## Assignment No:

## Title of the Assignment: Greedy Search Algorithms

**Problem statement:** Implement Greedy search algorithm for any ofthe following application:

1. Selection Sort
2. Minimum Spanning Tree
3. Single-Source Shortest Path Problem
4. Job Scheduling Problem
5. Prim's Minimal Spanning Tree Algorithm
6. Kruskal's Minimal Spanning Tree Algorithm
7. Dijkstra's Minimal Spanning Tree Algorithm

.

## Objective:

* To understand the concept of Greedy Search Algorithms.
* To implement algorithm of Selection Sort for given set of Numbers.

## Theory:

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

* 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.

o

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.

# Selection Sort

The selection sort enhances the bubble sort by making only a single swap for each pass through the rundown. In order to do this, a selection sort searches for the biggest value as it makes a pass and, after finishing the pass, places it in the best possible area. Similarly, as with a bubble sort, after the first pass, the biggest item is in the right place. After the second pass, the following biggest is set up. This procedure proceeds and requires n-1 goes to sort n item since the last item must be set up after the (n-1) th pass.

ALGORITHM: SELECTION SORT (A)

1. k ← length [A]
2. **for** j ←1 to n-1
3. smallest ← j
4. **for** I ← j + 1 to k
5. **if** A [i] < A [ smallest]
6. then smallest ← i
7. exchange (A [j], A [smallest]) How Selection Sort works
   1. In the selection sort, first of all, we set the initial element as a **minimum**.
   2. Now we will compare the minimum with the second element. If the second element turns out to be smaller than the minimum, we will swap them, followed by assigning to a minimum to the third element.
   3. Else if the second element is greater than the minimum, which is our first element, then we will do nothing and move on to the third element and then compare it with the minimum.

We will repeat this process until we reach the last element.

* 1. After the completion of each iteration, we will notice that our minimum has reached the start of the unsorted list.
  2. For each iteration, we will start the indexing from the first element of the unsorted list. We will repeat the Steps from 1 to 4 until the list gets sorted or all the elements get correctly positioned.

Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

A [] = (7, 4, 3, 6, 5). A [] =

**1st Iteration:**

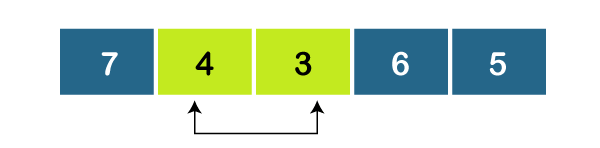
Set minimum = 7

* Compare a0 and a1



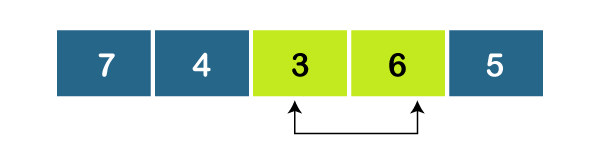
As, a0 > a1, set minimum = 4.

* Compare a1 and a2



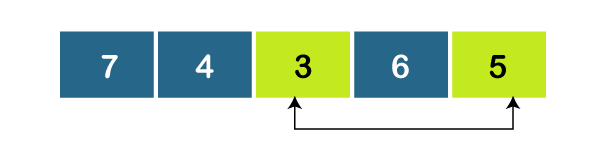
As, a1 > a2, set minimum = 3.

* Compare a2 and a3



As, a2 < a3, set minimum= 3.

* Compare a2 and a4



As, a2 < a4, set minimum =3.

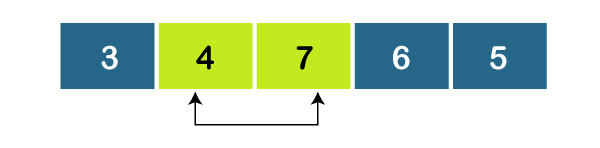
Since 3 is the smallest element, so we will swap a0 and a2.



**2nd Iteration:**

Set minimum = 4

* Compare a1 and a2



As, a1 < a2, set minimum = 4.

* Compare a1 and a3



As, A[1] < A[3], set minimum = 4.

* Compare a1 and a4



Again, a1 < a4, set minimum = 4.

