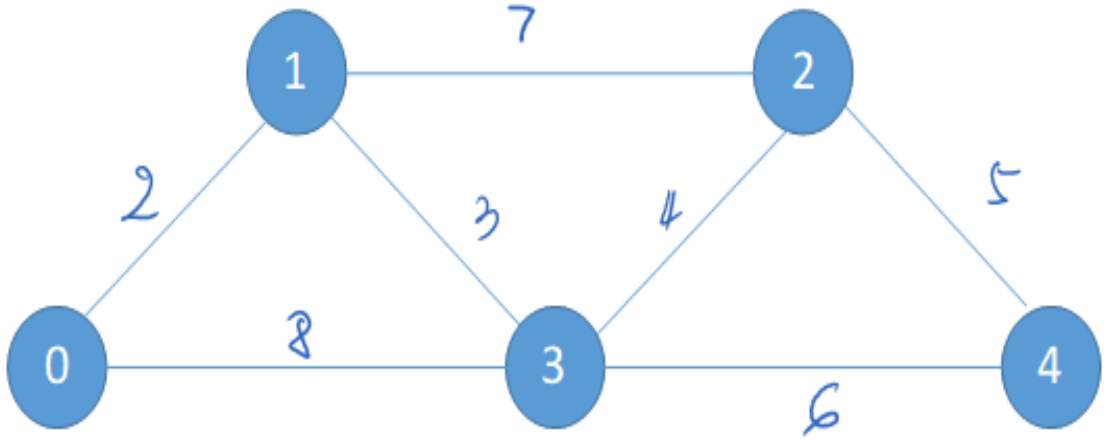
Shortest Path Algorithms

Representing Weighted Graphs

- Representing Weighted Edges: Edge Array
- Weighted Adjacency matrices
- Adjacency lists – it combines adjacency matrices with edge lists

Representing Weighted Edges: Edge Array




```
class Edge
{
        int StartVertex;
        int EndVertex;
        int weight;
};
```

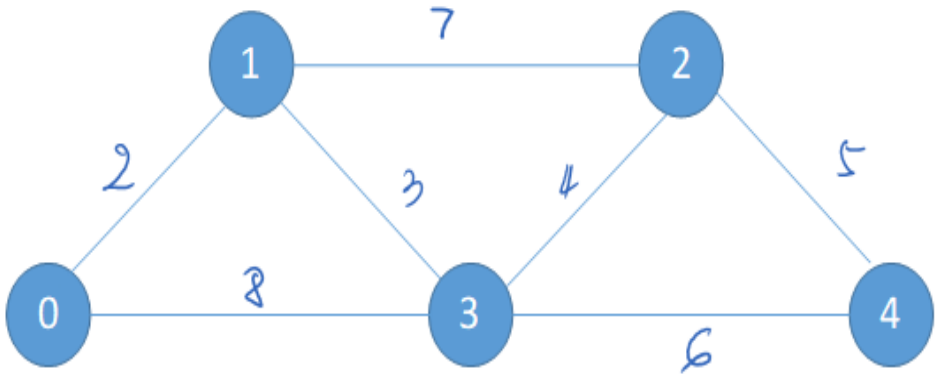
```
Struct Edge
{
        int StartVertex;
        int EndVertex;
        int weight;
};
```

Edge List		
StartVertex	EndVertex	Weight
0	1	2
0	3	8
1	0	2
1	3	3
1	2	7
2	1	7
2	3	4
2	4	5
3	0	8
3	1	3
3	2	4
3	4	6
4	2	5
4	3	6

Weighted Adjacency Matrices

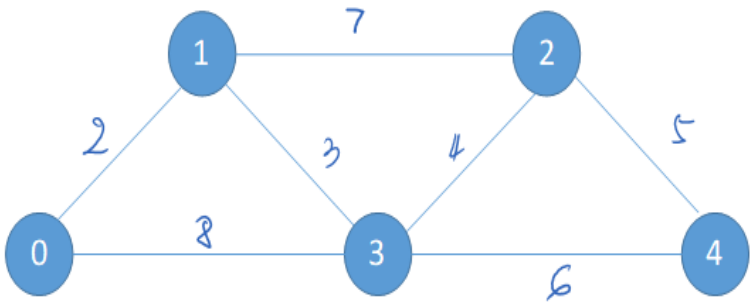
```
Integer[][] adjacencyMatrix = {
  {null, 2, null, 8, null },
  {2, null, 7, 3, null },
  {null, 7, null, 4, 5},
  {8, 3, 4, null, 6},
  {null, null, 5, 6, null}
```

	0	1	2	3	4
0	null	2	null	8	null
1	2	null	7	3	null
2	null	7	null	4	5
3	8	3	4	null	6
4	null	null	5	6	null



