SCS1201 Advanced Data Structures Unit IV
UNIT 4 ADVANCED GRAPH CONCEPTS

MINIMUM SPANNING TREES

Weight of an edge: *Weight of an edge is just of the value of the edge or the cost of the edge. For example, a graph representing cities, has the distance between two cites as the edge cost or its weight.*

Network: *A graph with weighted edges is called a network.*

Spanning Tree: *Any tree consisting of edges in the graph G and including all vertices in G is called a spanning tree.*

Given a network, we should try to connect all the nodes in the nodes in the graph with minimum number of edges, such that the total weight is minimized. To solve this problem, we shall devise an algorithm that converts a network into to tree structures called the minimum spanning tree of the network.

Given a network, the edges for the minimum spanning tree are chosen in such a way that:

(1) Every node in the network must be included in the spanning tree.

(2) The overall edge weight of the spanning tree is the minimum possible that will allow the existence of a path between any 2 nodes in the tree.

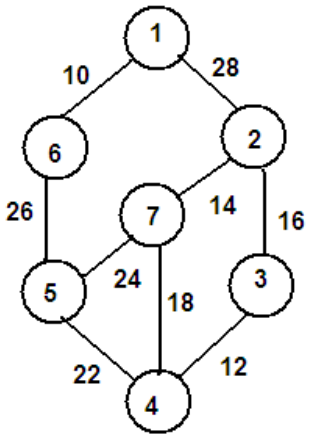
The two algorithms which are used for finding the minimum spanning tree for a graph are:

1. *Kruskal's Algorithm*
2. *Prim's Algorithm*
3. *Sollin's Algorithm*

KRUSKAL'S ALGORITHM

The Kruskal's algorithm follows greedy approach. At every stage of the solution, it takes that edge which has the minimum cost and builds the minimum spanning tree.

Example:



Consider the above graph. Now let us apply the Kruskal's algorithm to construct a minimum spanning tree.

Step 1:

Construct a queue with the cost of edges, such that the edges are placed in the queue in the ascending order of the cost as shown.

Queue of edge costs

10	12	14	16	18	22	24	26	28

Step 2:

Create N sets each consisting one node. N is the number of nodes in the graph. Then for the above problem, the sets which will be created are

S1 = {1}
S2 = {2}
S3 = {3}
S4 = {4}
S5 = {5}
S6 = {6}
S7 = {7}

Step 3:

Delete a cost from the queue. Let the nodes associated with that edge be (u,v). Now, 10 is deleted first from the queue. The nodes associated with 10 is (u,v) = (1,6). Check if u and v belong to the same set or different set. If they belong to the different set then enter that into the output matrix as shown. Since 1 belongs to S1 and 6 belong to S6, they can be entered into the T matrix. If the nodes belong to the same set, then entering them into the matrix will give an output which may form a cycle. Hence that is avoided. The T matrix has n-1 rows and 2 columns.

T matrix

	u	*v*
1	1	6
2		
3		
4		
5		
6		

After entering them in the T matrix, the sets S1 and S6 are merged.

S8 = {1, 6}

The above process in step 3 is repeated till the queue becomes empty. The solution is derived as shown.

Queue of edge costs

12	14	16	18	22	24	26	28

Delete 12 from the queue. The nodes associated with 12 are (u,v) = (3,4). The node 3 belongs to S3 and node 4 belongs to S4. As they are in different sets, they are entered in the T matrix.

T matrix

	u	*v*
1	1	6
2	3	4
3		
4		
5		
6		

The sets S3 and S4 are merged.

S9 = {3, 4}

Queue of edge costs

14	16	18	22	24	26	28

Delete 14 from the queue. The (u,v) = (2,7). 2 belong to S2 and 7 belong to S7. As they belong to different sets, they are entered into the T matrix and the sets S2 and S7 are merged.

T matrix

	u	*v*
1	1	6
2	3	4
3	2	7
4		
5		
6		

S10 = {2, 7}

Queue of edge costs

16	18	22	24	26	28

Delete 16 from the queue. The (u,v) = (2,3). 2 belong to S10 and 3 belong to S9. As they are from different sets, they are entered into the T matrix. The sets S9 and S10 are merged.

T matrix

	u	*v*
1	1	6
2	3	4

3	2	7
4	2	3
5		
6		

S11 = {2, 3, 4, 7}

Queue of edge costs

18	22	24	26	28

Delete 18. The (u, v) = (4, 7). 4 and 7 belong to same set S11. Hence they are not entered into the T matrix.

Queue of edge costs

22	24	26	28

Delete 22. The (u,v) = (4, 5). 4 belong to S11 and 5 belong to S5. As they belong to different set, they are entered into the T matrix. The sets S11 and S5 are merged.

T matrix

	u	*v*
1	1	6
2	3	4
3	2	7
4	2	3
5	4	5
6		

S12 = {2, 3, 4, 5, 7}

Queue of edge costs

24	26	28

Delete 24. (u, v) = (5, 7). Both 5 and 7 belong to S12. Hence they are not entered into the T matrix.