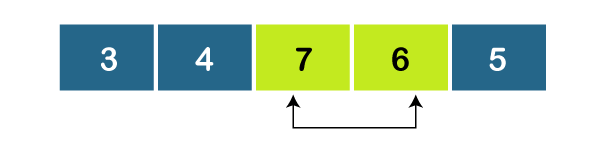
Since the minimum is already placed in the correct position, so there will be no swapping.



**3rd Iteration:**

Set minimum = 7

* Compare a2 and a3



As, a2 > a3, set minimum = 6.

* Compare a3 and a4



As, a3 > a4, set minimum = 5.

Since 5 is the smallest element among the leftover unsorted elements, so we will swap 7 and 5.



**4th Iteration:**

Set minimum = 6

* Compare a3 and a4



As a3 < a4, set minimum = 6.

Since the minimum is already placed in the correct position, so there will be no swapping.

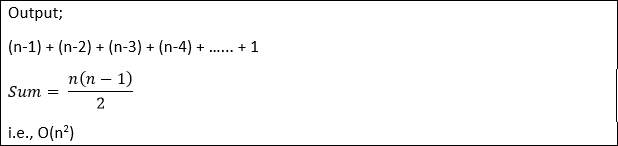


Complexity Analysis of Selection Sort

**Input:** Given **n** input elements.

**Output:** Number of steps incurred to sort a list.

**Logic:** If we are given n elements, then in the first pass, it will do **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;



Therefore, the selection sort algorithm encompasses a time complexity of **O(n2)** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

* **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of **O(n2)** for the already sorted array.
* **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is **O(n2)**, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
* **Worst Case Complexity:** The worst-case time complexity is also **O(n2)**, which occurs when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is **O(n2)** in all three cases. This is because, in each step, we are required to find **minimum** elements so that it can be placed in the correct position. Once we trace the complete array, we will get our minimum element.

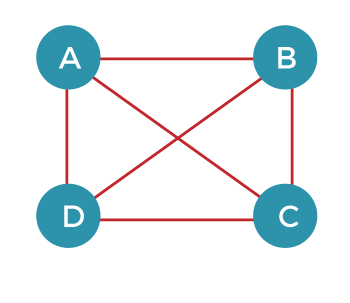
**Conclusion:**

We have implemented Selection Sort for given numbers.

# Minimum Spanning Tree

Before knowing about the minimum spanning tree, we should know about the spanning tree.

## To understand the concept of spanning tree, consider the below graph:



The above graph can be represented as G(V, E), where 'V' is the number of vertices, and 'E' is the number of edges. The spanning tree of the above graph would be represented as G`(V`, E`). In this case, V` = V means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph, but the number of edges would be different. The number of edges in the spanning tree is the subset of the number of edges in the original graph. Therefore, the number of edges can be written as:

E` € E

Competitive questions on Structures in Hindi Keep Watching

It can also be written as:

E` = |V| - 1

Two conditions exist in the spanning tree, which is as follows:

* The number of vertices in the spanning tree would be the same as the number of vertices in the original graph.

## V` = V

* The number of edges in the spanning tree would be equal to the number of edges minus 1.

