Greedy Algorithms

Greedy algorithms – overview

Greedy design technique is primarily used in Optimization problems.
Optimization problems are problems where in we would like to find the best of all possible solutions. In other words, we need to find the solution which has the optimal (maximum or minimum) value satisfying the given constraints.

The Greedy approach helps in constructing a solution for a problem through a sequence of steps where each step is considered to be a partial solution. This partial solution is extended progressively to get the complete solution.

In the greedy approach each step chosen has to satisfy the constraints given in the problem. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

Change making example:

Suppose, we want to make change for an amount 'A' using fewest no of currency notes. Assume the available denominations are Rs 1, 2, 5, 10, 20, 50, 100, 500, 1000.
To make a change for A=Rs 28, with the minimum number of notes, one would first choose a note of denomination Rs 20, 5, 2 and 1.

Denomination table					
for Rs 28		**for Rs 783**		**for Rs 3799**	
1000 X **0**	0	1000 X		1000 X	
500 X **0**	0	500 X		500 X	
100 X**0**	0	100 X		100 X	
50 X **0**	0	50 X		50 X	
20 X **1**	20	20 X		20 X	
10 X **0**	0	10 X		10 X	
5 X **1**	5	5 X		5 X	
2 X **1**	2	2 X		2 X	
1 X **1**	1	1 X		1 X	
Total	**28**	Total		Total	

```
Algorithm change making(denom_value[ ], TargetAmount)
{
        // denom={1000, 500, 100, 50, 20, 10, 5, 2, 1}

        selected amount (SA) :=0;
        i :=1;
        while (SA < TargetAmount){
                if (SA + denom_value [i] <= TargetAmount ) {       // Select & Feasible

                        denom_select[i] ++;    // Union
                        SA := SA + denom_value[i];
```

```
            }
            else {
                   i++;
                   }
        }
        Print denom _select
}
```

Greedy Algorithm-General method

In Greedy method the problems have **'n'** inputs called as candidate set, from which a subset is selected to form a solution for the given problem. Any subset that satisfies the given constraints is called a feasible solution. We need to find a feasible solution that maximizes or minimizes an objective function and such solution is called an optimal solution.

In the above ex currency notes denomination set { 1000 ….1000 ,500….500, 100….100, 50…50, 20…20,10…10,5…5,2..2,1…1}is candidate set.

In the above ex constraint is our solution make the exact target amount of cash. Hence, any feasible solution i.e. sum of selected notes should be equal to target amount.

In the above ex objective function is our solution should consist of the fewest number of currency notes. Hence, any optimal solution which is one of the feasible solutions that optimizes the objective function. There can be more than one optimal solution.

Control Abstraction for Greedy General Method

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

Greedy method consists of 3 functions (steps).

1) **Select:** it selects an input from array a[ ] (candidate set) and puts in the variable x.

2) **Feasible:** it is a Boolean function which checks whether the selected input meets the constraints or not.

3) **Union**: if the selected input i.e. 'x' makes the solution feasible, then x is included in the solution and objective function get updated.

Characteristics of Greedy:

1) These algorithms are simple and straightforward and easy to implement.
2) They take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.
3) They work in stages and never reconsider any decision.

Greedy Algorithms Applications

Knapsack problem:

A thief robbing a store finds n items, the items each worth v_i rupees and weights w_i grams, where v_i and w_i are positive numbers. He wants to take as valuable load as possible but he can carry at most w grams in his knapsack(bag). Which item should he take?

They are two types of knapsack problem.

1) 0-1 knapsack problem: Here the items may not be broken into smaller pieces, so thief may decide either to take an item or to leave to it(binary choice). It cannot be efficiently solved by greedy algorithm

2) Fractional (General) Knapsack problem: Here thief can take the fraction of items, meaning that the items can be broken into smaller pieces so that thief may be able to carry a fraction x_i of item i. This can be solved easily by greedy.

If a fraction x_i , $0 \leq x_i \leq 1$, of objects 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

$$\text{Maximize } \sum P_i x_i \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..(1)$$

$$\text{Subject to } \sum W_i x_i \leq m \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..(2)$$

$$\text{And } 0 \leq x_i \leq 1,\ 1 \leq i \leq n \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..(3)$$

The profit and weights are positive numbers. A feasible solution is any set $(x_1, x_2, \ldots\ldots\ldots\ldots\ldots\ldots.. x_n)$ satisfying (2) and(3). An optimal solution is feasible solution for which (1) is maximized.