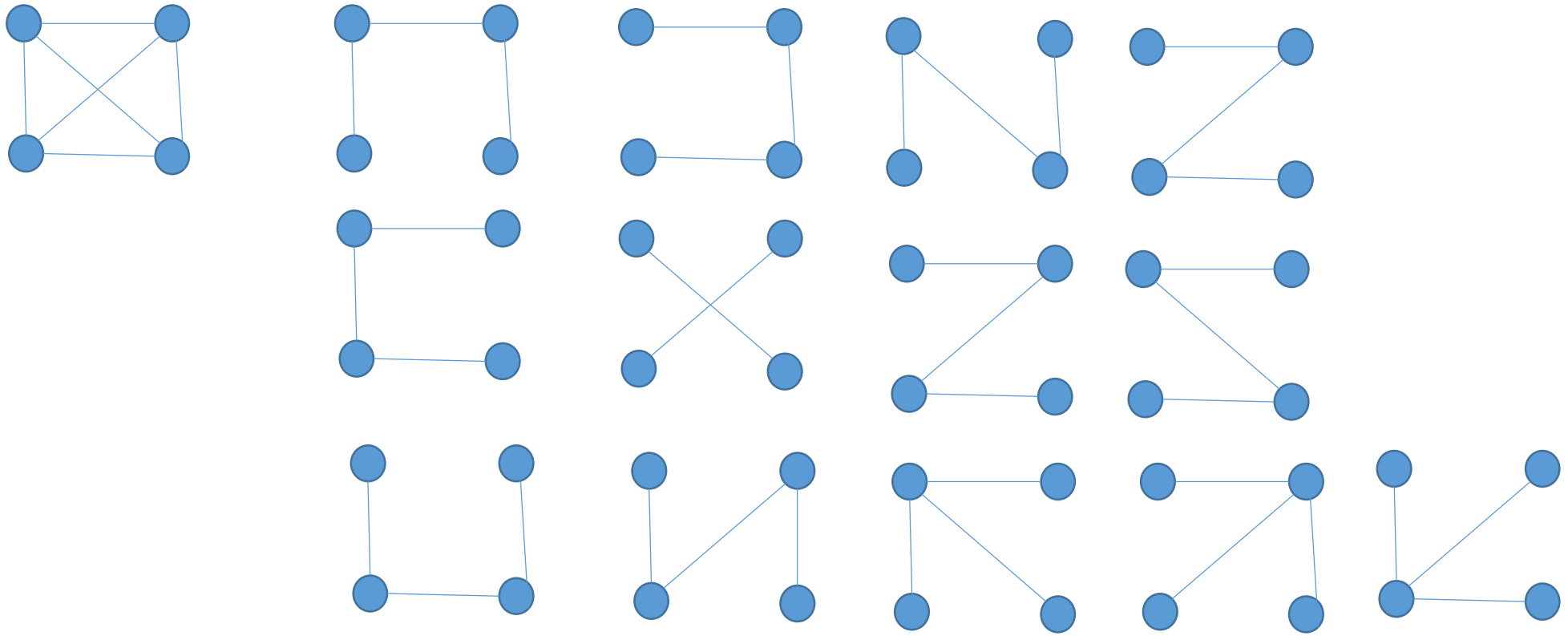
Adjacency lists

Neighbors	WeightedEdge	WeightedEdge	WeightedEdge	WeightedEdge
0 →	(0, 1, 2)	(0, 3, 8)		
1 →	(1, 0, 2)	(1, 3, 3)	(1, 2, 7)	
2 →	(2, 3, 4)	(2, 4, 5)	(2, 1, 7)	
3 →	(3, 1, 3)	(3, 2, 4)	(3, 4, 6)	(3, 0, 8)
4 →	(4, 2, 5)	(4, 3, 6)		



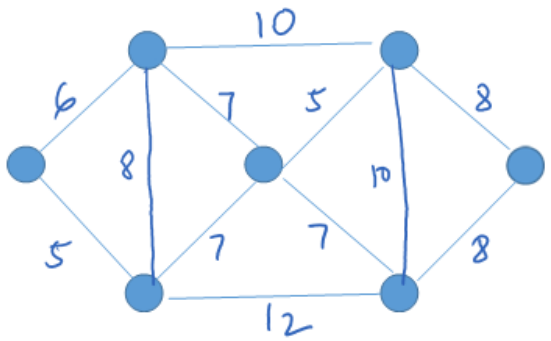
Minimum Spanning Trees

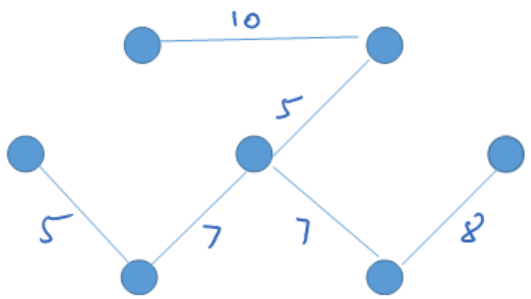
- A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees; for instance the complete graph on four vertices can have many spanning trees.

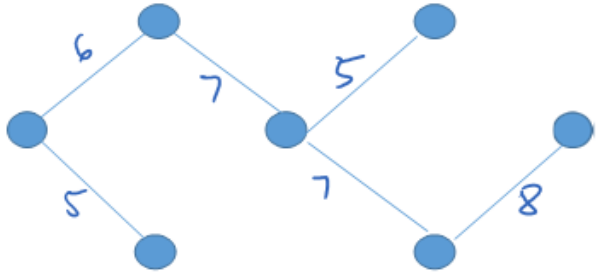


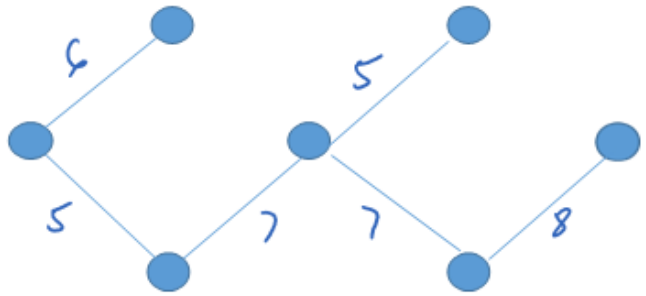
Minimum Spanning Trees

- Now suppose the edges of the graph have weights or lengths. The weight of a tree is just the sum of weights of its edges. Obviously, different trees have different lengths.