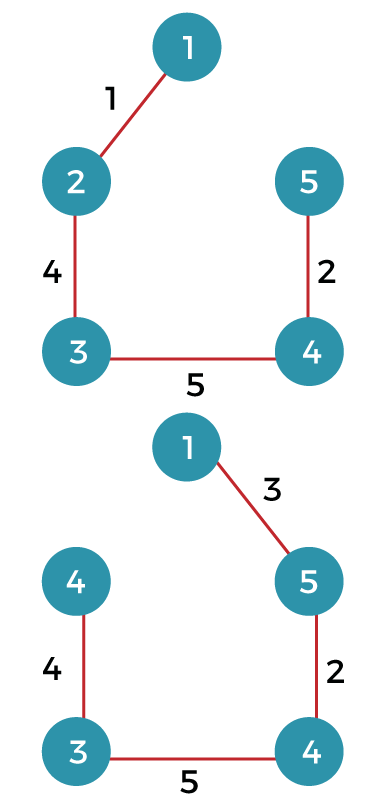
## E` = |V| - 1

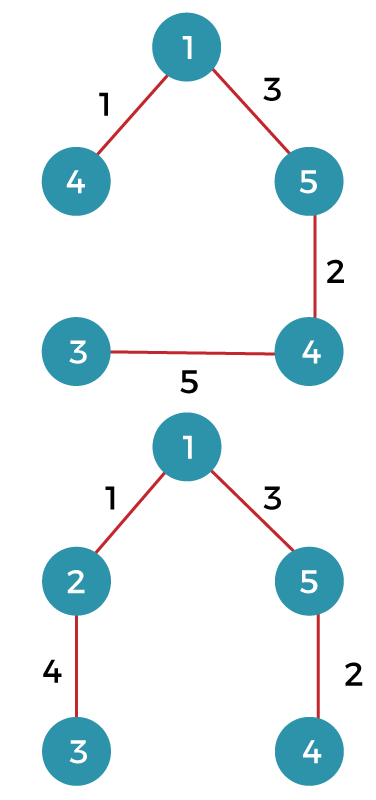
* The spanning tree should not contain any cycle.
* The spanning tree should not be disconnected.

Note: A graph can have more than one spanning tree.

Consider the below graph:

The above graph contains 5 vertices. As we know, the vertices in the spanning tree would be the same as the graph; therefore, V` is equal 5. The number of edges in the spanning tree would be equal to (5 - 1), i.e., 4. The following are the possible spanning trees:





## What is a minimum spanning tree?

The minimum spanning tree is a spanning tree whose sum of the edges is minimum. Consider the below graph that contains the edge weight:

## The following are the spanning trees that we can make from the above graph.

* The first spanning tree is a tree in which we have removed the edge between the vertices 1 and 5 shown as below:

The sum of the edges of the above tree is (1 + 4 + 5 + 2): 12

* The second spanning tree is a tree in which we have removed the edge between the vertices 1 and 2 shown as below:

The sum of the edges of the above tree is (3 + 2 + 5 + 4) : 14

* The third spanning tree is a tree in which we have removed the edge between the vertices 2 and 3 shown as below:

The sum of the edges of the above tree is (1 + 3 + 2 + 5) : 11

* The fourth spanning tree is a tree in which we have removed the edge between the vertices 3 and 4 shown as below:

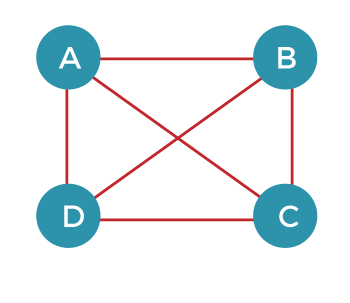
The sum of the edges of the above tree is (1 + 3 + 2 + 4) : 10. The edge cost 10 is minimum so it is a minimum spanning tree.

## General properties of minimum spanning tree:

* If we remove any edge from the spanning tree, then it becomes disconnected. Therefore, we cannot remove any edge from the spanning tree.
* If we add an edge to the spanning tree then it creates a loop. Therefore, we cannot add any edge to the spanning tree.
* In a graph, each edge has a distinct weight, then there exists only a single and unique minimum spanning tree. If the edge weight is not distinct, then there can be more than one minimum spanning tree.
* A complete undirected graph can have an nn-2 number of spanning trees.
* Every connected and undirected graph contains atleast one spanning tree.
* The disconnected graph does not have any spanning tree.
* In a complete graph, we can remove maximum (e-n+1) edges to construct a spanning tree.

## Let's understand the last property through an example.

Consider the complete graph which is given below:



The number of spanning trees that can be made from the above complete graph equals to nn-2 = 44-2 = 16.

Therefore, 16 spanning trees can be created from the above graph.

The maximum number of edges that can be removed to construct a spanning tree equals to e-n+1 = 6 - 4 + 1 = 3.