26	28

Delete 26. (u, v) = (5, 6). 5 belong to S12 and 6 belong to S8. As they are from different set, they are entered into the T matrix.

T matrix

	u	*v*
1	1	6
2	3	4
3	2	7
4	2	3
5	4	5

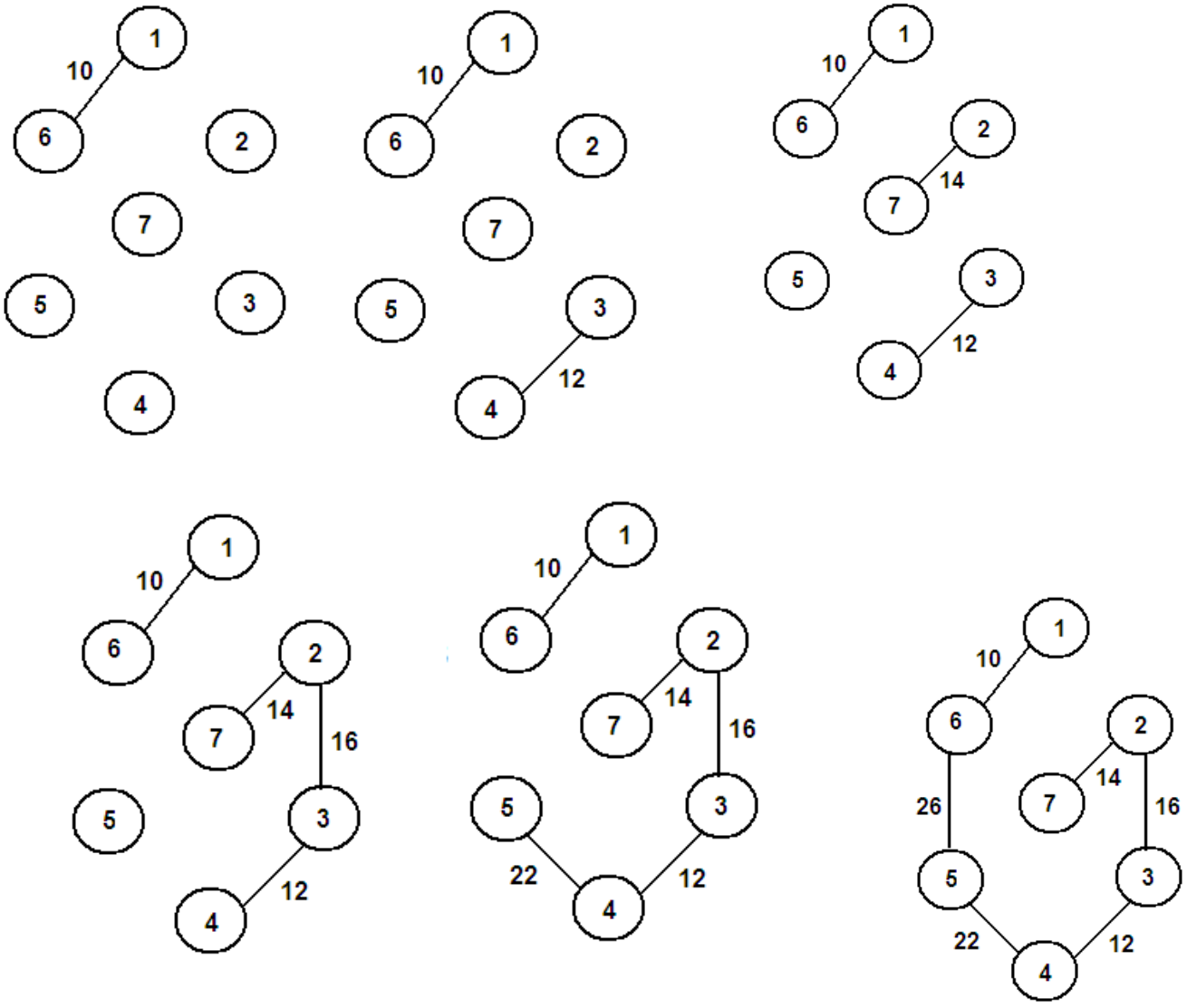
6	5	6

S13 = {1, 2, 3, 4, 5, 6, 7}

As all T matrix is completely filled, the algorithm comes to an end.

Step 4:

Using the edges in the T matrix connect the nodes of the graph. The resulting tree is the required minimum spanning tree.



