**Chhattisgarh Swami Vivekananda Technical University, Bhilai**

**B Tech Honours (Artificial Intelligence and Data Science)**

**Analysis & Design of Algorithm**

**Subject Code: B000372 (022) Semester: III**

**UNIT – III GREEDY METHODS**

Greedy Algorithm

The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results. Let's understand through some terms.

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Characteristics of Greedy method

**The following are the characteristics of a greedy method:**Hello Java Program for Beginners

* To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
* A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Components of Greedy Algorithm

**The components that can be used in the greedy algorithm are:**

* **Candidate set:** A solution that is created from the set is known as a candidate set.
* **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
* **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
* **Objective function:** A function is used to assign the value to the solution or the partial solution.
* **Solution function:** This function is used to intimate whether the complete function has been reached or not.

Applications of Greedy Algorithm

* It is used in finding the shortest path.
* It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
* It is used in a job sequencing with a deadline.
* This algorithm is also used to solve the fractional knapsack problem.

Pseudo code of Greedy Algorithm

1. Algorithm Greedy (a, n)
2. {
3. Solution : = 0;
4. for i = 0 to n do
5. {
6. x: = select(a);
7. if feasible(solution, x)
8. {
9. Solution: = union(solution , x)
10. }
11. return solution;
12. } }

The above is the greedy algorithm. Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

**Let's understand through an example.**

Suppose there is a problem 'P'. I want to travel from A to B shown as below:

**P : A → B**

The problem is that we have to travel this journey from A to B. There are various solutions to go from A to B. We can go from A to B by **walk, car, bike, train, aeroplane**, etc. There is a constraint in the journey that we have to travel this journey within 12 hrs. If I go by train or aeroplane then only, I can cover this distance within 12 hrs. There are many solutions to this problem but there are only two solutions that satisfy the constraint.

If we say that we have to cover the journey at the minimum cost. This means that we have to travel this distance as minimum as possible, so this problem is known as a minimization problem. Till now, we have two feasible solutions, i.e., one by train and another one by air. Since travelling by train will lead to the minimum cost so it is an optimal solution. An optimal solution is also the feasible solution, but providing the best result so that solution is the optimal solution with the minimum cost. There would be only one optimal solution.

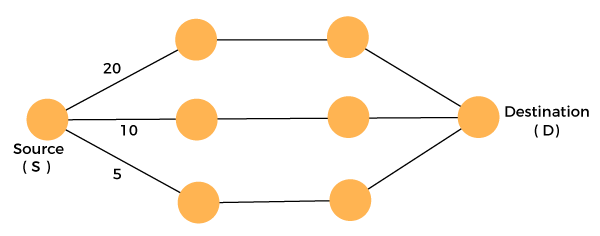
The problem that requires either minimum or maximum result then that problem is known as an optimization problem. Greedy method is one of the strategies used for solving the optimization problems.

Disadvantages of using Greedy algorithm

Greedy algorithm makes decisions based on the information available at each phase without considering the broader problem. So, there might be a possibility that the greedy solution does not give the best solution for every problem.

It follows the local optimum choice at each stage with a intend of finding the global optimum. Let's understand through an example.

**Consider the graph which is given below:**



We have to travel from the source to the destination at the minimum cost. Since we have three feasible solutions having cost paths as 10, 20, and 5. 5 is the minimum cost path so it is the optimal solution. This is the local optimum, and in this way, we find the local optimum at each stage in order to calculate the global optimal solution.

Fractional Knapsack Problem

The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

**This problem can be solved with the help of using two techniques:**

* Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
* Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.

**There are basically three approaches to solve the problem:**

* The first approach is to select the item based on the maximum profit.
* The second approach is to select the item based on the minimum weight.
* The third approach is to calculate the ratio of profit/weight.

**Consider the below example:**

Objects:         1     2     3     4     5     6     7

Profit (P):        5 10   15     7     8     9     4

Weight (w):     1     3     5     4     1     3     2

W (Weight of the knapsack): 15

n (no of items): 7

### First approach:

**First approach:**

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 3 | 15 | 5 | 15 - 5 = 10 |
| 2 | 10 | 3 | 10 - 3 = 7 |
| 6 | 9 | 3 | 7 - 3 = 4 |
| 5 | 8 | 1 | 4 - 1 = 3 |
| 4 | 7 \* ¾ = 5.25 | 3 | 3 - 3 = 0 |

The total profit would be equal to (15 + 10 + 9 + 8 + 5.25) = 47.25

### Second approach:

The second approach is to select the item based on the minimum weight.

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 1 | 5 | 1 | 15 - 1 = 14 |
| 5 | 7 | 1 | 14 - 1 = 13 |
| 7 | 4 | 2 | 13 - 2 = 11 |
| 2 | 10 | 3 | 11 - 3 = 8 |
| 6 | 9 | 3 | 8 - 3 = 5 |
| 4 | 7 | 4 | 5 - 4 = 1 |
| 3 | 15 \* 1/5 = 3 | 1 | 1 - 1 = 0 |

**In this case, the total profit would be equal to (5 + 7 + 4 + 10 + 9 + 7 + 3) = 46**

### Third approach:

**In the third approach, we will calculate the ratio of profit/weight.**

Objects:         1     2     3     4     5     6     7

Profit (P):        5    10     15     7     8     9     4

Weight(w):       1     3     5     4     1     3     2

In this case, we first calculate the profit/weight ratio.

Object 1: 5/1 = 5

Object 2: 10/3 = 3. 33

Object 3: 15/5 = 3

Object 4: 7/4 = 1.7

Object 5: 8/1 = 8

Object 6: 9/3 = 3

**Object 7: 4/2 = 2**

**P:w:         5     3.3     3     1.7     8     3     2**

In this approach, we will select the objects based on the maximum profit/weight ratio. Since the P/W of object 5 is maximum so we select object 5.

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 5 | 8 | 1 | 15 - 8 = 7 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

After object 5, object 1 has the maximum profit/weight ratio, i.e., 5. So, we select object 1 shown in the below table:

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 5 | 8 | 1 | 15 - 1 = 14 |
| 1 | 5 | 1 | 14 - 1 = 13 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

After object 1, object 2 has the maximum profit/weight ratio, i.e., 3.3. So, we select object 2 having profit/weight ratio as 3.3.

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 5 | 8 | 1 | 15 - 1 = 14 |
| 1 | 5 | 1 | 14 - 1 = 13 |
| 2 | 10 | 3 | 13 - 3 = 10 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

After object 2, object 3 has the maximum profit/weight ratio, i.e., 3. So, we select object 3 having profit/weight ratio as 3.

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 5 | 8 | 1 | 15 - 1 = 14 |
| 1 | 5 | 1 | 14 - 1 = 13 |
| 2 | 10 | 3 | 13 - 3 = 10 |
| 3 | 15 | 5 | 10 - 5 = 5 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

After object 3, object 6 has the maximum profit/weight ratio, i.e., 3. So we select object 6 having profit/weight ratio as 3.

