UNIT 1 GREEDY TECHNIQUES

Structure **Page Nos.**

1.0 INTRODUCTION

Greedy algorithms are typically used to solve an ***optimization problem.*** An Optimization problem is one in which we are given a set of input values, which are required to be either maximized or minimized w. r. t. some constraints or conditions. Generally an optimization problem has n inputs (call this set as **input domain** or **Candidate set**, C), we are required to obtain a subset of C (call it ***solution set***, S where S$\subseteq C$) that satisfies the given constraints or conditions. Any subset S $\subseteq C$, which satisfies the given constraints, is called a ***feasible*** solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that does this is called an **optimal solution**.

A greedy algorithm proceeds step–by-step, by considering one input at a time. At each stage, the decision is made regarding whether a particular input (say x) chosen gives an optimal solution or not. Our choice of selecting input x is being guided by the selection function (say ***select***). If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set. On the other hand, if the inclusion of that input x results in an infeasible solution, then this input x is not added to the partial solution. The input we tried and rejected is never considered again. When a greedy algorithm works correctly, the first solution found in this way is always optimal.
In brief, at each stage, the following activities are performed in greedy method:

1. First we select an element, say$x$, from input domain C.
2. Then we check whether the solution set S is feasible or not. That is we check whether x can be included into the solution set S or not. If yes, then solution set $S \leftarrow S \cup \{x\}$. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.
3. Continue until S is filled up (i.e. optimal solution found) or C is exhausted whichever is earlier.
   (Note: From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function (either maximize or minimize, as the case may be), is called ***optimal solution***.

Characteristics of greedy algorithm

- Used to solve optimization problem
- Most general, straightforward method to solve a problem.
- Easy to implement, and if exist, are efficient.
- Always makes the choice that looks best at the moment. That is, it makes a ***locally optimal choice in the hope that this choice will lead to a overall globally optimal solution.***
- Once any choice of input from C is rejected then it never considered again.
- Do not always yield an optimal solution; but for many problems they do.

In this unit, we will discuss those problems for which greedy algorithm gives an optimal solution such as Knapsack problem, Minimum cost spanning tree (MCST) problem and Single source shortest path problem.

1.1 OBJECTIVES

After going through this Unit, you will be able to:

- Understand the basic concept about Greedy approach to solve Optimization problem.

- Understand how Greedy method is applied to solve any optimization problem such as Knapsack problem, Minimum-spanning tree problem, Shortest path problem etc.

1.2 SOME EXAMPLES TO UNDERSTAND GREEDY TECHNIQUES

In order to better understand the greedy algorithms, let us consider some examples: Suppose we are given Indian currency notes of all denominations, e.g. {1,2,5,10,20,50,100,500,1000}. The **problem** is to ***find the minimum number of currency notes to make the required amount A, for payment***. Further, it is assumed that currency notes of each denomination are available in sufficient numbers, so that one may choose as many notes of the same denomination as are required for the purpose of using the minimum number of notes to make the amount A.

Now in the following examples we will notice that for a problem (discussed above) the greedy algorithm ***provides a solution*** (see example-1), some other cases, greedy algorithm does ***not provides a solution***, even when a solution by some other method exist (see example-2) and sometimes greedy algorithm does ***not provides an optimal solution*** Example-3).

Example 2

Solution: Intuitively, to begin with, we pick up a note of denomination D, satisfying the conditions.

i) $D \leq 289$ and
ii) if D1 is another denomination of a note such that $D1 \leq 289$, then $D1 \leq D$.

In other words, the picked-up note's denomination D is the largest among all the denominations satisfying condition (i) above.

The above-mentioned step of picking note of denomination D, satisfying the above two conditions, is repeated till either the amount of Rs.289/- is formed or we are clear that we can not make an amount or Rs.289/- out of the given denominations.

We apply the above-mentioned intuitive solution as follows:

To deliver Rs. 289 with minimum number of currency notes, the notes of different denominations are chosen and rejected as shown below:

Chosen-Note-Denomination	Total-Value-So far
100	0+100 ≤ 289
100	100+100= ≤ 289
~~100~~	~~200+100 > 289~~
50	200+50 ≤ 289
~~50~~	~~250+50 > 289~~
20	250 + 20 ≤ 289
~~20~~	~~270 + 20 > 289~~
10	270 + 10 ≤ 289
~~10~~	~~280 + 10 > 289~~
5	280 + 5 ≤ 289
~~5~~	~~285 + 5 > 289~~
2	285 + 2 < 289
2	287 + 2 = 289

The above sequence of steps based on Greedy technique, constitutes an algorithm to solve the problem.

To summarize, in the above mentioned solution, we have used the strategy of choosing, at any stage, the maximum denomination note, subject to the condition that the sum of the denominations of the chosen notes does not exceed the required amount A = 289.

The above strategy is the essence of greedy technique.

Example 2

Next, we consider an example in which for a given amount A and a set of available denominations, the greedy algorithm does not provide a solution, even when a solution by some other method exists.

Let us consider a hypothetical country in which notes available are of only the denominations 20, 30 and 50. We are required to collect an amount of 90.

Attempted solution through above-mentioned strategy of greedy technique:

i) First, pick up a note of denomination 50, because 50 ≤ 90. The amount obtained by adding denominations of all notes picked up so far is 50.

ii) Next, we can not pick up a note of denomination 50 again. However, if we pick up another note of denomination 50, then the amount of the picked-up notes becomes 100, which is greater than 90. Therefore, we do not pick up any note of denomination 50 or above.

iii) Therefore, we pick up a note of next denomination, viz., of 30. The amount made up by the sum of the denominations 50 and 30 is 80, which is less then 90. Therefore, we accept a note of denomination 30.

iv) Again, we can not pick up another note of denomination 30, because otherwise the sum of denominations of picked up notes, becomes 80+30=110, which is more than 90. Therefore, we do not pick up only note of denomination 30 or above.

v) Next, we attempt to pick up a note of next denomination, viz., 20. But, in that case the sum of the denomination of the picked up notes becomes 80+20=100, which is again greater than 90. Therefore, we do not pick up only note of denomination 20 or above.

vi) Next, we attempt to pick up a note of still next lesser denomination. However, there are no more lesser denominations available.

Hence greedy algorithm fails to deliver a solution to the problem.

However, by some other technique, we have the following solution to the problem: First pick up a note of denomination 50 then two notes each of denomination 20.

Thus, we get 90 and it can be easily seen that at least 3 notes are required to make an amount of 90. Another alternative solution is to pick up 3 notes each of denomination 30.

Example 3

Next, we consider an example in which the greedy technique, of course, leads to a solution, but the solution yielded by greedy technique is not optimal.

Again, we consider a hypothetical country in which notes available are of the only denominations 10, 40 and 60. We are required to collect an amount of 80.

Using the greedy technique, to make an amount of 80, first, we use a note of denomination 60. For the remaining amount of 20, we can choose note of only denomination 10. And , finally, for the remaining amount, we choose another note of denomination 10. Thus, greedy technique suggests the following solution using 3 notes: 80 = 60 + 10 + 10.

However, the following solution uses only two notes:

80 = 40 + 40

Thus, the solutions suggested by Greedy technique may not be optimal.

1.3 FORMALIZATION OF GREEY TECHNIQUE

In order to solve optimization problem using greedy technique, we need the following data structures and functions:

1) A candidate set from which a solution is created. It may be set of nodes, edges in a graph etc. call this set as:
   C: Set of given values or set of candidates

2) A solution set S (where $S \subseteq C$) , in which we build up a solution. This structure contains those candidate values, which are considered and chosen by the greedy technique to reach a solution. Call this set as:
   S: Set of selected candidates (or input) which is used to give optimal solution.

3) A function (say ***solution***) to test whether a given set of candidates give a solution (not necessarily optimal).

4) A selection function (say ***select***) which chooses the best candidate form C to be added to the solution set S,

5) A function (say ***feasible***) to test if a set S can be **extended** to a solution (not necessarily optimal) and

6) An objective function (say **ObjF**) which assigns a ***value*** to a solution, or a partial solution.

To better understanding of all above mentioned data structure and functions, consider the minimum number of notes problem of example1. In that problem:

1) C={1, 2, 5, 10,50,100,500,1000}, which is a list of available notes (in rupees). Here the set C is a multi-set, rather than set, where the values are repeated.

2) Suppose we want to collect an amount of Rs. 283 (with minimum no. of notes). If we allow a multi-set rather than set in the sense that values may be repeated, then S={100,100,50,20,10,2,1}

3) A function **solution** checks whether a solution is reached or not. However this function does not check for the optimality of the obtained solution. In case of minimum number of notes problem, the function ***solution*** finds the sum of all values in the multi-set S and compares with the fixed amount, say Rs. 283. If at any stage S={100,100, 50}, then sum of the values in the S is 250, which does not equal to the 283, then the function ***solution*** returns “solution not reached”. However, at the later stage, when S={100,100,50,20,10,2,1}, then the sum of values in S equals to the required amount, hence the function ***solution*** returns the message of the form “solution reached”.

4) A function ***select*** finds the “best” candidate value (say x) from C, then this value x is tried to add to the set S. At any stage, value x is added to the set S, if its addition leads to a partial (feasible) solution. Otherwise, x is rejected. For example, In case of minimum number of notes problem, for collecting Rs. 283, at the stage when S={100, 100,50}, then first the function ***select*** try to add the Rs 50 to S. But by using a function **solution**, we can found that the addition of Rs. 50 to S will lead us a infeasible solution, since the total value now becomes 300 which exceeds Rs. 283. So the value 50 is rejected. Next, the function ***select*** attempts the next lower denomination 20. The value 20 is added to the set S, since after adding 20, total sum in S is 270, which is less than Rs. 283. Hence, the value 20 is returned by the function ***select***.

5) When we select a new value (say x) using ***select*** function from set C, then before adding x to S we check its feasibility. If its addition gives a partial solution, then this value is added to S. Otherwise it is rejected. The feasibility checking of new selected value is done by the function ***feasible***. For example, In case of minimum number of notes problem, for collecting Rs. 283, at the stage when S={100, 100,50}, then first the function ***select*** try to add the Rs 50 to S. But by using a function **solution**, we can found that the addition of Rs. 50 to S will lead us an infeasible solution, since the total value now becomes 300 which exceeds Rs. 283. So the value 50 is rejected. Next, the function ***select*** attempts the next lower denomination 20. The value 20 is added to the set S, since after adding 20, total sum in S is 270, which is less than Rs. 283. Hence feasible.

6) The objective function (say ***ObjF***), gives the value of the solution. For example, In case of minimum number of notes problem, for collecting Rs. 283; and when S={100,100,50,20,10,2,1}, then the sum of values in S equals to the required amount 283; the function ***ObjF*** returns the number of notes in S, i.e., the number 7.