Algorithm

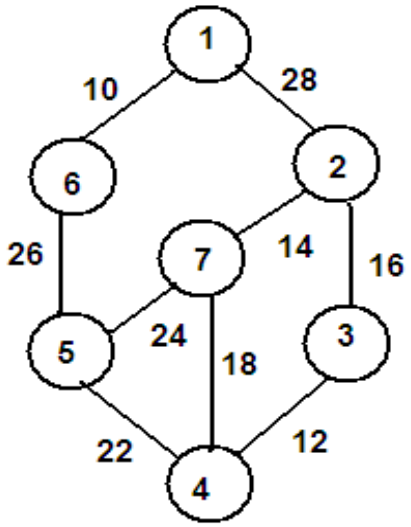
KRUSKAL(E, cost, n, t) Construct a queue with edge costs such that they are in ascending order

```
i = 0, mincost = 0
while i < n – 1 and queue is not empty
        Delete minimum cost edge (u, v) from queue
        j = Find(u), k = Find(v)
        If j ≠ k
                i = i + 1
                t[i, 1] = u, t[i, 2] = v
                mincost = mincost + cost[u, v]
                Union(j, k)
        End if
End while
If i ≠ n – 1
        Print "No spanning tree"
Else
        Return mincost
End if
End KRUSKAL
```

PRIM'S ALGORITHM

The other popular algorithm used for constructing the minimum spanning tree is the Prim's algorithm, which also follows the greedy approach. We can consider the same example as above and solve it using Prim's algorithm.

Example:



Cost Matrix is

	1	*2*	*3*	*4*	*5*	*6*	*7*
1	0	28	∞	∞	∞	10	∞

2	28	0	16	∞	∞	∞	14
3	∞	16	0	12	∞	∞	∞
4	∞	∞	12	0	22	∞	18
5	∞	∞	∞	22	0	26	24
6	10	26	∞	∞	∞	0	∞
7	∞	14	∞	18	24	∞	0

Step 1:

Select the least cost edge from the graph and enter into the T matrix. The least cost edge is (1, 6) with cost 10.

T matrix

	u	*v*
1	1	6
2		
3		
4		
5		
6		

Let us consider an array NEAR[ ], which is filled as follows:

```
If cost[i, l] < cost[i, k]
        Near[i] = l
    Else
        Near[i] = k
```

In the first iteration i = 1 and (k, l) = (1, 6). Using the above condition the NEAR array is filled as follows.

NEAR

1	1
2	1
3	1
4	1
5	6
6	6
7	1

Step 2:

Make the entries in the NEAR array corresponding to 1 and 6 as 0. For all non-zero entries in the near array, find out the cost[j][near[j]]. Select the minimum among these costs and enter the corresponding nodes into the T matrix.

NEAR

1	0	
2	1	28
3	1	∞
4	1	∞
5	6	26
6	0	
7	1	∞

Among the costs, 26 is minimum. Hence (5, 6) is entered into the T matrix. The corresponding entry into the NEAR array is made 0.

T matrix

	u	*v*
1	1	6
2	5	6
3		
4		
5		
6		

Step 3:

Now in every iteration the NEAR array is updated using the following condition and procedure in step 2 is followed to fill up the T matrix. The solution is as follows:

If Near[k] ≠ 0 and cost[k, Near[k]] > cost[k, j]
Near[k] = j

Updated NEAR

1	0	
2	1	28
3	1	∞
4	5	22
J=5	0	
6	0	
7	5	24

Among the cost computed, 22 is minimum and hence (4,5) is selected as the minimum edge.
T matrix

	u	*v*
1	1	6
2	5	6
3	4	5
4		
5		
6		

Updated NEAR

1	0	
2	1	28
3	4	12
J=4	0	
5	0	
6	0	
7	4	18

Among the cost computed, 12 is minimum and hence (3, 4) is selected as the minimum edge.
T matrix

	u	*v*
1	1	6
2	5	6
3	4	5
4	3	4
5		
6		

Updated NEAR

1	0	
2	3	16
J=3	0	
4	0	
5	0	
6	0	
7	4	18

Among the cost computed, 16 is minimum and hence (2, 3) is selected as the minimum edge.

