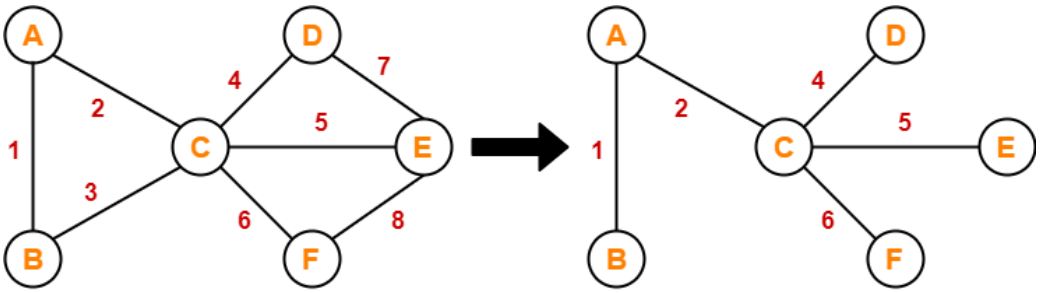
Important comparision

Concept-01:

If all the edge weights are distinct, then both the algorithms are guaranteed to find the same MST.

Example-

Consider the following example-



Given Graph

Minimum Spanning Tree (MST)

(Cost = 18 units)

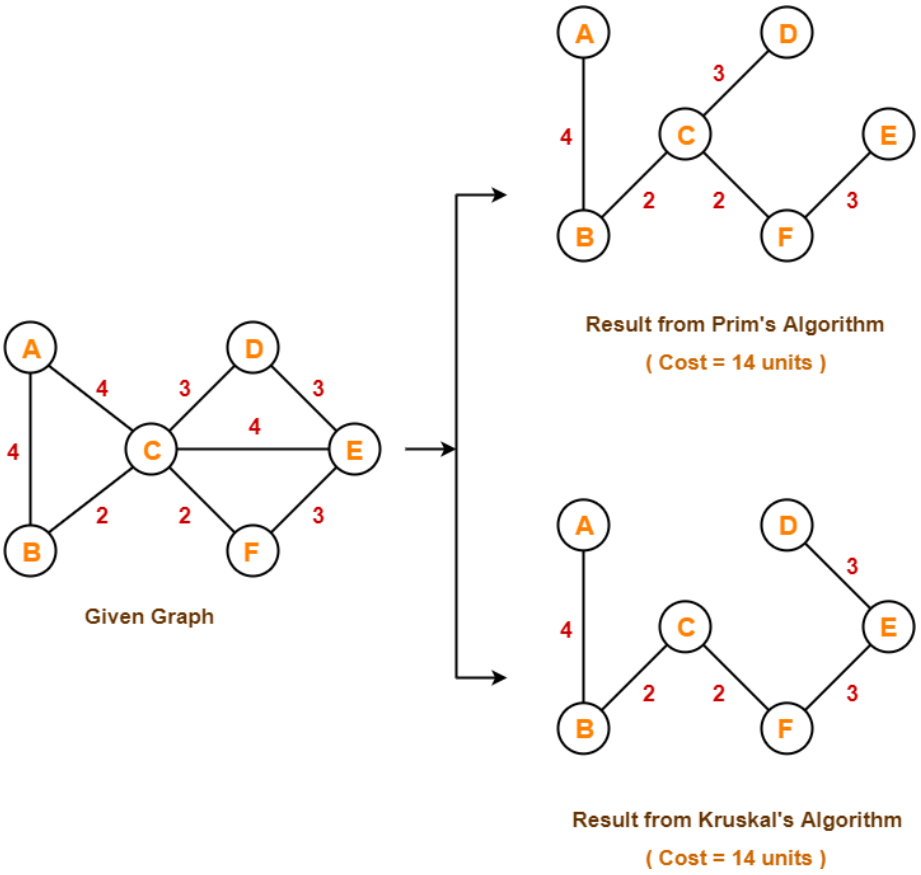
Here, both the algorithms on the above given graph produces the same MST as shown.

Concept-02:

- If all the edge weights are not distinct, then both the algorithms may not always produce the same MST.
- However, cost of both the MST_s would always be same in both the cases.

Example-

Consider the following example-



Here, both the algorithms on the above given graph produces different MST_s as shown but the cost is same in both the cases.

