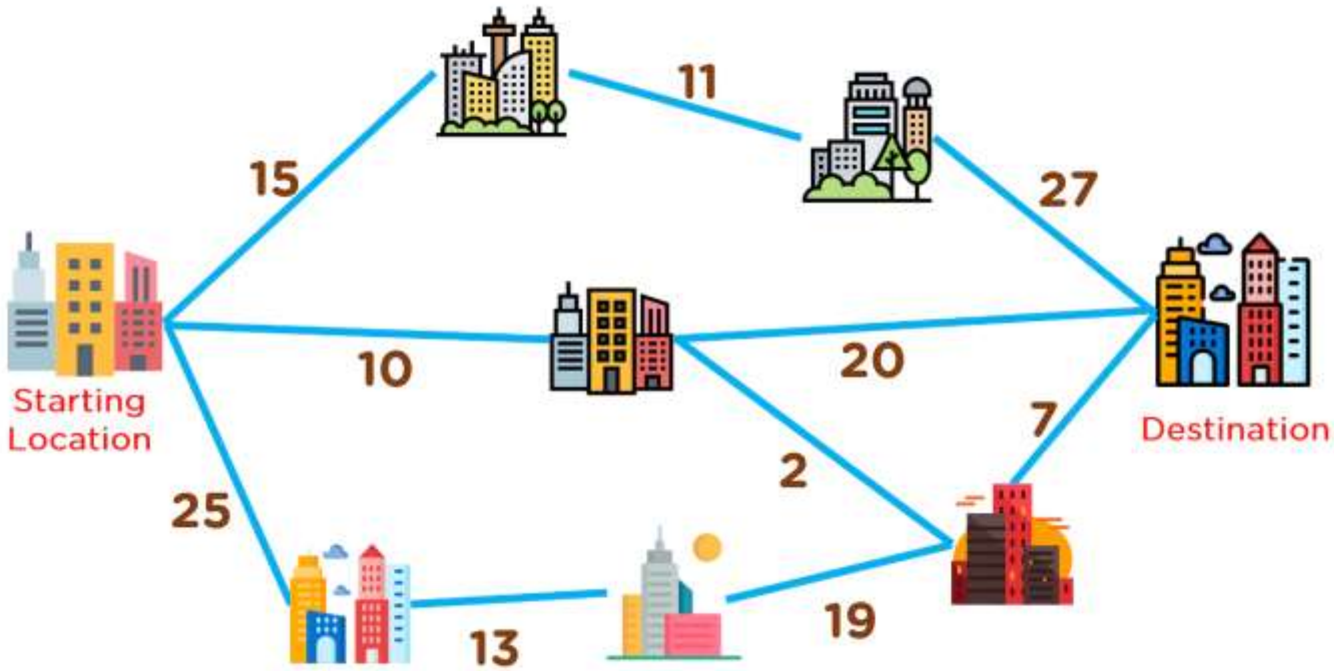
Greedy Algorithms

- *Approach for solving a problem, piece by piece, always choosing the best option available at the time*
- *Doesn't worry **whether the current best will bring the overall optimal result***
- *There is no going back once we decided!*
  - *Even it would be wrong!*

Problem Statement: *Find the best route to reach the destination city from the given starting point using a greedy method.*