T matrix

	u	*v*
1	1	6
2	5	6
3	4	5
4	3	4
5	2	3
6		

Updated NEAR

1	0	
J=2	0	
3	0	
4	0	
5	0	
6	0	
7	2	14

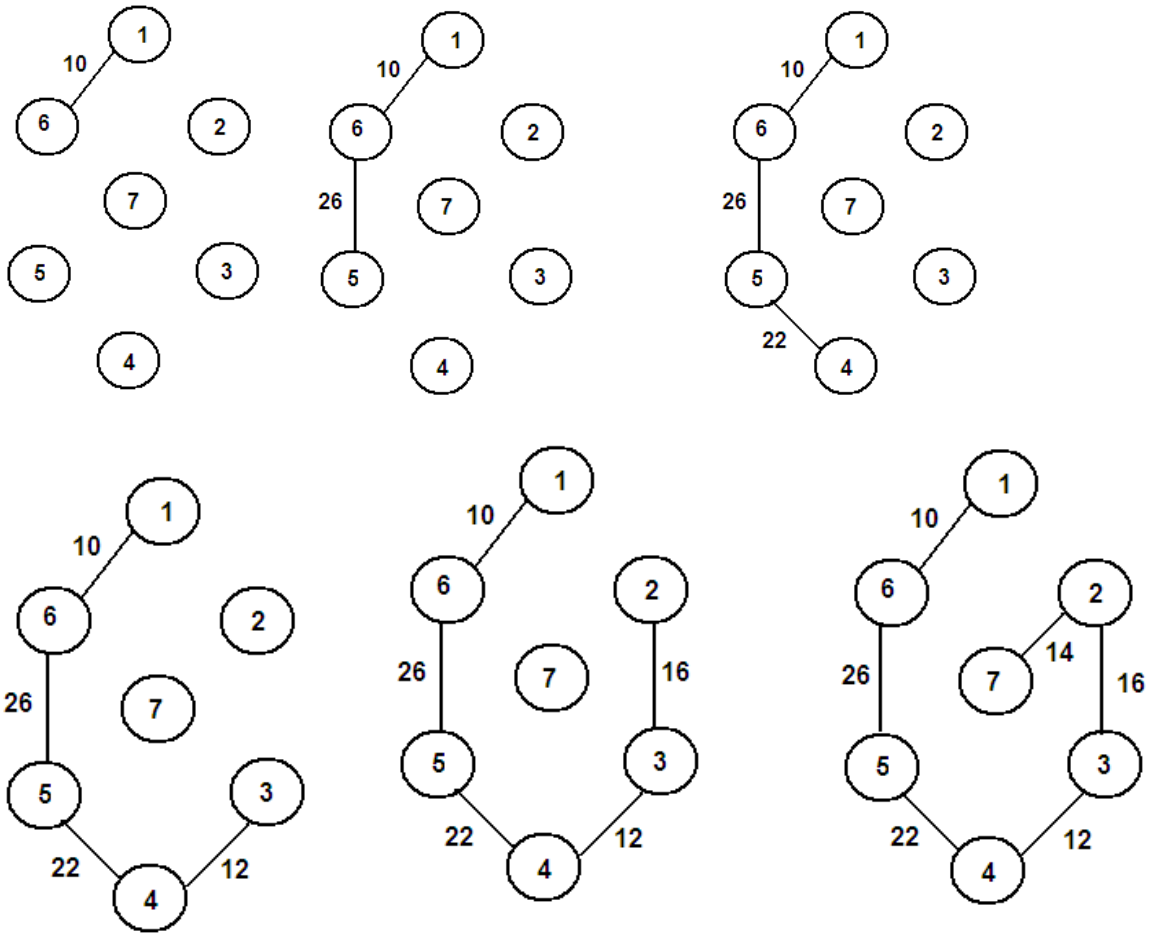
The last edge (7, 2) is selected and entered into the T matrix.

T matrix

	u	*v*
1	1	6
2	5	6
3	4	5
4	3	4
5	2	3
6	7	2

Step 4:

Now using the edges in the T matrix connect the nodes in the graph. The resulting tree is the minimum spanning tree.



Algorithm

PRIM(E, cost, n, t)
Let (k, L) be an edge of minimum cost in E
mincost = cost[k, L]
t[1, 1] = k, t[1, 2] =L
For i = 1 to n
    If cost[i, L] < cost[i, k]
        Near[i] = L
    Else
        Near[i] = k
    End if
End for
Near[k] = Near[L] = 0
For i = 2 to n -1
    Let j be an index such that near[j] ≠ 0 and cost[j, near[j]] is minimum
    T[i, 1] = j, t[i, 2] = Near[j]

```
        mincost = mincost + cost[j, near[j]]
        Near[j] = 0
        For k = 1 to n
        If Near[k] ≠ 0 and cost[k, Near[k]] > cost[k, j]
                Near[k] = j
        End if
End for
Return mincost
End PRIM
```

SOLLIN'S ALGORITHM

A minimum spanning tree (MST) of a weighted graph G is a spanning tree of G whose edges sum to minimum weight. In other words, a minimum spanning tree is a tree formed from a subset of the edges in a given undirected graph, with two properties: (1) it spans the graph, i.e., it includes every vertex in the graph, and (2) it is a minimum, i.e., the total weight of all the edges is as low as possible.

Sollin's algorithm selects several edges at each stage. At the start of a stage, the selected edges, together with all n graph vertices, form a spanning forest. During a stage we select one edge for each tree in this forest. The edge is a minimum-cost edge that has exactly one vertex in the tree. This selected edges are added to the spanning tree being constructed . Note that it is possible for two trees in the forest to select the same edge. So , multiple copies of the same edge are to be eliminated . Also , when the graph has several edges with the same cost , it is possible for two trees to select two different edges that connect them together . At the start of the first stage , the set of selected edges is empty . The algorithm terminates when there is only one tree at the end of a stage or when no edges remain to be selected.
The Sollin's Algorithm based on two basic operations:

- Nearest Neighbor – This operation takes a an input a tree spanning the nodes N_k and determines an arc (i_k , j_k) with the minimum cost among all arcs emanating from N_k.
- Merge $(i_k\ j_k)$ – This operation takes as an input two nodes i_k and j_k,and if the two nodes belong to two different trees, then merge these two trees into a single tree

