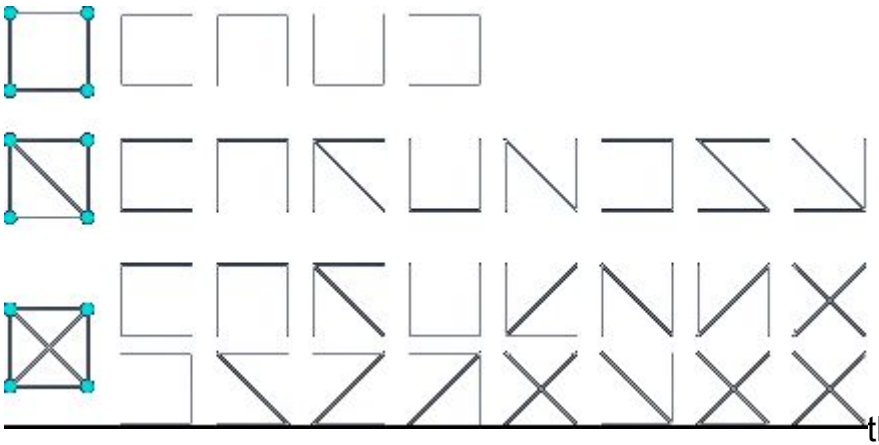
Spanning Tree

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 can not be disconnected.

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

Examples



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 number of nodes. In addressed example, n is 4, hence $4^{4-2} = 16$ maximum number of spanning trees are possible.

General properties of spanning tree

We now understand that one graph can have more than one spanning trees. Below are few properties is spanning tree of given connected graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have same number of edges and vertices.
- Spanning tree does not have any cycle loops
- Removing one edge from spanning tree will make the graph disconnected i.e. spanning tree is minimally connected.
- Adding one edge to a spanning tree will create a circuit or loop i.e. spanning tree is maximally acyclic.

Mathematical properties of spanning tree

- Spanning tree has n-1 edges, where n is 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.

Minimum Spanning Tree

A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Application of Minimum Spanning Tree

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.

2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm.

Kruskal's Algorithm

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

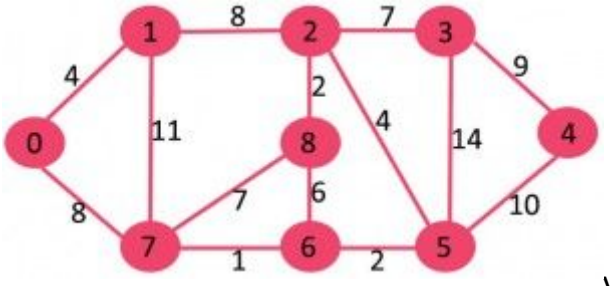
1. Sort all the edges from low weight to high

2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge **created a cycle**, then reject this edge.

3. Keep adding edges until we reach all vertices.

Example

Lets consider the following example



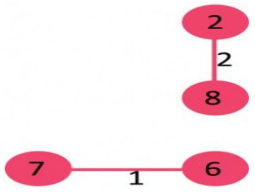
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting

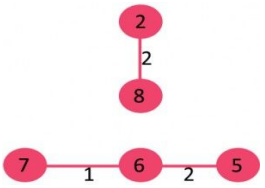
Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges
1. *Pick edge 7-6:* No cycle is formed, include it i.e. {7, 6}

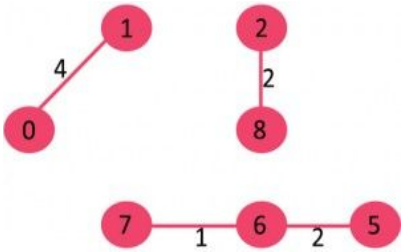
2. *Pick edge 8-2:* No cycle is formed, include it i.e. {7, 6}, {8, 2}



3. *Pick edge 6-5:* No cycle is formed, include it i.e.{7, **6**}, {**6**, 5}, {8, 2}={7, 6, 5},{8, 2}

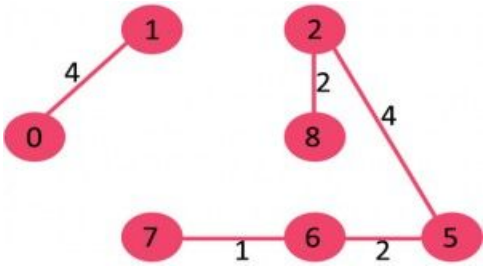


4. *Pick edge 0-1:* No cycle is formed, include it i.e.{7, 6, 5},{8, 2},{0, 1}



5. *Pick edge 2-5:* No cycle is formed, include it i.e.

