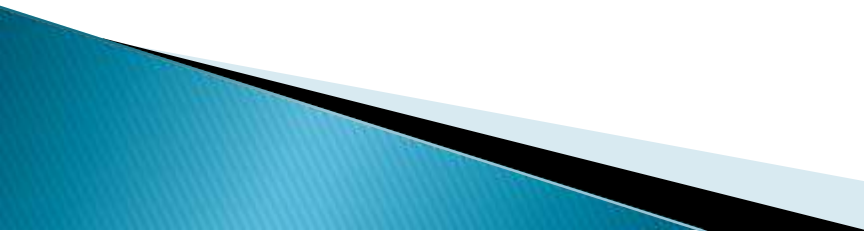
```

Algorithm to generate a two-way merge tree

- ▶ If list is kept in non decreasing order according to the weight value in the roots, then $\text{Least}(\text{list})$ requires only $O(1)$ time and $\text{insert}(\text{list})$, can be done in $O(n)$ time. Hence the total time taken is $O(n^2)$.
- ▶ In case list is represented as a minheap in which the root value is less than or equal to the values of its children, then $\text{Least}(\text{list})$ and $\text{Insert}(\text{list})$, can be done in $O(\log n)$ time. In this case the computing time for Tree is $O(n \log n)$.

1. Find an optimal binary merge pattern for ten files whose lengths are 28, 32, 12, 5, 84, 53, 91, 35, 3, and 11.

Huffman Code

- ▶ Another application of binary trees with minimal weighted external path Length is to obtain an optimal set of codes for message M_1, M_2, \dots, M_{n+1} .
 - ▶ Each code is a binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using a decode tree.
 - ▶ A decode tree is a binary tree in which external nodes represent messages.
- 

after
iteration

list

initial

2

3

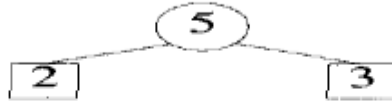
5

7

9

13

1



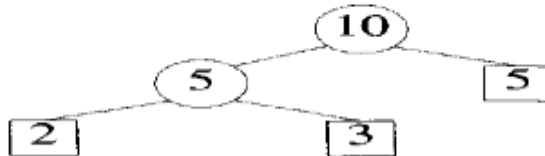
5

7

9

13

2



7

9

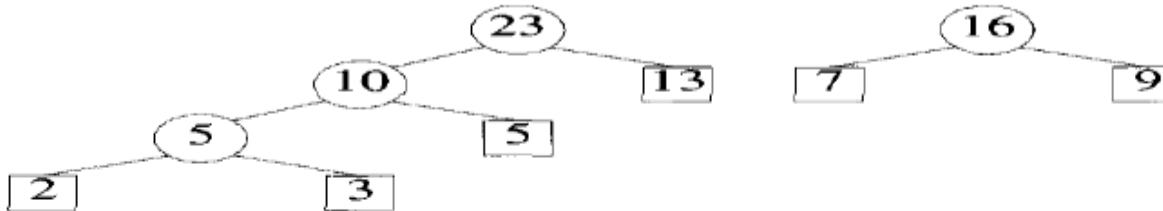
13

3

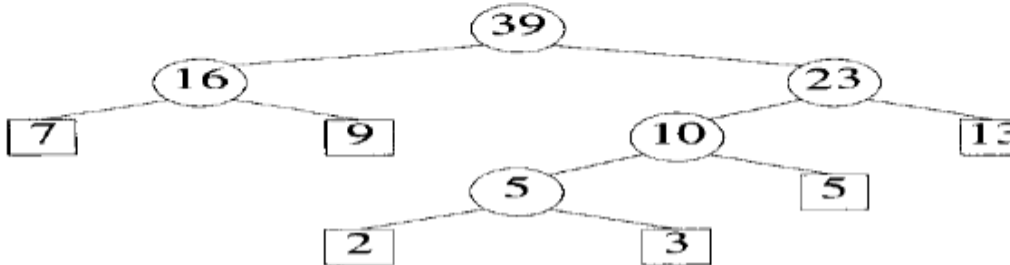


13

4



5



- ▶ The binary bits in the code word for a message determine the branching
Needed at each level of the decode tree to reach the correct external node.