Algorithm

```
Sollin's Algorithm
{
Form a forest consisting of the nodes of the graph while the forest has more than one tree
For each tree in the forest
Choose the cheapest edge
```

Form a vertex in the tree to a vertex not in the tree
Merge trees with common vertices
}
This algorithm keeps a forest of minimum spanning trees which it continuously connects via the least cost arc live in each tree. To begin each node is made its minimum spanning tree from here the shortest path live in each tree (which doesn't connect to a node already belonging to the current tree) is added along the minimum spanning tree it connect to . This continues until exit a single spanning tree.
Example :

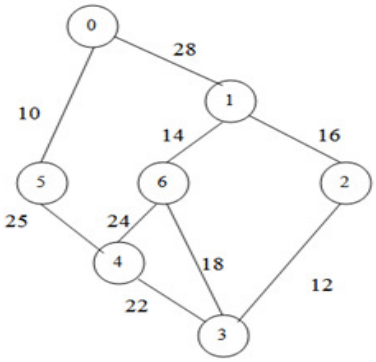


Figure (a)

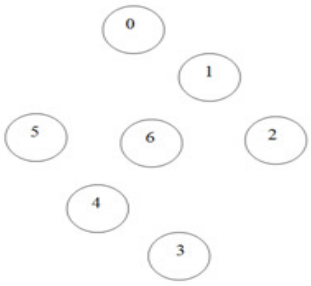


Figure (b)

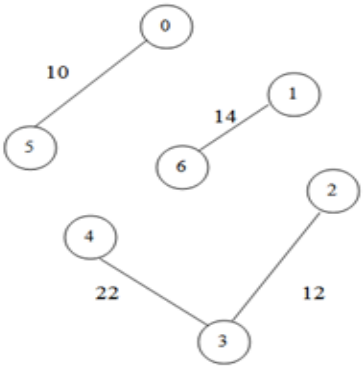


Figure (c)

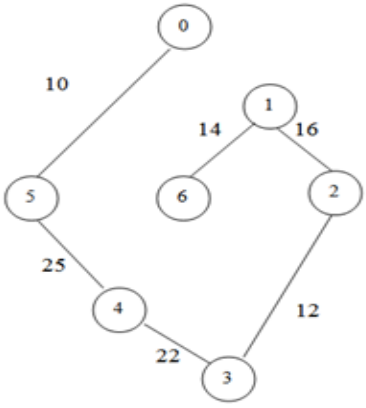


Figure (d)

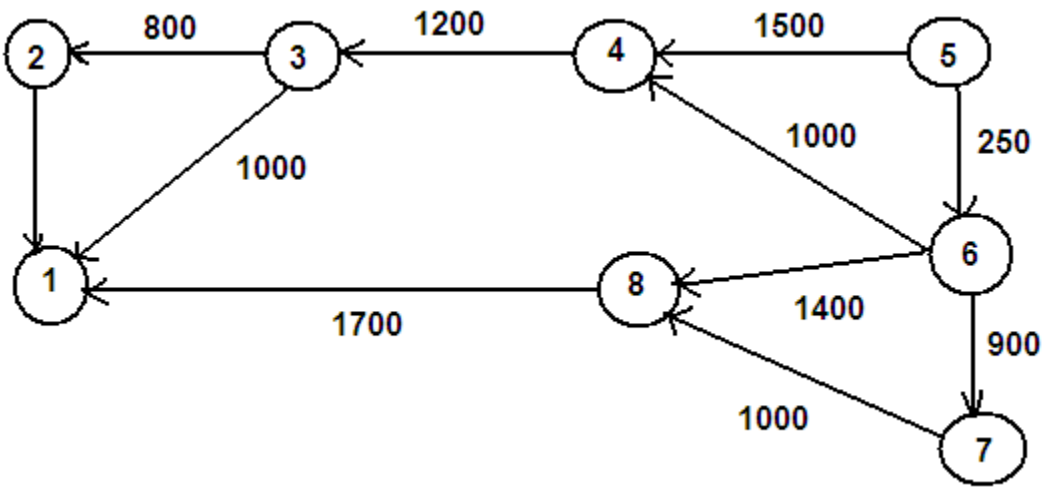
Figures shows the stages in Sollin's algorithm when it begins with the graph of fig (a). The initial configuration of zero selected edges is the same as that shows in fig(b) .Each tree in this spanning forest is a single vertex. The edges selected by vertices 0,1,…,6 are , respectively , (0,5),(1,6),(2,3),(3,2),(4,3),(5,0), and (6,1). The distinct edges in this selection are (0,5),(1,6),(2,3), and (4,3). Adding these to the set of selected edges results in the configuration

of fig(c). In the next stage, the tree with vertex set {0,5} selects the edge (5,4),and the remaining two trees select the edge (1,2).Following the addition of these two edges to the set of selected edges, construction of the spanning tree is complete. The resulting spanning tree is shown in fig (d).