A general form for greedy technique can be illustrated as:

```
Algorithm Greedy(C, n)

   /* Input: A input domain (or Candidate set ) C of size n, from which solution is to be
          Obtained. */

   // function select (C: candidate_set)  return an element (or candidate).
  // function solution (S: candidate_set) return Boolean
  // function feasible (S: candidate_set) return Boolean
   /* Output: A solution set S, where S ⊆ $C$, which maximize or minimize the selection
           criteria w. r. t. given constraints */

{

        $S \leftarrow \phi$                      // Initially a solution set S is empty.
       While ( not solution(S) and $C \neq \phi$)
         {
           $x \leftarrow \boldsymbol{select}(C)$          /* A "best" element x is selected from C which
                                       maximize or minimize the selection criteria. */
           $C \leftarrow \boldsymbol{C} - \{\boldsymbol{x}\}$                  /* once x is selected , it is removed from C
           if ( feasible($S \cup \{x\}$)  then    /* x is now checked for feasibility
              $S \leftarrow S \cup \{x\}$
         }
       If (solution (S))
           return S;
        else
          return " No Solution"
    } // end of while
```

Now in the following sections, we apply greedy method to solve some optimization problem such as knapsack (fractional) problem, Minimum Spanning tree and Single source shortest path problem etc.

1.4 KNAPSACK (FRACTIONAL) PROBLEM

The fractional knapsack problem is defined as:

- Given a list of n objects say $\{I_1, I_2 \dots \dots, I_n\}$ and a Knapsack (or bag).
- Capacity of Knapsack is M.
- Each object I_i has a weight w_i and a profit of p_i .
- If a fraction x_i (where $x_i \in \{0, \dots .,1\}$) of an object I_i is placed into a knapsack then a profit of $p_i x_i$ is earned.

The **problem** (or Objective) is to fill a knapsack (up to its maximum capacity M) which maximizes the total profit earned.

Mathematically:

$$Maximize\ (the\ profit) \sum_{i=1}^{n} p_i x_i \ \ ; \ \ subjected\ to\ the\ constraints$$

$$\sum_{i=1}^{n} w_i x_i \ \leq \text{M and } \ x_i \in \{0, \dots ,1\}, 1 \leq \text{i} \leq \text{n}$$

Note that the value of x_i will be any value between 0 and 1 (inclusive). If any object I_i is completely placed into a knapsack then its value is 1 $(i.e. x_i = 1)$, if we do not pick (or select) that object to fill into a knapsack then its value is 0 $(i.e. x_i = 0)$. Otherwise if we take a fraction of any object then its value will be any value between 0 and 1.

To understand this problem, consider the following instance of a knapsack problem:

$Number\ of\ objects;\ n = 3$
Capacity of Knapsack; M=20
$(p_1, p_2, p_3) = (25,24,15)$
$(w_1, w_2, w_3) = (18,15,10)$

To solve this problem, Greedy method may apply any one of the following strategies:

- From the remaining objects, select the object with maximum profit that fit into the knapsack.
- From the remaining objects, select the object that has minimum weight and also fits into knapsack.
- From the remaining objects, select the object with maximum p_i/w_i that fits into the knapsack.

Let us apply all above 3 approaches on the given knapsack instance:

Approach	(x_1, x_2, x_3)	$\sum_{i=1}^{3} w_i x_i$	$\sum_{i=1}^{3} p_i x_i$
1	$(1, \frac{2}{15}, 0)$	18+2+0=20	28.2
2	$< 0, \frac{2}{3}, 1 >$	0+10+10=20	31.0
3	$< 0, 1, \frac{1}{2} >$	0+15+5=20	31.5

Approach 1: (selection of object in decreasing order of profit):

In this approach, we select those object first which has maximum profit, then next maximum profit and so on. Thus we select 1st object (since its profit is 25, which is maximum among all profits) first to fill into a knapsack, now after filling this object ($w_1 = 18$) into knapsack remaining capacity is now 2 (i.e. 20-18=2). Next we select the 2nd object, but its weight w_2=15, so we take a fraction of this object (i.e. $x_2 = \frac{2}{15}$). Now knapsack is full (i.e. $\sum_{i=1}^{3} w_i x_i = 20$) so 3rd object is not selected. Hence we get total profit $\sum_{i=1}^{3} p_i x_i$ = 28 units and the solution set $(x_1, x_2, x_3) = (1, \frac{2}{15}, 0)$

Approach 2: (Selection of object in increasing order of weights).
In this approach, we select those object first which has minimum weight, then next minimum weight and so on. Thus we select objects in the sequence 2nd then 3rd then 1st. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i$ = 31.0 units and the solution set $(x_1, x_2, x_3) = (0, \frac{2}{3}, 1)$.

Approach 3: (Selection of object in decreasing order of the ratio p_i/w_i).
In this approach, we select those object first which has maximum value of p_i/w_i , that is we select those object first which has maximum profit per unit weight .
Since (p_1/w_1 , p_2/w_2 , p_3/w_3)=(1.3, 1.6, 1.5). Thus we select 2nd object first , then 3rd object then 1st object. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i$ = 31.5 units and the solution set $(x_1, x_2, x_3) = (0, 1, \frac{1}{2},)$.

Thus from above all 3 approaches, it may be noticed that

- Greedy approaches **do not always yield an optimal solution**. In such cases the greedy method is frequently the basis of a heuristic approach.
- Approach3 **(S**election of object in decreasing order of the ratio p_i/w_i) gives a optimal solution for knapsack problem.

A pseudo-code for solving knapsack problem using greedy approach is :

```
          Greedy Fractional-Knapsack (P[1..n], W[1..n], X [1..n], M)
                /* P[1..n] and W[1..n] contains the profit and weight of the n-objects ordered such
                that
                X[1..n] is a solution set and M is the capacity of KnapSack*/
          {
1:              For i ← 1 to n do
2:                      X[i] ← 0
3:                      profit ← 0        //Total profit of item filled in Knapsack
4:                      weight ← 0        // Total weight of items packed in KnapSack
5:                      i←1
6:              While (Weight < M) // M is the Knapsack Capacity
                {
7:                      if (weight + W[i] ≤ M)
8:                              X[i] = 1
9:                              weight = weight + W[i]
10:                     else
11:                             X[i] = (M-wright)/w[i]
12:                             weight = M
13:             Profit = profit = profit + p [i]*X[i]
14:             i++;
          }//end of while
          }//end of Algorithm
```

Running time of Knapsack (fractional) problem:

Sorting of n items (or objects) in decreasing order of the ratio p_i/w_i takes $O(nlogn)\ time$. Since this is the lower bound for any comparison based sorting algorithm. Line 6 of ***Greedy Fractional-Knapsack*** takes $O(n)$ time. Therefore, the total time including sort is $O(nlogn)$.

Example: 1: Find an optimal solution for the knapsack instance n=7 and M=15 ,

$(p_1, p_2 \ldots\ldots, p_7) = (10, 5, 15, 7, 6, 18, 3)$
$(w_1, w_2, \ldots., w_7) = (2, 3, 5, 7, 1, 4, 1)$

Solution:

Greedy algorithm gives a optimal solution for knapsack problem if you select the object in decreasing order of the ratio p_i/w_i . That is we select those object first which has maximum value of the ratio p_i/w_i $; for\ all\ i = 1, .., 7$. This ratio is also called profit per unit weight .

Since $\left(\frac{p_1}{w_1}, \frac{p_2}{w_2}, \ldots, \frac{p_7}{w_7}\right)$ = (5,1.67,3,1,6,4.5,3). Thus we select 5th object first , then 1st object, then 3rd (or 7th) object, and so on.

Approach	$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\sum_{i=1}^{7} w_i x_i$	$\sum_{i=1}^{7} p_i x_i$
Selection of object in decreasing order of the ratio p_i/w_i	$(1, \frac{2}{3}, 1, 0, 1, 1, 1)$	1+2+4+5+1+2 =15	6+10+18+15+3+3.33 =55.33

1.5 MINIMUM COST SPANNING TREE (MCST) PROBLEM

Definition: **(Spanning tree):** Let G=(V,E) be an undirected connected graph. A ***subgraph*** T=(V,E') of G is a spanning tree of G if and only if T is a tree (i.e. no cycle exist in T) and contains **all the vertices** of G.

Definition: **(Minimum cost Spanning tree):**

Suppose G is a **weighted connected graph**. A ***weighted graph*** is one in which every edge of G is assigned some positive weight (or length). A graph G is having several spanning tree.

In general, a *complete graph* (each vertex in G is connected to every other vertices) with n vertices has total n^{n-2} spanning tree. For example, if n=4 then total number of spanning tree is 16.
A ***minimum cost spanning tree*** (MCST) of a weighted connected graph G is that spanning tree whose sum of length (or weight) of all its edges is minimum, among all the possible spanning tree of G.

For example: consider the following **weighted connected graph** G (as shown in figure-1). There are so many spanning trees (say $T_1, T_2, T_3, \dots \dots$) are possible for G. Out of all possible spanning trees, four spanning trees T_1, T_2, T_3 and $, T_4$ of G are shown in figure a to figure d.

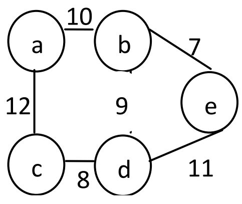
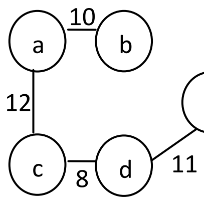
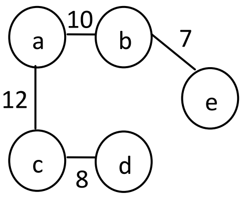


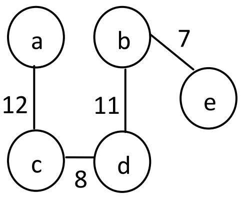
Figure-1



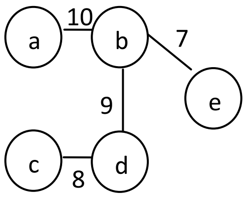
(a)



(b)



(c)



(d)

A sum of the weights of the edges in T_1, T_2, T_3 and $, T_4$ is: 41, 37, 38 and 34 (some other spanning trees $T_5, T_6, \dots$ are also possible). We are interested to find that spanning tree, out of all possible spanning trees $T_1, T_2, T_3, T_4, T_5 \dots\dots$; whose sum of weights of all its edges are minimum. For a given graph G , T_3 is the MCST, since weight of all its edges is minimum among all possible spanning trees of G.