Concept-03:

Kruskal's Algorithm is preferred when-

- The graph is sparse.
- There are less number of edges in the graph like E = O(V)
- The edges are already sorted or can be sorted in linear time.

Prim's Algorithm is preferred when-

- The graph is dense.
- There are large number of edges in the graph like E = $O(V^2)$.

Kruskal's Algorithm	Prim's Algorithm
It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.	It starts to build the Minimum Spanning Tree from any vertex in the graph.
It traverses one node only once.	It traverses one node more than one time to get the minimum distance.
Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components	Prim's algorithm gives connected component as well as it works only on connected graph.
Kruskal's algorithm runs faster in sparse graphs.	Prim's algorithm runs faster in dense graphs.
Kruskal's algorithm uses Heap Data Structure.	Prim's algorithm uses List Data Structure.

Dijkstra’s shortest path algorithm

Problem: Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

- Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph.

  - Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree.
  - Like Prim’s MST, we generate a *SPT (shortest path tree)* with a given source as a root.
  - We maintain two sets, one set contains vertices included in the shortest-path tree,
  - other set includes vertices not yet included in the shortest-path tree.

  - At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm
1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While *sptSet* doesn’t include all vertices

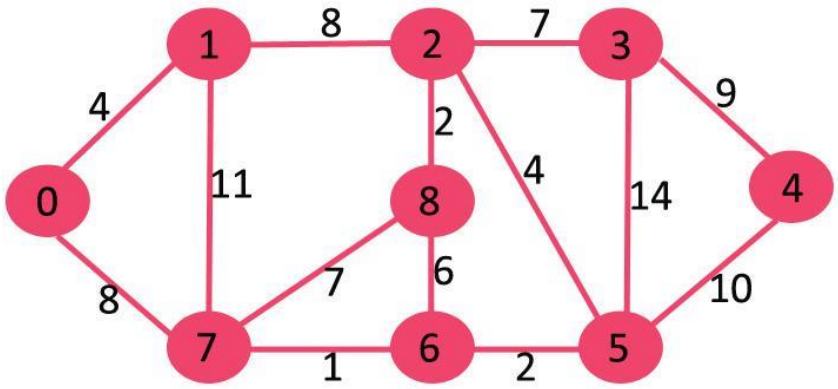
a) Pick a vertex u which is not there in *sptSet* and has a minimum distance value.

b) Include u to *sptSet*.

c) Update distance value of all adjacent vertices of u.

i. To update the distance values, iterate through all adjacent vertices.

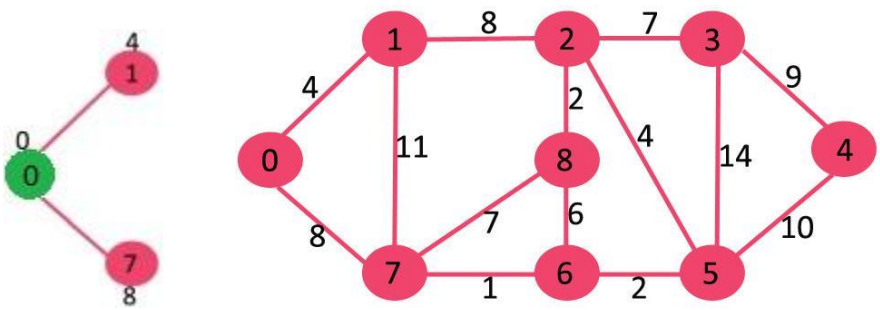
ii. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.



- sptset = { }
- Key_Value = { 0, ∞,∞,∞,∞,∞,∞, ∞,∞ }

  - The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}.
  - After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.
  - The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

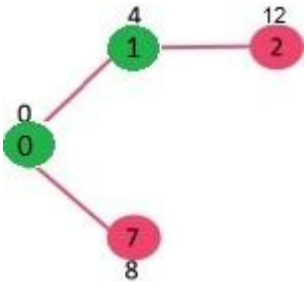
Mstset={0} Key_value = {0,4, ∞,∞,∞,∞,∞,8, ∞}



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET).

The vertex 1 is picked and added to sptSet.

- So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.
  Mstset={0,1}
- Key_value = {0,4, 12,∞,∞,∞,∞,8, ∞}



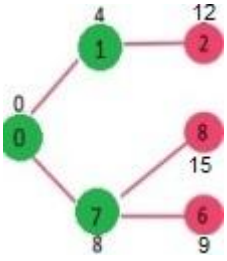
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET).