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 5 | 8 | 1 | 15 - 1 = 14 |
| 1 | 5 | 1 | 14 - 1 = 13 |
| 2 | 10 | 3 | 13 - 3 = 10 |
| 3 | 15 | 5 | 10 - 5 = 5 |
| 6 | 9 | 3 | 5 - 3 = 2 |
|  |  |  |  |
|  |  |  |  |

After object 6, object 7 has the maximum profit/weight ratio, i.e., 2. So we select object 7 having profit/weight ratio as 2.

|  |  |  |  |
| --- | --- | --- | --- |
| Object | Profit | Weight | Remaining weight |
| 5 | 8 | 1 | 15 - 1 = 14 |
| 1 | 5 | 1 | 14 - 1 = 13 |
| 2 | 10 | 3 | 13 - 3 = 10 |
| 3 | 15 | 5 | 10 - 5 = 5 |
| 6 | 9 | 3 | 5 - 3 = 2 |
| 7 | 4 | 2 | 2 - 2 = 0 |
|  |  |  |  |

As we can observe in the above table that the remaining weight is zero which means that the knapsack is full. We cannot add more objects in the knapsack. Therefore, the total profit would be equal to (8 + 5 + 10 + 15 + 9 + 4), i.e., 51.

In the first approach, the maximum profit is 47.25. The maximum profit in the second approach is 46. The maximum profit in the third approach is 51. Therefore, we can say that the third approach, i.e., maximum profit/weight ratio is the best approach among all the approaches.

Huffman Codes

* (i) Data can be encoded efficiently using Huffman Codes.
* (ii) It is a widely used and beneficial technique for compressing data.
* (iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 105 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x 105 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

Huffman Codes

How can we represent the data in a Compact way?

**(i) Fixed length Code:** Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

**For example:**

a 000

b 001

c 010

d 011

e 100

f 101

For a file with 105 characters, we need 3 x 105 bits.

**(ii) A variable-length code:** It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long codewords.

**For example:**

a 0

b 101

c 100

d 111

e 1101

f 1100

Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000

**= 2.24 x 105bits**

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

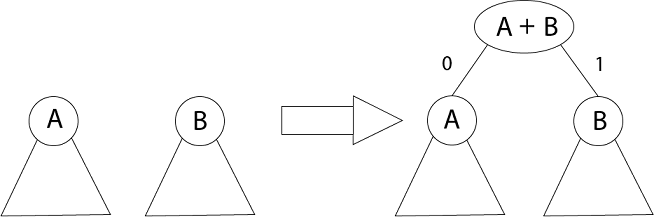
## Prefix Codes:

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the codeword that starts with an encoded data is unambiguous.

## Greedy Algorithm for constructing a Huffman Code:

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.



The algorithm builds the tree T analogous to the optimal code in a bottom-up manner. It starts with a set of |C| leaves (C is the number of characters) and performs |C| - 1 'merging' operations to create the final tree. In the Huffman algorithm 'n' denotes the quantity of a set of characters, z indicates the parent node, and x & y are the left & right child of z respectively.

Algorithm of Huffman Code

**Huffman (C)**

1. n=|C|

2. Q ← C

3. for i=1 to n-1

4. do

5. z= allocate-Node ()

6. x= left[z]=Extract-Min(Q)

7. y= right[z] =Extract-Min(Q)

8. f [z]=f[x]+f[y]

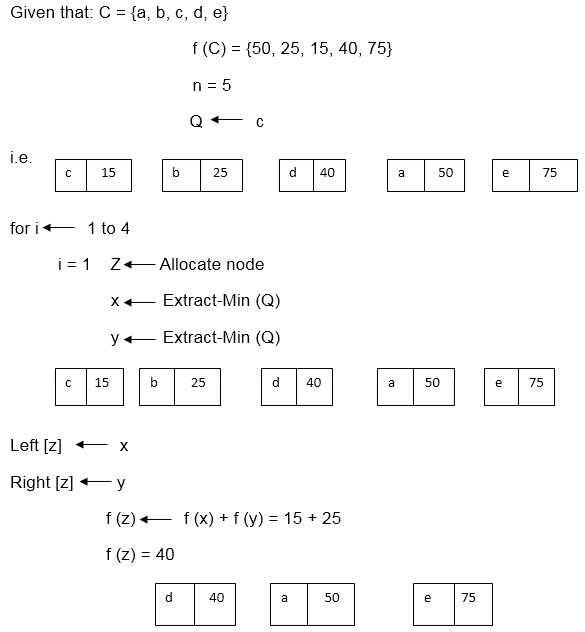
9. Insert (Q, z)

10. return Extract-Min (Q)

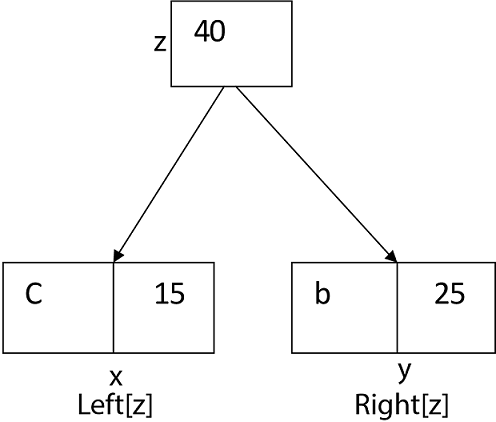
**Example:** Find an optimal Huffman Code for the following set of frequencies:

1. a: 50   b: 25   c: 15   d: 40   e: 75

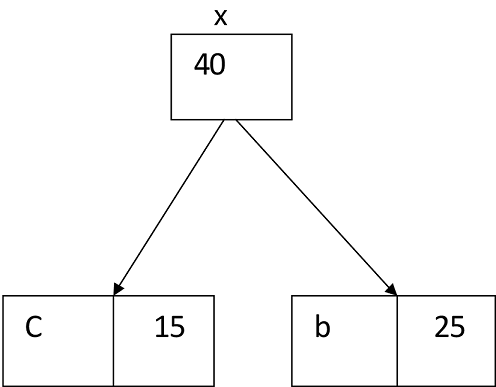
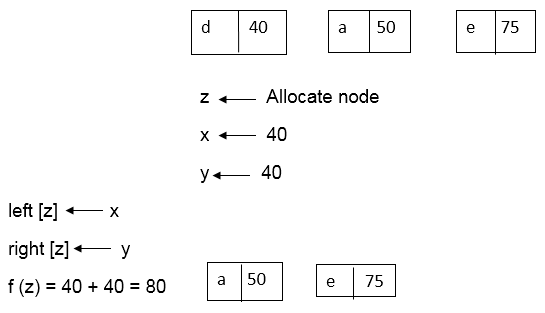
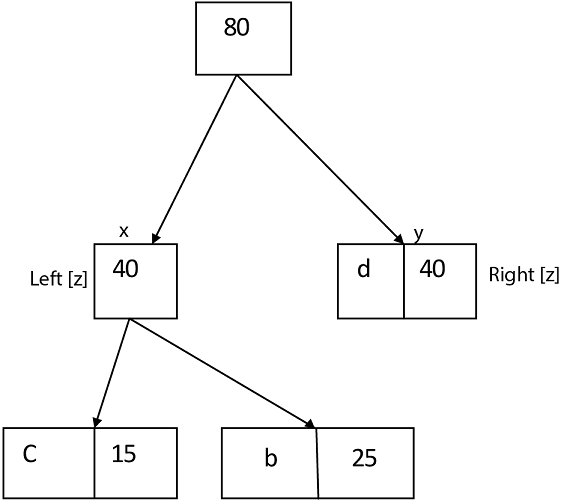
**Solution:**