SINGLE SOURCE SHORTEST PATH ALGORITHM

DJIKSTRA'S ALGORITHM

The Djikstra's algorithm finds out the shortest path between the single source and every other node in the graph. For example consider the following graph. Let the node 5 be the source. Let solve this using Djiksta's algorithm to find the shortest paths between 5 and every other node in the graph.



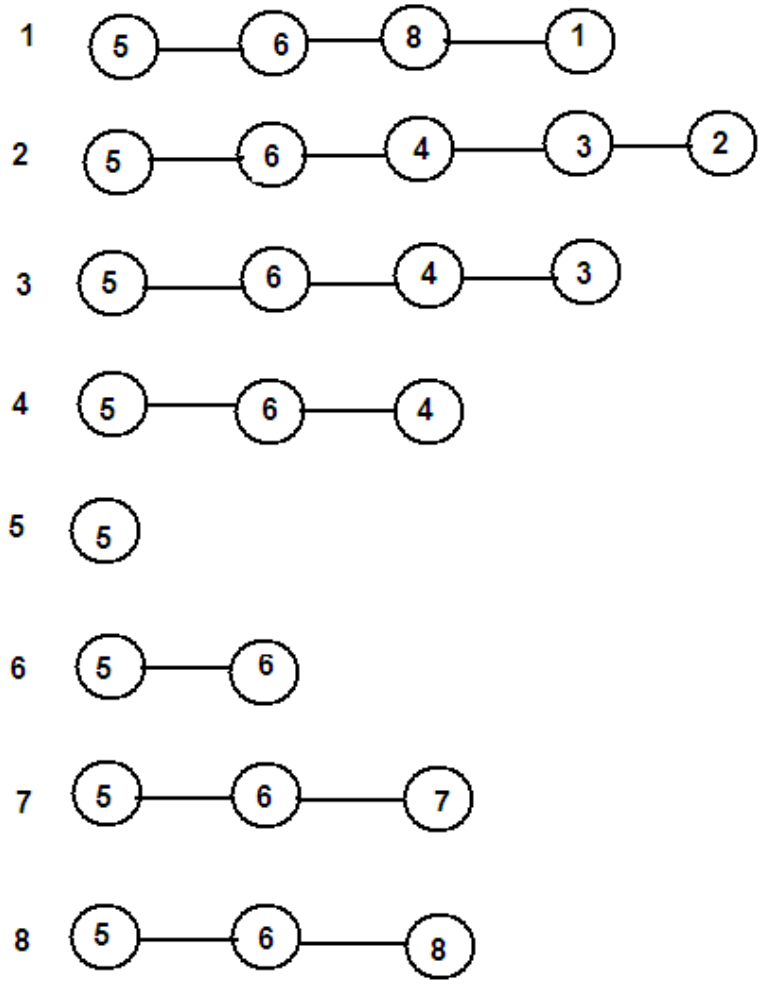
A Distance array Dist[] is initially filled with infinity. The entry corresponding to the source node 5 alone is made 0. Now find out the adjacent nodes of 5 and update their values using the cost matrix. In the next iteration the node with minimum distance is the vertex selected and again the above process is repeated. The Column S shows the set of vertices already selected. In every iteration the node with minimum distance and which is not yet selected is taken as the new vertex.
The solution is obtained as follows.

Iteration	S	Vertex Selected	Dist[ ]							
			1	2	3	4	5	6	7	8
Initial	-	-	∞	∞	∞	1500	0	250	∞	∞
	5	6	∞	∞	∞	1250	0	250	1150	1650
	5,6	7	∞	∞	∞	1250	0	250	1150	1650
	5,6,7	4	∞	∞	2450	1250	0	250	1150	1650
	5,6,7,4	8	3350	∞	2450	1250	0	250	1150	1650

	5,6,7,4,8	3	3350	3250	2450	1250	0	250	1150	1650
	5,6,7,4,8,3	2	3350	3250	2450	1250	0	250	1150	1650

Now we can see that the values in the last row, gives the distance of the shortest path between the source node 5 and every other node in the graph.

The shortest paths corresponding to this can also be generated. The paths are represented using linked lists. Whenever a value in the distance array is updated, the shortest path linked lists are also adjusted.
The shortest paths are represented using linked lists as shown.



Algorithm

DJIKSTRA(v, cost, dist, n)