## Application of Minimum Spanning Tree

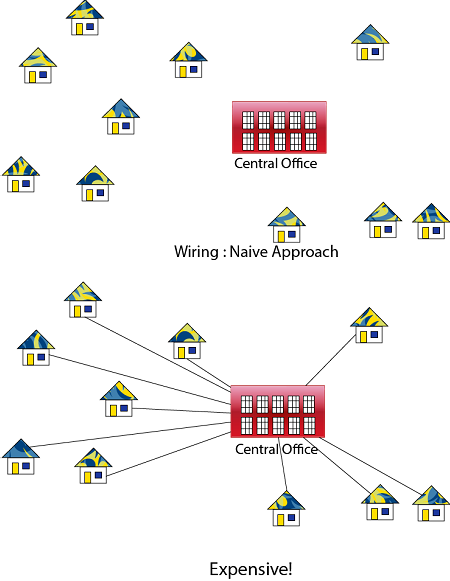
1. Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost.

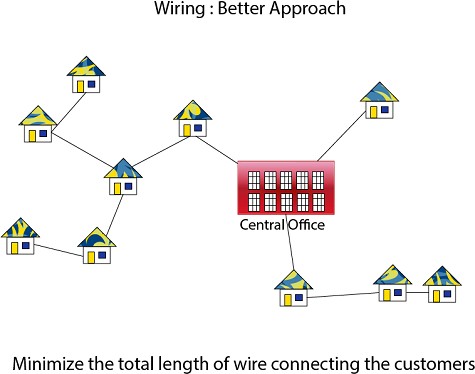
The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.

1. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
2. Designing Local Area Networks.
3. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
4. Suppose you want to apply a set of houses with
   * Electric Power
   * Water
   * Telephone lines
   * Sewage lines

To reduce cost, you can connect houses with minimum cost spanning trees.

## For Example, Problem laying Telephone Wire.





Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

**Conclusion:**

We have implemented Minimum Spanning tree for given input.

# Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

## If the graph is not linked, then it finds a Minimum Spanning Tree. Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

## MST- KRUSKAL (G, w)

1. A ← ∅

1. for each vertex v ∈ V [G]
2. do MAKE - SET (v)
3. sort the edges of E into non decreasing order by weight w
4. for each edge (u, v) ∈ E, taken in non decreasing order by weight
5. do if FIND-SET (μ) ≠ if FIND-SET (v)
6. then A ← A ∪ {(u, v)}
7. UNION (u, v)
8. return A

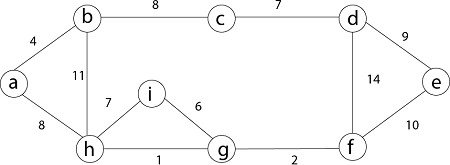
**Analysis:** Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

* + E is at most V2 and log V2= 2 x log V is O (log V).
  + If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

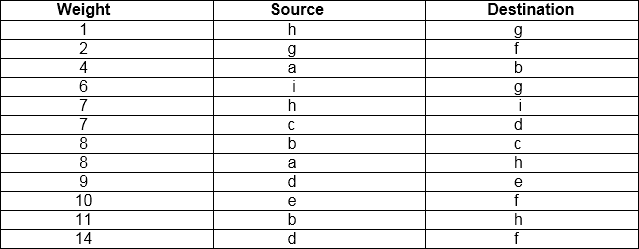
1. O (E log E) = O (E log V).

**For Example:** Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



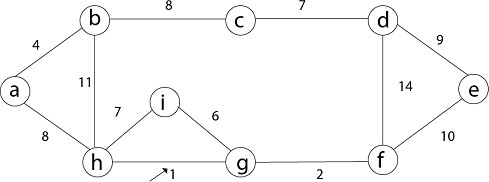
**Solution:** First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

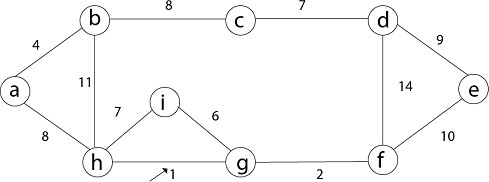


Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

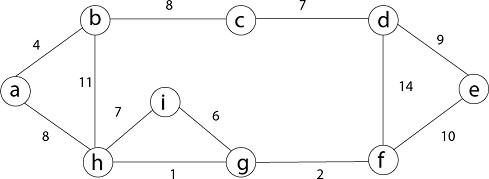
**Step1:** So, first take (h, g) edge



**Step 2:** then (g, f) edge.

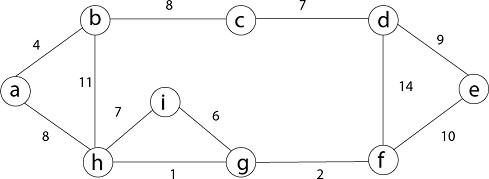


**Step 3:** then (a, b) and (i, g) edges are considered, and the forest becomes



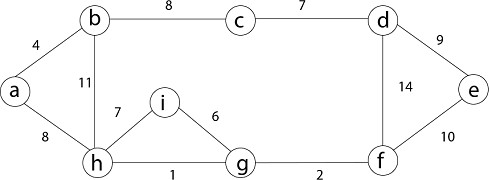
**Step 4:** Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