Why Minimum Spanning Trees?

- The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.
- A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Minimum Spanning Tree Algorithms

- How to find a minimum spanning tree? There are several well-known algorithms for doing so. We will introduce Kruskal's algorithm and Prim's Algorithm.
- Let's start with Kruskal's algorithm, which is easier to understand and probably the best one for solving problems by hand.

Kruskal’s Algorithm

Kruskal's Algorithm

1. Sort all the edges of G in *non-decreasing* order of their weight.
2. Keep a subgraph S of G, initially empty

    for each edge E in sorted order

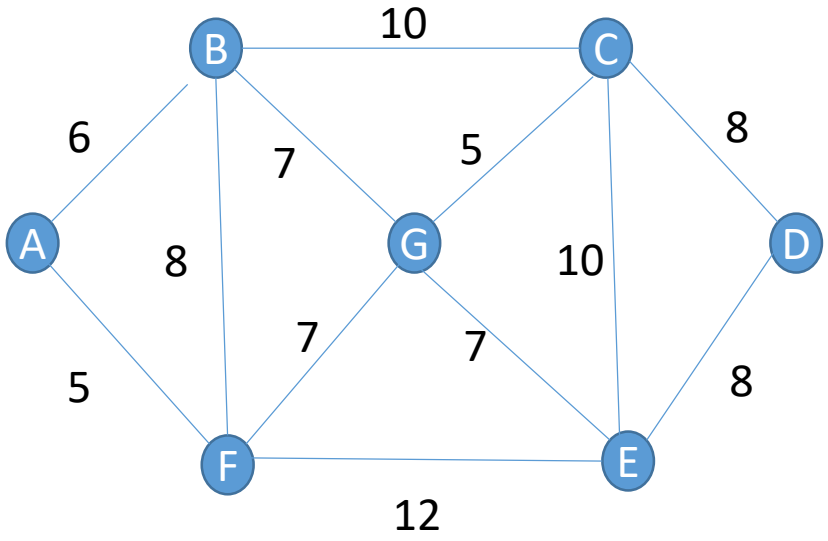
        If the endpoints of E are not formed the cycle in S, add E to S

        otherwise discard it.

    return S;

3. Repeat step #2 until there are (V-1) edges in the spanning tree, where V is the number of vertices.

Kruskal's Algorithm



Sorted weighted edges

AF = 5

CG = 5

AB = 6

BG = 7

FG = 7 (no)

EG = 7

DE = 8

CD = 8 (No)

BF = 8 (no)

BC = 10 (no)

CE = 10 (no)

EF = 12 (no)

MST = 38

AF = 5

CG = 5

AB = 6

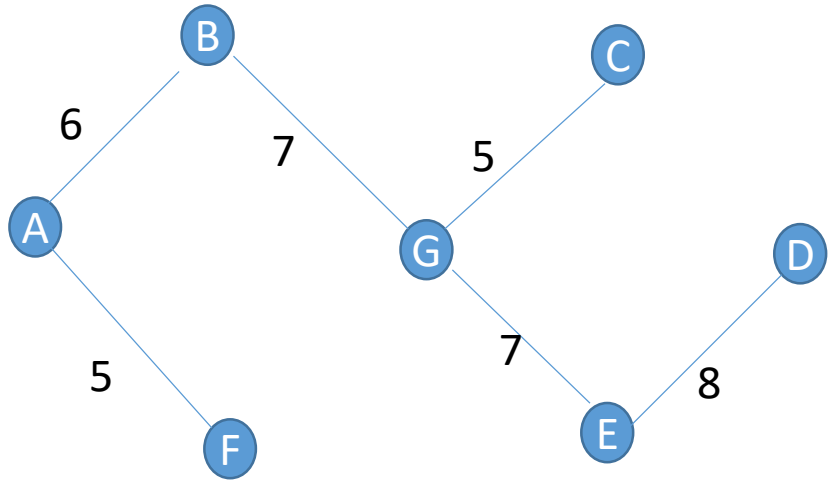
BG = 7

EG = 7

DE = 8

Vertices = 7

Edges = 7- 1 = 6



Kruskal's minimum spanning tree

Analysis: Kruskal's Algorithm

"If the endpoints of E are not formed the cycle in S, add E to S"

- The line testing whether two endpoints are not formed the cycle it should be slow (linear time per iteration, or O(mn) total).
- But actually there are some complicated data structures that let us perform each test in close to constant time; this is known as the *union-find* problems.
- The slowest part turns out to be the sorting step, which takes **O(m log n)** time.

Kruskal's Implementation

```
class Edge
{
    int StartVertex;
    int EndVertex;
    int weight;
};
```

```
class Graph
{
    int V, E;
    int Edge* edge;
 //graph is represented as an
array of edges.
};
```

```
Graph* createGraph(int V, int E)
{
      Graph* graph = new Graph;
      graph->V = V;
      graph->E = E;
      graph->edge = new Edge[E];
      return graph;
};
```

```
//  Disjoint Sets Data Structure
int parents[100];

int find(int x) {
   if (parents[x] == x) {
      return x;
   }
   return find(parents[x]);
}
void unite(int x, int y) {
   int px = find(x);
   int py = find(y);
   parents[px] = py;
}
```