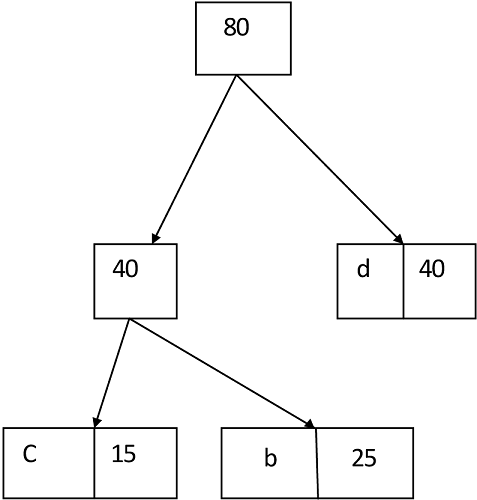
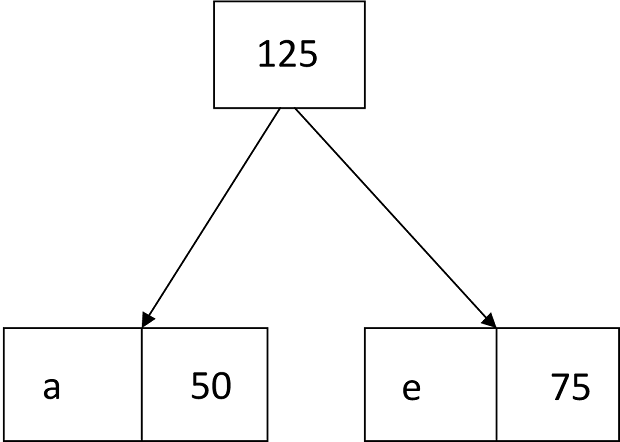
i.e.



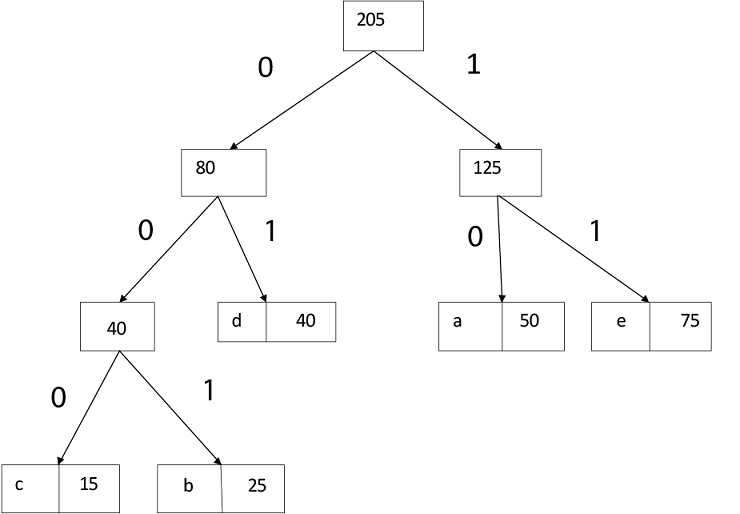
Again for i=2

Similarly, we apply the same process we get

Thus, the final output is:



# Spanning tree

In this article, we will discuss the spanning tree and the minimum spanning tree. But before moving directly towards the spanning tree, let's first see a brief description of the graph and its types.

### Graph

A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -

* **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
* **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
* **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

Now, let's move towards the topic spanning tree.

### What is a spanning tree?

A spanning tree can be defined as the sub graph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have **nn-2** number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be **55-2 = 125.**

### Applications of the spanning tree

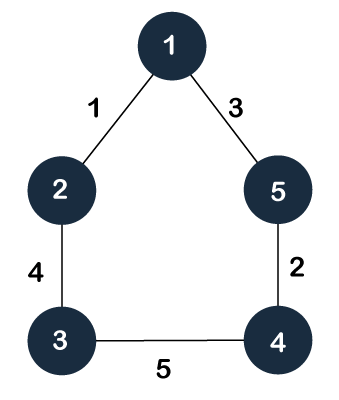
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

* Cluster Analysis
* Civil network planning
* Computer network routing protocol

Now, let's understand the spanning tree with the help of an example.

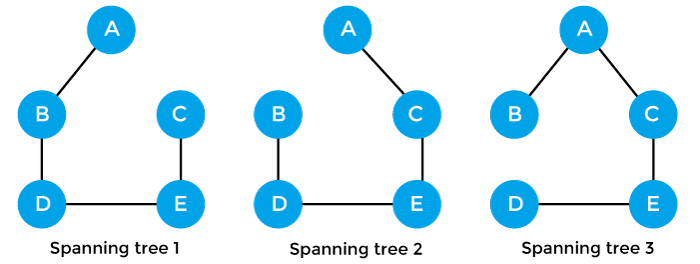
### Example of Spanning tree

Suppose the graph be -



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

Some of the possible spanning trees that will be created from the above graph are given as follows -



## Properties of spanning-tree

Some of the properties of the spanning tree are given as follows -

* There can be more than one spanning tree of a connected graph G.
* A spanning tree does not have any cycles or loop.
* A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
* A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
* There can be a maximum **nn-2** number of spanning trees that can be created from a complete graph.
* A spanning tree has **n-1** edges, where 'n' is the number of nodes.
* If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

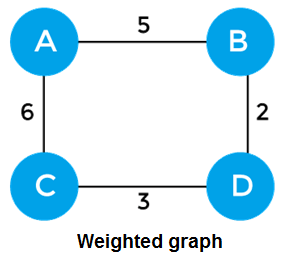
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

## Minimum Spanning tree

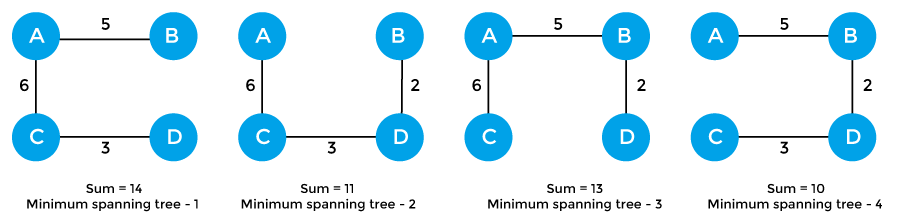
A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

### Example of minimum spanning tree

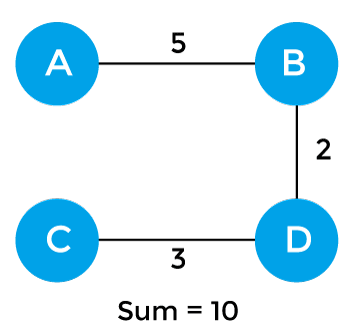
Let's understand the minimum spanning tree with the help of an example.



The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



### Applications of minimum spanning tree

The applications of the minimum spanning tree are given as follows -

* Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
* It can be used to find paths in the map.

### Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

* Prim's Algorithm
* Kruskal's Algorithm

Let's see a brief description of both of the algorithms listed above.

**Prim's algorithm -** It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

**Kruskal's algorithm -** This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

# Prim's Algorithm

In this article, we will discuss the prim's algorithm. Along with the algorithm, we will also see the complexity, working, example, and implementation of prim's algorithm.

Before starting the main topic, we should discuss the basic and important terms such as spanning tree and minimum spanning tree.

**Spanning tree -** A spanning tree is the subgraph of an undirected connected graph.

**Minimum Spanning tree -** Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start the main topic.

**Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

## How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

* First, we have to initialize an MST with the randomly chosen vertex.
* Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
* Repeat step 2 until the minimum spanning tree is formed.

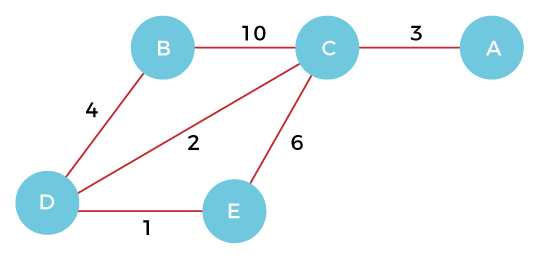
The applications of prim's algorithm are -

* Prim's algorithm can be used in network designing.
* It can be used to make network cycles.
* It can also be used to lay down electrical wiring cables.

