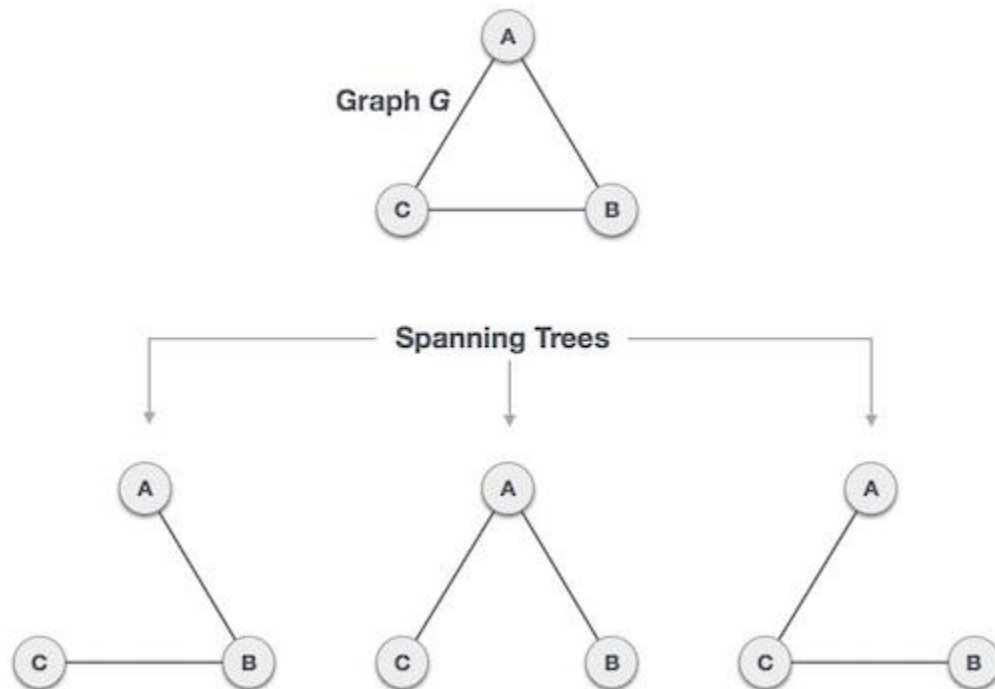


Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.

- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

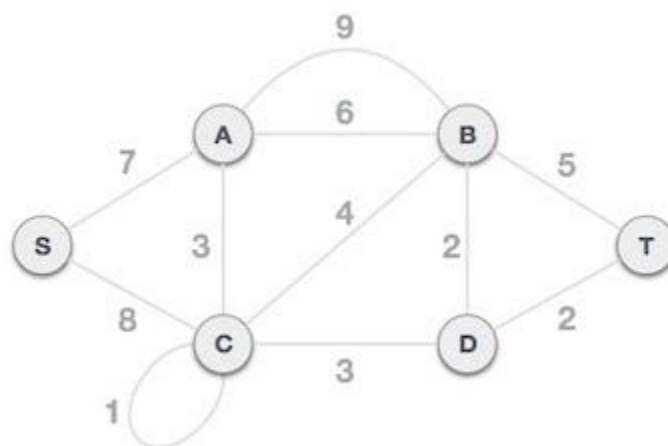
- Spanning tree has **$n-1$** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **$e - n + 1$** edges, we can construct a spanning tree.
- A complete graph can have maximum **n^{n-2}** number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Kruskal's Spanning Tree Algorithm

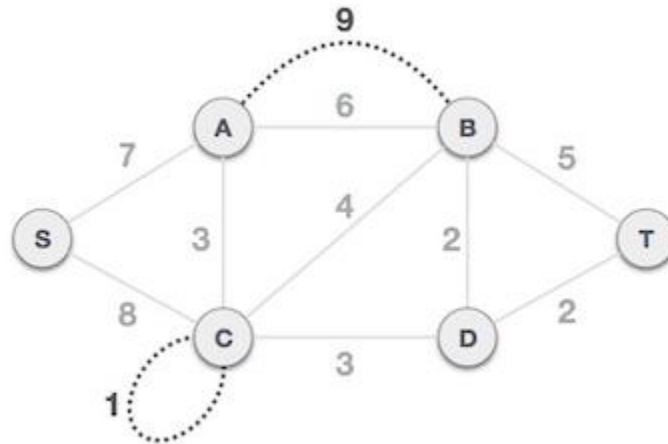
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example –

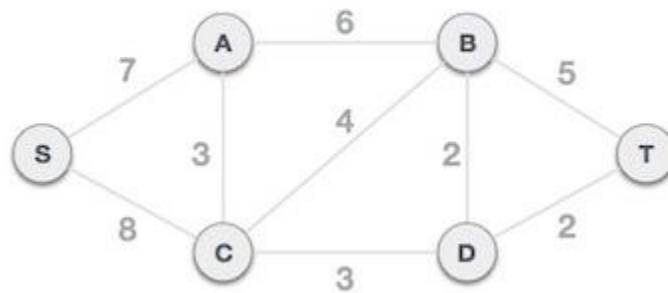


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



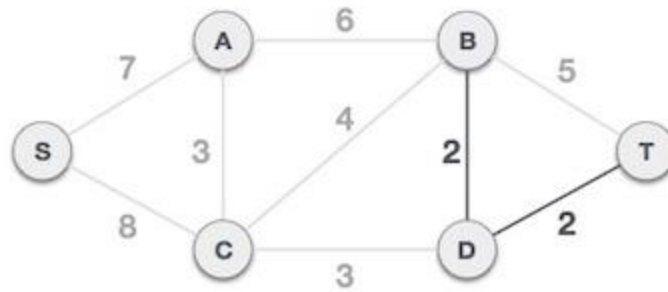
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
|------|------|------|------|------|------|------|------|------|
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |

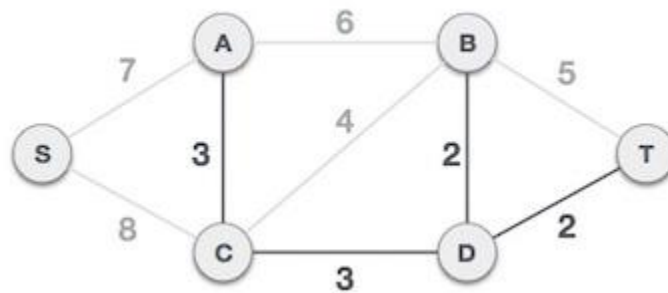
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

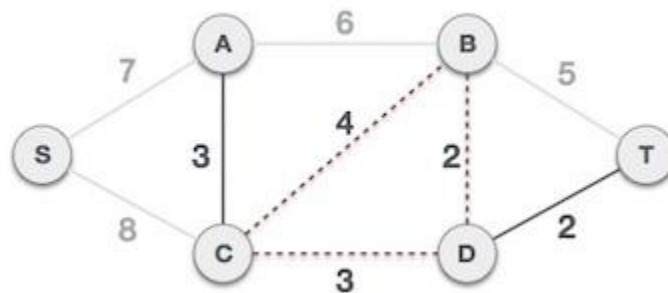


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

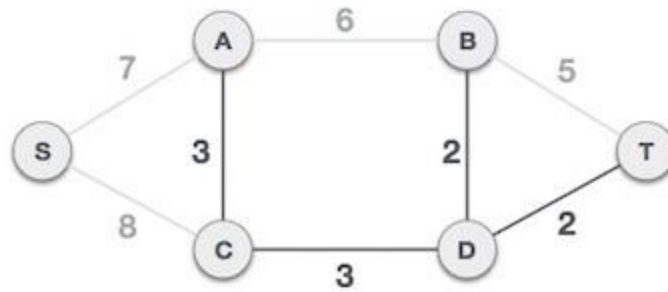
Next cost is 3, and associated edges are A,C and C,D. We add them again –



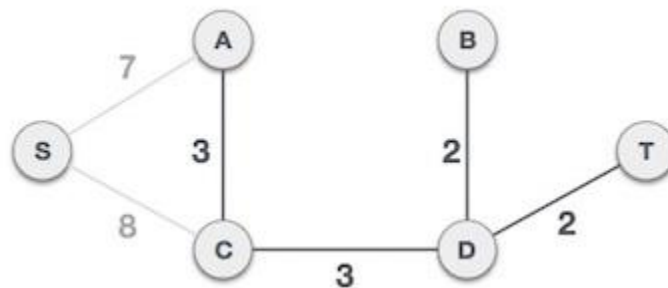
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



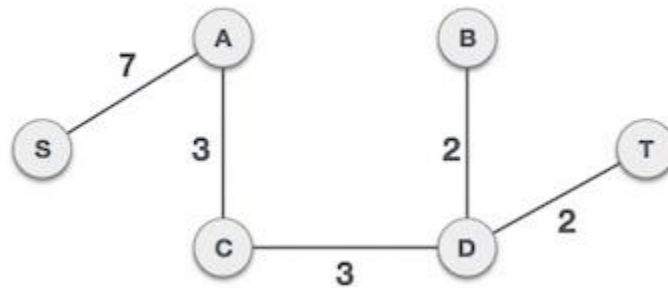
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



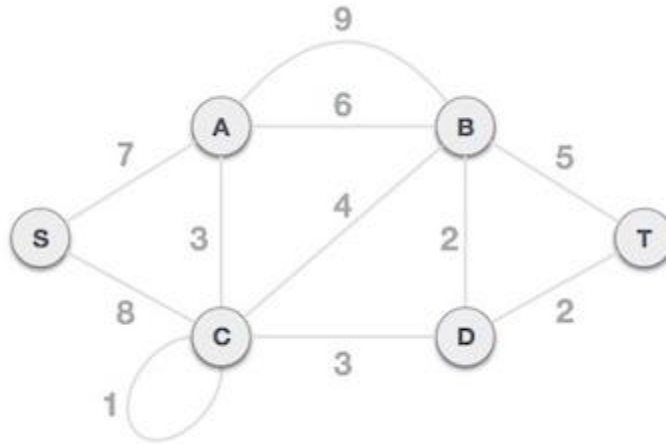
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's Spanning Tree Algorithm