Prim’s Algorithm

```
1  getShortestPath(s)    {
3    Let T be a set that contains the vertices whose
4      paths to s are known;
5    Set cost[v] = infinity for all vertices;
6    Set cost[s] = 0 and parent[s] = -1;
7
8    while (size of T < n)
9    {
10     Find u not in T with the smallest cost[u];
11     Add u to T;
12     for (each v not in T and (u, v) in E)
13     {
14       if (cost[v] > w(u, v))  //w(u, v) is weight on edge (u, v)
15             cost[v] = w(u, v); // cost[v] is the smallest weight among all
         V's edges.
16             parent[v] = u;
16     }
17   }
```

Prim's Algorithm

Input: a graph G = (V, E) with nonnegative weights and a source vertex s

Output: a shortest path tree with the source vertex s as the root

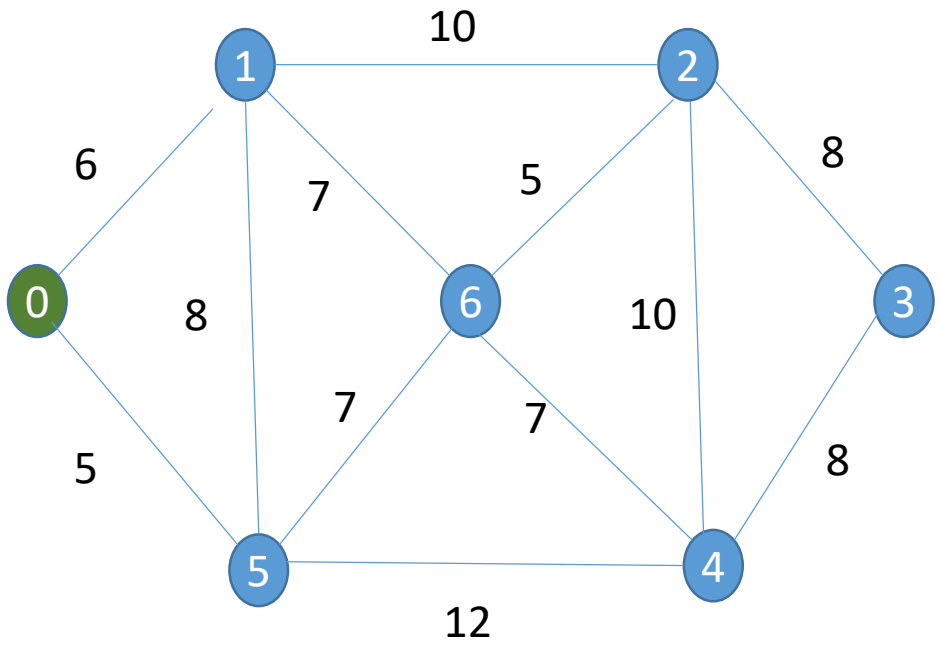
Analysis: Prim's Algorithm

- Rather than build a subgraph one edge at a time, Prim's algorithm builds a tree one vertex at a time. At each step adding the cheapest possible connection from the tree to another vertex.
- Every time after a vertex u is added into T, the function updates cost[v] and parent[v] for each vertex v adjacent to u. The total time for finding the vertex to be added into T is O(V^2).
- Each edge incident to u is examined only once. So the total time for examining the edges is O(E), there E denotes the number of edges.
- Therefore the time complexity of this implementation is O(V^2 + E) = O(V^2)
- **Note:** "this implementation" is adjacent matrix implementation. That meant the time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight.

Pick 0 as a starting vertex (s). Set cost[s] = 0 and cost[i] to infinity for all other vertices.

parent[0] is set to -1, which indicates that vertex 0 is the root.

Initially, T is empty. The algorithm will select the vertex with the smallest cost and add it to T in the next step. In this case, the vertex is 0, which will be added to T.



cost[v]

0	inf	inf	inf	inf	inf	inf
0	1	2	3	4	5	6

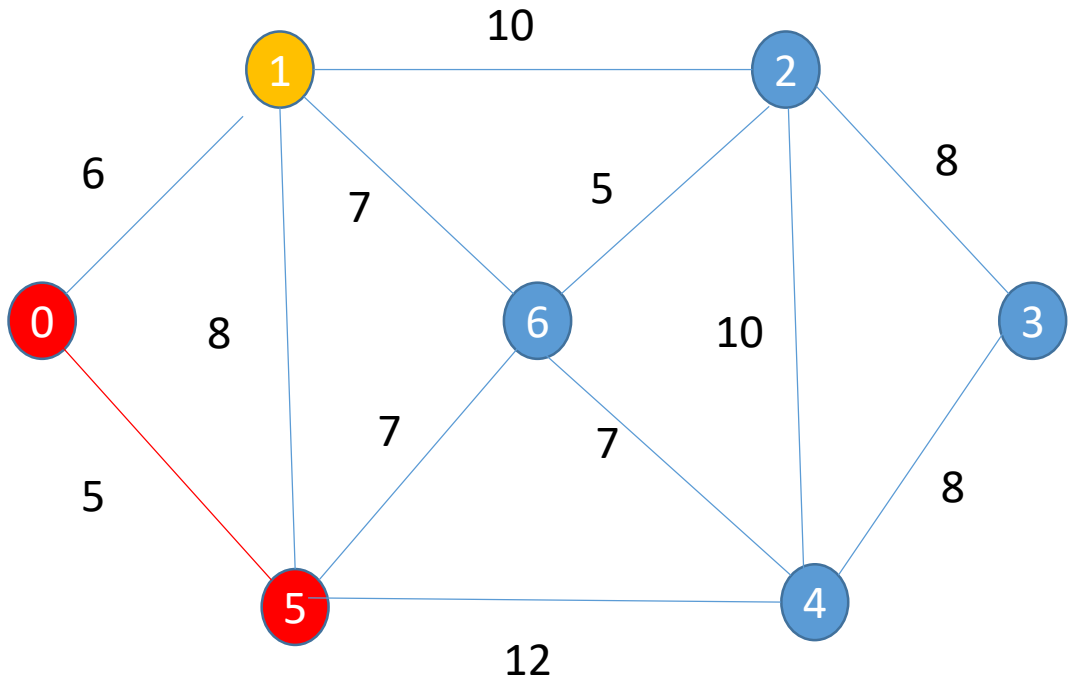
parent[s]