<u>Application of spanning tree</u>:

- ***Spanning trees are widely used in designing an efficient network***.
  For example, suppose we are asked to design a network in which a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. This problem can be converted into a graph problem in which nodes are telephones (or computers), undirected edges are potential links. The goal is to pick enough of these edges that the nodes are connected. Each link (edge) also has a maintenance cost, reflected in that edge's weight. Now question is "what is the cheapest possible network? An answer to this question is MCST, which connects everyone at a minimum possible cost.
- Another application of MCST is in the **designing of efficient routing algorithm**.
  Suppose we want to find a airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, when we travel more, the more it will cost. So MCST can be applied to optimize airline routes by finding the least costly paths with no cycle.

To find a MCST of a given graph G, one of the following algorithms is used:

1. **Kruskal's algorithm**
2. **Prim's algorithm**

These two algorithms use Greedy approach. A greedy algorithm selects the edges one-by-one in some given order. The next edge to include is chosen according to some optimization criteria. The simplest such criteria would be to choose an edge (u, v) that results in a minimum increase in the sum of the costs (or weights) of the edges so for included.

In General for constructing a MCST:

- We will build a set *A* of edges that is always a subset of some MCST.
- Initially, *A* has no edges (i.e. empty set).
- At each step, an edge (u, v) is determined such that $A \cup \{(u, v)\}$ is also a subset of a MCST. This edge *(u, v)* is called a ***safe edge.***
- At each step, we always add only ***safe edges*** to set A.
- **Termination:** when all *safe* edges are added to *A*, we stop. Now *A* contains a edges of spanning tree that is also an MCST.

Thus a general MCST algorithm is:

```
GENERIC_MCST(G, w)
   {
     $A \leftarrow \phi$
     While A is not a spanning tree
       {
          find  an edge (u, v) that is safe for A
          $A \leftarrow A \cup \{(u,v)\}$
        }
     return A
   }
```

A main ***difference*** between ***kruskal's*** and ***Prim's*** algorithm to solve MCST problem is that the order in which the edges are selected.

Kruskal's Algorithm	Prim's algorithm
• Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST. • In kruskal's algorithm for getting MCST, it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there are may be more than one connected components are possible. • Time complexity: $O(\|E\|log\|V\|)$	• Prim's algorithm always selects a vertex (say, v) to find MCST. • In Prim's algorithm for getting MCST, it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there will be only one connected components are possible • Time complexity: $O(\|V\|^2)$

For solving MCST problem using Greedy algorithm, we use the following data structure and functions, as mentioned earlier:

i) **C: The set of candidates (or given values):** Here C=E, the set of edges of $G(V, E)$.
ii) **S: Set of selected candidates (or input) which is used to give optimal solution.** Here the subset of edges, E' $(i.e. E' \subseteq E)$ is a solution, if the graph $T(V, E')$ is a spanning tree of $G(V, E)$.
iii) In case of MCST problem, the function ***Solution*** checks whether a solution is reached or not. This function basically checks :
  a) All the edges in S form a tree.
  b) The set of vertices of the edges in S equal to V.
  c) The sum of the weights of the edges in S is minimum possible of the edges which satisfy (a) and (b) above.

1) ff selection function (say ***select***) which chooses the best candidate form C to be added to the solution set S,

iv) The ***select*** function chooses the best candidate from C. In case of **Kruskal's algorithm**, it selects an edge, whose **length is smallest** (from the remaining candidates). But in case of Prim's algorithm, it select a vertex, which is added to the already selected vertices, to minimize the cost of the spanning tree.

v) A function ***feasible*** checks the feasibility of the newly selected candidate (i.e. edge (u,v)). It checks whether a newly selected edge (u, v) form a cycle with the earlier selected edges. If answer is "yes" then the edge (u,v) is rejected, otherwise an edge (u,v) is added to the solution set S.

vi) Here the objective function **ObjF** gives the **sum of the edge lengths** in a **Solution.**

1.5.1 Kruskal's Algorithm

Let $G(V,E)$ is a connected, weighted graph.

Kruskal's algorithm finds a minimum-cost spanning tree (MCST) of a given graph G. It uses a ***greedy approach*** to find MCST, because at each step it adds an edge of least possible weight to the set A. In this algorithm:

- First we examine the edges of G in order of increasing weight.
- Then we select an edge $(u, v) \in E$ of minimum weight and checks whether its end points belongs to same component or different connected components.
- If u and v belongs to different connected components then we add it to set A, otherwise it is rejected because it create a cycle.
- The algorithm stops, when only one connected components remains (i.e. all the vertices of G have been reached).

Following pseudo-code is used to constructing a MCST, using Kruskal's algorithm:

```
KRUSKAL_MCST(G, w)
    /* Input: A undirected connected weighted graph G=(V,E).
    /* Output:  A minimum cost spanning tree T(V, E') of G
  {
 1.   Sort the edges of E in order of increasing weight
 2.    $A \leftarrow \phi$
 3.    for (each vertex $v \in V[G]$)
 4.        do MAKE_SET(v)
 5.   for (each edge $(u, v) \in E$, taken in increasing order of weight
          {
 6.          $if(\boldsymbol{FIND\_SET}(u) \neq \boldsymbol{FIND\_SET}(v))$
 7.            $A \leftarrow A \cup \{(u,v)\}$
 8.            $\boldsymbol{MERGE}(u,v)$
          }
 9.    return $A$
  }
```

Kruskal's algorithm works as follows:

- First, we sorts the edges of E in order of increasing weight
- We build a set A of edges that contains the edges of the MCST. Initially A is empty.
- At line 3-4, the function **MAKE_SET(v),** make a new set {v} for all vertices of G. For a graph with n vertices, it makes n components of disjoint set such as {1},{2},… and so on.

- In line 5-8: An edge *(u, v)*$\in E$, of minimum weight is added to the set A, if and only if it joins two nodes which belongs to ***different*** components (to check this we use a $\boldsymbol{FIND\_SET}(\ )$ function, which returns a same integer value, if u and v belongs to same components (In this case adding (u,v) to A creates a cycle), otherwise it returns a different integer value)
- If an edge added to A then the two components containing its end points are merged into a single component.
- Finally the algorithm stops, when there is just a single component.

Analysis of Kruskal's algorithm:

Let $|V| = n, the\ number\ of\ vertices$ and

$|E| = a, the\ number\ of Edges$

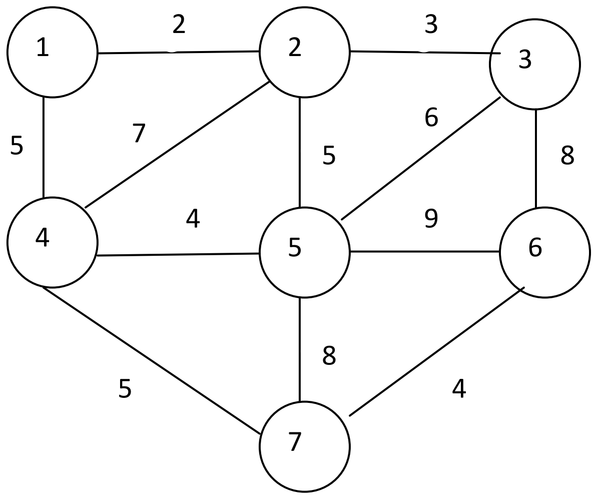
1. Sorting of edges requires $O(|E|log|E|) = O(aloga)\ time.$
2. Since, in any graph, minimum number of edges is $(n-1)$ and maximum number of edges (when graph is complete) is $n(n-1)/2$. Hence $n-1 \leq a \leq n(n-1)/2$. Thus $O(aloga) = O(alog\,(n(n-1)/2)) = O(alogn)$
3. Initializing n-disjoint set (in line 3-4) using **MAKE_SET** will requires O(n) time.
4. There are at most $2a$ **FIND_SET** operations (since there are a edges and each edge has 2 vertices) and $(n-1)$ **MERGE** operations. Thus we requires $O((2a + (n-1)logn)$ time.
5. At worst, O(a) time for the remaining operations.
6. For a connected graph, we know that $\boldsymbol{a} \geq (\boldsymbol{n}-\boldsymbol{1})$**.** So the total time for Kruskal's algorithm is $O((2a + (n-1)logn) = O(alogn) = \boldsymbol{O}(|\boldsymbol{E}|\ \mathbf{log}|\boldsymbol{V}|)$

Example: Apply Kruskal's algorithm on the following graph to find minimum-cost-spanning –

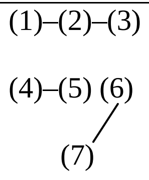
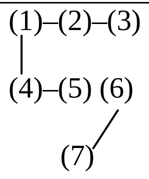
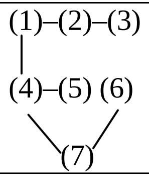
tree (MCST).

Solution: First, we sorts the edges of G=(V,E) in order of increasing weights as:

Edges	(1,2)	(2,3)	(4,5)	(6,7)	(1,4)	(2,5)	(4,7)	(3,5)	(2,4)	(3,6)	(5,7)	(5,6)
weights	2	3	4	4	5	5	5	6	7	8	8	9



The kruskal's Algorithm proceeds as follows:

STEP	EDGE CONSIDERED	CONNECTED COMPONENTS	SPANNING FORESTS (A)
Initialization	___	{1}{2}{3}{4}{5}{6}{7} (using line 3-4)	(1) (2) (3) (4) (5) (6) (7)
1.	(1, 2)	{1, 2},{3},{4},{5},{6},{7}	(1) – (2) (3) (4) (5) (6) (7)
2.	(2, 3)	{1,2,3},{4},{5},{6},{7}	(1)–(2)–(3) (4) (5) (6) (7)
3.	(4, 5)	{1,2,3},{4,5},{6},{7}	(1)–(2)–(3) (6) (7) (4)–(5)
4.	(6, 7)	{1,2,3},{4,5},{6,7}	
5.	(1, 4)	{1,2,3,4,5},{6,7}	
6.	(2, 5)	Edge (2,5) is rejected, because its end point belongs to same connected component, so create a cycle.	
7.	(4, 7)	{1,2,3,4,5,6,7}	

Total Cost of Spanning tree, T = 2+3+5+4+5+4=23

1.5.2 Prim's Algorithm

PRIM's algorithm has the property that the edges in the set A (this set A contains the edges of the minimum spanning tree, when algorithm proceed step-by step) always form a single tree, i.e. at each step we have only one connected component.

- We begin with one starting vertex (say v) of a given graph G(V,E).
- Then, in each iteration, we choose a minimum weight edge *(u, v)* connecting a vertex *v* in the set A to the vertices in the set$(V - A)$. That is, we always find an edge *(u, v)* of minimum weight such that $v \in A$ and $u \in$ *V-A*. Then we modify the set A by adding u i.e.$A \leftarrow A \cup \{u\}$
- This process is repeated until $\neq V$, i.e. until all the vertices are not in the set A.