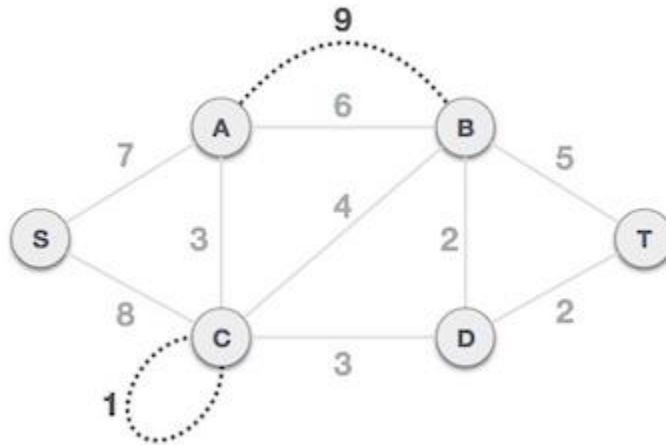
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

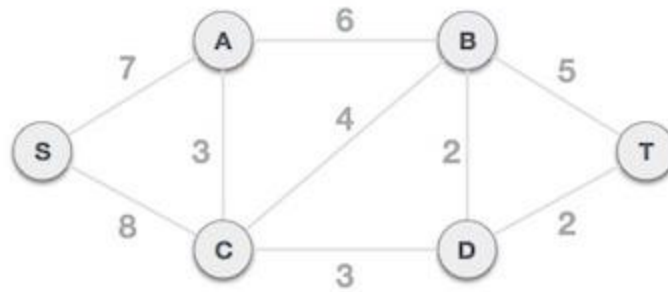
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

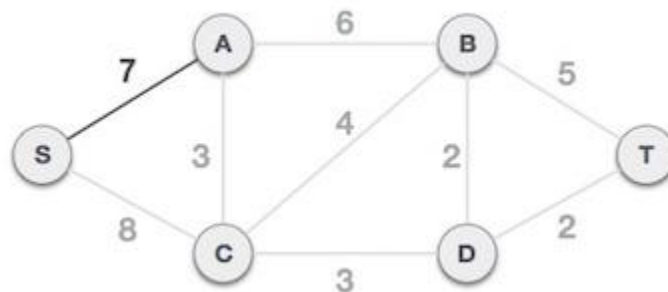


Step 2 - Choose any arbitrary node as root node

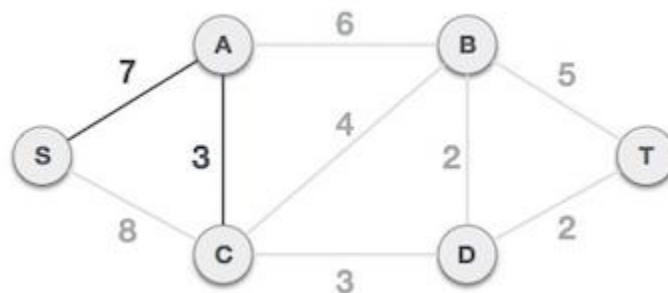
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

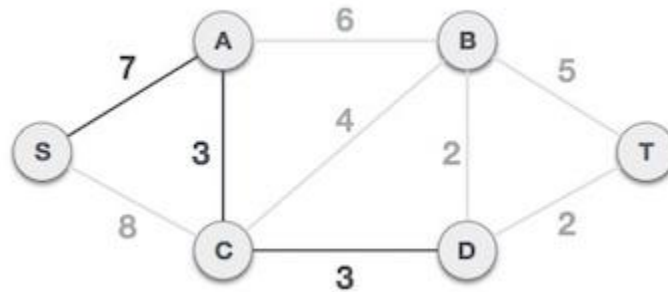
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



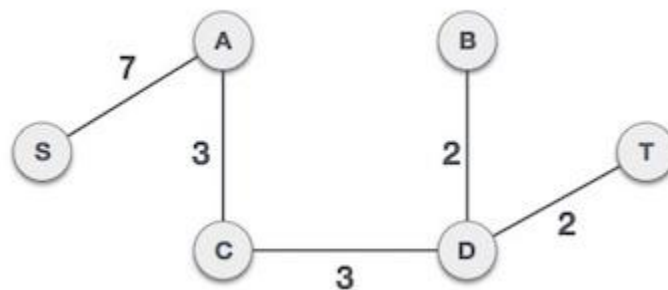
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

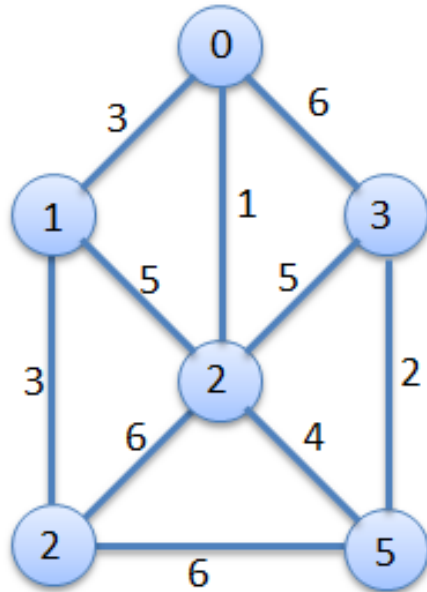
Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.