-1						
0	1	2	3	4	5	6

T: { }

After node is added into T, adjust cost[v] for each v adjacent to node 0, which are node 1 and 5. Update the cost of 1 and 5; update parents of node 1 and 5 to node 0. Find u not in T with the smallest cost[u] => node 5. Node 5 is picked and added int to T.

At this moment: w(u, v) is 5 and cost[v] is 5. nothing to update.



cost[v]

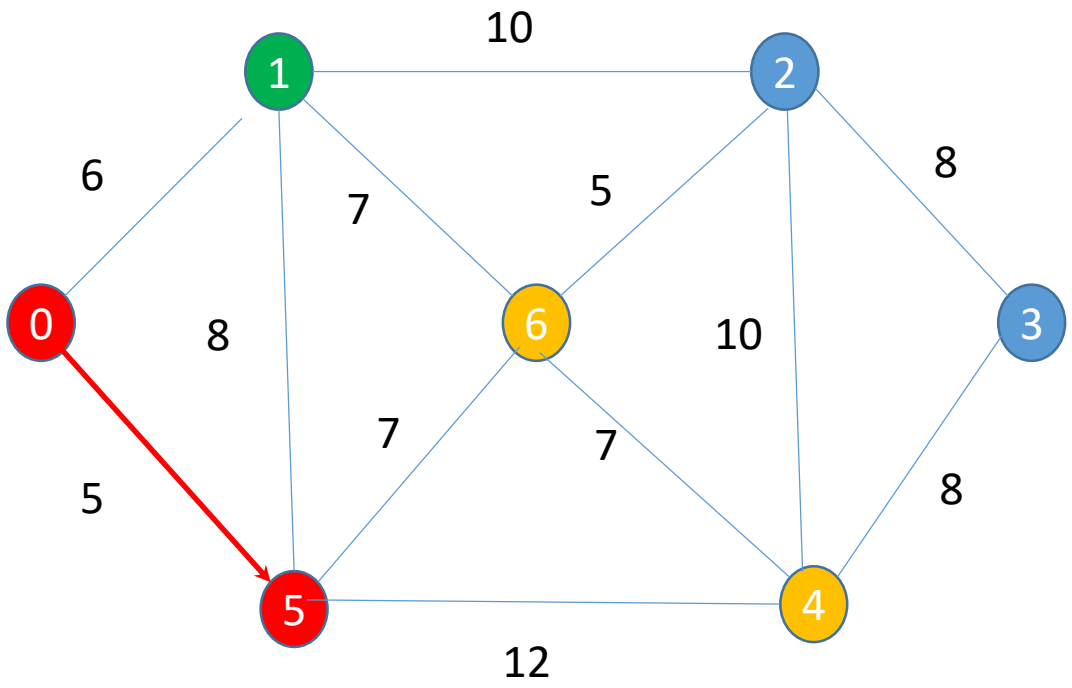
0	6	int	inf	inf	5	inf
0	1	2	3	4	5	6

parent[s]

-1	0				0	
0	1	2	3	4	5	6

T: {0,5}

Repeat the procedure, starting at vertex 5. Vertex 5 connects to 1, 6, and 4. Update cost of 6 and 4; update the parent of 6 and 4.



cost[v]

0	6	int	inf	12	5	7
0	1	2	3	4	5	6

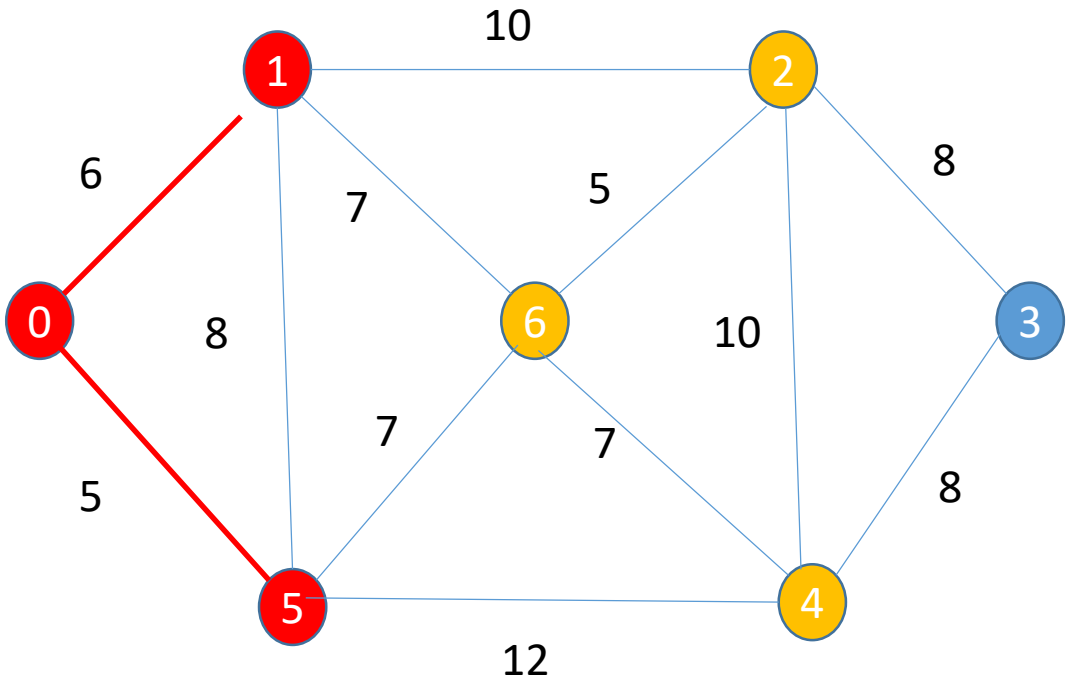
parent[s]