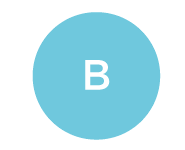
## Example of prim's algorithm

Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

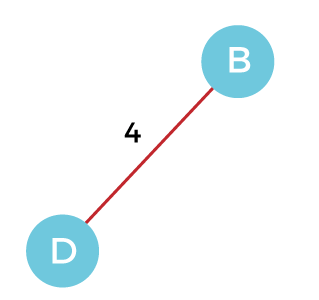
Suppose, a weighted graph is -



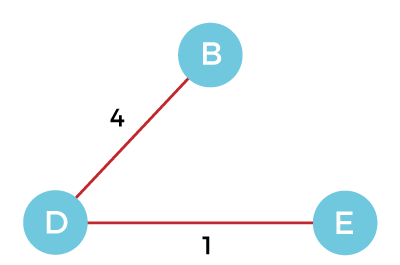
**Step 1 -** First, we have to choose a vertex from the above graph. Let's choose B.



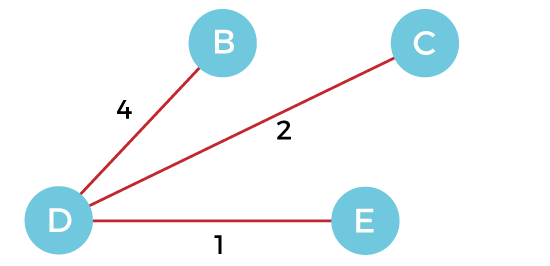
**Step 2 -** Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



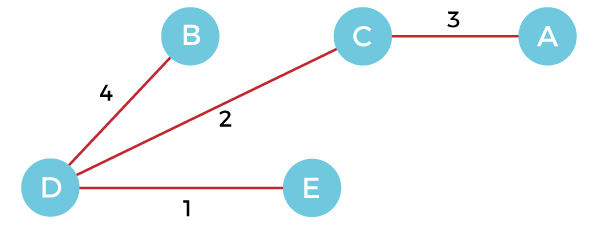
**Step 3 -** Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



**Step 4 -** Now, select the edge CD, and add it to the MST.



**Step 5 -** Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

## Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

## Complexity of Prim's algorithm

Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -

* **Time Complexity**

|  |  |
| --- | --- |
| **Data structure used for the minimum edge weight** | **Time Complexity** |
| Adjacency matrix, linear searching | O(|V|2) |
| Adjacency list and binary heap | O(|E| log |V|) |
| Adjacency list and Fibonacci heap | O(|E|+ |V| log |V|) |

Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires O(|V|2) running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.

The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

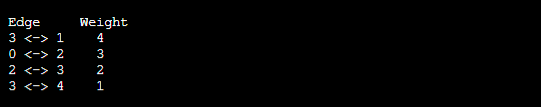
## Implementation of Prim's algorithm

Now, let's see the implementation of prim's algorithm.

**Program:** Write a program to implement prim's algorithm in C language.

1. #include <stdio.h>
2. #include <limits.h>
3. #define vertices 5  /\*Define the number of vertices in the graph\*/
4. /\* create minimum\_key() method for finding the vertex that has minimum key-value and that is not added in MST yet \*/
5. **int** minimum\_key(**int** k[], **int** mst[])
6. {
7. **int** minimum  = INT\_MAX, min,i;
9. /\*iterate over all vertices to find the vertex with minimum key-value\*/
10. **for** (i = 0; i < vertices; i++)
11. **if** (mst[i] == 0 && k[i] < minimum )
12. minimum = k[i], min = i;
13. **return** min;
14. }
15. /\* create prim() method for constructing and printing the MST.
16. The g[vertices][vertices] is an adjacency matrix that defines the graph for MST.\*/
17. **void** prim(**int** g[vertices][vertices])
18. {
19. /\* create array of size equal to total number of vertices for storing the MST\*/
20. **int** parent[vertices];
21. /\* create k[vertices] array for selecting an edge having minimum weight\*/
22. **int** k[vertices];
23. **int** mst[vertices];
24. **int** i, count,edge,v; /\*Here 'v' is the vertex\*/
25. **for** (i = 0; i < vertices; i++)
26. {
27. k[i] = INT\_MAX;
28. mst[i] = 0;
29. }
30. k[0] = 0; /\*It select as first vertex\*/
31. parent[0] = -1;   /\* set first value of parent[] array to -1 to make it root of MST\*/
32. **for** (count = 0; count < vertices-1; count++)
33. {
34. /\*select the vertex having minimum key and that is not added in the MST yet from the set of vertices\*/
35. edge = minimum\_key(k, mst);
36. mst[edge] = 1;
37. **for** (v = 0; v < vertices; v++)
38. {
39. **if** (g[edge][v] && mst[v] == 0 && g[edge][v] <  k[v])
40. {
41. parent[v]  = edge, k[v] = g[edge][v];
42. }
43. }
44. }
45. /\*Print the constructed Minimum spanning tree\*/
46. printf("\n Edge \t  Weight\n");
47. **for** (i = 1; i < vertices; i++)
48. printf(" %d <-> %d    %d \n", parent[i], i, g[i][parent[i]]);
50. }
51. **int** main()
52. {
53. **int** g[vertices][vertices] = {{0, 0, 3, 0, 0},
54. {0, 0, 10, 4, 0},
55. {3, 10, 0, 2, 6},
56. {0, 4, 2, 0, 1},
57. {0, 0, 6, 1, 0},
58. };
59. prim(g);
60. **return** 0;
61. }

**Output**



So, that's all about the article. Hope, the article will be helpful and informative to you.

# Kruskal's Algorithm

In this article, we will discuss Kruskal's algorithm. Here, we will also see the complexity, working, example, and implementation of the Kruskal's algorithm.

But before moving directly towards the algorithm, we should first understand the basic terms such as spanning tree and minimum spanning tree.

**Spanning tree -** A spanning tree is the subgraph of an undirected connected graph.

**Minimum Spanning tree -** Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start with the main topic.

**Kruskal's Algorithm** is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

## How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

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

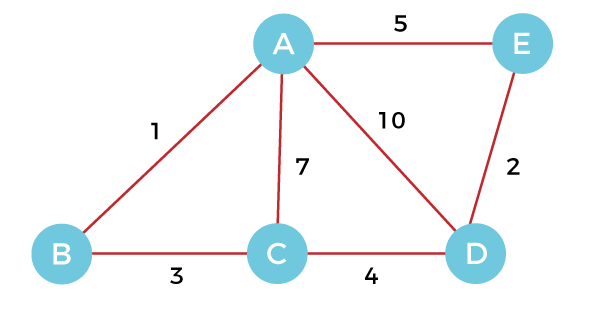
The applications of Kruskal's algorithm are -

* Kruskal's algorithm can be used to layout electrical wiring among cities.
* It can be used to lay down LAN connections.

## Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

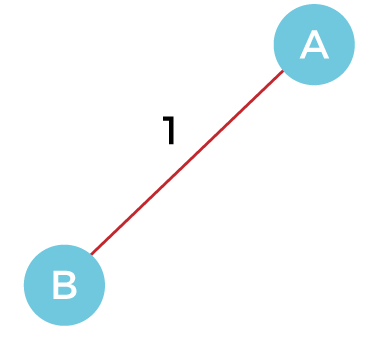
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | AB | AC | AD | AE | BC | CD | DE |
| **Weight** | 1 | 7 | 10 | 5 | 3 | 4 | 2 |

Now, sort the edges given above in the ascending order of their weights.