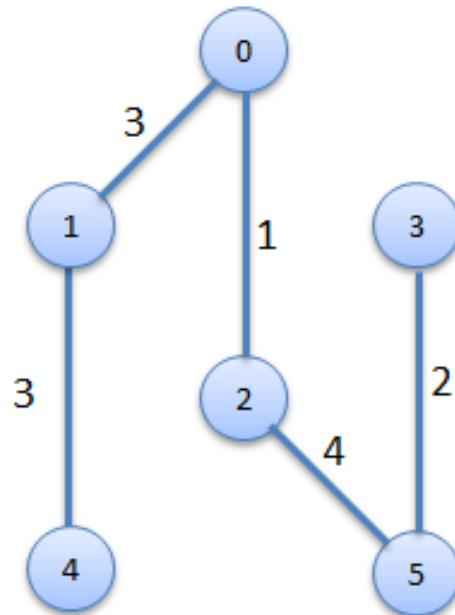
It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

This algorithm is directly based on the MST(minimum spanning tree) property.

Example



A Simple Weighted Graph

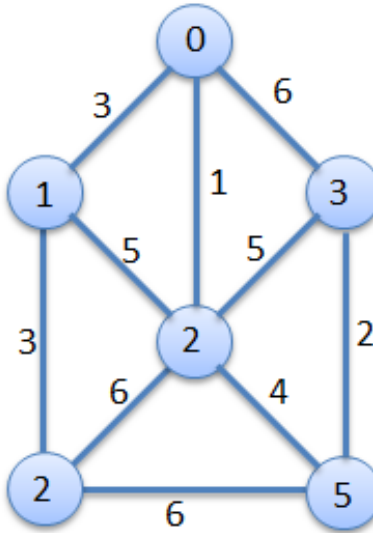


Minimum-Cost Spanning Tree

Kruskal's Algorithm

```
1. MST-KRUSKAL( $G, w$ )
2. 1.    $A \leftarrow \emptyset$ 
3. 2.   for each vertex  $v \in V[G]$ 
4. 3.     do MAKE-SET( $v$ )
5. 4.   sort the edges of  $E$  into nondecreasing order by weight  $w$ 
6. 5.   for each edge  $(u, v) \in E$ , taken in nondecreasing order by
   weight
7. 6.     do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
8. 7.       then  $A \leftarrow A \cup \{(u, v)\}$ 
9. 8.       UNION( $u, v$ )
10. 9.   return  $A$ 
```