-1	0			5	0	5
0	1	2	3	4	5	6

T: {0,5}

The cost of vertex 1 is the lowest one, vertex 1 is the root. Visit 1.

Vertex 1 connects to 2 and 6. Update the cost and parent of vertex 2.



cost[v]

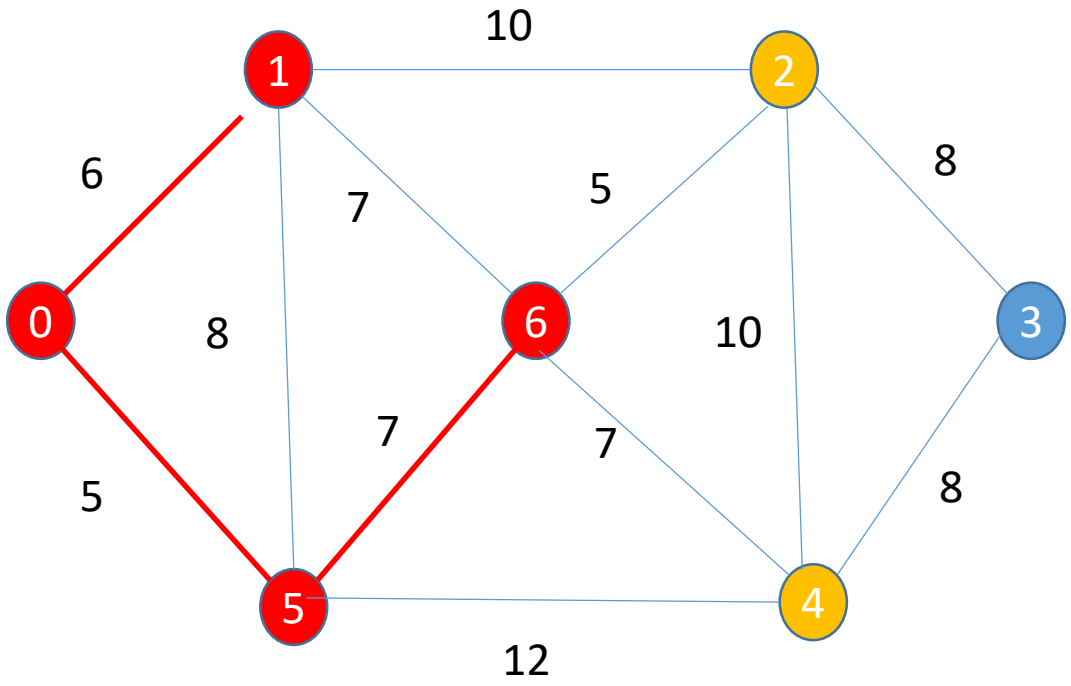
0	6	10	inf	12	5	7
0	**1**	2	3	4	**5**	6

parent[s]

-1	0	1		5	0	5
0	**1**	2	3	4	**5**	6

T: {0,5,1}

The cost table indicates vertex 6 is lowest one with parent is 5. Visit 6.
Vertex 6 is now the root; it connects to 2 and 4.
For each v not in T and (u, v) in E:
If (cost[v] > w(u,v)) then cost[v] = w(u,v);



cost[v]

0	6	10	inf	7	5	7
0	1	2	3	4	5	6

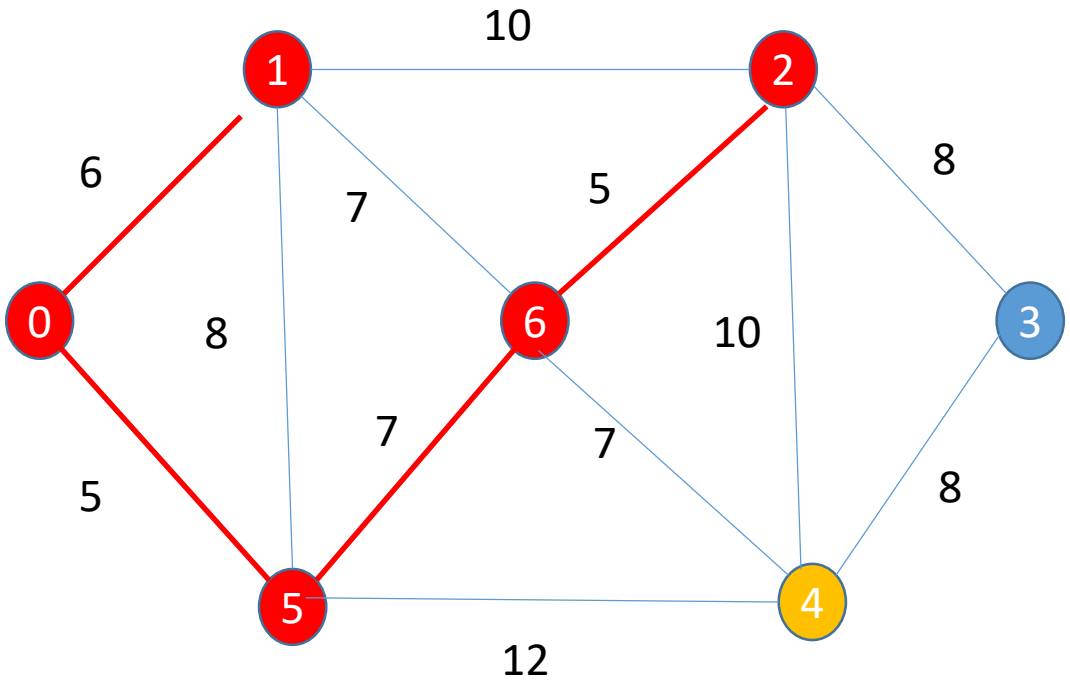
parent[s]