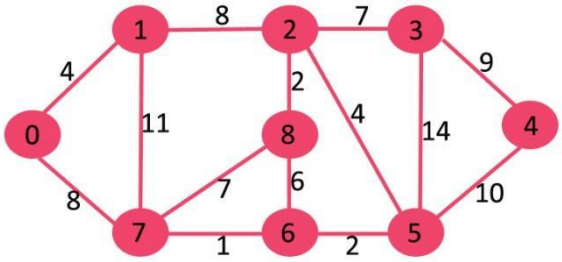
Vertex 7 is picked.

So sptSet now becomes {0, 1, 7}.

Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).

Mstset={0,1,7}
Key_value = {0,4, 12,∞,∞,∞,9,8,15}





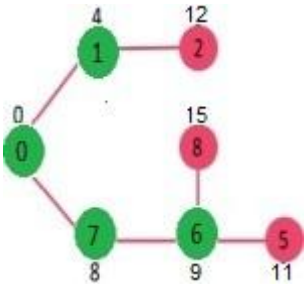
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET).

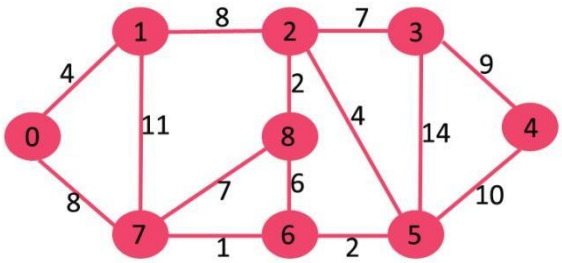
Vertex 6 is picked.

So sptSet now becomes {0, 1, 7, 6}.

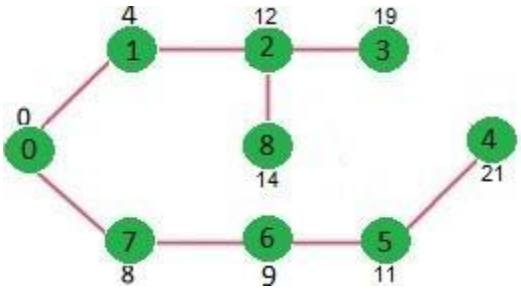
Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

Mstset={0,1,7,6}
Key_value = {0,4, 12,∞,∞,11,9,8,15}





We repeat the above steps until *sptSet* includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).

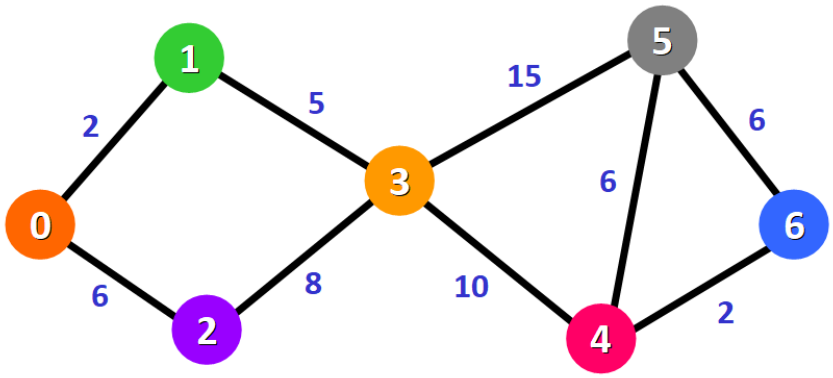


For detailed algorithm

https://www.javatpoint.com/dijkstras-algorithm

Example 3:

Find the shortest path from node 0 to all the other nodes in the graph. Here in this process Key value is Distance. We can maintain ist of Unvisited nodes also instead of Visied node list