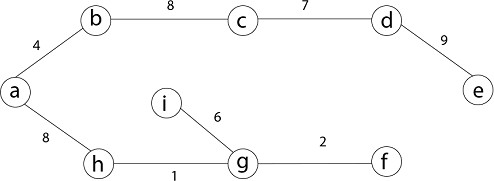
**Step 5:** In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

**Step 6:** After that edge (d, f) and the final spanning tree is shown as in dark lines.



**Step 7:** This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

**Conclusion:**

We have implemented Kruskal’s algorithm for given input graph.

# Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

* + Contain vertices already included in MST.
  + Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

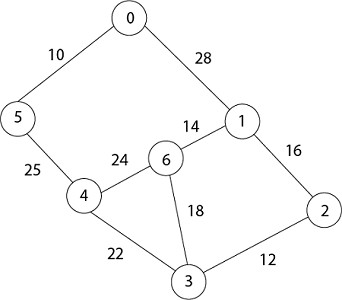
Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
   1. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.
   2. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

## MST-PRIM (G, w, r)

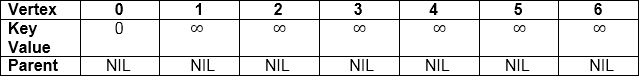
1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q ? ∅
7. do u ← EXTRACT - MIN (Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u, v) < key [v]
10. then π [v] ← u
11. key [v] ← w (u, v)

**Example:** Generate minimum cost spanning tree for the following graph using Prim's algorithm.



**Solution:** In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.



1. Taking 0 as starting vertex
2. Root = 0

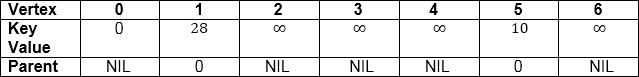
3. Adj [0] = 5, 1

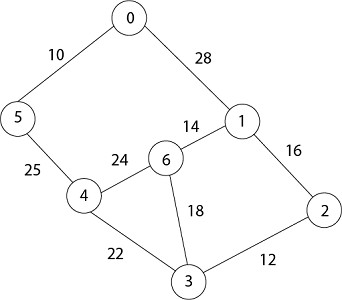
4. Parent, π [5] = 0 and π [1] = 0

5. Key [5] = ∞ and key [1] = ∞ 6. w [0, 5) = 10 and w (0,1) = 28

7. w (u, v) < key [5] , w (u, v) < key [1] 8. Key [5] = 10 and key [1] = 28

9. So update key value of 5 and 1 is:





Now by EXTRACT\_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

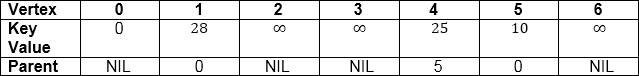
1. Adj [5] = {0, 4} and 0 is already in heap

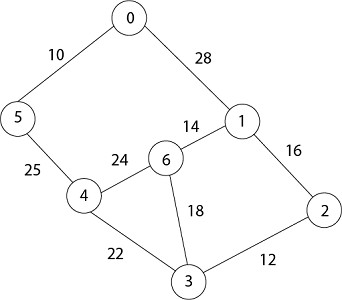
2. Taking 4, key [4] = ∞ π [4] = 5

3. (u, v) < key [v] then key [4] = 25 4. w (5,4) = 25

5. w (5,4) < key [4]

6. date key value and parent of 4.