-1	0	1		6	0	5
0	1	2	3	4	5	6

T: {0,5,1,6}

Vertex 6 connects to 2 and 4. Cost from 6 to 2 if 5; modify cost and update the parent of 2.

Vertex 2 has a lowest cost. Visit 2.



cost[v]

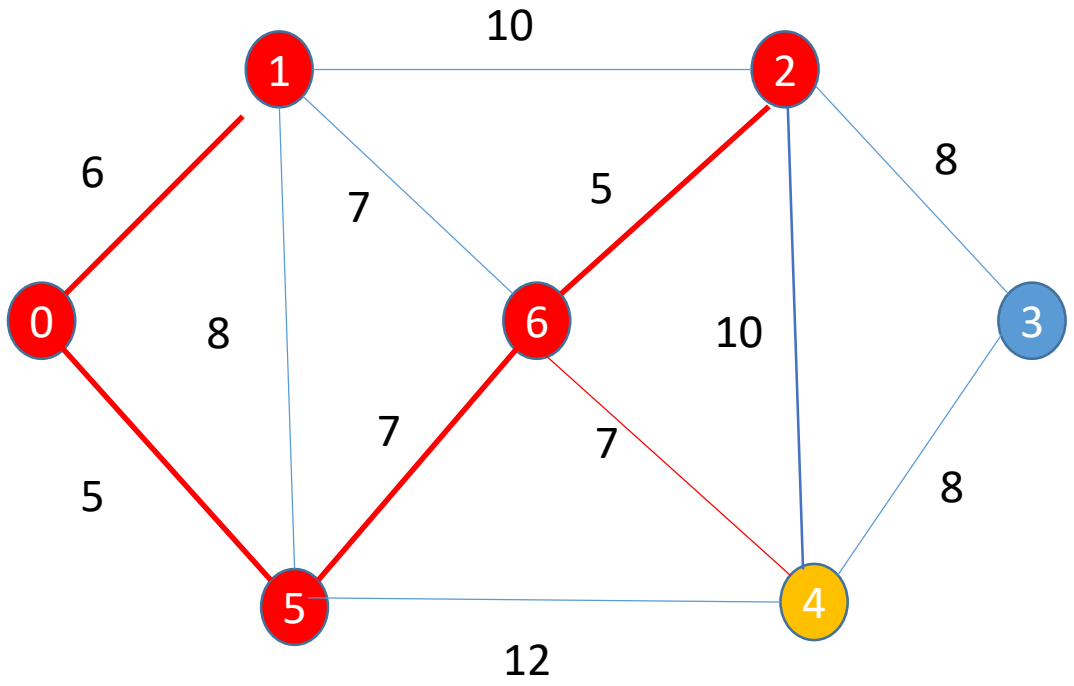
0	6	5	8	7	5	7
0	1	2	3	4	5	6

parent[s]

-1	0	6	2	6	0	5
0	1	2	3	4	5	6

T: {0,5,1,6,2}

Vertex 2 has a lowest cost. Visit 2. Vertex 2 connects to 3 and 4. Update cost and parent of vertex 3. *Don't need to change cost of node 4; the cost[4] is 7 which is lower than w(2, 4).*



cost[v]