- Distance List contains distance from Root

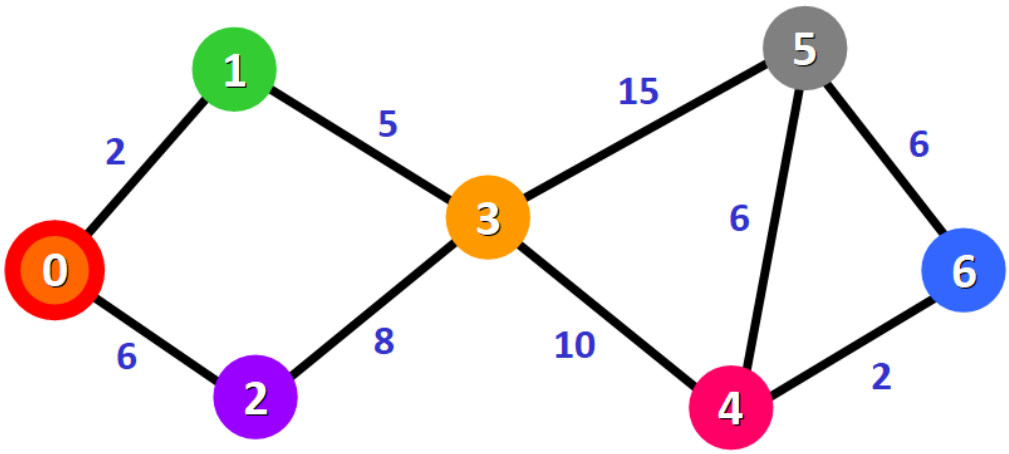
• Unvisited List = {0,1,2,3,4,5,6} • Distance = {0, ∞, ∞,∞,∞,∞,∞}

Step 1: Choose the start node

Check distance of adjunct nodes

Update the distance in the distance list

• Unvisited List = {1,2,3,4,5,6} • Distance = {0, 2, 6,∞,∞,∞,∞} • Spt={0} • Active Edge list {}

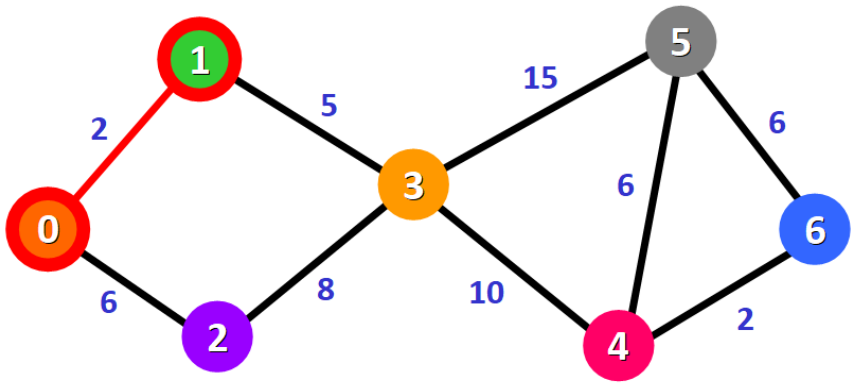


Step 2: After updating the distances of the adjacent nodes, we need to:

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

  - Unvisited List = {2,3,4,5,6}
  - Distance = {0, 2, 6,∞,∞,∞,∞}
  - Spt={0,1}
  - Active Edge ={01}

-

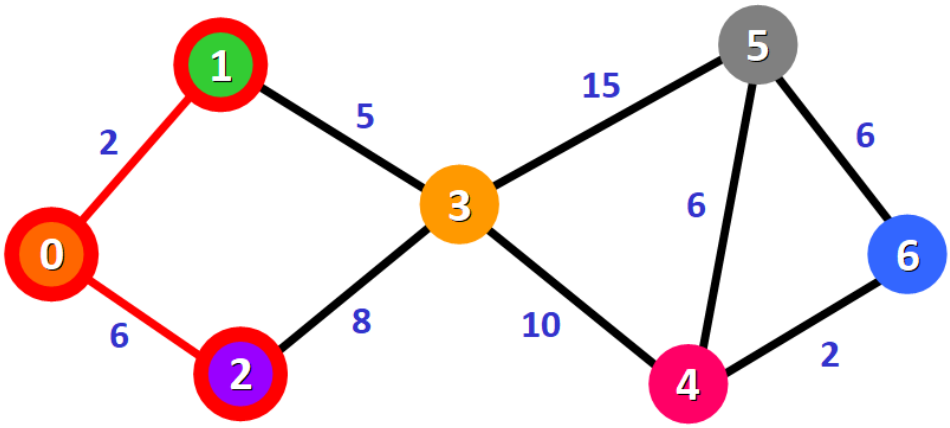


Node `3` and node `2` are both adjacent to nodes that are already in the path because they are directly connected to node `0` and node `1`, respectively, as you can see below. These are the nodes that we will analyze in the next step.