Eg; consider the following instance of the knapsack problem.

n=3, m=20 , (P_1, P_2, P_3) =(25, 24, 15) & (w_1, w_2, w_3) = (18,15,10)

Note that knapsack problem calls for select a subset of the objects hence fits the subset paradigm.

1) We can try to fill the knapsack by including the object with largest profit(greedy approach to the profit) .If an object under consideration does not fit, then a fraction of it is included to fit the knapsack. Object 1 has the largest profit value.P_1=25. So it is placed into the knapsack first. Then x_1=1 and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Objects 2 has the next largest profit P_2=24. But W_2=15 & it does not fit into the knapsack. Using x_2=2/15 fills the knapsack exactly with the part of the object 2.

The method used to obtain this solution is termed a greedy method at each step, we chose to introduce that object which would increase the objective function value the most.

(x1 , x2 , x3)	∑ wi xi	∑ pi xi
(1 , 2/15 , 0)	20	28.2

This is **not an optimal solution.**

2)We apply greedy approach by choosing value per unit weight is as high as possible

Item(n)	Value(p1,p2,p3)	Weight(w1,w2,w3)	Val/weight
1	25	18	1.388
2	24	15	1.6
3	15	10	1.5

Here $\mathbf{p_2/w_2 > p_3/w_3 > p_1/w_1}$. Now the items are arranged into non increasing order of p_i/w_i. Second item is the most valuable item. We chose item 2 first. Item 3 is second most valuable item. But we cannot choose the entire item3 as the weight of item 3 exceeds the capacity of knapsack. We can take ½ of the third item. Therefore the solution is $x_1 = 0$, $x_2 = 1$, $x_3 = ½$ and maximum profit is $\sum p_i x_i$ = 0*25 + 1*24 + ½ * 15 = 31.5

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
(1, 2/15, 0)	20	28.2
(0, 2/3, 1)	20	31
(0, 1, 1/2)	20	31.5

If the items are already arranged in non increasing order of pi/wi , then the function greedy knapsack obtains solution corresponding to this strategy.

```
Algorithm Greedy Knapsack(a,n)
// Objects are sorted in the non-increasing order of p[i]/w[i]
{
   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; U:=U - w[i];
       }
  if (i <= n) then x[i] := U / w[i];
}
```

Analysis of Greedy Knapsack

- If the items are already sorted into decreasing order of vi/wi, then time complexity is **O(n)**
- Therefore Time complexity including sort is **O(n log n)**

Job Sequencing with deadlines:

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $P_i \geq 0$.For any job i profit P_i is earned iff the job is completed by its deadline. To complete a job, one has to process 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 $\sum P_i$. An optimal solution is a feasible solution with maximum value. Here the problem involves the identification of a subset, it fits the subset paradigm.

Eg; Let n=4. $(P_1,P_2,P_3,P_4) = (100,10,15,27)$ and (d1,d2,d3,d4)=(2,1,2,1). d1 =2 means first job should be completed by first 2 units of time. d2 = 1 means second job should be completed by first 1 unit of time. 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

Solution 3 is optimal. In this solution job 1 & 4 are produced 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 & that of job 1 is completed at time2

We can choose the objective function $\sum P_i$, $i \varepsilon J$ as one optimization measure. Using this measure, the next job to include is the one that increases $\sum P_i$, $i \varepsilon J$ the most, subject to the constraint that the resulting J is a feasible solution. This requires us to consider jobs in decreasing order of P_i's.

From the above example we begin with J=ø and $\sum P_i = 0$, $i \varepsilon J$. Jobs 1 is added to J as it has the largest profit and J={1} is a feasible solution. Next job 4 is considered. The solution J= {1,4} is also feasible. Next job 3 is considered and discarded as J={1,3,4} is not feasible. Hence we are left with the solution. J={1,4} with value 127. This is the optimal solution for the problem instance.

<u>High level description of Greedy Algorithm for Job Sequencing with deadlines:</u>

```
Algorithm GreedyJob  (d, J,n)
{
        // job 1, job 2, job 3,.... job n are in the non-increasing order of their profits
        J := { job1 };
        for (i =2 to n do
        {
                if all jobs in J  and job i  together can be completed by their deadlines, then
                    add job i to J;
        }
}
```

<u>Detailed Greedy Algorithm for Job Sequencing with deadlines:</u>

```
Algorithm   JS(d, j, s)
{
        //d[i]≥ 1, 1≤ i ≤ n are the dead lines, n ≥ 1. The jobs
        //are ordered such that  P[1]≥ p[2] ≥…≥p[n]. J[i]
        //is the i th job in the optimal solution , 1≤ i≤k.
        //Also, at termination d[J[i]] ≤d[J[i + 1]],1 ≤ 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 non-increasing order of p[i]: Find
                // Position for  i and check feasibility of insertion.
                r :=k;
                While (d[J[r]]>r]] > d[i]) and( d[J[r]] ≠ r) do
                        r : = r – 1;
                  if ((d[J[r]] ≤ 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;
        }
```

Minimum Spanning Tree

A tree is defined to be an undirected, acyclic and connected graph (or more simply, a graph in which there is only one path connecting each pair of vertices).