Example

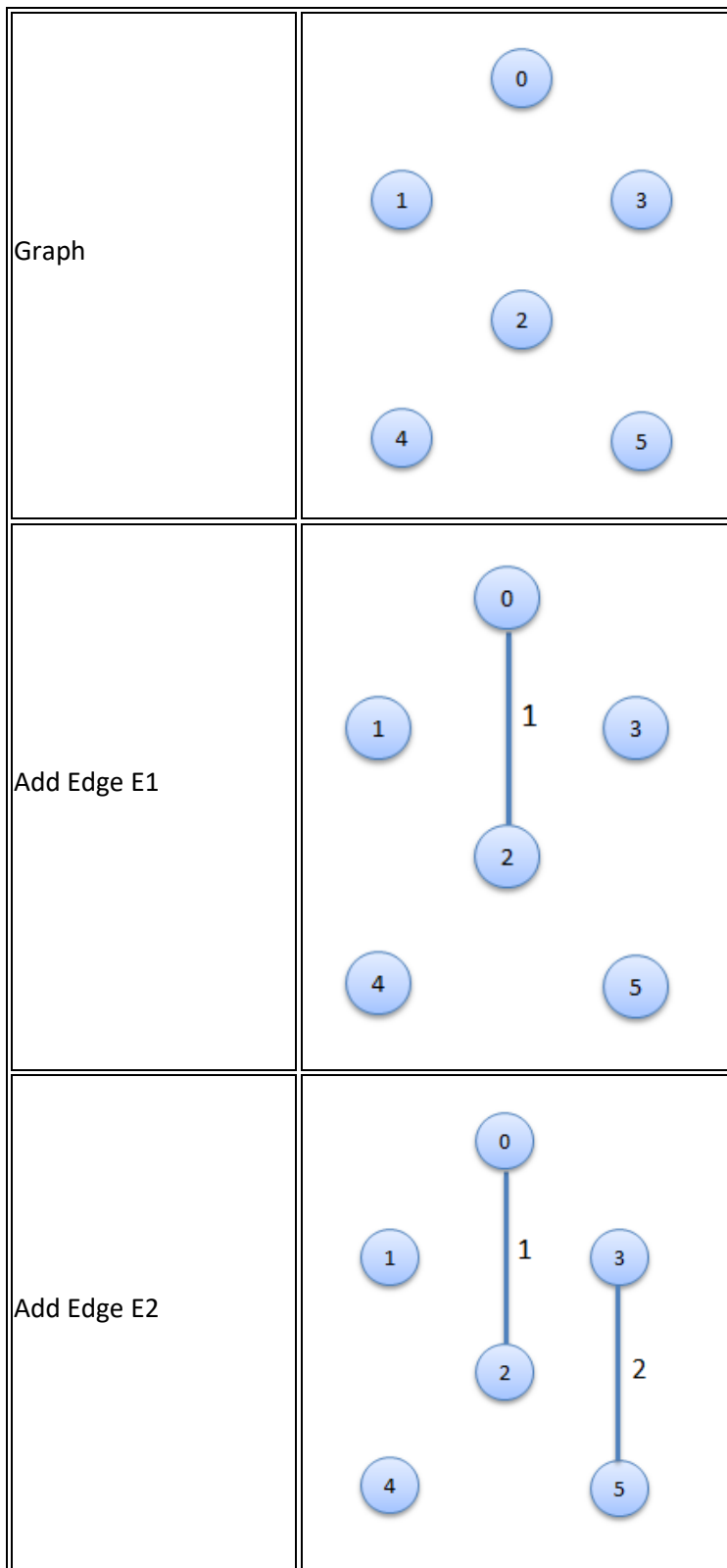


Procedure for finding Minimum Spanning Tree

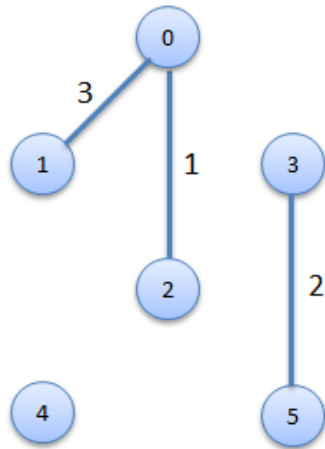
Step1. Edges are sorted in ascending order by weight.

| Edge No. | Vertex Pair | Edge Weight |
|----------|-------------|-------------|
| E1 | (0,2) | 1 |
| E2 | (3,5) | 2 |
| E3 | (0,1) | 3 |
| E4 | (1,4) | 3 |
| E5 | (2,5) | 4 |
| E6 | (1,2) | 5 |
| E7 | (2,3) | 5 |
| E8 | (0,3) | 6 |
| E9 | (2,4) | 6 |
| E10 | (4,5) | 6 |

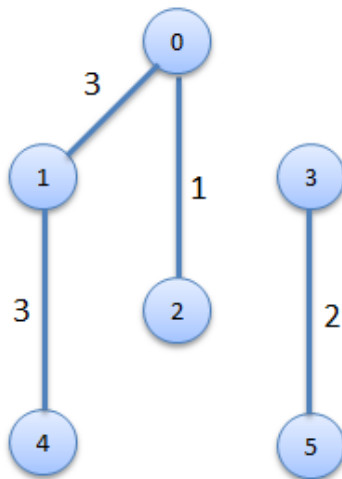
Step2. Edges are added in sequence.



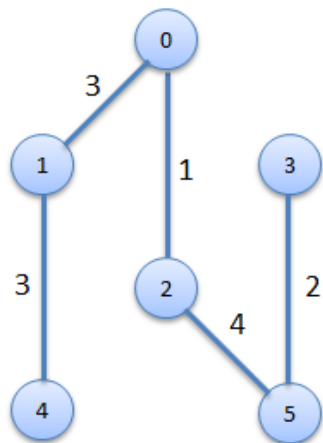
Add Edge E3



Add Edge E4



Add Edge E5



Total Cost = $1+2+3+3+4 = 13s$

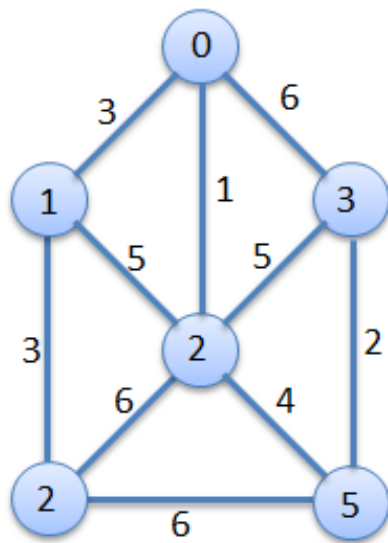
Prim's Algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph.

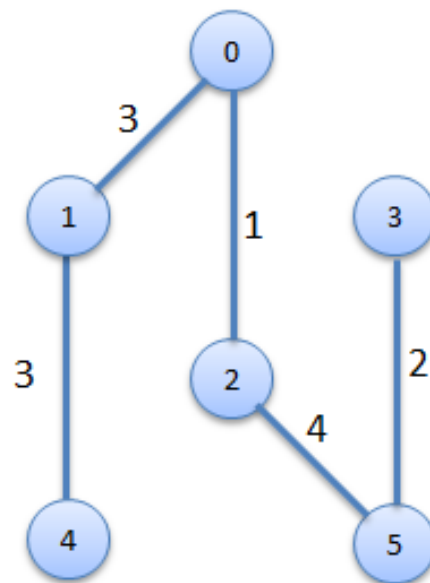
It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

This algorithm is directly based on the MST(minimum spanning tree) property.