Now remove 4 because key [4] = 25 which is minimum, so u =4 1. Adj [4] = {6, 3}

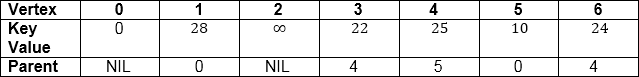
2. Key [3] = ∞ key [6] = ∞ 3. w (4,3) = 22 w (4,6) = 24

4. w (u, v) < key [v] w (u, v) < key [v] 5. w (4,3) < key [3] w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

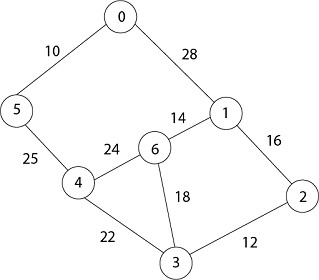
And the parent of 3, 6 as 4.

1. π[3]= 4 π[6]= 4



1. u = EXTRACT\_MIN (3, 6) [key [3] < key [6]] 2. u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.



1. Adj [3] = {4, 6, 2}

1. 4 is already in heap
2. 4 ≠ Q key [6] = 24 now becomes key [6] = 18 4. Key [2] = ∞ key [6] = 24

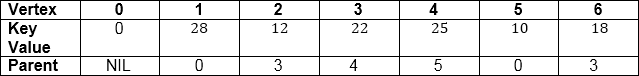
5. w (3, 2) = 12 w (3, 6) = 18

6. w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

1. π [2] = 3 π[6]=3

Now by EXTRACT\_MIN (Q) Removes 2, because key [2] = 12 is minimum.



1. u = EXTRACT\_MIN (2, 6) 2. u = 2 [key [2] < key [6]] 3. 12 < 18

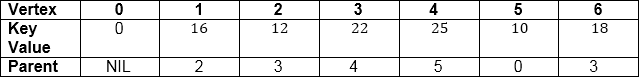
4. Now the root is 2 5. Adj [2] = {3, 1}

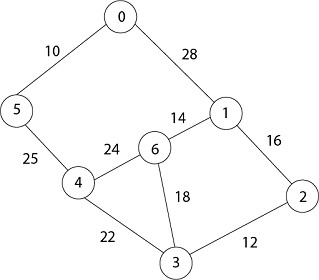
1. 3 is already in a heap
2. Taking 1, key [1] = 28 8. w (2,1) = 16

9. w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

1. π[1]= 2





Now by EXTRACT\_MIN (Q) Removes 1 because key [1] = 16 is minimum.

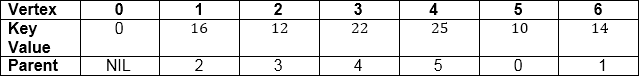
1. Adj [1] = {0, 6, 2}

1. 0 and 2 are already in heap.
2. Taking 6, key [6] = 18 4. w [1, 6] = 14

5. w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

1. Π [6] = 1

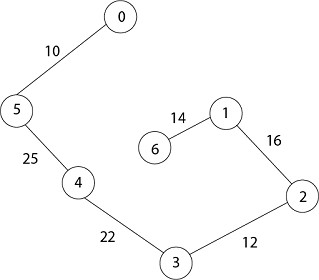


Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

1. 0 → 5 → 4 → 3 → 2 → 1 → 6

2. [Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

## Thus the final spanning Tree is



**Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99**

**Conclusion:**

We have implemented Prim’s algorithm for given input graph.

# Single Source Shortest Paths

In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = (v0,v1,. vk) is

the total of the weights of its constituent edges:



We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

The **breadth-first- search algorithm** is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

In a **Single Source Shortest Paths Problem**, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.

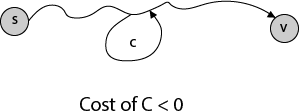
Variants:

There are some variants of the shortest path problem.

* **Single- destination shortest - paths problem:** Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.
* **Single - pair shortest - path problem:** Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
* **All - pairs shortest - paths problem:** Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.

Shortest Path: Existence:

If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.