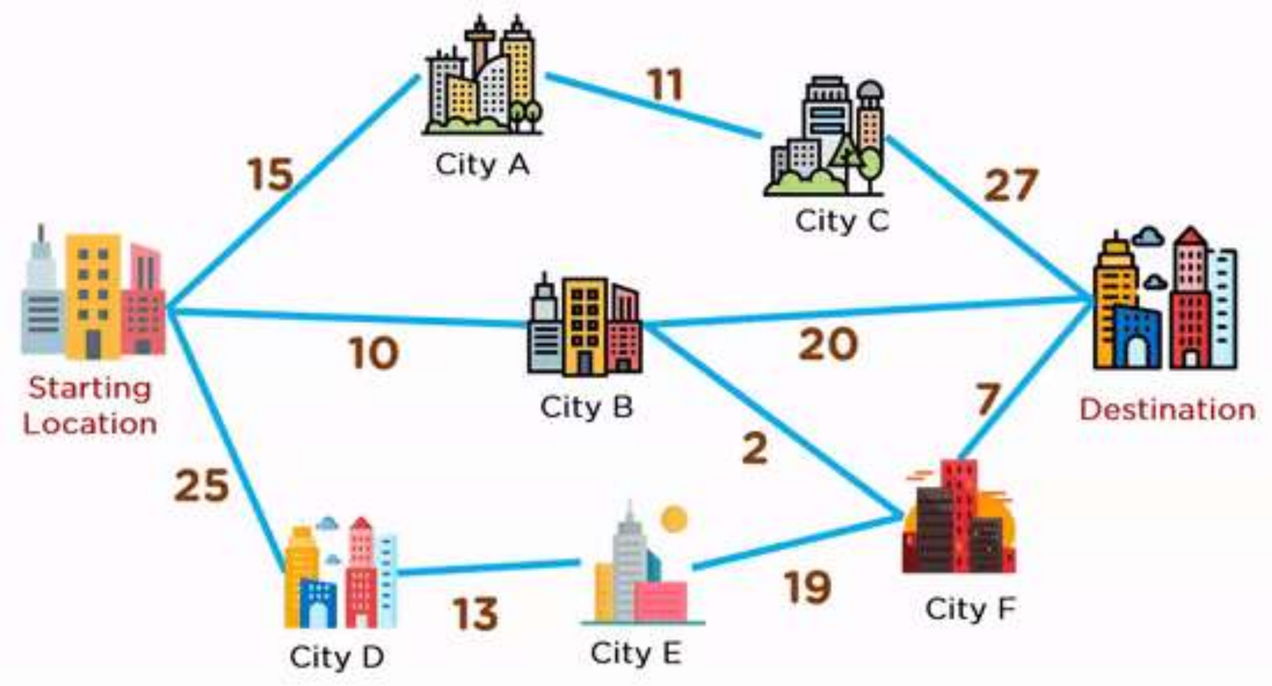
Greedy Solution

In order to tackle this problem, we need to maintain a graph structure.

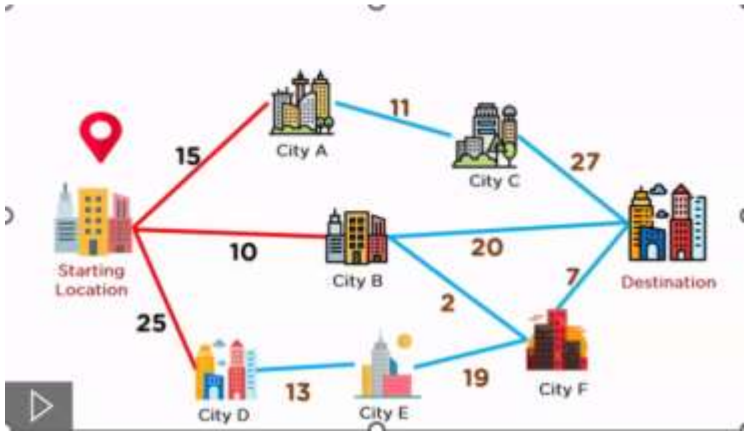
The steps to generate this solution are given below:

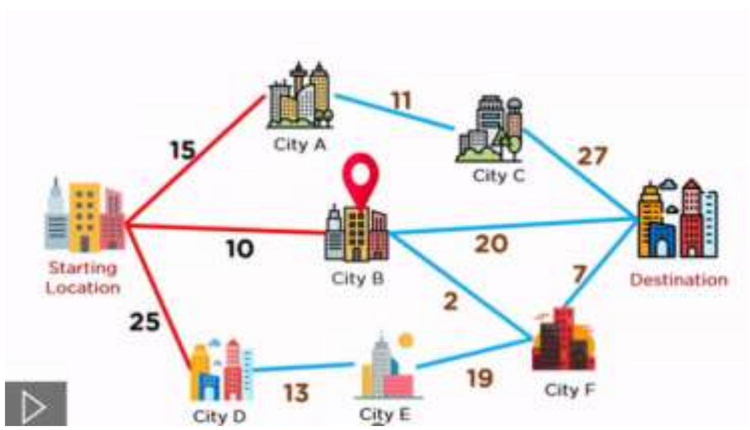
- *Start from the source vertex.*
- *Pick one vertex at a time with a minimum edge weight (distance) from the source vertex.*
- *Keep adding minimum adjacent edges to the tree until you reach the destination vertex.*

The animation given below explains how paths will be picked up in order to reach the destination city.