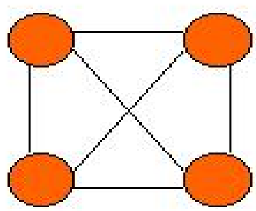
Assume there is an undirected, connected graph G. A spanning tree is a sub-graph of G, is a tree, and contains all the vertices of G. A minimum spanning tree is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum

Application of MST

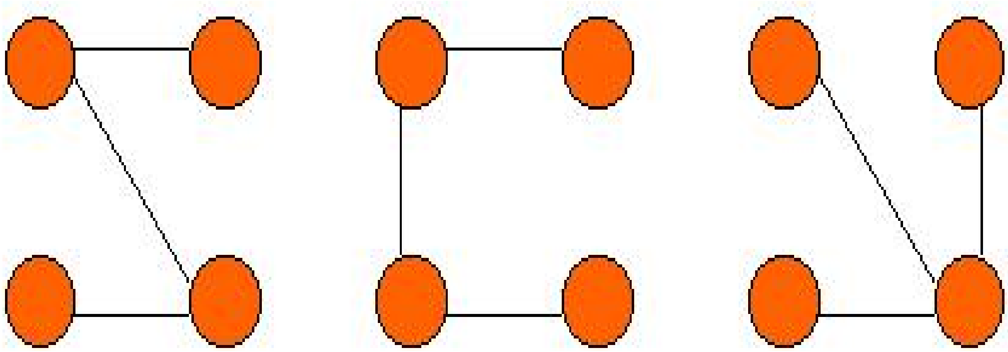
1) practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. MST can be used to determine the least costly paths with no cycles in this network, thereby connecting everyone at a minimum cost.

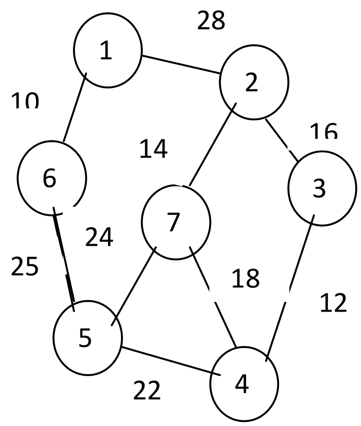
2) Another useful application of MST would be finding airline routes MST can be applied to optimize airline routes by finding the least costly paths with no cycles

G:

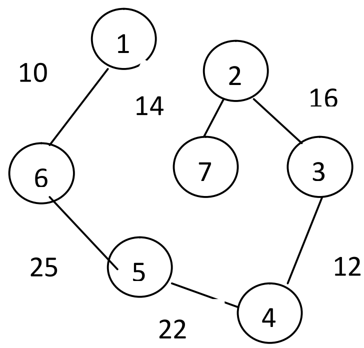


Three (of the many possible) spanning trees from graph G





(a)



(b)

Prim's Algorithm (DJP algorithm,the Jarník algorithm, or the Prim-Jarník algorithm).

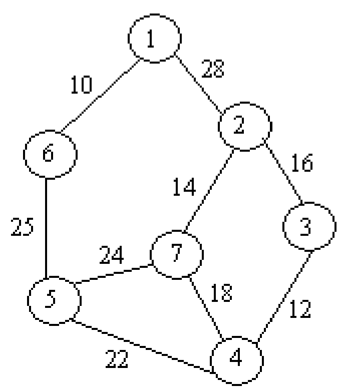
Prim's algorithm finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Steps