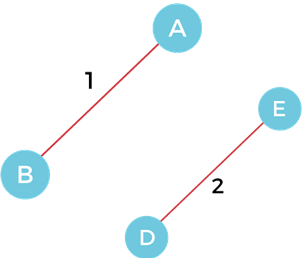
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | AB | DE | BC | CD | AE | AC | AD |
| **Weight** | 1 | 2 | 3 | 4 | 5 | 7 | 10 |

Now, let's start constructing the minimum spanning tree.

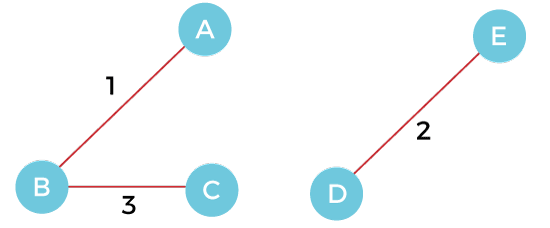
**Step 1 -** First, add the edge **AB** with weight **1** to the MST.



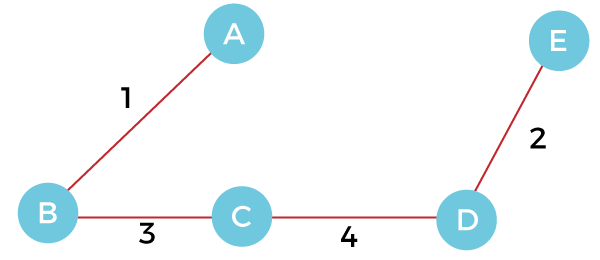
**Step 2 -** Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



**Step 3 -** Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



**Step 4 -** Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

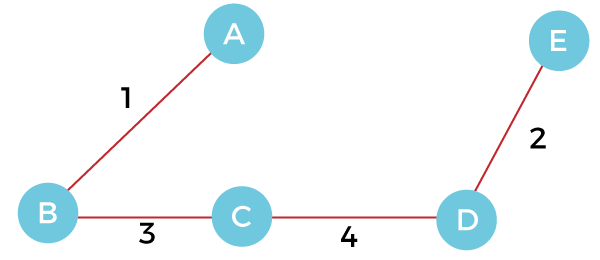


**Step 5 -** After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

**Step 6 -** Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

**Step 7 -** Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

## Algorithm

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
6. (**for** combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

## Complexity of Kruskal's algorithm

Now, let's see the time complexity of Kruskal's algorithm.

* **Time Complexity**  
  The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

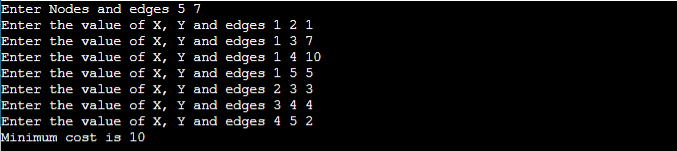
## Implementation of Kruskal's algorithm

Now, let's see the implementation of kruskal's algorithm.

**Program:** Write a program to implement kruskal's algorithm in C++.

1. #include <iostream>
2. #include <algorithm>
3. **using** **namespace** std;
4. **const** **int** MAX = 1e4 + 5;
5. **int** id[MAX], nodes, edges;
6. pair <**long** **long**, pair<**int**, **int**> > p[MAX];
7. **void** init()
8. {
9. **for**(**int** i = 0;i < MAX;++i)
10. id[i] = i;
11. }
12. **int** root(**int** x)
13. {
14. **while**(id[x] != x)
15. {
16. id[x] = id[id[x]];
17. x = id[x];
18. }
19. **return** x;
20. }
21. **void** union1(**int** x, **int** y)
22. {
23. **int** p = root(x);
24. **int** q = root(y);
25. id[p] = id[q];
26. }
27. **long** **long** kruskal(pair<**long** **long**, pair<**int**, **int**> > p[])
28. {
29. **int** x, y;
30. **long** **long** cost, minimumCost = 0;
31. **for**(**int** i = 0;i < edges;++i)
32. {
33. x = p[i].second.first;
34. y = p[i].second.second;
35. cost = p[i].first;
36. **if**(root(x) != root(y))
37. {
38. minimumCost += cost;
39. union1(x, y);
40. }
41. }
42. **return** minimumCost;
43. }
44. **int** main()
45. {
46. **int** x, y;
47. **long** **long** weight, cost, minimumCost;
48. init();
49. cout <<"Enter Nodes and edges";
50. cin >> nodes >> edges;
51. **for**(**int** i = 0;i < edges;++i)
52. {
53. cout<<"Enter the value of X, Y and edges";
54. cin >> x >> y >> weight;
55. p[i] = make\_pair(weight, make\_pair(x, y));
56. }
57. sort(p, p + edges);
58. minimumCost = kruskal(p);
59. cout <<"Minimum cost is "<< minimumCost << endl;
60. **return** 0;
61. }

**Output**

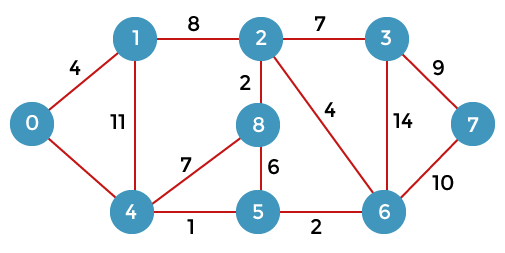


So, that's all about the article. Hope the article will be helpful and informative to you.

Dijkstra's Algorithm

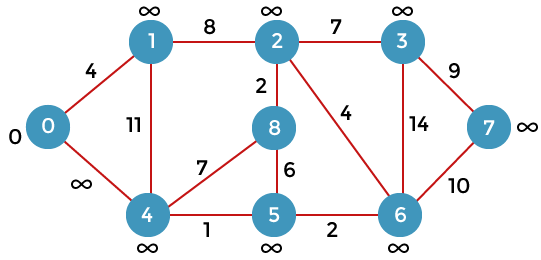
Dijkstra algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

**Let's understand the working of Dijkstra's algorithm. Consider the below graph.**



First, we have to consider any vertex as a source vertex. Suppose we consider vertex 0 as a source vertex.

Here we assume that 0 as a source vertex, and distance to all the other vertices is infinity. Initially, we do not know the distances. First, we will find out the vertices which are directly connected to the vertex 0. As we can observe in the above graph that two vertices are directly connected to vertex 0.



Let's assume that the vertex 0 is represented by 'x' and the vertex 1 is represented by 'y'. The distance between the vertices can be calculated by using the below formula:

1. d(x, y) = d(x) + c(x, y)  < d(y)
2. = (0 + 4) < ∞
3. = 4 < ∞

Since 4<∞ so we will update d(v) from ∞ to 4.

Therefore, we come to the conclusion that the formula for calculating the distance between the vertices:

1. {**if**( d(u) + c(u, v) < d(v))
2. d(v) = d(u)  +c(u, v) }

Now we consider vertex 0 same as 'x' and vertex 4 as 'y'.