Example



A Simple Weighted Graph



Minimum-Cost Spanning Tree

Prim's Algorithm

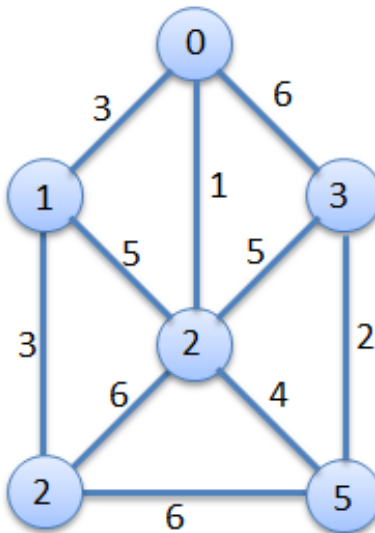
```
1. MST-PRIM( $G, w, r$ )
2.
3. 1.   for each  $u \in V[G]$ 
4.
5. 2.   do  $key[u] \leftarrow \infty$ 
6.
7. 3.    $\pi[u] \leftarrow NIL$ 
```

```

8.
9. 4.      key[r] ← 0
10.
11. 5.      Q ← V [G]
12.
13. 6.      while Q ≠ ∅
14.
15. 7.          do u ← EXTRACT-MIN(Q)
16.
17. 8.              for each v Adj[u]
18.
19. 9.                  do if v ∉ Q and w(u, v) < key[v]
20.
21. 10.                      then π[v] ← u
22.
23. 11.                          key[v] ← w(u, v)
24.
25.
26.

```

Example



Procedure for finding Minimum Spanning Tree

Step1

| | | | | | | |
|---------------|---|---|----------|---|----------|----------|
| | | | | | | |
| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 |
| Distance | 0 | 3 | 1 | 6 | ∞ | ∞ |
| Distance From | | 0 | 0 | 0 | | |
| | | | | | | |

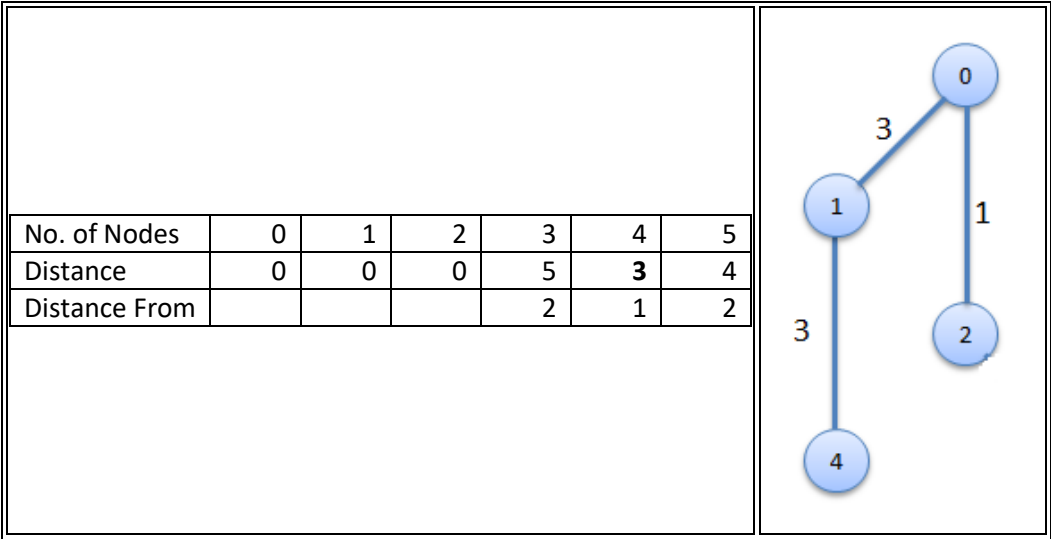


Step2

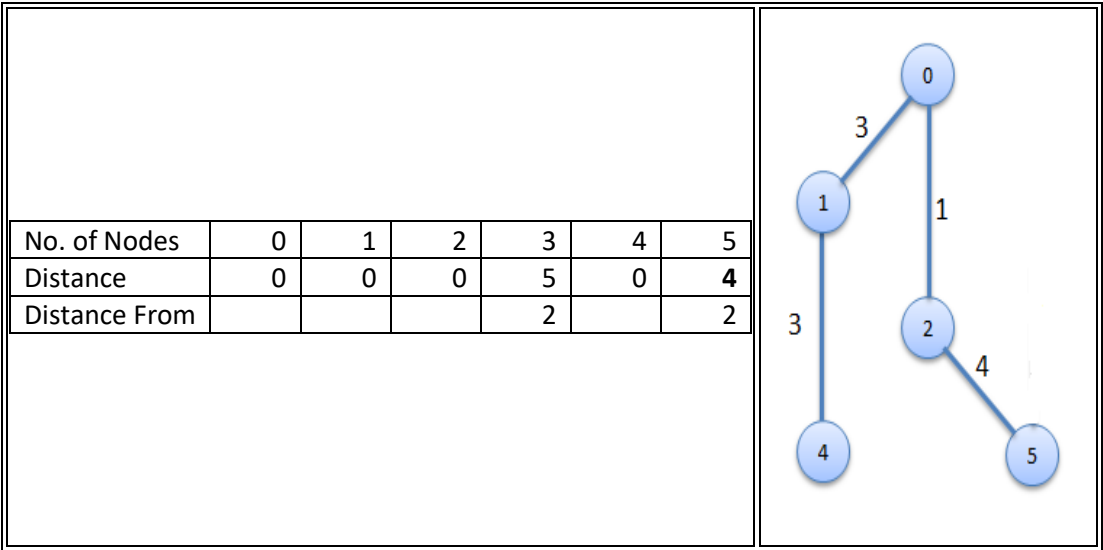
| | | | | | | |
|---------------|---|----------|---|---|---|---|
| | | | | | | |
| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 |
| Distance | 0 | 3 | 0 | 5 | 6 | 4 |
| Distance From | | 0 | | 2 | 2 | 2 |
| | | | | | | |