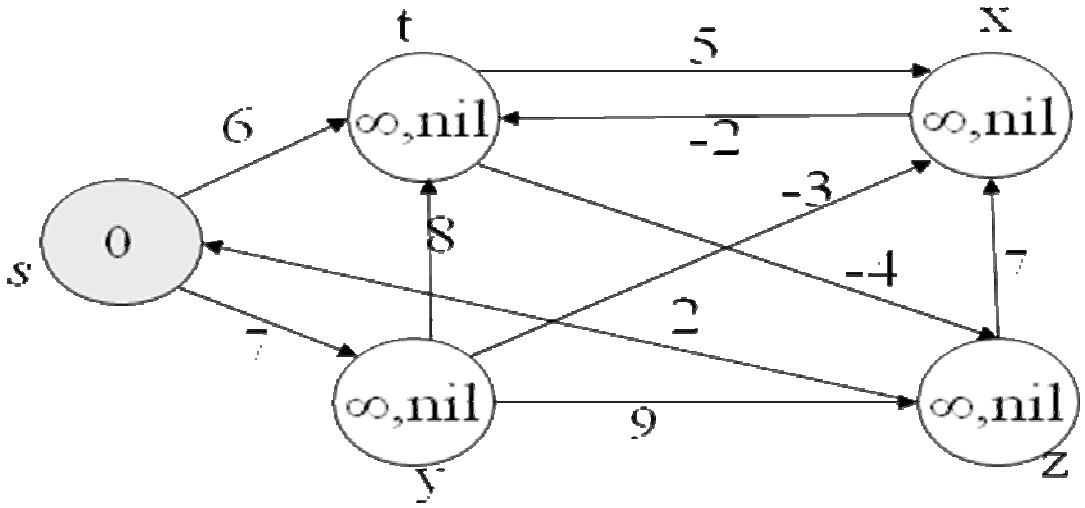
```
For i = 1 to n
        S[i] = false
        Dist[i] = ∞
End for
S[v] = true
```

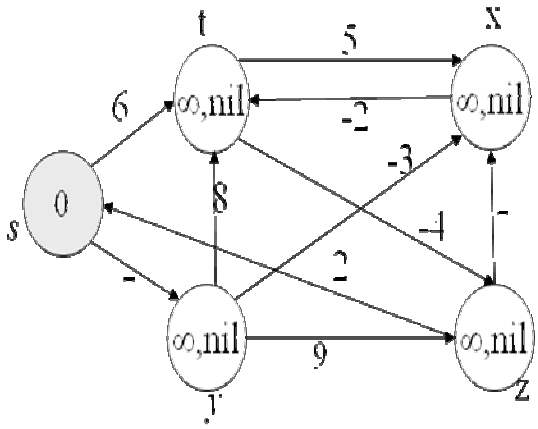
```
Dist[v] = 0
Create n lists each beginning with v
For num = 1 to n – 1
        Choose u from among those vertices not in S such that dis[u] is minimum
        S[u] = true
        For each w adjacent to u with s[w] = false
                If Dist[w] > Dist[u] + cost[u, w]
                        Dist[w] = Dist[u] + cost [u, w]
                        List[w] = List[u] + w
                End if
        End for
End for
```

THE BELLMAN-FORD SHORTEST PATH ALGORITHM

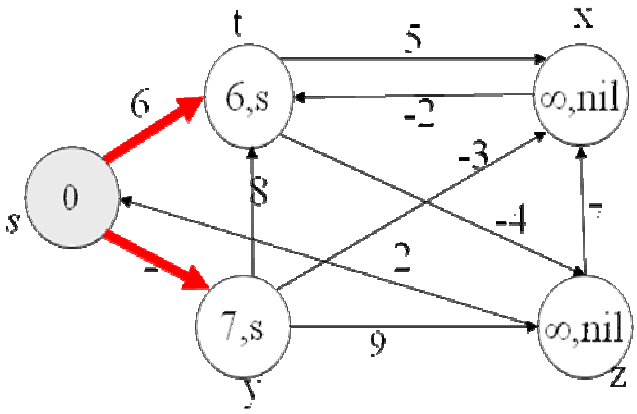
Given a weighted graph G and a source vertex s, Bellman-Ford algorithm finds the shortest (minimum cost) path from s to every other vertex in G. The weighted path length (cost) is the sum of the weights of all links on the path.

.