Since we already have the distance from the source node to node `2` written down in our list, we don't need to update the distance this time. We only need to update the distance from the source node to the new adjacent node (node `3`):

- Unvisited List = {_,_,_,3,4,5,6}
- Distance = {0, 2, 6,7,∞,∞,∞}
- Spt={0,1,2}
- Active Edge = {01,02}

Now that we have the distance to the adjacent nodes, we have to choose which node will be added to the path.
We must select the **unvisited** node with the shortest (currently known) distance to the source node.
From the list of distances, we can immediately detect that this is node `2` with distance **6**:

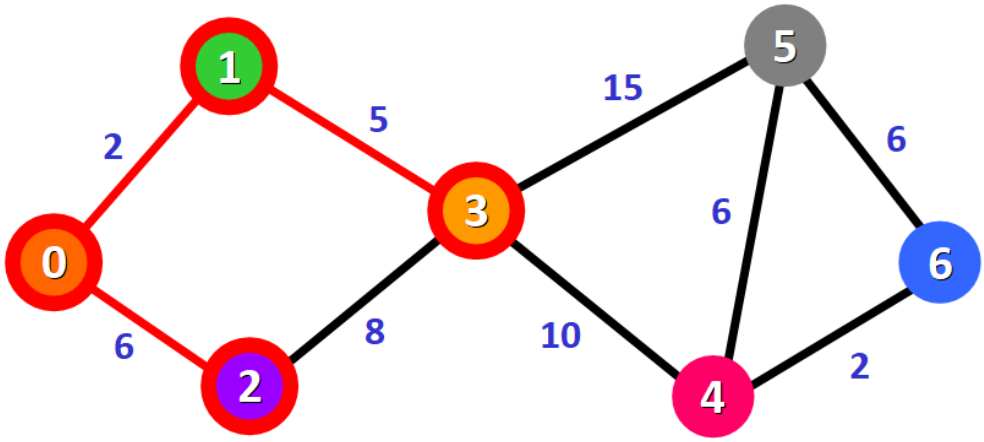


Now we need to repeat the process to find the shortest path from the source node to the new adjacent node, which is node `3`.
You can see that we have two possible paths `0 -> 1 -> 3` or `0 -> 2 -> 3`. Let's see how we can decide which one is the shortest path.

Dist(3) = Dist(1) + 5 = 7
Dist(3) = Dist(2)+8 = 14

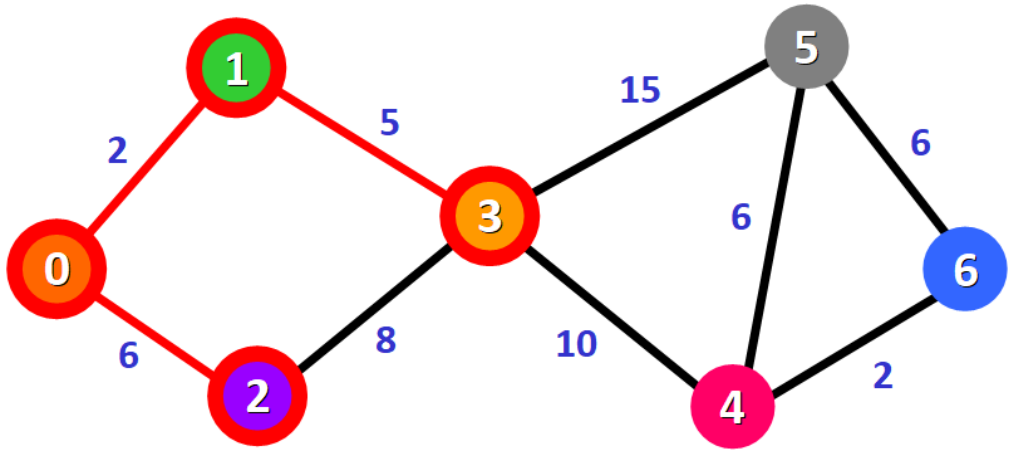
So path 1 □ 3 will be selected and 3 will be added in visited lsit

- Unvisited List = {4,5,6}
- Distance = {0, 2, 6,7,∞,∞,∞}
- Spt={0,1,2,3}
- Active Edge = {01,02,1,3}



Now we repeat the process again.

We need to check the new adjacent nodes that we have not visited so far.
This time, these nodes are node `4` and node `5` since they are adjacent to node `3`.