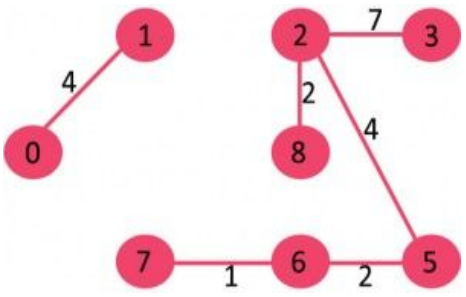
{7, 6, 5},{8, **2**},{**2**, 5},{0, 1}={7, 6, **5**},{8, 2, **5**},{0, 1}={ 7, 6, 5, 8, 2},{0, 1}



6. Pick edge 8-6: Since including this edge results in cycle, discard it because both the vertices of edge {8, 6} present in one subset i.e.{ 7, **6**, 5, **8**, 2},{0, 1}

7. Pick edge 2-3: No cycle is formed, include it i.e.

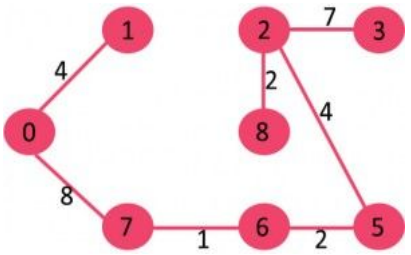
{ 7, 6, 5, 8, **2**},{**2**, 3},{0, 1}={ 7, 6, 5, 8, 2, 3},{0, 1}



8. Pick edge 7-8: Since including this edge results in cycle, discard it because both the vertices of edge {7, 8} present in one subset i.e.{ 7, 6, 5, 8, 2, 3},{0, 1}

9. Pick edge 0-7: No cycle is formed, include it i.e.

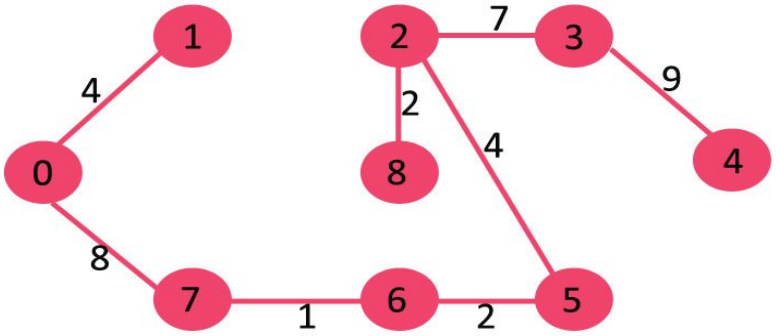
{ 7, 6, 5, 8, 2, 3},{0, 1},{0, 7}={ 7, 6, 5, 8, 2, 3}{0, 1, 7}={ 7, 6, 5, 8, 2, 3, 0, 1}



10. Pick edge 1-2: Since including this edge results in cycle, discard it. because both the vertices of edge {1, 2} present in one subset i.e.{ 7, 6, 5, 8, 2, 3, 0, 1}

11. Pick edge 3-4: No cycle is formed, include it i.e.

{ 7, 6, 5, 8, 2, 3, 0, 1},{3, 4}={ 7, 6, 5, 8, 2, 3, 0, 1, 4}



Since the number of edges included equals (V – 1), the algorithm stops here.

Conclusion

1. If **both the vertex** of the pick edge is **present in one subset** then this pick edge will be discard because **cycle** will formed(i.e. step 6, 8, 10 of the above example)

2. If **both the vertex** of the pick edge is **present in different subset** then this pick edge will be include and two subset will be merge(i.e. step 5, 9 of the above example)

3. If **any one of the vertex** of the pick edge is **present in one subset** then this pick edge will be include and merge with the subset.(i.e. step 3, 7, 11 of the above example)

4. If **no vertex** of the pick edge is **present in any subset** then this pick edge will be include but not merge with any subset(i.e. step 1, 2, 4 of the above example)

Algorithm

```
Algorithm kruskal()
{
    sort E in increasing order by weight w;        //O(|E| log |E|)
    A={Φ};                                         //O(1)
    for each vertex v Є G.V                        //O(|V|)
        create_set(v);
    for each edge (u_i, v_i) Є G.E do              //O(|E| log |V|)
        if(Find_set(u_i) ≠ Find_set(v_i))
        {
            A=A U {(u_i, v_i)};
            union(u_i, v_i);
        }
    return(A);
}
```

Time complexity:

$$T(n) = O(|E| \log |E|) + O(1) + O(|V|) + O(|E| \log |V|)$$

$$= O(|E| \log |E|) + O(|E| \log |V|)$$

as $|E| \geq |V| - 1$ //number edges dominant number of vertices

$$T(n) = O(|E| \log |E|) + O(|E| \log |E|)$$

$$= O(|E| \log |E|)$$

Union-Find

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not.