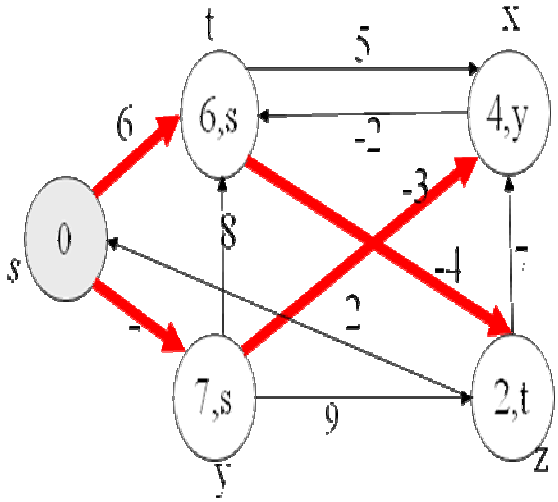




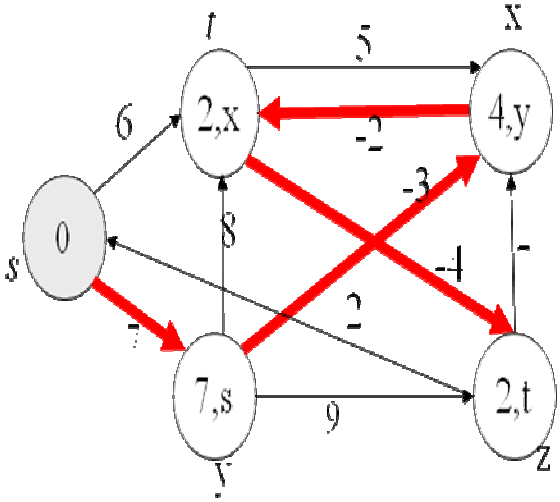
Initialisation



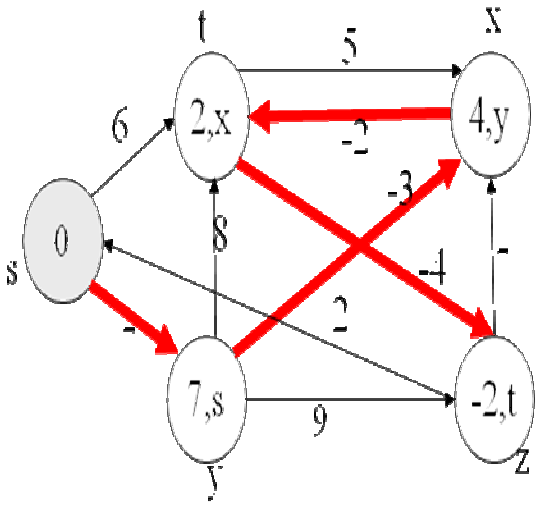
After Pass1



After Pass 2



After Pass3



After Pass 4

The order of edges examined in each pass:
(t, x), (t, z), (x, t), (y, x), (y, t), (y, z), (z, x), (z, s), (s, t), (s, y)

Algorithm

Bellman-Ford(G, w, s)

```
1.  Initialize-Single-Source(G, s)
2.  for i := 1 to |V| - 1 do
3.          for each edge (u, v) ∈ E do
4.                  Relax(u, v, w)
5.  for each vertex v ∈ u.adj do
6.          if d[v] > d[u] + w(u, v)
7.                  then return False        // there is a negative cycle
8.  return True

Relax(u, v, w)
 if d[v] > d[u] + w(u, v)
        then  d[v] := d[u] + w(u, v)
              parent[v] := u
```

Difference between Djikstra's and Bellman Ford Algorithm

Sl.No	Djikstra's Algorithm	Bellman Ford Algorithm
1	It doesn't work for Negative Link weight	It works for Negative Link weight

2	It cann't be implemented in a distributed way	It can be easily implemented in a distributed way
3	It has less time complexity	It has higher time complexity

ALL PAIRS SHORTEST PATH ALGORITHM

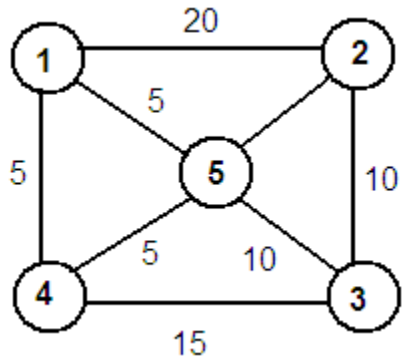
FLOYD WARSHALL ALGORITHM

The All pairs shortest path algorithm is used to find the shortest path between every pair of nodes in the graph. Consider the following example.

We will use the following condition solve the problem

A[i, j] = min(A[i, j], A[i, k] + A[k, j])

Solution is derived as follows using the above condition.



Cost Matrix

	1	*2*	*3*	*4*	*5*
1	0	20	∞	5	5
2	20	0	10	∞	10
3	∞	10	0	15	10
4	5	∞	15	0	5
5	5	10	10	5	0

Iteration 1 Iteration 2

	1	*2*	*3*	*4*	*5*
1	0	20	∞	5	5
2	20	0	10	∞	10
3	∞	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

	1	2	3	4	5
1	0	20	30	5	5
2	20	0	10	∞	10
3	30	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

Iteration 3

	1	2	3	4	5
1	0	20	30	5	5
2	20	0	10	25	10
3	30	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

Iteration 4

	1	2	3	4	5
1	0	20	20	5	5
2	20	0	10	25	10
3	20	10	0	15	10
4	5	25	15	0	5
5	5	10	10	5	0

Iteration 5

	1	2	3	4	5
1	0	15	15	5	5
2	15	0	10	15	10
3	15	10	0	15	10
4	5	15	15	0	5
5	5	10	10	5	0

Final Output matrix

	1	2	3	4	5
1	0	15	15	5	5
2	15	0	10	15	10
3	15	10	0	15	10
4	5	15	15	0	5
5	5	10	10	5	0

The output matrix gives the shortest path distance between all pairs of nodes.

Algorithm

```
ALLPAIRSHORTESTPATH(cost, A, n)
For i = 1 to n
        For j = 1 to n
                A[i, j] = cost[i, j]
        End for
For k = 1 to n
        For i = 1 to n
                For j = 1 to n
                        A[i, j] = min(A[i, j], A[i, k] + A[k, j])
                End for
        End for
End for
```