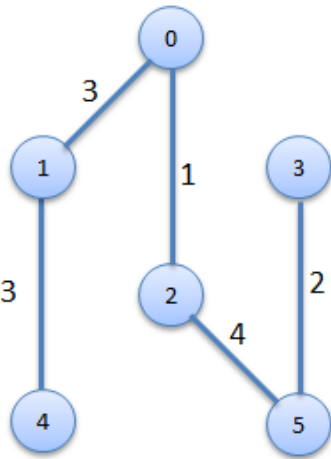
Step3



Step4



Step5

| | | | | | | | | | | | | | |
|-------------------------------|---|---|---|---|---|---|--|--|--|--|--|--|--|
| | | | | | | |  | | | | | | |
| No. of Nodes | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |
| Distance | 0 | 0 | 0 | 3 | 0 | 0 | | | | | | | |
| Distance From | | | | 2 | | 2 | | | | | | | |
| | | | | | | | | | | | | | |
| Minimum Cost = 1+2+3+3+4 = 13 | | | | | | | | | | | | | |

Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that solves the shortest path problem for a directed graph G . Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights.

Dijkstra's Algorithm

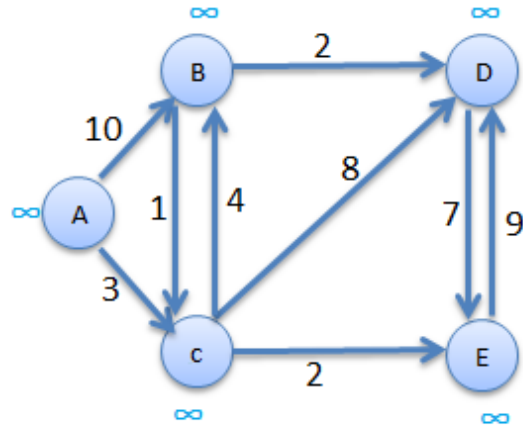
```

1. DIJKSTRA( $G, s$ )
2. 1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
3. 2  $S \leftarrow \emptyset$ 
4. 3  $Q \leftarrow V[G]$ 
5. 4 while  $Q \neq \emptyset$ 
6. 5     do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
7. 6          $S \leftarrow S \cup \{u\}$ 
8. 7         for each vertex  $v \in \text{Adj}[u]$ 
9. 8             do if  $\text{dist}[v] > \text{dist}[u] + w(u, v)$ 
10. 9                 then  $d[v] \leftarrow d[u] + w(u, v)$ 

1. INITIALIZE-SINGLE-SOURCE( Graph  $g$ , Node  $s$  )
2.      $\text{dist}[s] = 0$ ;
3.     for each vertex  $v$  in Vertices  $V[G] - s$ 
4.          $\text{dist}[v] \leftarrow \infty$ 

```

Example

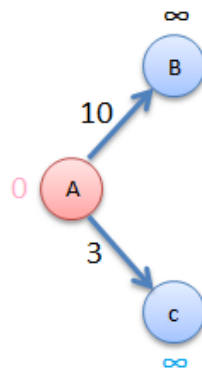


Procedure for Dijkstra's Algorithm

Step1

Consider A as source vertex

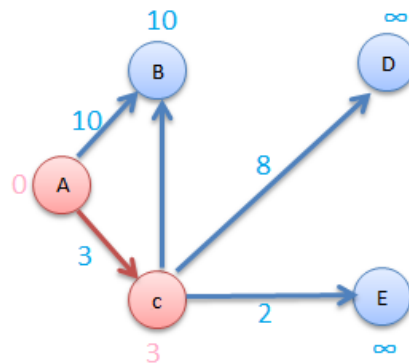
| | | | | | |
|---------------|---|----|---|---|---|
| No. of Nodes | A | B | C | D | E |
| Distance | 0 | 10 | 3 | ∞ | ∞ |
| Distance From | | A | A | | |



Step2

Now consider vertex C

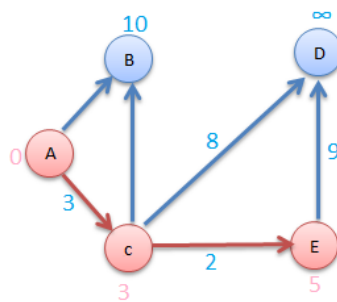
| | | | | | |
|---------------|---|---|---|----|---|
| No. of Nodes | A | B | C | D | E |
| Distance | 0 | 7 | 3 | 11 | 5 |
| Distance From | | C | C | C | C |



Step3

Now consider vertex E

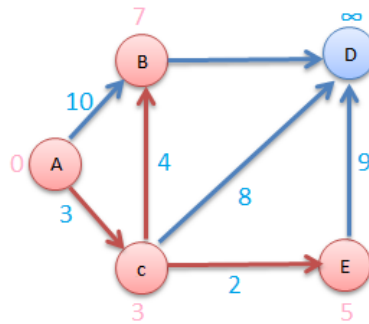
| | | | | | |
|---------------|---|---|---|----|---|
| No. of Nodes | A | B | C | D | E |
| Distance | 0 | 7 | 3 | 11 | 0 |
| Distance From | | C | A | C | E |