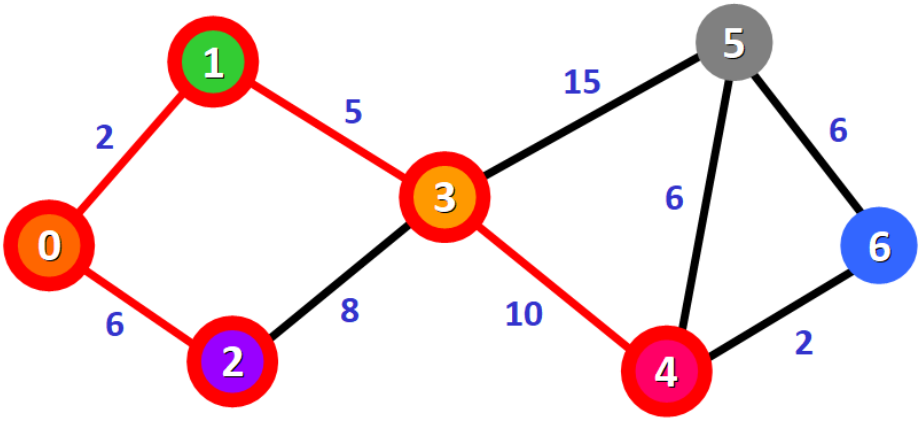


We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:

- **For node 4:** the distance is **17** from the path `0 -> 1 -> 3 -> 4`.
- **For node 5:** the distance is **22** from the path `0 -> 1 -> 3 -> 5`.

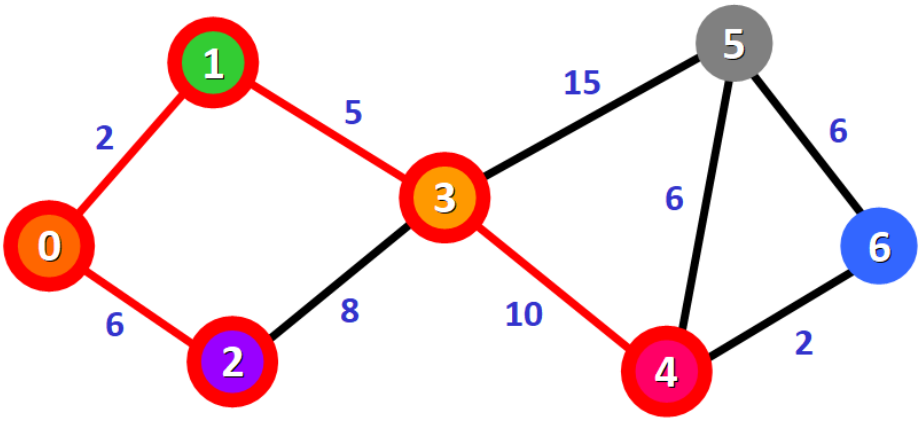
  - Unvisited List = {4,5,6}
  - Distance = {0, 2, 6,7,17,22,∞}
  - Spt={0,1,2,3}
  - Active Edge = {01,02,1,3}
- Node with least value will be selected that is node 4
- Node 4 marked as visited

  - Unvisited List = {5,6}
  - Distance = {0, 2, 6,7,17,22,∞}
  - Spt={0,1,2,3,4}
  - Active Edge = {01,02,13,34}
-



And we repeat the process again.
We check the adjacent nodes: node `5` and node `6`.

We need to analyze each possible path that we can follow to reach them from nodes that have already been marked as visited and added to the path.