0	6	5	8	7	5	7
0	1	2	3	4	5	6

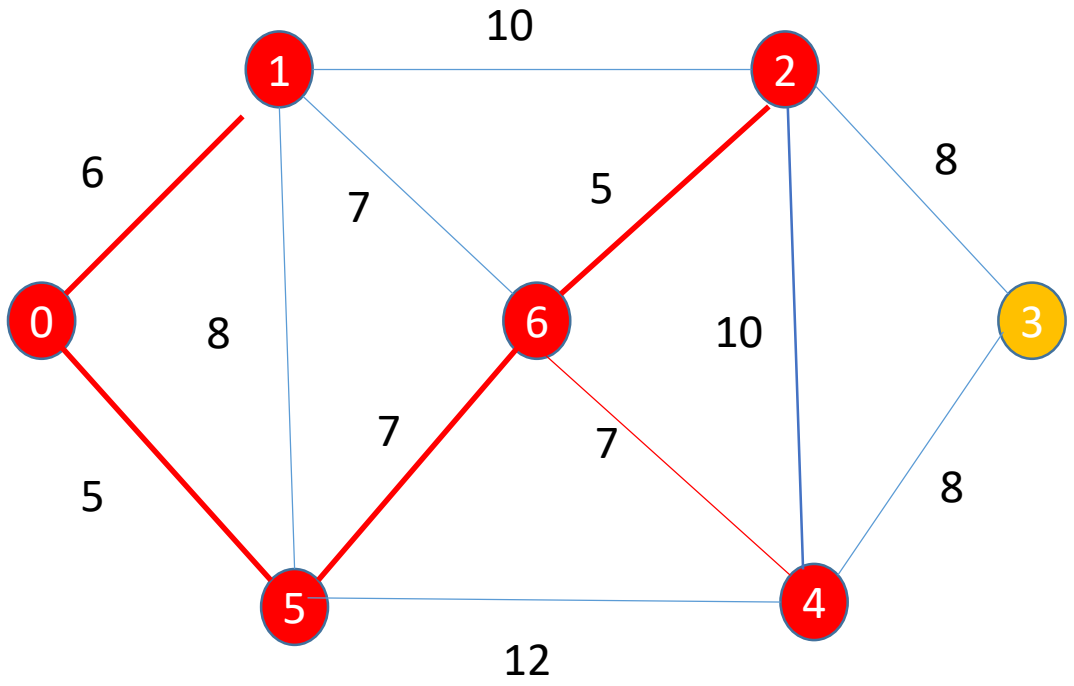
parent[s]

-1	0	6	2	6	0	5
0	1	2	3	4	5	6

T: {0,5,1,6,2}

The cost table indicates vertex 4 with parent 6 is lowest cost.

Visit 4.



cost

0	6	5	8	7	5	7
0	1	2	3	4	5	6

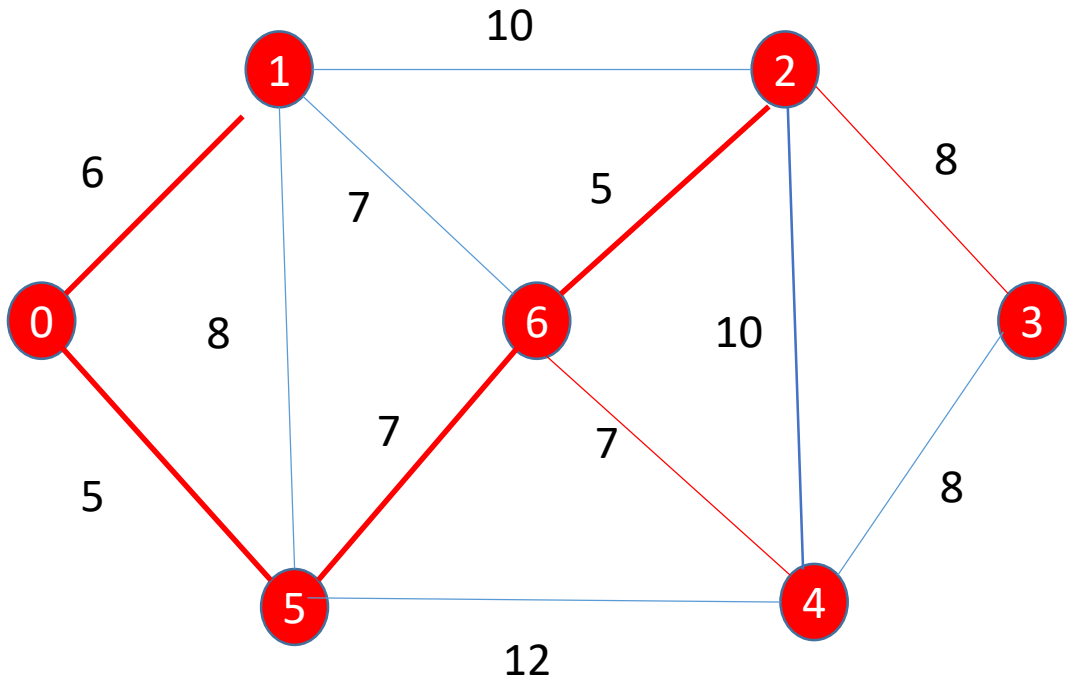
parent

-1	0	6	2	6	0	5
0	1	2	3	4	5	6

T: {0,5,1,6,2,4}

To connect 3, either we can start at vertex 4 or vertex 2 as both has the same cost.

Note: a minimum spanning tree is not unique. If the weighs are distinct, the graphs has unique minimum spanning tree.



cost

0	6	5	8	7	5	7
0	1	2	3	4	5	6

parent

-1	0	6	2	6	0	5
0	1	2	3	4	5	6

```
T: {0,5,1,6,2,4,3}
Total cost: 38
```

Find Prim's MST using Distance Matrix

Steps - Rows and columns present nodes

1. Choose any vertex to start, highlight the chosen column
2. Delete row from chosen vertex
3. Number column in the matrix for chosen vertex
4. Find the lowest undeleted entry in numbered column(s). Highlight the column corresponding to the lowest node.
5. Repeat step 3 until finish all columns.

If nodes = n; then edges = n – 1;

Example - Prim's MST using Distance Matrix

	1	3	5	7	6	2	4
	A	B	C	D	E	F	G
A	0	6	inf	inf	inf	5	inf
B	6	0	10	inf	int	8	7
C	inf	10	0	8	10	inf	5
D	inf	inf	8	0	8	inf	inf
E	inf	inf	10	8	0	12	7
F	5	8	inf	inf	12	0	7
G	inf	7	5	inf	7	7	0

Edges:

FA = 5

AB = 6

GB = 7

CG = 5

EG = 7

DC = 8

Total distance: 38

The End.