Step4

Now consider vertex B

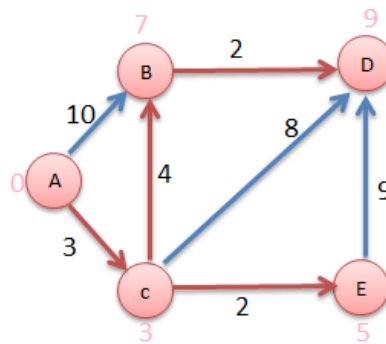
| | | | | | |
|---------------|---|---|---|---|---|
| No. of Nodes | A | B | C | D | E |
| Distance | 0 | 7 | 3 | 9 | 5 |
| Distance From | | C | A | B | C |



Step5

Now consider vertex D

| | | | | | |
|---------------|---|---|---|---|----|
| No. of Nodes | A | B | C | D | E |
| Distance | 0 | 7 | 3 | 9 | 11 |
| Distance From | A | C | A | B | C |



| | A | B | C | D | E |
|---|---|----------|----------|----------|----------|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| A | 0 | 10 | 3 | ∞ | ∞ |
| C | | 7 | 3 | 11 | 5 |
| E | | | | 14 | 5 |
| B | | | | 9 | |
| D | | | | | 16 |

Bellman Ford's Algorithm

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

Why would one ever have edges with negative weights in real life?

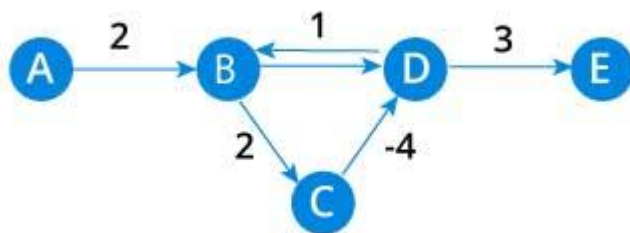
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, heat released/absorbed in a chemical reaction etc.

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

Why we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle which will reduce the total path distance by coming back to the same point.



Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

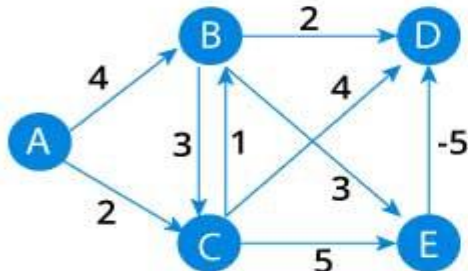
How Bellman Ford's algorithm works

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

By doing this repeatedly for all vertices, we are able to guarantee that the end result is optimized.

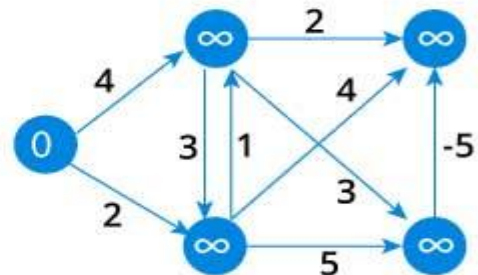
1

Start with a weighted graph



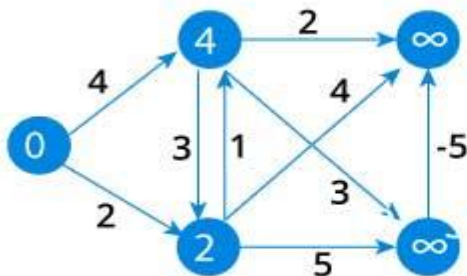
2

Choose a starting vertex and assign infinity path values to all other vertices



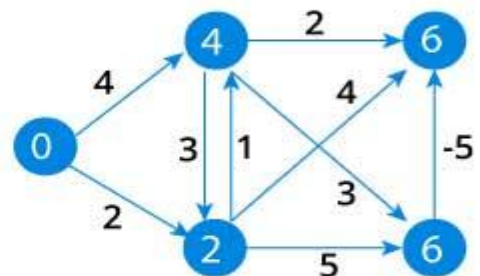
3

Visit each edge and relax the path distances if they are inaccurate



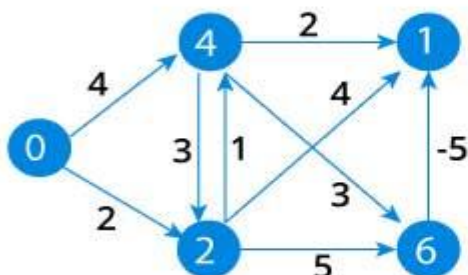
4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



5

Notice how the vertex at the top right corner had its path length adjusted



6

After all the vertices have their path lengths, we check if a negative cycle is present.

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | 4 | 2 | ∞ | ∞ |
| 0 | 3 | 2 | 6 | 6 |
| 0 | 3 | 2 | 1 | 6 |
| 0 | 3 | 2 | 1 | 6 |

Bellman Ford Pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v , where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
    distance[S] <- 0
    for each vertex V in G
        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V]
            Error: Negative Cycle Exists

    return distance[], previous[]
```