1. d(x, y) = d(x) + c(x, y)  < d(y)
2. = (0 + 8) < ∞
3. = 8 < ∞

Therefore, the value of d(y) is 8. We replace the infinity value of vertices 1 and 4 with the values 4 and 8 respectively. Now, we have found the shortest path from the vertex 0 to 1 and 0 to 4. Therefore, vertex 0 is selected. Now, we will compare all the vertices except the vertex 0. Since vertex 1 has the lowest value, i.e., 4; therefore, vertex 1 is selected.

Since vertex 1 is selected, so we consider the path from 1 to 2, and 1 to 4. We will not consider the path from 1 to 0 as the vertex 0 is already selected.

First, we calculate the distance between the vertex 1 and 2. Consider the vertex 1 as 'x', and the vertex 2 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (4 + 8) < ∞
3. = 12 < ∞

Since 12<∞ so we will update d(2) from ∞ to 12.

Now, we calculate the distance between the vertex 1 and vertex 4. Consider the vertex 1 as 'x' and the vertex 4 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (4 + 11) < 8
3. = 15 < 8

Since 15 is not less than 8, we will not update the value d(4) from 8 to 12.

Till now, two nodes have been selected, i.e., 0 and 1. Now we have to compare the nodes except the node 0 and 1. The node 4 has the minimum distance, i.e., 8. Therefore, vertex 4 is selected.

Since vertex 4 is selected, so we will consider all the direct paths from the vertex 4. The direct paths from vertex 4 are 4 to 0, 4 to 1, 4 to 8, and 4 to 5. Since the vertices 0 and 1 have already been selected so we will not consider the vertices 0 and 1. We will consider only two vertices, i.e., 8 and 5.

First, we consider the vertex 8. First, we calculate the distance between the vertex 4 and 8. Consider the vertex 4 as 'x', and the vertex 8 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (8 + 7) < ∞
3. = 15 < ∞

Since 15 is less than the infinity so we update d(8) from infinity to 15.

Now, we consider the vertex 5. First, we calculate the distance between the vertex 4 and 5. Consider the vertex 4 as 'x', and the vertex 5 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (8 + 1) < ∞
3. = 9 < ∞

Since 5 is less than the infinity, we update d(5) from infinity to 9.

Till now, three nodes have been selected, i.e., 0, 1, and 4. Now we have to compare the nodes except the nodes 0, 1 and 4. The node 5 has the minimum value, i.e., 9. Therefore, vertex 5 is selected.

Since the vertex 5 is selected, so we will consider all the direct paths from vertex 5. The direct paths from vertex 5 are 5 to 8, and 5 to 6.

First, we consider the vertex 8. First, we calculate the distance between the vertex 5 and 8. Consider the vertex 5 as 'x', and the vertex 8 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (9 + 15) < 15
3. = 24 < 15

Since 24 is not less than 15 so we will not update the value d(8) from 15 to 24.

Now, we consider the vertex 6. First, we calculate the distance between the vertex 5 and 6. Consider the vertex 5 as 'x', and the vertex 6 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (9 + 2) < ∞</p>
3. = 11 < ∞

Since 11 is less than infinity, we update d(6) from infinity to 11.

Till now, nodes 0, 1, 4 and 5 have been selected. We will compare the nodes except the selected nodes. The node 6 has the lowest value as compared to other nodes. Therefore, vertex 6 is selected.

Since vertex 6 is selected, we consider all the direct paths from vertex 6. The direct paths from vertex 6 are 6 to 2, 6 to 3, and 6 to 7.

First, we consider the vertex 2. Consider the vertex 6 as 'x', and the vertex 2 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (11 + 4) < 12
3. = 15 < 12

Since 15 is not less than 12, we will not update d(2) from 12 to 15

Now we consider the vertex 3. Consider the vertex 6 as 'x', and the vertex 3 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (11 + 14) < ∞
3. = 25 < ∞

Since 25 is less than ∞, so we will update d(3) from ∞ to 25.

Now we consider the vertex 7. Consider the vertex 6 as 'x', and the vertex 7 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (11 + 10) < ∞
3. = 22 < ∞

Since 22 is less than ∞ so, we will update d(7) from ∞ to 22.

Till now, nodes 0, 1, 4, 5, and 6 have been selected. Now we have to compare all the unvisited nodes, i.e., 2, 3, 7, and 8. Since node 2 has the minimum value, i.e., 12 among all the other unvisited nodes. Therefore, node 2 is selected.

Since node 2 is selected, so we consider all the direct paths from node 2. The direct paths from node 2 are 2 to 8, 2 to 6, and 2 to 3.

First, we consider the vertex 8. Consider the vertex 2 as 'x' and 8 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (12 + 2) < 15
3. = 14 < 15

Since 14 is less than 15, we will update d(8) from 15 to 14.

Now, we consider the vertex 6. Consider the vertex 2 as 'x' and 6 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (12 + 4) < 11
3. = 16 < 11

Since 16 is not less than 11 so we will not update d(6) from 11 to 16.

Now, we consider the vertex 3. Consider the vertex 2 as 'x' and 3 as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (12 + 7) < 25
3. = 19 < 25

Since 19 is less than 25, we will update d(3) from 25 to 19.

Till now, nodes 0, 1, 2, 4, 5, and 6 have been selected. We compare all the unvisited nodes, i.e., 3, 7, and 8. Among nodes 3, 7, and 8, node 8 has the minimum value. The nodes which are directly connected to node 8 are 2, 4, and 5. Since all the directly connected nodes are selected so we will not consider any node for the updation.

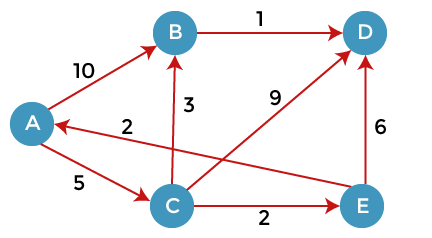
The unvisited nodes are 3 and 7. Among the nodes 3 and 7, node 3 has the minimum value, i.e., 19. Therefore, the node 3 is selected. The nodes which are directly connected to the node 3 are 2, 6, and 7. Since the nodes 2 and 6 have been selected so we will consider these two nodes.

Now, we consider the vertex 7. Consider the vertex 3 as 'x' and 7 as 'y'.

1. d(x, y) = d(x) + c(x, y)  < d(y)
2. = (19 + 9) < 21
3. = 28 < 21

Since 28 is not less than 21, so we will not update d(7) from 28 to 21.

**Let's consider the directed graph.**



Here, we consider A as a source vertex. A vertex is a source vertex so entry is filled with 0 while other vertices filled with ∞. The distance from source vertex to source vertex is 0, and the distance from the source vertex to other vertices is ∞.

We will solve this problem using the below table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **D** | **E** |
| ∞ | ∞ | ∞ | ∞ | ∞ |

Since 0 is the minimum value in the above table, so we select vertex A and added in the second row shown as below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |

As we can observe in the above graph that there are two vertices directly connected to the vertex A, i.e., B and C. The vertex A is not directly connected to the vertex E, i.e., the edge is from E to A. Here we can calculate the two distances, i.e., from A to B and A to C. The same formula will be used as in the previous problem.

1. If(d(x) + c(x, y)  < d(y))
2. Then we update d(y) = d(x) + c(x, y)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
|  |  | 10 | 5 | ∞ | ∞ |

As we can observe in the third row that 5 is the lowest value so vertex C will be added in the third row.

We have calculated the distance of vertices B and C from A. Now we will compare the vertices to find the vertex with the lowest value. Since the vertex C has the minimum value, i.e., 5 so vertex C will be selected.

Since the vertex C is selected, so we consider all the direct paths from the vertex C. The direct paths from the vertex C are C to B, C to D, and C to E.