- ▶ For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure corresponds to codes 000, 001, 01, and 1 for message M1, M2, M3 and M4 respectively.
- ▶ These codes are called Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is

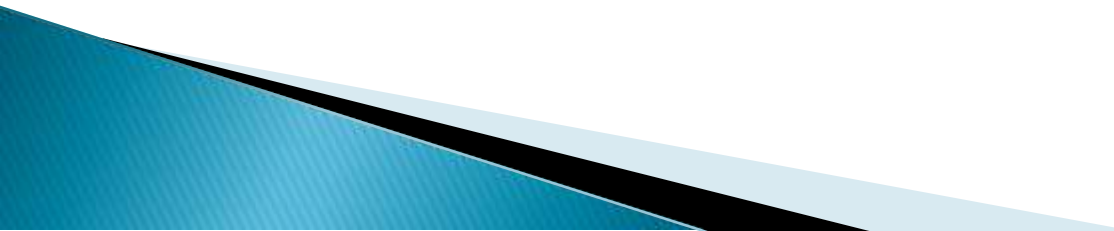
$$\sum_{1 \leq i \leq n+1} q_i d_i$$

where d is the distance of the external node for message M_i from the root node.

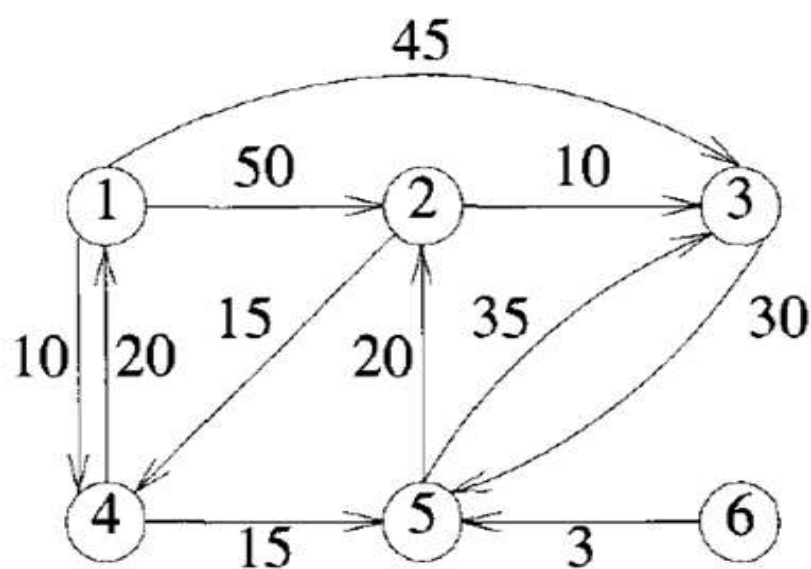
- ▶ The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length
- ▶ Hence the code that minimizes expected decode time also minimizes the expected length of a message.

Obtain a set of optimal Huffman codes for the messages (M_1, \dots, M_7) with relative frequencies $(q_1, \dots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree for this set of codes.

5.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 average 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?
- 

- ▶ 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.
- ▶ In the problem we consider, we are given a directed graph $G = (V, E)$, a weighting function cost for the edges of G , and a source vertex v_0 . The problem is to determine the shortest paths from v_0 to all the remaining vertices of G .
- ▶ It is assumed that all the weights are positive. The shortest path between v_0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.