Following pseudo-code is used to constructing a MCST, using PRIM's algorithm:

```
PRIMS_MCST(G, w)
    /* Input: A undirected connected weighted graph G=(V,E).
    /* Output:  A minimum cost spanning tree T(V, E') of G
{
1.   $T \leftarrow \phi$          // T contains the edges of the MST
2.   $A \leftarrow \{$ Any arbitrary menber of V $\}$
3.   while $(A \neq V)$
     {
4.       find an edge $(u, v)$ of minimum weight s.t. $u \in V-A$ and $v \in A$
5.         $A \leftarrow A \cup \{(u, v)\}$
6.         $B \leftarrow B \cup \{u\}$
     }
7.   return $T$
}
```

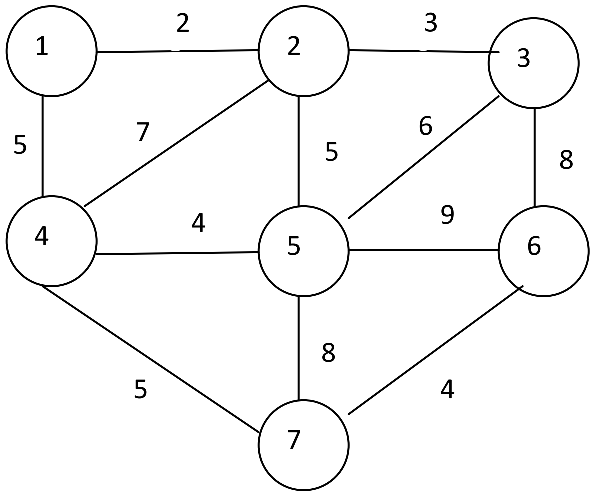
PRIM's algorithm works as follows:

1) Initially the set A of nodes contains a single arbitrary node (i.e. starting vertex) and the set T of edges are empty.
2) At each step PRIM's algorithm looks for the shortest possible edge (u, v) such that
   u*∈ *V-A *and* ***v*∈*A***
3) In this way the edges in T form at any instance a minimal spanning tree for the nodes in A. We repeat this process until $\boldsymbol{A \neq V}$**.**

Time complexity of PRIM's algorithm

Running time of PRIM's algorithm can be calculated as follows:
- While loop at line-3 is repeated $|V| - 1 = (n - 1)$ times.
- For each iteration of while loop, the inside statements will require $O(n)$ time.
- So the overall time complexity is$O(n^2)$.

Example2: Apply PRIM’s algorithm on the following graph to find minimum-cost-spanning – tree (MCST).



Solution: In PRIM’s, First we select an arbitrary member of V as a starting vertex (say 1), then the algorithm proceeds as follows:

STEP	EDGE CONSIDERED (4, v)	CONNECTED COMPONENTS (Set A)	SPANNING FORESTS (set T)
Initialization	__	{1}	(1)
1	(1,2)	{1,2}	(1)−(2)
2	(2,3)	{1,2,3}	(1)−(2)−(3)
3	(1,4)	{1,2,3,4}	(1)−(2)−(3) | (4)
4.	(4,5)	{1,2,3,4,5}	(1)−(2)−(3) | (4)−(5)
5	(4,7)	{1,2,3,4,5,7}	(1)−(2)−(3) | (4)−(5) \\ (7)
6	(6,7)	{1,2,3,4,5,6,7}	(1)−(2)−(3) | (4)−(5) (6) \\ / (7)

Total Cost of the minimum spanning tree = 2+3+5+4+5+4
= 23

☞ Check Your Progress 1

Choose correct options from Q.1 to Q.7

Q.1: The essence of greedy algorithm is the ………….. policy.
a) Maximization b) Minimization c) selection d) either a) or b)

Q2: The running time of KRUSKAL's **algorithm**, where |E| is the number of edges and |V| is the number of nodes in a graph:
a) $O(|E|)$ b) $O(|E|\log|E|)$ c) $O(|E|\log|V|)$ d) $O(|V|\log|V|)$

Q3: The running time of PRIM's algorithm, where |E| is the number of edges and |V| is the number of nodes in a graph:
a) $O(|E|^2)$ b) $O(|V|^2)$ c) $O(|E|\log|V|)$ d) $O(|V|\log|V|)$

Q.4: The optimal solution to the knapsack instance n=3, M=15, $(P_1,P_2,P_3)=(25,24,15)$ and $(W_1,W_2,W_3)=(18,15,10)$ is:
a) 28.2 b) 31.0 c) 31.5 d) 41.5

Q5: The solution set $X = (X_1, X_2, X_3)$ for the problem given in Q.4 is
a) $(\frac{1}{15},\frac{2}{15},0)$ b) $(0,\frac{2}{3},1)$ c) $(0,1,\frac{1}{2})$ d) None of these

Q.6: Total number of spanning tree in a complete graph with 5 nodes are
a) 5^2 b) 5^3 c) 10 d) 100

Q.7: Let (i, j, C), where i and j indicates vertices of a graph & C denotes cost between edges. Consider the following edges & cost in order of increasing length: (b,e,3),(a,c,4),(e,f,4), (b,c,5),(f,g,5),(a,b,6), (c,d,,6),(e,f,6), (b,d,7), (d,e,7),(d,f,7),(c,f,7). Which of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?

a) $(b,e),(e,f),(a,c),(b,c),(f,g),(c,d)$

b) $(b,e),(e,f),(a,c),(f,g),(b,c)(c,d)$

c) $(b,e),(a,c),(e,f),(b,c),(f,g),(c,d)$

d) $(b,e),(e,f),(b,c),(a,c),(f,g),(c,d)$

Q.8: Consider a question given in Q.7. Applying Kruskal's algorithm to find total cost of a Minimum spanning tree.

Q.9: State whether the following Statements are TRUE or FALSE. Justify your answer:

a) If e is a minimum edge weight in a connected weighted graph , it must be among the edges of at least one minimum spanning tree of the graph.

b) If e is a minimum edge weight in a connected weighted graph , it must be among the edges of each one minimum spanning tree of the graph.

c) If edge weights of a connected weighted graph are all distinct, the graph must have exactly one minimum spanning tree.

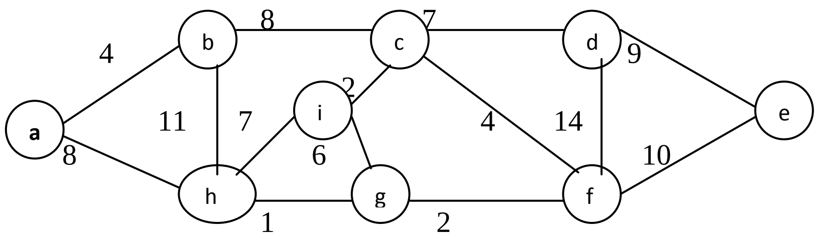
d) If edge weights of a connected weighted graph are not all distinct, the graph must have more than one minimum spanning tree.

e) If edge weights of a connected weighted graph are not all distinct, the minimum cost of each one minimum spanning tree is same.

Q.10: What is "*Greedy algorithm*" ? Write its pseudo code

Q.11: Differentiate between Kruskal's and Prim's algorithm to find a Minimum cost of a spanning tree of a graph G..

Q.12: Are the Minimum spanning tree of any graph is unique? Apply PRIM's algorithm to find a minimum cost spanning tree for the following.
Graph following using Prim's Algorithm. (a is a starting vertex).



Q.13: Find the optimal solution to the knapsack instance n=5, M=10, $(P_1, P_2, \ldots, P_5) = (12,32,40,30,50)$ $(W_1, W_2, \ldots, W_5) = (4,8,2,6,1)$.

Q.14: Let S={a, b, c, d, e, f, g} be a collection of objects with Profit-Weight values as follows: a:(12,4), b:(10,6), c:(8,5), d:(11,7), e:(14,3), f:(7,1) and g:(9,6). What is the optimal solution to the *fractional* knapsack problem for S, assuming we have a knapsack that can hold objects with total weight 18? What is the *complexity* of this method.

1.6 SINGLE SOURCE SHORTEST PATH PROBLEM (SSSPP)

Given: A directed graph $G(V, E)$ with weight edge $w(u, v)$.

- We define the **weight of path** $p = < v_0, v_1, \ldots, v_k >$ as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) = sum\ of\ edge\ weights\ on\ path\ p.$$

- We can define the **shortest-path weight** from u to v as:

$$\delta(u, v) = \begin{cases} \min\{w(p): u \rightsquigarrow v\}\ ; & if\ there\ exist\ a\ path\ u \rightsquigarrow v. \\ \infty\ ; & otherwise \end{cases}$$

Single-source-shortest path problem (SSSPP) problem:
Given a directed graph $G(V, E)$ with weight edge $w(u, v)$. We have to find a shortest path from source vertex $s \in V$ to every other vertex $v \in V - s$.

SSSP Problem can also be used to solve some other related problems:

- **Single-destination shortest path** problem(**SDSPP**): Find the transpose graph (i.e. reverse the edge directions) and use **single-source-shortest** path.

- **Single-pair shortest path (i.e.** a specific destination, say v): If we solve the SSSPP with source vertex s, we also solved this problem. Moreover, no algorithm for this problem is known that asymptotically faster than the SSSP in worst case.

- **All pair shortest path problem (APSPP):** Find a shortest path between every pair of vertices u and v. One technique is to use SSSP for each vertex, but there are some more efficient algorithm (known as Floyed-warshall's algorithm).

To find a SSSP for directed graphs $G(V, E)$, we have two different algorithms:

1. Bellman-Ford algorithm
2. Dijkstra's algorithm

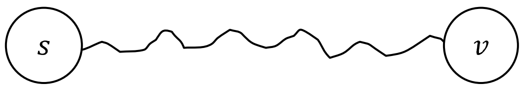
- Bellman-ford algorithm, allow ***negative weight edges*** in the input graph. This algorithm either finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$ or detect a negative weight cycles in G, hence ***no solution***. If there is no negative weight cycles are reachable (or exist) from source vertex s, then we can find a shortest path form source vertex $s \in V$ to every other vertex $v \in V$. If there exist a negative weight cycles in the input graph, then the algorithm can detect it, and hence "No solution".

- **Dijkstra's algorithm** allows only positive ***weight edges*** in the input graph and finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$.