First, we consider the vertex B. We calculate the distance from C to B. Consider vertex C as 'x' and vertex B as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (5 + 3) < ∞
3. = 8 < ∞

Since 8 is less than the infinity so we update d(B) from ∞ to 8. Now the new row will be inserted in which value 8 will be added under the B column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
|  |  | 10 | 5 | ∞ | ∞ |
|  |  | 8 |  |  |  |

We consider the vertex D. We calculate the distance from C to D. Consider vertex C as 'x' and vertex D as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (5 + 9) < ∞
3. = 14 < ∞

Since 14 is less than the infinity so we update d(D) from ∞ to 14. The value 14 will be added under the D column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
|  |  | 8 |  | 14 |  |

We consider the vertex E. We calculate the distance from C to E. Consider vertex C as 'x' and vertex E as 'y'.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (5 + 2) < ∞
3. = 7 < ∞

Since 14 is less than the infinity so we update d(D) from ∞ to 14. The value 14 will be added under the D column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
|  |  | 8 |  | 14 | 7 |

As we can observe in the above table that 7 is the minimum value among 8, 14, and

7. Therefore, the vertex E is added on the left as shown in the below table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
| E |  | 8 |  | 14 | 7 |

The vertex E is selected so we consider all the direct paths from the vertex E. The direct paths from the vertex E are E to A and E to D. Since the vertex A is selected, so we will not consider the path from E to A.

**Consider the path from E to D.**

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (7 + 6) < 14
3. = 13 < 14

Since 13 is less than the infinity so we update d(D) from ∞ to 13. The value 13 will be added under the D column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
| E |  | 8 |  | 14 | 7 |
| B |  | 8 |  | 13 |  |

The value 8 is minimum among 8 and 13. Therefore, vertex B is selected. The direct path from B is B to D.

1. d(x, y) = d(x) + c(x, y) < d(y)
2. = (8 + 1) < 13
3. = 9 < 13

Since 9 is less than 13 so we update d(D) from 13 to 9. The value 9 will be added under the D column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C |  | 10 | 5 | ∞ | ∞ |
| E |  | 8 |  | 14 | 7 |
| B |  | 8 |  | 13 |  |
| D |  |  |  | 9 |  |

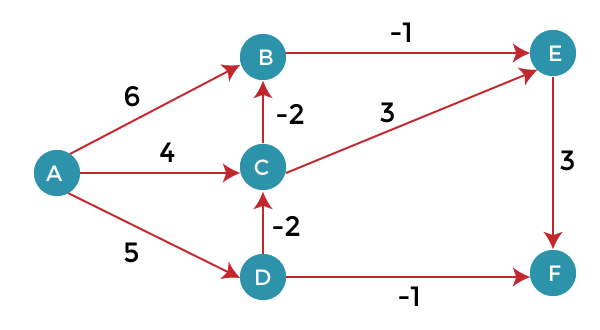
Bellman Ford Algorithm

Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

**Rule of this algorithm**

1. We will go on relaxing all the edges (n - 1) times where,
2. n = number of vertices

**Consider the below graph:**



As we can observe in the above graph that some of the weights are negative. The above graph contains 6 vertices so we will go on relaxing till the 5 vertices. Here, we will relax all the edges 5 times. The loop will iterate 5 times to get the correct answer. If the loop is iterated more than 5 times then also the answer will be the same, i.e., there would be no change in the distance between the vertices.

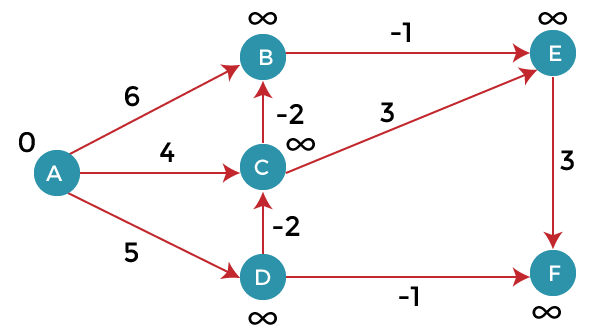
**Relaxing means:**

1. If (d(u) + c(u , v) **<** **d**(v))
2. d(v) = d(u) + c(u , v)

To find the shortest path of the above graph, the first step is note down all the edges which are given below:

(A, B), (A, C), (A, D), (B, E), (C, E), (D, C), (D, F), (E, F), (C, B)

Let's consider the source vertex as 'A'; therefore, the distance value at vertex A is 0 and the distance value at all the other vertices as infinity shown as below:



Since the graph has six vertices so it will have five iterations.

**First iteration**

Consider the edge (A, B). Denote vertex 'A' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:

d(u) = 0

d(v) = ∞

c(u , v) = 6

Since (0 + 6) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 0 + 6 = 6

Therefore, the distance of vertex B is 6.

Consider the edge (A, C). Denote vertex 'A' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:

d(u) = 0

d(v) = ∞

c(u , v) = 4

Since (0 + 4) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 0 + 4 = 4

Therefore, the distance of vertex C is 4.

Consider the edge (A, D). Denote vertex 'A' as 'u' and vertex 'D' as 'v'. Now use the relaxing formula:

d(u) = 0

d(v) = ∞

c(u , v) = 5

Since (0 + 5) is less than ∞, so update

1. d(v) = 0 + 5 = 5 d(v) = d(u) + c(u , v)

Therefore, the distance of vertex D is 5.

Consider the edge (B, E). Denote vertex 'B' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:

d(u) = 6

d(v) = ∞

c(u , v) = -1

Since (6 - 1) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 6 - 1= 5

Therefore, the distance of vertex E is 5.

Consider the edge (C, E). Denote vertex 'C' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:

d(u) = 4

d(v) = 5

c(u , v) = 3

Since (4 + 3) is greater than 5, so there will be no updation. The value at vertex E is 5.

Consider the edge (D, C). Denote vertex 'D' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = 4

c(u , v) = -2

Since (5 -2) is less than 4, so update

1. d(v) = d(u) + c(u , v)

d(v) = 5 - 2 = 3

Therefore, the distance of vertex C is 3.

Consider the edge (D, F). Denote vertex 'D' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = ∞

c(u , v) = -1

Since (5 -1) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 5 - 1 = 4

Therefore, the distance of vertex F is 4.

Consider the edge (E, F). Denote vertex 'E' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = ∞

c(u , v) = 3

Since (5 + 3) is greater than 4, so there would be no updation on the distance value of vertex F.

Consider the edge (C, B). Denote vertex 'C' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:

d(u) = 3

d(v) = 6

c(u , v) = -2

Since (3 - 2) is less than 6, so update

1. d(v) = d(u) + c(u , v)

d(v) = 3 - 2 = 1

Therefore, the distance of vertex B is 1.

Now the first iteration is completed. We move to the second iteration.

**Second iteration:**

In the second iteration, we again check all the edges. The first edge is (A, B). Since (0 + 6) is greater than 1 so there would be no updation in the vertex B.

The next edge is (A, C). Since (0 + 4) is greater than 3 so there would be no updation in the vertex C.

The next edge is (A, D). Since (0 + 5) equals to 5 so there would be no updation in the vertex D.