(a) Graph

Path	Length
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

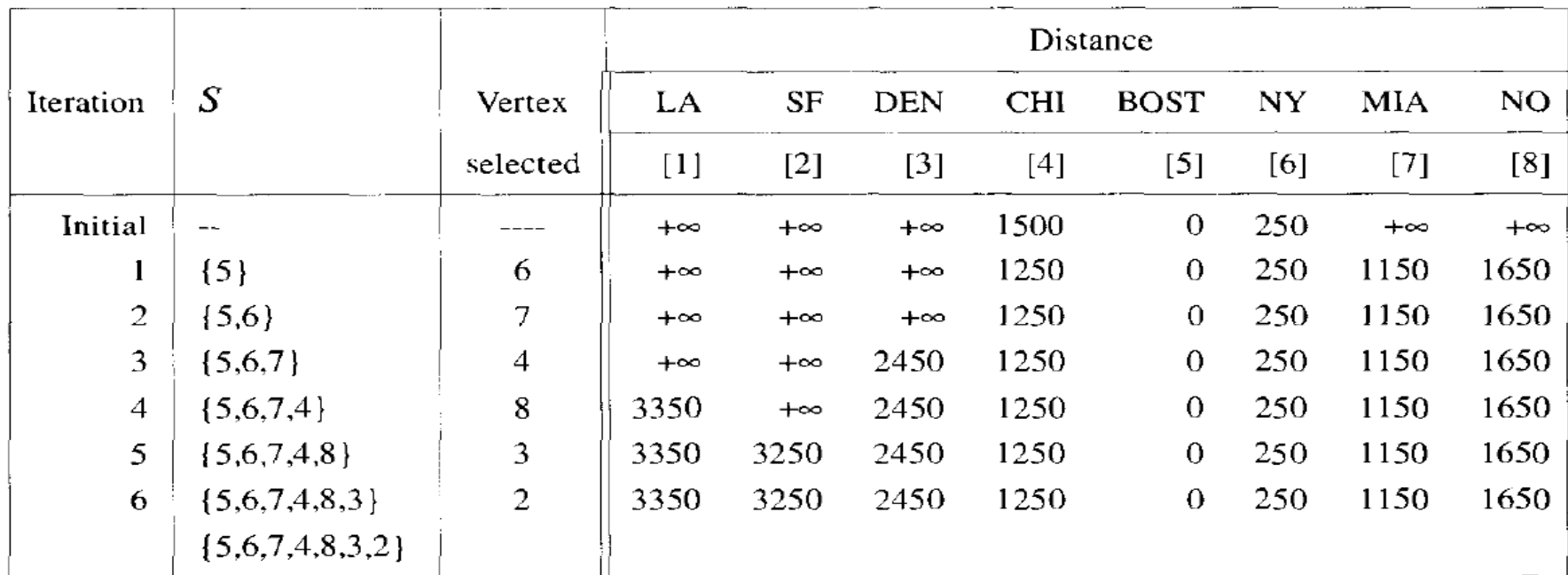
(b) Shortest paths from 1

Sources	Destinations				
1	2	3	4	5	6
1	50	45	10	∞	∞
1, 4	50	45	10	25	∞
1, 4, 5	45	45	10	25	∞
1, 4, 5, 2	45	45	10	25	∞
1, 3	45	45	10	25	∞

```

1  Algorithm ShortestPaths( $v, cost, dist, n$ )
2  //  $dist[j]$ ,  $1 \leq j \leq n$ , is set to the length of the shortest
3  // path from vertex  $v$  to vertex  $j$  in a digraph  $G$  with  $n$ 
4  // vertices.  $dist[v]$  is set to zero.  $G$  is represented by its
5  // cost adjacency matrix  $cost[1 : n, 1 : n]$ .
6  {
7      for  $i := 1$  to  $n$  do
8      { // Initialize  $S$ .
9           $S[i] := \mathbf{false}$ ;  $dist[i] := cost[v, i]$ ;
10     }
11      $S[v] := \mathbf{true}$ ;  $dist[v] := 0.0$ ; // Put  $v$  in  $S$ .
12     for  $num := 2$  to  $n - 1$  do
13     {
14         // Determine  $n - 1$  paths from  $v$ .
15         Choose  $u$  from among those vertices not
16         in  $S$  such that  $dist[u]$  is minimum;
17          $S[u] := \mathbf{true}$ ; // Put  $u$  in  $S$ .
18         for (each  $w$  adjacent to  $u$  with  $S[w] = \mathbf{false}$ ) do
19             // Update distances.
20             if ( $dist[w] > dist[u] + cost[u, w]$ ) then
21                  $dist[w] := dist[u] + cost[u, w]$ ;
22     }
23 }
```

- ▶ The time taken by the algorithm on a graph with n vertices is $O(n^2)$
- ▶ The running time of Dijkstra's algorithm using Binary min-heap method is $O(E \log V)$



- ▶ Use algorithm Shortest Paths to obtain in non decreasing order the lengths of the shortest paths from vertex 1 to all remaining vertices in the digraph