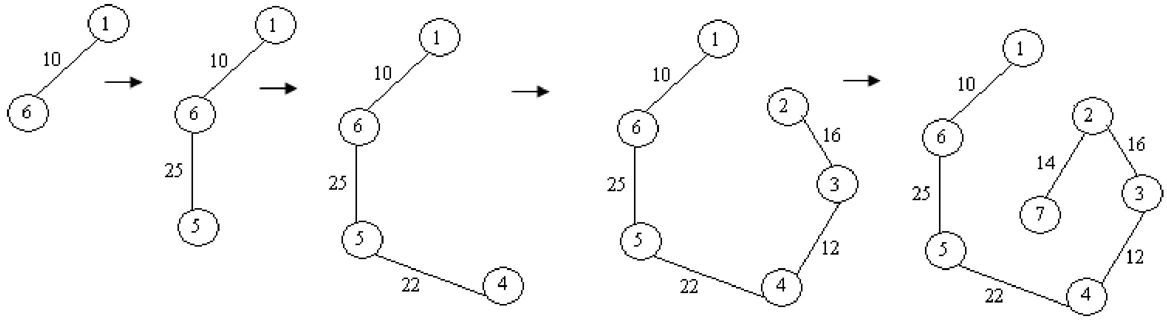
- Builds the tree edge by edge
- Next edge to be selected is one that result in a minimum increase in the sum of costs of the edges so far included
- Always verify that resultant is a tree

Ex:

Consider the connected graph given below



Minimum spanning tree using Prim's algorithm can be formed as follows.



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 ∞ if no edges(i, j) exists,
//A minimum spanning tree is computed and stored as 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] ≠ 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]≠0) and (cost[k,near[k]] > cost[k,j]))then
                                near[k]:=j;
        }
        return mincost;
}
```

Time complexity of the above algorithm is **O(n^2).**

Kruskal's Algorithm

Kruskal's algorithm is another algorithm that finds a minimum spanning tree for a connected weighted graph. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

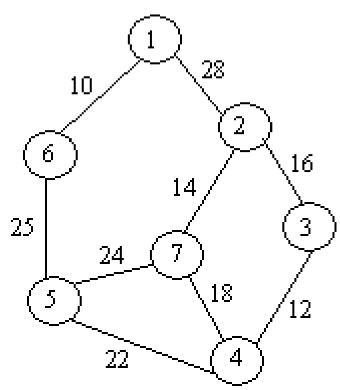
Kruskal's Algorithm builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm considers each edge in turn, order by increasing weight. If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree on the other hand, if an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded. The resultant may not be a tree in all stages. But can be completed into a tree at the end.

```
t = EMPTY;
while ((t has fewer than n-1 edges) && (E != EMPTY))
{
    choose an edge(v, w) from E of lowest cost;
    delete (v, w) from E;
    if (v, w) does not create a cycle in t
            add (v, w) to t;
    else
            discard (v, w);
}
```

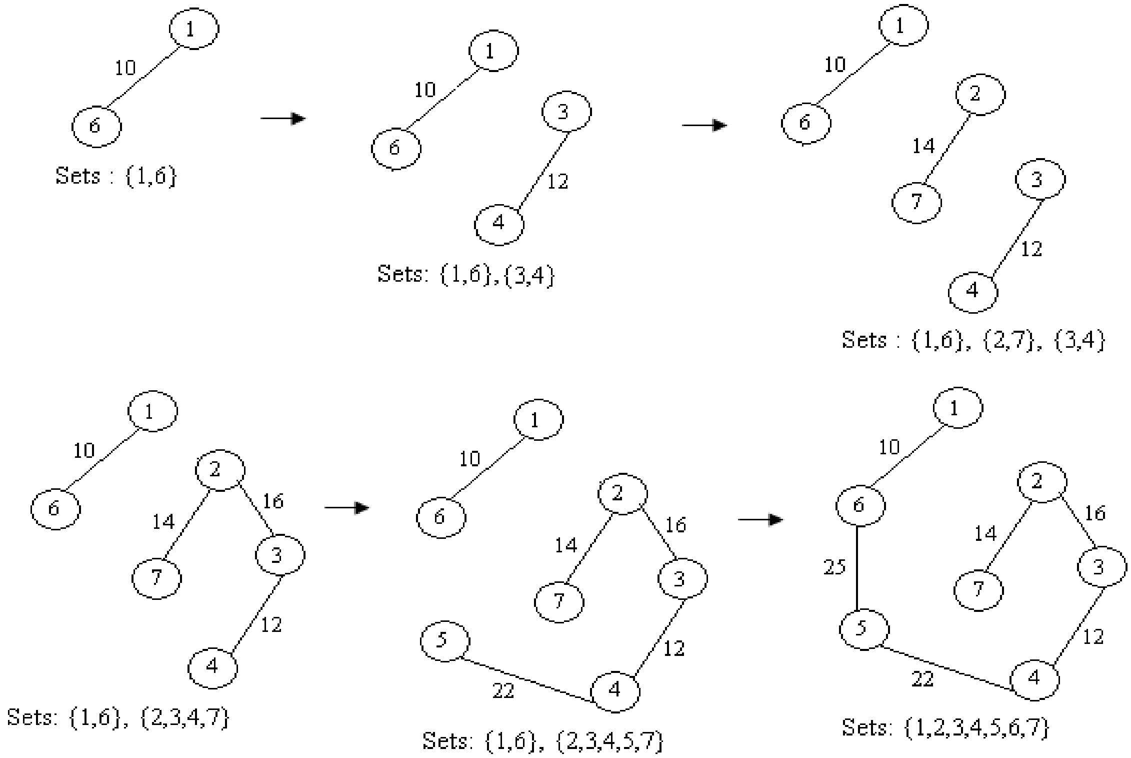
To check whether there exist a cycle, place all vertices in the same connected component of t into a set. Then two vertices v and w are connected in t then they are in the same set.

Example:

Consider the connected graph given below:



Minimum spanning tree using Kruskal's algorithm can be formed as given below.

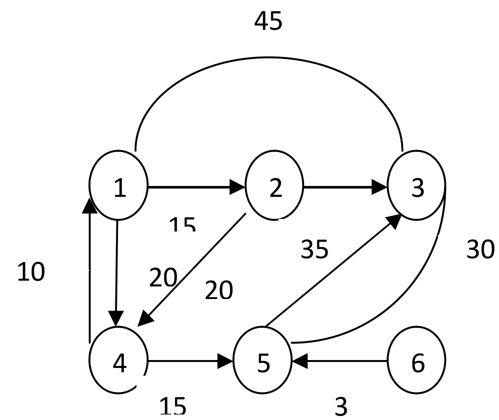


Kruskal's Algorithm