The next edge is (B, E). Since (1 - 1) equals to 0 which is less than 5 so update:

d(v) = d(u) + c(u, v)

d(E) = d(B) +c(B , E)

= 1 - 1 = 0

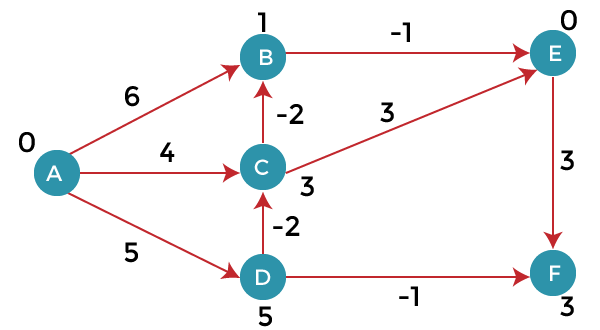
The next edge is (C, E). Since (3 + 3) equals to 6 which is greater than 5 so there would be no updation in the vertex E.

The next edge is (D, C). Since (5 - 2) equals to 3 so there would be no updation in the vertex C.

The next edge is (D, F). Since (5 - 1) equals to 4 so there would be no updation in the vertex F.

The next edge is (E, F). Since (5 + 3) equals to 8 which is greater than 4 so there would be no updation in the vertex F.

The next edge is (C, B). Since (3 - 2) equals to 1` so there would be no updation in the vertex B.

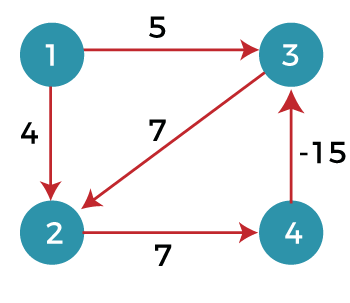
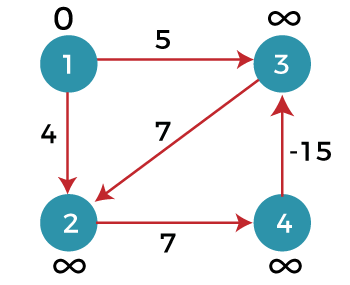


**Third iteration**

We will perform the same steps as we did in the previous iterations. We will observe that there will be no updation in the distance of vertices.

1. The following are the distances of vertices:
2. A: 0
3. B: 1
4. C: 3
5. D: 5
6. E: 0
7. F: 3

**Drawbacks of Bellman ford algorithm**

* The bellman ford algorithm does not produce a correct answer if the sum of the edges of a cycle is negative. Let's understand this property through an example. Consider the below graph.  
  
* In the above graph, we consider vertex 1 as the source vertex and provides 0 value to it. We provide infinity value to other vertices shown as below:  
    
  Edges can be written as:  
  (1, 3), (1, 2), (2, 4), (3, 2), (4, 3)

**First iteration**

**Consider the edge (1, 3). Denote vertex '1' as 'u' and vertex '3' as 'v'. Now use the relaxing formula:**

d(u) = 0

d(v) = ∞

c(u , v) = 5

Since (0 + 5) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 0 + 5 = 5

Therefore, the distance of vertex 3 is 5.

**Consider the edge (1, 2). Denote vertex '1' as 'u' and vertex '2' as 'v'. Now use the relaxing formula:**

d(u) = 0

d(v) = ∞

c(u , v) = 4

Since (0 + 4) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 0 + 4 = 4

Therefore, the distance of vertex 2 is 4.

**Consider the edge (3, 2). Denote vertex '3' as 'u' and vertex '2' as 'v'. Now use the relaxing formula:**

d(u) = 5

d(v) = 4

c(u , v) = 7

Since (5 + 7) is greater than 4, so there would be no updation in the vertex 2.

**Consider the edge (2, 4). Denote vertex '2' as 'u' and vertex '4' as 'v'. Now use the relaxing formula:**

d(u) = 4

d(v) = ∞

c(u , v) = 7

Since (4 + 7) equals to 11 which is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 4 + 7 = 11

Therefore, the distance of vertex 4 is 11.

**Consider the edge (4, 3). Denote vertex '4' as 'u' and vertex '3' as 'v'. Now use the relaxing formula:**

d(u) = 11

d(v) = 5

c(u , v) = -15

Since (11 - 15) equals to -4 which is less than 5, so update

1. d(v) = d(u) + c(u , v)

d(v) = 11 - 15 = -4

Therefore, the distance of vertex 3 is -4.

Now we move to the second iteration.

**Second iteration**

Now, again we will check all the edges. The first edge is (1, 3). Since (0 + 5) equals to 5 which is greater than -4 so there would be no updation in the vertex 3.

The next edge is (1, 2). Since (0 + 4) equals to 4 so there would be no updation in the vertex 2.

The next edge is (3, 2). Since (-4 + 7) equals to 3 which is less than 4 so update:

d(v) = d(u) + c(u, v)

d(2) = d(3) +c(3, 2)

= -4 + 7 = 3

Therefore, the value at vertex 2 is 3.

The next edge is (2, 4). Since ( 3+7) equals to 10 which is less than 11 so update

d(v) = d(u) + c(u, v)

d(4) = d(2) +c(2, 4)

= 3 + 7 = 10

Therefore, the value at vertex 4 is 10.

The next edge is (4, 3). Since (10 - 15) equals to -5 which is less than -4 so update:

d(v) = d(u) + c(u, v)

d(3) = d(4) +c(4, 3)

= 10 - 15 = -5

Therefore, the value at vertex 3 is -5.

Now we move to the third iteration.

**Third iteration**

Now again we will check all the edges. The first edge is (1, 3). Since (0 + 5) equals to 5 which is greater than -5 so there would be no updation in the vertex 3.

The next edge is (1, 2). Since (0 + 4) equals to 4 which is greater than 3 so there would be no updation in the vertex 2.

The next edge is (3, 2). Since (-5 + 7) equals to 2 which is less than 3 so update:

d(v) = d(u) + c(u, v)

d(2) = d(3) +c(3, 2)

= -5 + 7 = 2

Therefore, the value at vertex 2 is 2.

The next edge is (2, 4). Since (2 + 7) equals to 9 which is less than 10 so update:

d(v) = d(u) + c(u, v)

d(4) = d(2) +c(2, 4)

= 2 + 7 = 9

Therefore, the value at vertex 4 is 9.

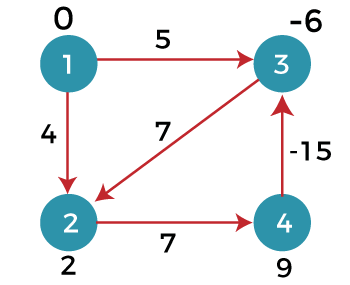
The next edge is (4, 3). Since (9 - 15) equals to -6 which is less than -5 so update:

d(v) = d(u) + c(u, v)

d(3) = d(4) +c(4, 3)

= 9 - 15 = -6

Therefore, the value at vertex 3 is -6.



Since the graph contains 4 vertices, so according to the bellman ford algorithm, there would be only 3 iterations. If we try to perform 4th iteration on the graph, the distance of the vertices from the given vertex should not change. If the distance varies, it means that the bellman ford algorithm is not providing the correct answer.

**4th iteration**

The first edge is (1, 3). Since (0 +5) equals to 5 which is greater than -6 so there would be no change in the vertex 3.

The next edge is (1, 2). Since (0 + 4) is greater than 2 so there would be no updation.

The next edge is (3, 2). Since (-6 + 7) equals to 1 which is less than 3 so update:

d(v) = d(u) + c(u, v)

d(2) = d(3) +c(3, 2)

= -6 + 7 = 1

In this case, the value of the vertex is updated. So, we conclude that the bellman ford algorithm does not work when the graph contains the negative weight cycle.

Therefore, the value at vertex 2 is 1.