To understand the basic concept of negative-weight cycle, consider the following 2 cases:

Case1: ***Shortest-path cannot contain a cycle***; ***it is just a simple path*** (i.e. no repeated vertex):

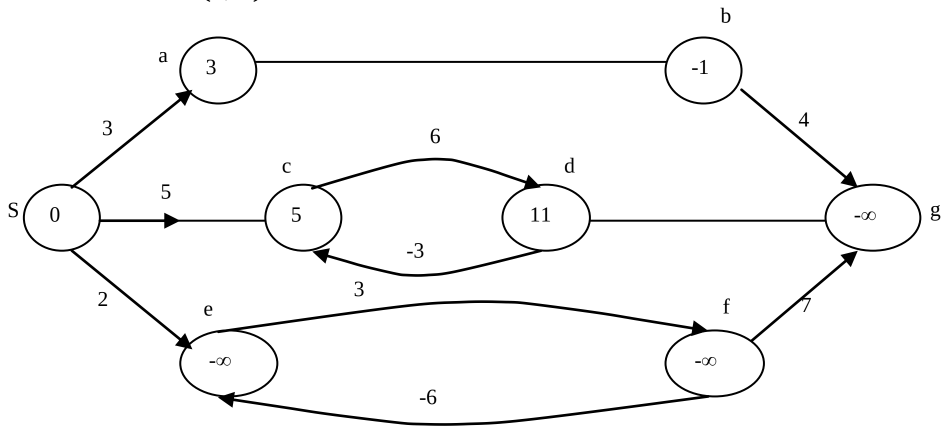
If some path from s to v contains a negative cost cycle, then there does not exist a shortest path. Otherwise there exists a shortest path (i.e. a simple path) from $u \rightsquigarrow v$.



Case 2: Graph containing a negative-weight cycle:

- No problem, if it is not reachable from the source vertex s.
- If it is reachable from source vertex s, then we just keep going around it and producing a path weight of $-\infty$. If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty\, for\ all\ v\ on\ the\ cycle.$

For example: consider a graph with negative weight cycle:
If there is a negatives weight cycle on some path from s to v,
we define $\delta(s,v) = -\infty$.



There are infinitely many paths from s to c: <s,c>,<a,c,d,c>,<s,c,d,c,d,c>, and so on.

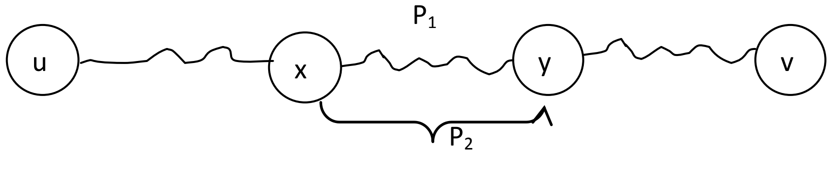
There are infinitely many paths from *s* to *c*: ⟨*s*, *c*⟩ , ⟨*s*, *c*, *d*, *c*⟩ , ⟨*s*, *c*, *d*, *c*, *d*, *c*⟩ , and so on. Because the cycle ⟨*c*, *d*, *c*⟩ has weight 6 + (-3) = 3 > 0, the shortest path from *s* to *c* is ⟨*s*, *c*⟩ , with weight *δ*(*s*, *c*) = 5. Similarly, the shortest path from *s* to *d* is ⟨*s*, *c*, *d*⟩ , with weight *δ*(*s*, *d*) = *w*(*s*, *c*) + *w*(*c*, *d*) = 11. Analogously, there are infinitely many paths from *s* to *e*: ⟨*s*, *e*⟩ , ⟨*s*, *e*, *f*, *e*⟩ , ⟨*s*, *e*, *f*, *e*, *f*, *e*⟩ , and so on. Since the cycle ⟨*e*, *f*, *e*⟩ has weight 3 + (-6) = -3 < 0, however, there is no shortest path from *s* to *e*. By traversing the negative-weight cycle ⟨*e*, *f*, *e*⟩ arbitrarily many times, we can find paths from *s* to *e* with arbitrarily large negative weights, and so *δ*(*s*, *e*) = -∞. Similarly, *δ*(*s*, *f*) = -∞. Because *g* is reachable from *f* , we can also find paths with arbitrarily large negative weights from *s* to *g*, and *δ*(*s*, *g*) = -∞. Vertices *h*, *i*, and *j* also form a negative-weight cycle. They are not reachable from *s*, however, and so *δ*(*s*, *h*) = *δ*(*s*, *i*) = *δ*(*s*, *j*) = ∞.

Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are non negative, as in the road-map example. Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

Shortest path : Properties

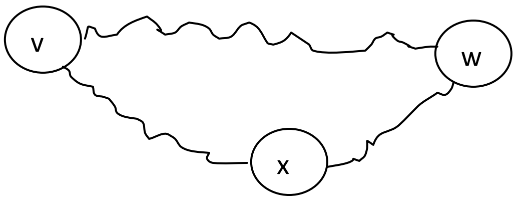
1. **Optimal sub-structure property**: Any sub-path of a shortest path is also a shortest path.

   Let P_1 is any sub-path (say x↝y) of a shortest s ↝ v, and P_2 is axy x ↝ y path; then the cost of $P_1 \leq$ cost of P_2; otherwise s ↝ v is not a shortest path.



2. **Triangle inequality**: Let $d(v, w)$ be the length of the shortest path from v to w, then

$$\boldsymbol{d(v, w) \leq d(v, x) + d(x, w}$$



3. Shortest path does not contains a cycle:

- Negative weight cycles are not allowed when it is reachable from source vertex s, since in this case there is no shortest path.
- If Positive –weight cycles are there then by removing the cycle, we can get a shorter path.

Generic Algorithm for solving single-source-shortest path (SSSP) problem:

Given a directed weighted graph $G(V, E)$, algorithm always maintain the following two fields for each vertex $v \in V$.

1. $\boldsymbol{d[v] = \delta(s, v)} =$
   $the\ length\ of\ the\ shortest\ path\ from\ starting\ vertex\ s\ to\ v,$
   (initially $d[v] = \infty$), and the value of $d[v]$ reduces as the algorithm progress. Thus we call $\boldsymbol{d[v]}$**,** a **shortest path estimate.**

2. $\boldsymbol{Pred[v]}$, $which\ is\ a\ predecessor\ of\ v\ on\ a\ shortest\ path\ from\ s.$ The value of $\boldsymbol{Pred[v]}$ is either a another vertex or NIL

Initialization:

All the shortest-paths algorithms start with initializing $\boldsymbol{d[v]}$ and $\boldsymbol{Pred[v]}$ **by us**ing the following procedure INITIALIZE_SIGLE_SOURCE.

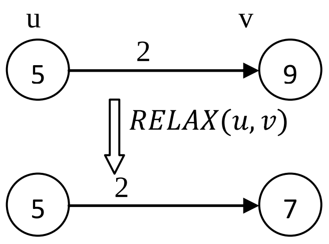
```
INITIALIZE_SIGLE_SOURCE(V,s)
    1. for each $v \in V$
    2.    do $d[v] \leftarrow \infty$
    3.       $pred[v] \leftarrow NIL$
    4. $d[s] \leftarrow 0$
```

After this initialization procedure, $d[v] = 0$ for start vertex s, and $d[v] = \infty$ for $v \in V - \{s\}$ and $pred[v] = NIL$ for all $v \in V$.

<u>Relaxing an edge (u, v):</u>

The SSSP algorithms are based on the technique known as ***edge relaxation***. The process of relaxing an edge (u, v) consists of testing whether we can improve (or reduce) the shortest path to v found so far (i.e $d[v]$) by going through u and taking (u, v) and, if so, update $d[v]$ and $pred[v]$. This is accomplished by the following procedure:

```
RELAX(u,v,w)
    1. $if\ (d[v] > d[u] + w(u,v)$
    2.     then $d[v] \leftarrow d[u] + w(u,v)$
    3.         $pred[v] \leftarrow u$
```



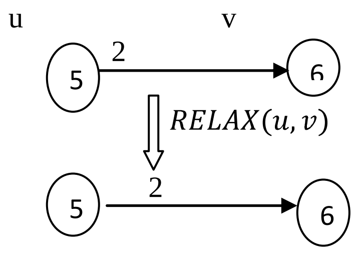


Figure (a): $d[v] > d[u] + w(u,v)$ i.e. $9 > 5 + 2$, hence $d[v] = 7$

Figure (b): No change in $d[v]$ since, $d[v] < d[u] + w(u,v)$.

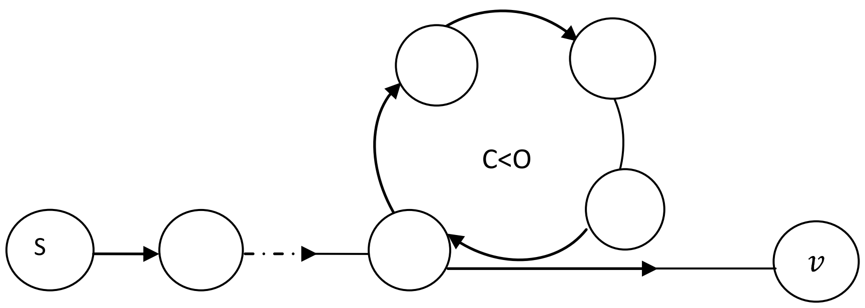
Note:

- For all the SSSP algorithm, we always start by calling **INITIALIZE_SIGLE_SOURCE(V,s)** and then relax edges.
- In **Dijkstra's algorithm** and the other shortest-path algorithm for directed acyclic graph, each edge is relaxed exactly once. In a **Bellman-Ford** algorithm, each edge is relaxed several times

1.6.1 Bellman-Ford Algorithm

- Bellman-ford algorithm, allow ***negative weight edges*** in the input graph. This algorithm either finds a shortest path from a source vertex $s \in V$ to every other vertex $v \in V - \{s\}$

or detect a negative weight cycles exist in G, hence ***no shortest path exist for some vertices***.



- Thus given a weighted, directed graph with $G = (V, E)$ with weight function $w: E \rightarrow R$, the Bellman-ford algorithm returns a **Boolean value** (either **TRUE** or **FALSE)** indicating whether or not there is a negative weight cycle is reachable from the source vertex. The algorithm returns a Boolean TRUE if the given graph G contains no negative weight cycle that are reachable from source vertex s, otherwise it returns Boolean FALSE.

- The algorithm uses a technique of ***relaxation*** and progressively decreases an estimate $d[v]$ on the weight of a shortest path from the source vertex s to each vertex $v \in V$ until it achieves the actual shortest path. We also maintain a predecessor value $\boldsymbol{pred}[\boldsymbol{v}]$ for all $v \in V$.