## Single Source Shortest Path in a directed Acyclic Graphs

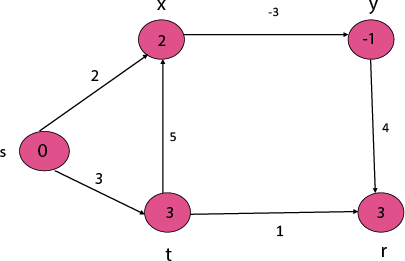
By relaxing the edges of a weighted DAG (Directed Acyclic Graph) G = (V, E) according to a topological sort of its vertices, we can figure out shortest paths from a single source in ∅(V+E) time. Shortest paths are always well described in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

## DAG - SHORTEST - PATHS (G, w, s)

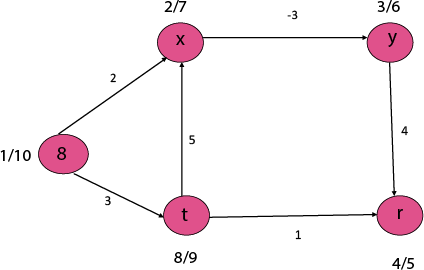
* 1. Topologically sort the vertices of G.
  2. INITIALIZE - SINGLE- SOURCE (G, s)
  3. for each vertex u taken in topologically sorted order
  4. do for each vertex v ∈ Adj [u]
  5. do RELAX (u, v, w)

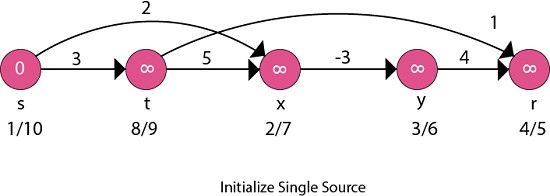
The running time of this data is determined by line 1 and by the for loop of lines 3 - 5. The topological sort can be implemented in ∅ (V + E) time. In the for loop of lines 3 - 5, as in Dijkstra's algorithm, there is one repetition per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, we use only O (1) time per edge. The running time is thus ∅ (V + E), which is linear in the size of an adjacency list depiction of the graph.

## Example:

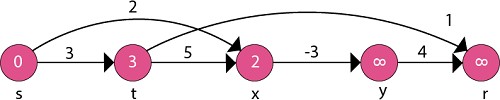


**Step1:** To topologically sort vertices apply **DFS (Depth First Search)** and then arrange vertices in linear order by decreasing order of finish time.



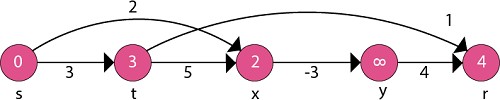


Now, take each vertex in topologically sorted order and relax each edge.

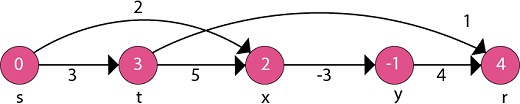


1. adj [s] →t, x 2. 0 + 3 < ∞

3. d [t] ← 3 4. 0 + 2 < ∞

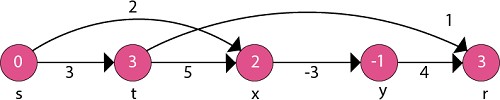
5. d [x] ← 2

1. adj [t] → r, x 2. 3 + 1 < ∞

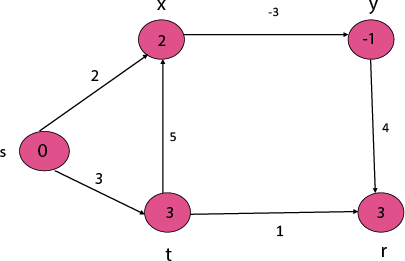
3. d [r] ← 4 4. 3 + 5 ≤ 2

1. adj [x] → y

2. 2 - 3 < ∞

3. d [y] ← -1

1. adj [y] → r 2. -1 + 4 < 4 3. 3 <4

4. d [r] ← 3

## Thus the Shortest Path is:

1. s to x is 2
2. s to y is -1
3. s to t is 3
4. s to r is 3

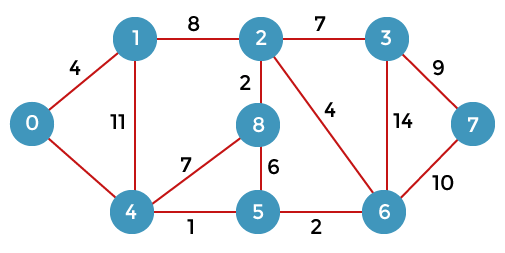
**Conclusion:**

We have implemented Single Source Shortest path algorithm for given graph.