```
Float kruskal (int E[ ][ ], float cost[ ][ ], int n, int t[ ][2])
{
    int parent[w];
    consider heap out of edge cost;
    for (i=1; i<=n; i++)
            parent[i] = -1;            //Each vertex in different set
    i=0;
    mincost = 0;
    while((i<n-1) && (heap not empty))
    {
            Delete a minimum cost edge (u,v) from the heap and re heapify;
            j = Find(u);     k = Find(v);              // Find the set
            if (j != k)
            {
                    i++;
                    t[i][1] = u;
                    t[i][2] = v;
                    mincost += cost[u][v];
                    Union(j, k);
            }
            if (i != n-1)
                    printf("No spanning tree \n");
            else
                    return(mincost);
    }
}
```

Time complexity of the above algorithm is **O(n log n)**

Greedy Algorithm for Single-source shortest paths to all other vertices

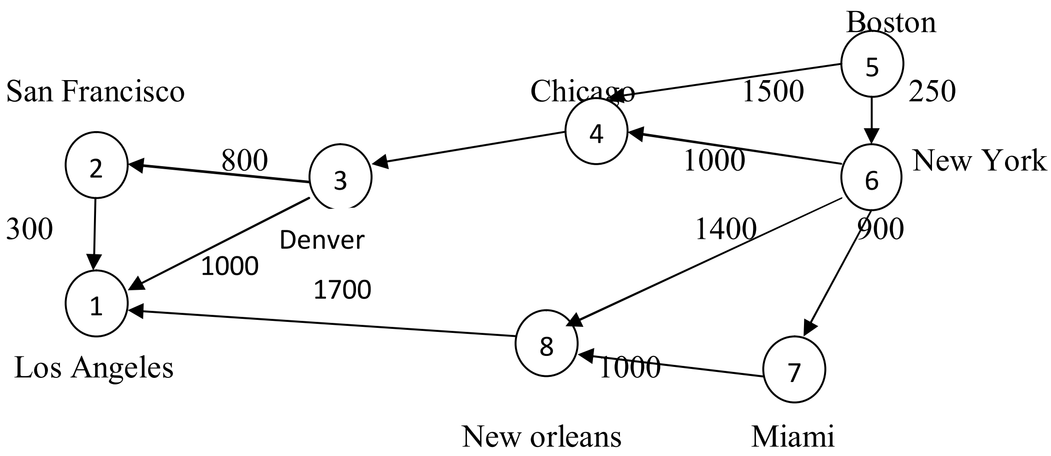


Path	Length
1) 1.4	10
2) 1.4.5	25
3) 1.4.5.2	45
4) 1.3	45

(b) Shortest path from 1

Greedy algorithm to generate shortest paths

```
Algorithm Shortest paths (v, cost ,dist, n)
// dist[j]. 1≤ j≤ 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 do
    {
      // Determine n - 1 paths from v.
        Choose u from among those vertices not
        in S such that dist[u] is minimum;
         S[n] :=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];
      }
}
```



(a) Digraph

	1	2	3	4	5	6	7	8
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

(a) Length – adjacency matrix