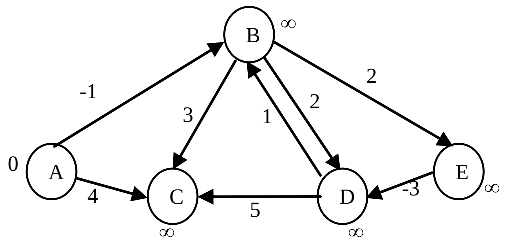
```
BELLMAN_FORD(G,w,s)

    1. INITIALIZE_SIGLE_SOURCE(G,s)
    2. $for\ i \leftarrow 1\ to\ |V| - 1$
    3.   $do\ for\ each\ edge\ (u, v) \in E[G]$
    4.     do $RELAX(u, v, w)$
    5. $for\ each\ edge\ (u, v) \in E[G]$
    6.     do $if\ (d[v] > d[u] + w(u, v)$
    7.       then $return\ FALSE$          // we detect a negative weight cycle
exist
    7. $return\ TRUE$
```

Analysis of Bellman-ford algorithm

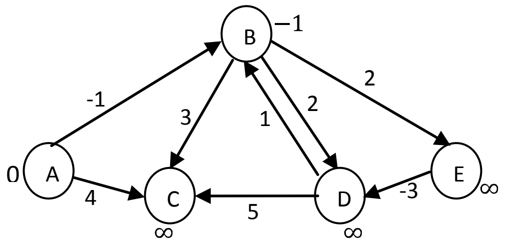
1. Line 1 for initializing $d[v]'s$, $Pred[v]'s$ and setting $v[s] = 0$ takes $\boldsymbol{O}(\boldsymbol{V})$ time.
2. For loop at line-2 executed $(|V| - 1)$ times which Relaxes all the E edges, so line 2-4 requires $(|V| - 1). O(E) = O(VE)$.
3. For loop at line 5 checks negative weight cycle for all the E edges, which requires O(E) time.

   Thus the run time of Bellman ford algorithm is $O(V + E + VE) = O(VE)$.

Order of edge: (B,E), (D,B), (B,D), (A,C), (D,C), (B,C), (E,D)

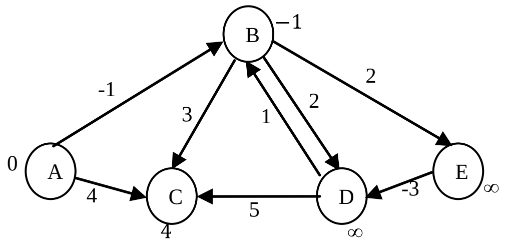


A	B	C	D	E
0	∞	∞	∞	∞

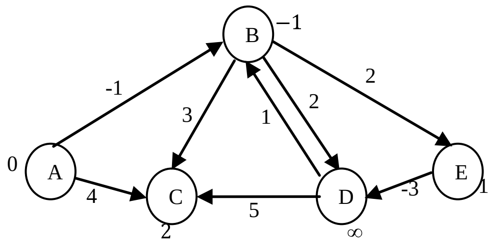


A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞

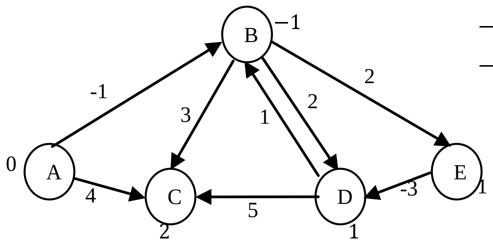
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞



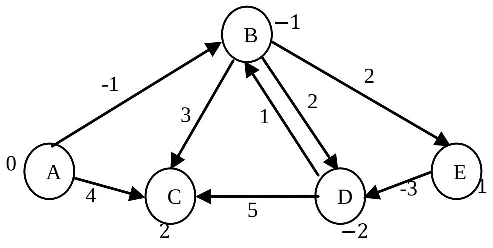
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



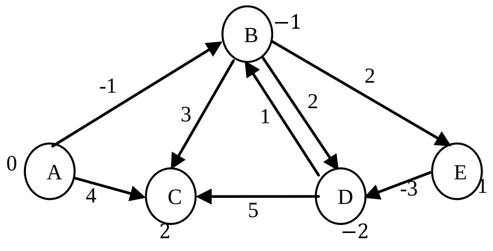
A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

1.6.2 Dijkstra's Algorithm

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a greedy algorithm that solves the single-source shortest path problem for a directed graph G=(V,E) with ***non-negative edge*** weights i.e. we assume that w (u,v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's algorithm maintains a set of S of vertices whose final shortest-path weights from the source have already been determined. That is, all vertices $v \in$ S , we have d[v] = $\delta(s, v)$. the algorithm repeatedly selects the vertex u ∈V-S with the minimum shortest-path estimate, inserts u into S and relaxes all edges leaving u. We maintain a min-priority queue Q that contains all the vertices in $V - s$ keyed by their d values. Graph G is represented by adjacency lists.

DIJKSTRA(G, w, s)

```
1    INITIALIZE-SINGLE-SOURCE($G$, $s$)
2    $S \leftarrow \emptyset$
3    $Q \leftarrow V[G]$
4    while $Q \neq \emptyset$
5        do $u \leftarrow$ EXTRACT-MIN($Q$)
6           $S \leftarrow S \cup \{u\}$
7           for each vertex $v \in Adj[u]$
8              do RELAX($u$, $v$, $w$)
```

Because Dijkstra's algorithm always choose the "lightest" or "closest" vertex in V-S to insert into set S, we say that it uses a greedy strategy.

Dijkstra's algorithm bears some similarly to both breadth-first search and Prim's algorithm for computing minimum spanning trees. It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so do black vertices in a breadth-first search have their correct breadth- first distances.
Dijkstra's algorithm is like prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

Analysis of Dijkstra's algorithm

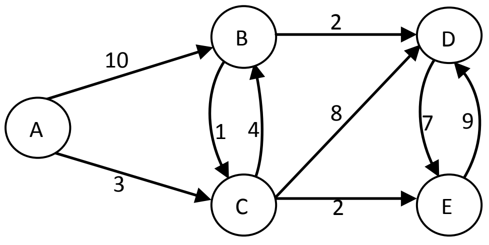
The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as function of |E| and |V| using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q.

in this case, the running time is $O(|V|^2 + |E|) = O(V^2)$.

For sparse graphs, that is, graphs with many fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently, storing the graph in the form of adjacency lists and using a binary heap or Fibonacci heap as a priority queue to implement the Extract-Min function. With a binary heap, the algorithm requires O((|E| + |V|) time (which is dominated by O|E|log|V|) assuming every vertex is connected, and the Fibonacci heap improves this to O|E| + |V|log|V|).

Example1:

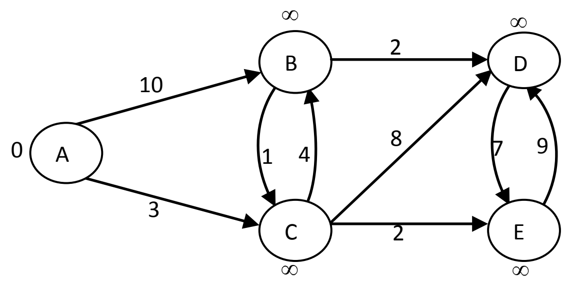
Apply Dijkstra's algorithm to find shortest path from source vertex A to each of the other vertices of the following directed graph.



Solution:

Dijkstra's algorithm maintains a set of S of vertices whose final shortest-path weights from the source have already been determined. The algorithm repeatedly selects the vertex u ∈V-S with the minimum shortest-path estimate, inserts u into S and relaxes all edges leaving u. We maintain a min-priority queue Q that contains all the vertices in $V - s$ keyed by their d values.

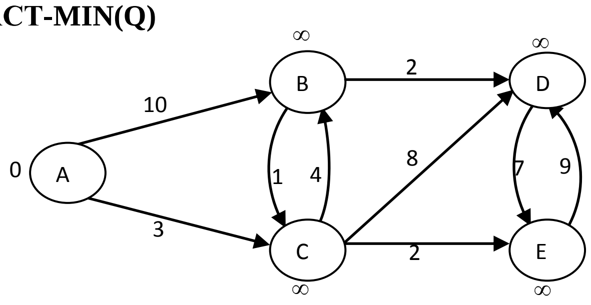
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S={}

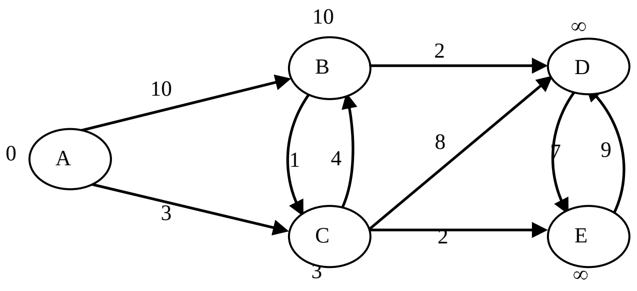
"A" ← EXTRACT-MIN(Q)



Q:	**A**	**B**	**C**	**D**	**E**
	0	∞	∞	∞	∞

S:{A}

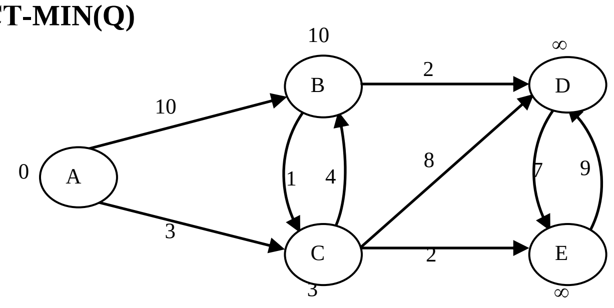
Relax all edges leaving A:



Q:	**A**	**B**	**C**	**D**	**E**
	0	∞	∞	∞	∞
		10	**3**	**-**	**-**

S:{A}

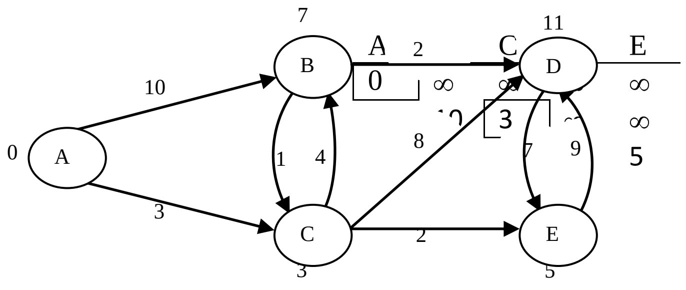
"C" ← EXTRACT-MIN(Q)



Q:	**A**	**B**	**C**	**D**	**E**
	$\boxed{\textbf{0}}$	∞	∞	∞	∞
		10	$\boxed{\textbf{3}}$	**-**	**-**

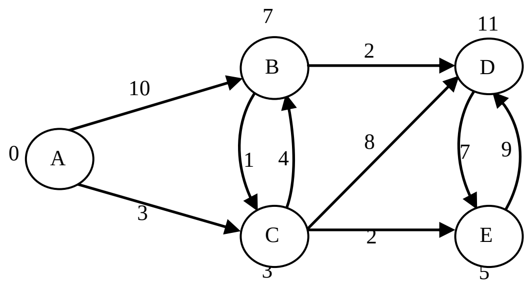
S:{A,C}

Relax all edges leaving C:



S:{A,C}

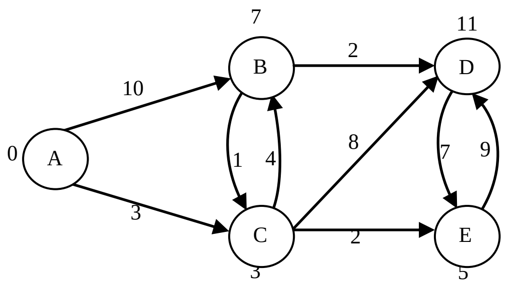
“E”← EXTRAXT-MIN(Q)



S:{A,C,E}

Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	-	-
		7		11	$\boxed{5}$

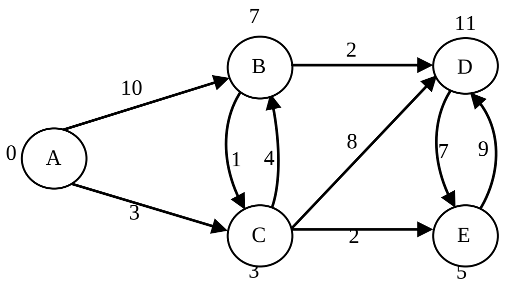
Relax all edges leaving E:



S:{A,C,E}

Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		7		11	

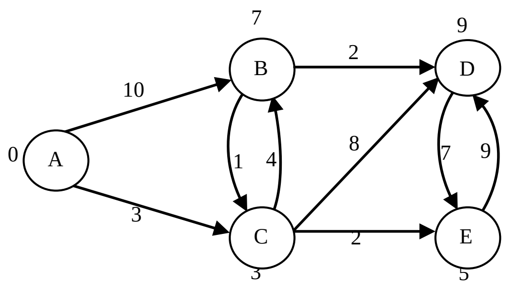
“B” ← EXTRACT-MIN(Q):



S:{A,C,E,B}

Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	

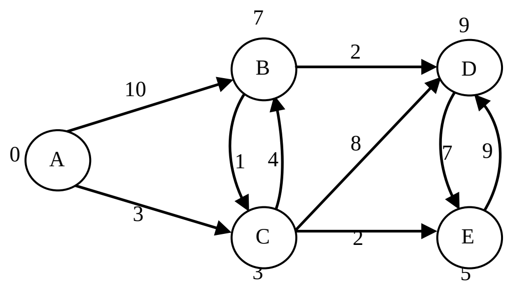
Relax all edges leaving B:



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				9	

S:{A,C,E,B}

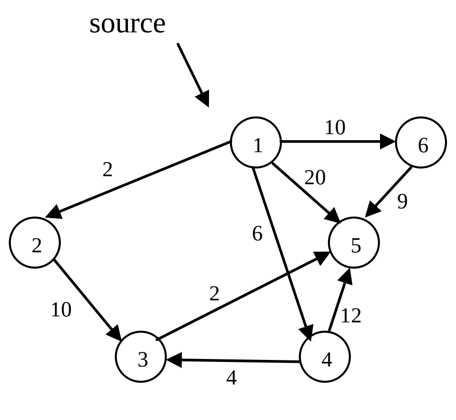
"D" ←EXTRACT-MIN(Q):



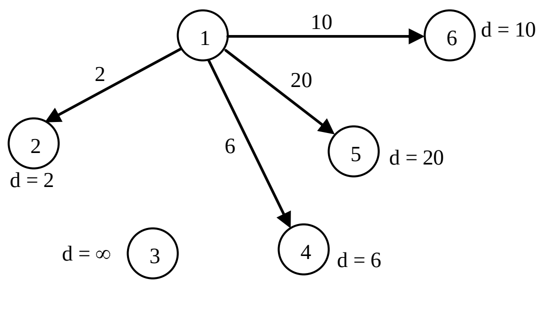
Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

S:{A,C,E,B,D}

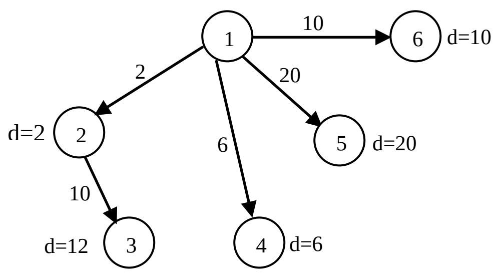
Example2: Apply dijkstra's algorithm on the following digraph (1 is starting vertex)



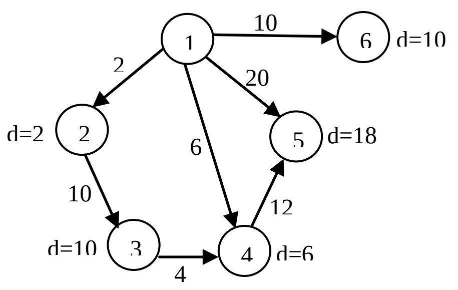
S={ }



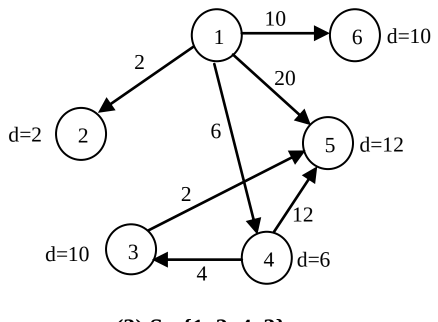
(initial) S={1}



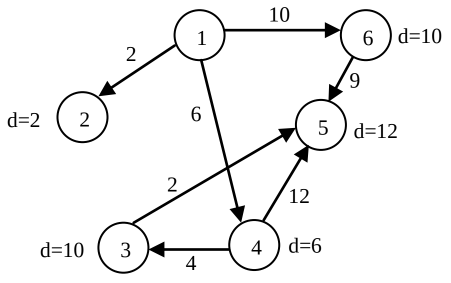
(1) S= {1, 2}



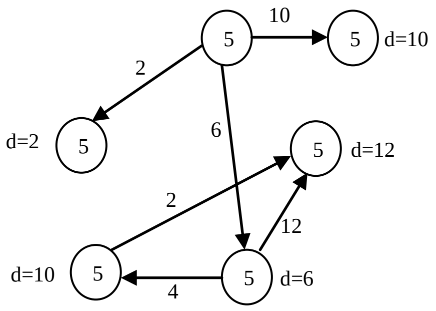
(2) S={1, 2, 4}



(3) S= {1, 2, 4, 3}



(4) S= {1, 2, 4, 3, 6}



(5) S={1, 2, 4, 3, 6, 5}

Figure 2: Stages of Dijkstra's algorithm

Iterations	S	Q (Priority Queue)						EXTRACT_ MIN(Q)
		d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	
Initial	{ }	0	∞	∞	∞	∞	∞	1
1	(1}	**[0]**	2	∞	6	20	10	2
2	{1,2}		[2]	**12**	**6**	**20**	**10**	4
3	(1,2,4}			10	[6]	18	10	3
4	{1,2,4,3}			**[10]**		**18**	**10**	6
5	{1,2,4,3,6}					**12**	**[10]**	5
	{1,2,4,3,6,5}					**[12]**		

Table2: Computation of Dijkstra's algorithm on digraph of Figure 2

☞ Check Your Progress 2

Choose correct option for Q.1 to Q.5

Q.1: Dijkstra's algorithm running time, where n is the number of nodes in a graph is:

a) $O(n^2)$ b) $O(n^3)$ c) $O(n)$ d) $O(n \log n)$

Q2: This of the following algorithm allows negative edge weight in a graph to find shortest path?

a) Dijkstra's algorithm b) Bellman-ford algorithm c) Kruskal algo. d) Prim's Algo

Q3: The running time of Bellman-ford algorithm is

a) $O(|E|^2)$ b) $O(|V|^2)$ c) $O(|E|\log|V|)$ d) $O(|E||V|)$

Q.4: Consider a weighted undirected graph with positive edge weights and let (u,v) be an edge in the graph. It is known that the shortest path from source vertex s to u has weight 60 and the shortest path from source vertex s to v has weight 75. which statement is always true?

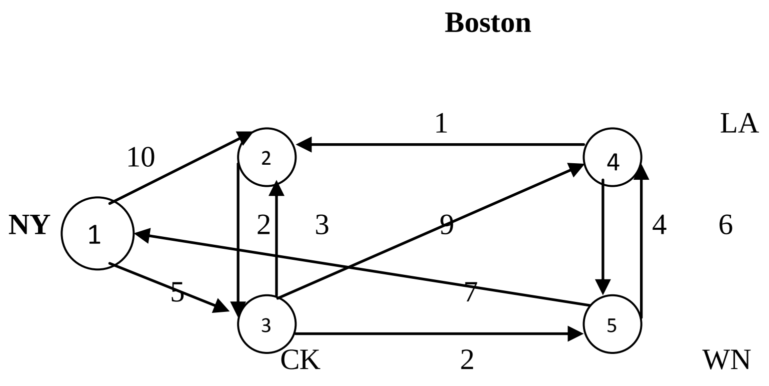
a) weight $(u,v) \le 15$ b) weight $(u,v) = 15$ c) weight $(u,v) \ge 15$ d) weight $(u,v) > 15$

Q.5: Which data structure is used to maintained the distance of each node in a Dijkstras's algorithm.

a) Stack b) Queue c) Priority Queue d) Tree

Q.6: Differentiate between Bellman-ford and Dijkstra's algorithm to find a shortest path in a graph?

Q.7: Find the minimum distance of each station from New York (**NY**) using Dijkstra's algorithm. Show all the steps.



Q.8: Analyze the running time of the Dijkstra's algorithm?