# Dijkstra Algorithm

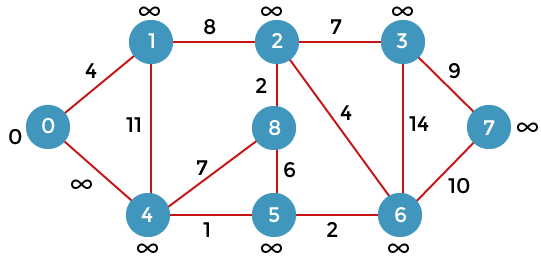
Dijkstra algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

## 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:

## d(x, y) = d(x) + c(x, y) < d(y)

= **(0 + 4)** < ∞

= **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:

## {if( d(u) + c(u, v) < d(v))

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

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

## d(x, y) = d(x) + c(x, y) < d(y)

= **(0 + 8)** < ∞

= **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'.

## d(x, y) = d(x) + c(x, y) < d(y)

= **(4 + 8)** < ∞

= **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'.

## d(x, y) = d(x) + c(x, y) < d(y)

**= (4 + 11) < 8**

## = 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'.

## d(x, y) = d(x) + c(x, y) < d(y)

**= (8 + 7) < ∞**

= **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'.

## d(x, y) = d(x) + c(x, y) < d(y)

= **(8 + 1)** < ∞

= **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'.

## d(x, y) = d(x) + c(x, y) < d(y)

**= (9 + 15) < 15**

## = 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'.

## d(x, y) = d(x) + c(x, y) < d(y)

= **(9 + 2)** < ∞

= **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'. d(x, y) = d(x) + c(x, y) < d(y)

= (11 + 4) < 12

= 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'. d(x, y) = d(x) + c(x, y) < d(y)

= (11 + 14) < ∞

= 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'. d(x, y) = d(x) + c(x, y) < d(y)

= (11 + 10) < ∞

= 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'. d(x, y) = d(x) + c(x, y) < d(y)

= (12 + 2) < 15

= 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'. d(x, y) = d(x) + c(x, y) < d(y)

= (12 + 4) < 11

= 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'. d(x, y) = d(x) + c(x, y) < d(y)

= (12 + 7) < 25

= 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.

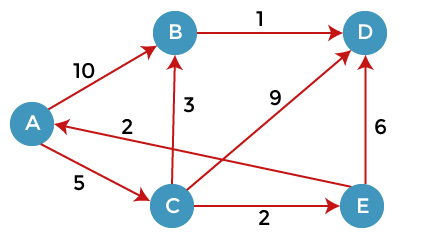
Now, we consider the vertex 7. Consider the vertex 3 as 'x' and 7 as 'y'. d(x, y) = d(x) + c(x, y) < d(y)

= (19 + 9) < 21

= 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))

1. 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'.

d(x, y) = d(x) + c(x, y) < d(y)

= (5 + 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'.

d(x, y) = d(x) + c(x, y) < d(y)

= (5 + 9) < ∞

= 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'.

d(x, y) = d(x) + c(x, y) < d(y)

= (5 + 2) < ∞

= 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.

d(x, y) = d(x) + c(x, y) < d(y)

= (7 + 6) < 14

= 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.

d(x, y) = d(x) + c(x, y) < d(y)

= (8 + 1) < 13

= 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 |  |

**Conclusion:**

We have implemented Dijkstra’s Shortest path algorithm for given graph.

# Job Sequencing Problem

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

## Examples:

Input: Four Jobs with following

deadlines and profits JobID Deadline Profit a 4 20

b 1 10

c 1 40

d 1 30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID Deadline Profit a 2 100

b 1 19

c 2 27

d 1 25

e 3 15

Output: Following is maximum profit sequence of jobs

c, a, e

A **Simple Solution** is to generate all subsets of a given set of jobs and check individual subsets for the feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential.

This is a standard [Greedy Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/) problem. Following is the algorithm.

* 1. *Sort all jobs in decreasing order of profit.*
  2. *Iterate on jobs in decreasing order of profit.For each job , do the following :*
     1. *Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in this slot and mark this slot filled.*
     2. *If no such i exists, then ignore the job.*

**Conclusion:**

We have implemented Job Sequencing algorithm for given input.