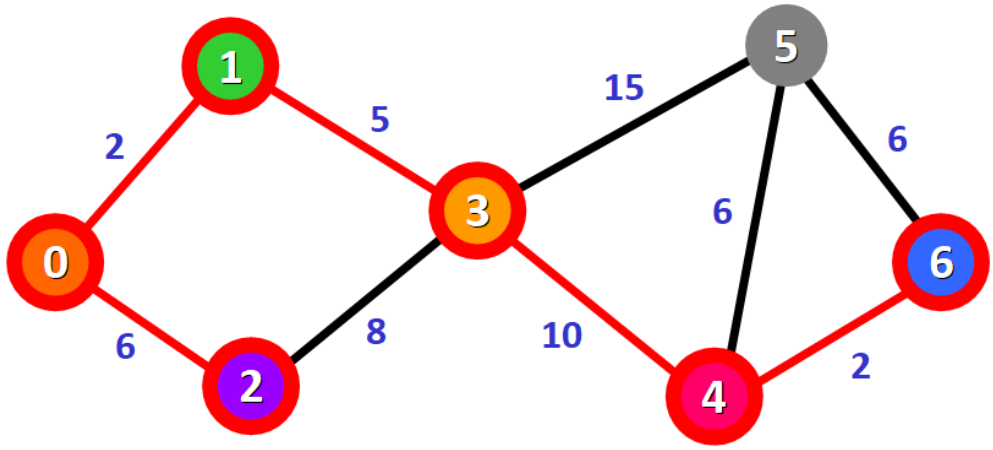


For node 5:
3→5 = 7+15= 22
4→5=17+6 = 23

For Node 6:
4→6 = 19
5→6 = 28

Least cost node is selected

- Unvisited List = {5}
- Distance = {0, 2, 6,7,17,22,19}
- Spt={0,1,2,3,4,6}
- Active Edge = {01,02,13,34,46}

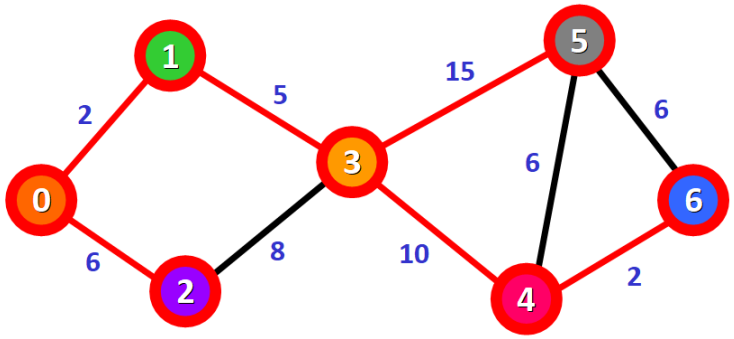


Only one node has not been visited yet, node 5. Let's see how we can include it in the path.
There are three different paths that we can take to reach node 5 from the nodes that have been added to the path:

- 3→5 =22

- 4□5 = 23
- 6□5 =25

  - Unvisited List = {5}
  - Distance = {0, 2, 6,7,17,22,19}
  - Spt={0,1,2,3,4,5,6}
  - Active Edge = {01,02,13,34,34,46}

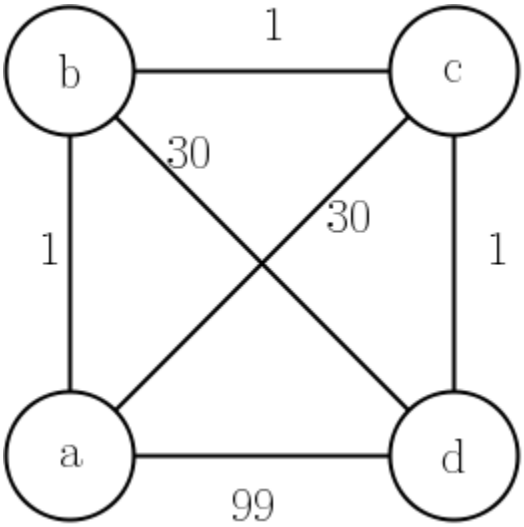


- Graphs are used to model connections between objects, people, or entities. They have two main elements: nodes and edges. Nodes represent objects and edges represent the connections between these objects.
- Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph.
- This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

Important Notes;

Dijkstra's algorithm returns a shortest path tree, containing the shortest path from a starting vertex to each other vertex, but not necessarily the shortest paths between the other vertices, or a shortest route that visits all the vertices.

Here's a counter example where the greedy algorithm you describe will not work:



Starting from a, the greedy algorithm will choose the route [a,b,c,d,a][a,b,c,d,a], but the shortest route starting and ending at a is [a,b,d,c,a][a,b,d,c,a].

Since the TSP route is not allowed to repeat vertices, once the greedy algorithm chooses a,b,c,d or a,b,c,d, it is forced to take the longest edge d,a to return to the starting city.

Good question

https://www.careercup.com/question?id=5085331422445568

How would you use Dijkstra's algorithm to solve travel salesman problem, which is to find a shortest path from a starting node back to the starting node and visits all other node exactly once.