1.7 SUMMARY

- Greedy algorithms are typically used to solve an ***optimization problem.***
- An Optimization problem is one in which, we are given a set of input values, which are required to be either maximized or minimized w. r. t. some constraints or conditions.
- Generally an optimization problem has n inputs (call this set as **input domain** or **Candidate set**, C), we are required to obtain a subset of C (call it ***solution set***, S where S$\subseteq C$) that satisfies the given constraints or conditions. Any subset S $\subseteq C$, which satisfies the given constraints, is called a ***feasible*** solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that does this is called a **optimal solution**.
- Greedy algorithm always makes the choice that looks best at the moment. That is, it makes a ***locally optimal choice in the hope that this choice will lead to a overall globally optimal solution.***
- Greedy algorithm does not always yield an optimal solution; but for many problems they do.
- The (fractional) Knapsack problem is to fill a knapsack or bag (up to its maximum capacity M) with the given $n\ items,$ which maximizes the total profit earned.
- Let G=(V,E) be an undirected connected graph. A ***subgraph*** T=(V,E') of G is a ***spanning tree*** of G if and only if T is a tree (i.e. no cycle exist in T) and contains **all the vertices** of G.
- A *complete graph* (each vertex in G is connected to every other vertices) with n vertices has total n^{n-2} spanning tree. For example, if n=5 then total number of spanning tree is 125.
- A ***minimum cost spanning tree*** (MCST) of a weighted connected graph G is that spanning tree whose sum of length (or weight) of all its edges is minimum, among all the possible spanning tree of G.
- There are two algorithm to find a MCST of a given directed graph G, namely Kruskal's algorithm and Prim's algorithm.
- The basic difference between Kruskal's and Prim's algorithm is that in kruskal's algorithm it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). Thus At intermediate step of algorithm, there are may be more than one connected components of trees are possible. But in case of Prim's algorithm it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). Thus at intermediate step of algorithm, there will be only one connected components are possible.
    - Kruskal's algorithm runs in $O(|E|log|V|)$ time and Prim's algorithm runs in time (n^2) , where n is the number of nodes in the graph.

- ***Single-source-shortest path problem (SSSPP) problem*** is to find a shortest path from source vertex $s \in V$ to every other vertex $v \in V - s$ in a given graph $G = (V, E)$

- To find a SSSP for directed graphs $G(V, E)$, we have two different algorithms: ***Bellman-Ford algorithm*** and ***Dijkstra's algorithm***

- ***Bellman-ford algorithm***, allow ***negative weight edges*** also in the input graph where as **Dijkstra's algorithm** allows only ***positive weight edges*** in the input graph.

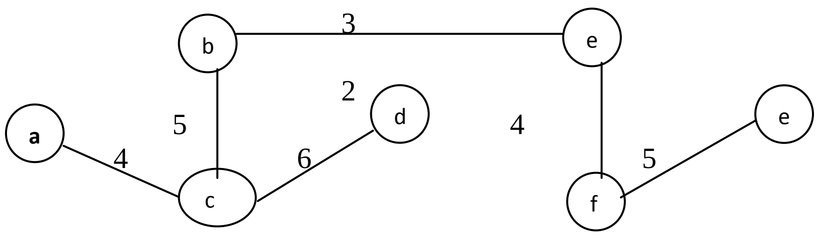
- ***Bellman-ford algorithm*** runs in time $O(|V||E|)$ **whereas Dijkstra's algorithm runs in time $\boldsymbol{O(n^2)}$.**

1.8 SOLUTIONS/ANSWERS

Check your Progress 1:

1-d, 2-c, 3-b, 4-c, 5-c, 6-b, 7-d

Solution 8:



Total minimum cost of given graph G = $3 + 4 + 5 + 6 + 4 + 3 = 27$

Solution 9:

(a) FALSE, since edge with the smallest weight will be part of every minimum spanning tree.
(b) **TRUE:** edge with the smallest weight will be part of every minimum spanning tree.
(c) **TRUE:**
(d) **TRUE:** Since more than one edges in a Graph may have the same weight.
(e) **TRUE**: In a connected weighted graph in which edge weights are not all distinct, then the graph must have more than one spanning tree but the minimum cost of those spanning tree will be same.

Solution 10:

A greedy algorithm proceeds step–by-step, by considering one input at a time. At each stage, the decision is made regarding whether a particular input (say x) chosen gives an optimal solution or not. Our choice of selecting input x is being guided by the selection function (say ***select***). If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set. On the other hand, if the inclusion of that input x results in an infeasible solution, then this input x is not added to the partial solution. When a greedy algorithm works correctly, the first solution found in this way is always optimal. In brief, at each stage, the following activities are performed in greedy method:

1. First we select an element, say$x$, from input domain C.
2. Then we check whether the solution set S is feasible or not. That is we check whether x can be included into the solution set S or not. If yes, then solution set $S \leftarrow S \cup \{x\}$. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.
3. Continue until S is filled up (i.e. optimal solution found) or C is exhausted whichever is earlier.

A general form for greedy technique can be illustrated as:

```
Algorithm Greedy(C, n)
   /* Input: A input domain (or Candidate set ) C of size n, from which solution is to be
             Obtained. */
   // function select (C: candidate_set)  return an element (or candidate).
  // function solution (S: candidate_set) return Boolean
  // function feasible (S: candidate_set) return Boolean
   /* Output: A solution set S, where S ⊆ $C$, which maximize or minimize the selection
              criteria w. r. t. given constraints */
  {

      $S \leftarrow \phi$                     // Initially a solution set S is empty.
     While ( not solution(S) and $C \neq \phi$)
       {
         $x \leftarrow \boldsymbol{select}(C)$          /* A "best" element x is selected from C which
                                    maximize or minimize the selection criteria. */
         $C \leftarrow \boldsymbol{C} - \{\boldsymbol{x}\}$                 /* once x is selected , it is removed from C
         if ( feasible($S \cup \{x\}$)  then   /* x is now checked for feasibility
            $S \leftarrow S \cup \{x\}$
       }
     If (solution (S))
         return S;
      else
        return " No Solution"
   } // end of while
```

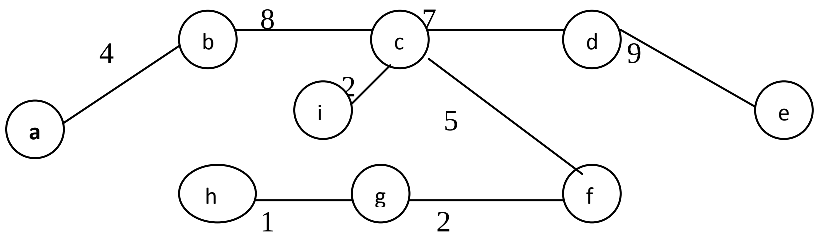
Solution11:

A main ***difference*** between ***kruskal's*** and ***Prim's*** algorithm to solve MCST problem is that the order in which the edges are selected.

Kruskal's Algorithm	**Prim's algorithm**
• Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST. • In kruskal's algorithm for getting MCST, it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there are may be more than one connected components are possible. • Time complexity: $O(\|E\|log\|V\|)$	• Prim's algorithm always selects a vertex (say, v) to find MCST. • In Prim's algorithm for getting MCST, it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). Thus • At intermediate step of algorithm, there will be only one connected components are possible • Time complexity: $O(\|V\|^2)$

Solution 12: No, spanning tree of a graph is not unique in general, because more than one edges of a graph may have the same weight.

For complete solution refer Q.2 , given in this booklet.



Total minimum cost of the spanning tree =$4 + 8 + 2 + 1 + 2 + 5 + 7 + 9 = 38$

Solution 13:

Given n=5, M=12,

$(P_1, P_2, ..., P_5) = (12,32,40,30,50)$

$(W_1, W_2, ..., W_5) = (4,8,2,6,1)$.

$$\left(\frac{P_1}{w_1}, \frac{P_2}{w_2}, \dots, \frac{P_5}{w_5}\right) = (3,4,20,5,50)$$

Thus the item which has maximum P_i/w_i value will be placed into a knapsack first, That is 5th item first, then 3rd item then 4th item then 2nd and then 1st item (if capacity of knapsack is remaining). The following table shows a solution of this knapsack problem.

S.No	Solution Set $(x_1, x_2, x_3, , x_4, x_5)$	$\sum_{i=1}^{5} w_i x_i$	$\sum_{i=1}^{5} p_i x_i$
1	$(\frac{1}{2}, 1,1,1,1)$	$1 + 2 + 6 + 1 + 4 \times \frac{1}{2} = 12$	158

Solution 14:

Given n=5, M=18,

$(P_1, P_2, ..., P_7) = (12,10,8,11,14,7,9)$

$(W_1, W_2, ..., W_7) = (4,6,5,7,3,1,6)$.

$$\left(\frac{P_1}{w_1}, \frac{P_2}{w_2}, \dots, \frac{P_7}{w_7}\right) = (3.0, 1.67, 1.60, 1.57, 4.67, 7.0, 1.50)$$

The item which has maximum P_i/w_i value will be placed into a knapsack first. Thus the sequence of items placed into a knapsack is: 6th , 5th ,1st ,2nd, 3rd , 4th and then 7th item. The following table shows a solution of this knapsack problem.

S.No	Solution Set $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\sum_{i=1}^{7} w_i x_i$	$\sum_{i=1}^{7} p_i x_i$
1	$(1,1,\frac{4}{5}, 0,1,1,0)$	$1 + 3 + 4 + 6 + 5 \times \frac{4}{5} =$ 18	49.4

Check Your Progress 2

1-a, 2-b, 3-d, 4-a, 5-c

Solution 6:

- Bellman-ford algorithm, allow ***negative weight edges*** in the input graph. This algorithm either finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$ or detect a negative weight cycles in G, hence ***no solution***. If there is no negative weight cycles are reachable (or exist) from source vertex s, then we can find a shortest path form source vertex $s \in V$ to every other vertex $v \in V$. If there exist a negative weight cycles in the input graph, then the algorithm can detect it, and hence "No solution".

- **Dijkstra's algorithm** allows only positive ***weight edges*** in the input graph and finds a shortest path from source vertex $s \in V$ to every other vertex$v \in V$.

Solution 7:

Following Table summarizes the Computation of Dijkstra's algorithm for the given digraph of Question 7.

Iterations	**S**	**Q (Priority Queue)** d[1]	d[2]	d[3]	d[4]	d[5]	**EXTRACT _MIN(Q)**
Initial	{ }	0	∞	∞	∞	∞	1
1	(1}	**[0]**	10	5	∞	∞	3
2	{1,3}		8	**[5]**	14	7	2
3	(1,3,2}		**[8]**		14	7	5
4	{1,3,2,5}				13	[7]	4
5	{1,3,2,5,4)				**[13]**		

Table1: Computation of Dijkstra's algorithm on digraph of question 7

Solution 8: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as function of |E| and |V| using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q. in this case, the running time is $O(|V|^2 + |E|) = O(V^2)$.

3.9 FURTHER READING

1. *Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson (PHI)*
2. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).
3. *Algoritmics: The Spirit of Computing,* D. Harel*:* (Addison-Wesley Publishing Company, 1987).
4. *Fundamentals of Algorithmics*, G. Brassard & P. Brately: (Prentice-Hall International, 1996).
5. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth: (Narosa Publishing House).
6. *Fundamentals of Computer Algorithms*, E. Horowitz & S. Sahni: (Galgotia Publications).
7. *The Design and Analysis of Algorithms*, Anany Levitin: (Pearson Education, 2003).
8. *Programming Languages* (*Second Edition*) ─ *Concepts and Constructs,* Ravi Sethi: (Pearson Education, Asia, 1996).