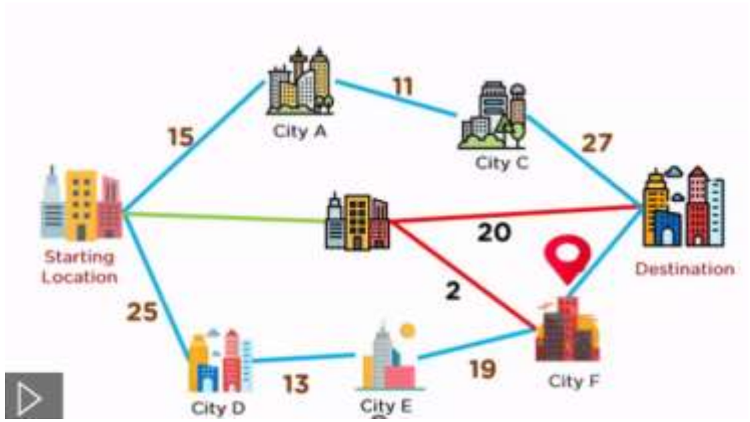


Here we are choosing edge that going to ***city B*** *which has minimum weight(10) compared to edge to the city A and city D*

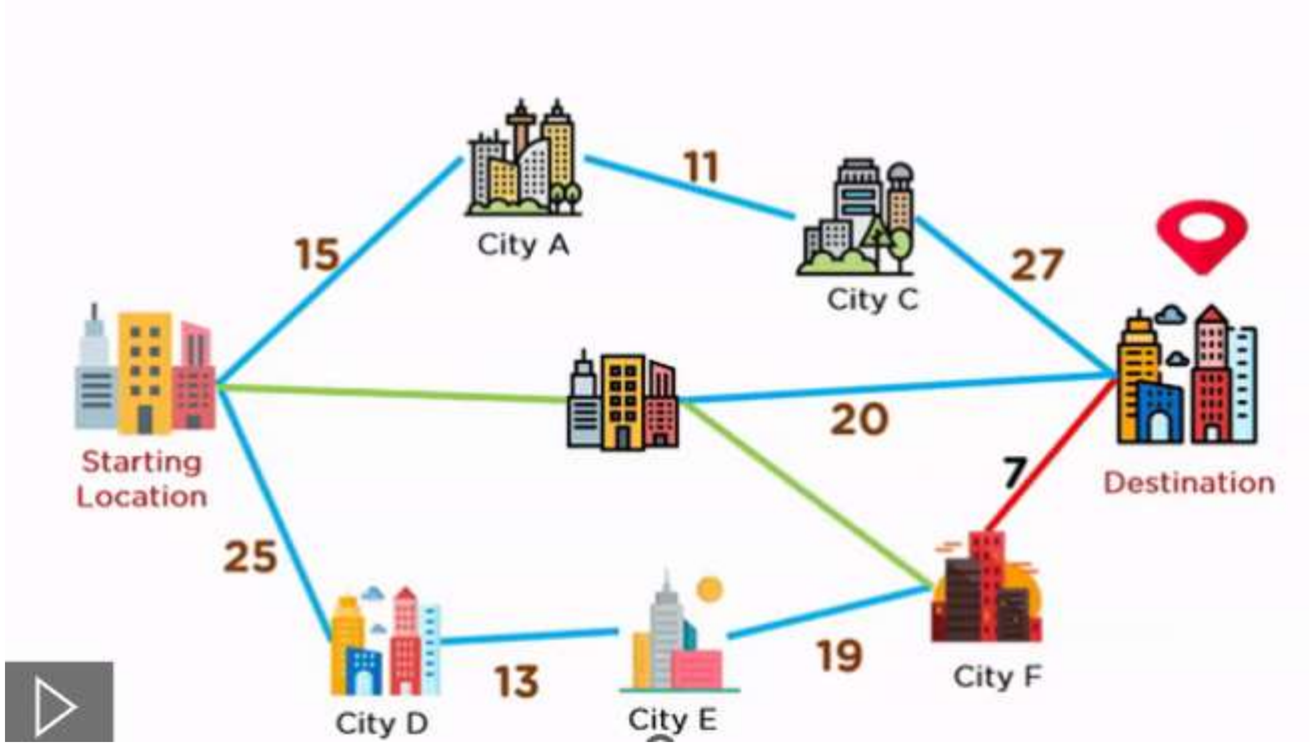




*Here we are choosing edge that going to the **city F** which has minimum weight(2) compared to edge directed to the destination*



Here we are choosing the edge going to destination which has weight (7) compared to edge directed to the destination



Thus, using greedy approach.. We are reaching the destination… Each step we're choosing the minimum i.e. best at the time.

Advantages of Greedy Approach

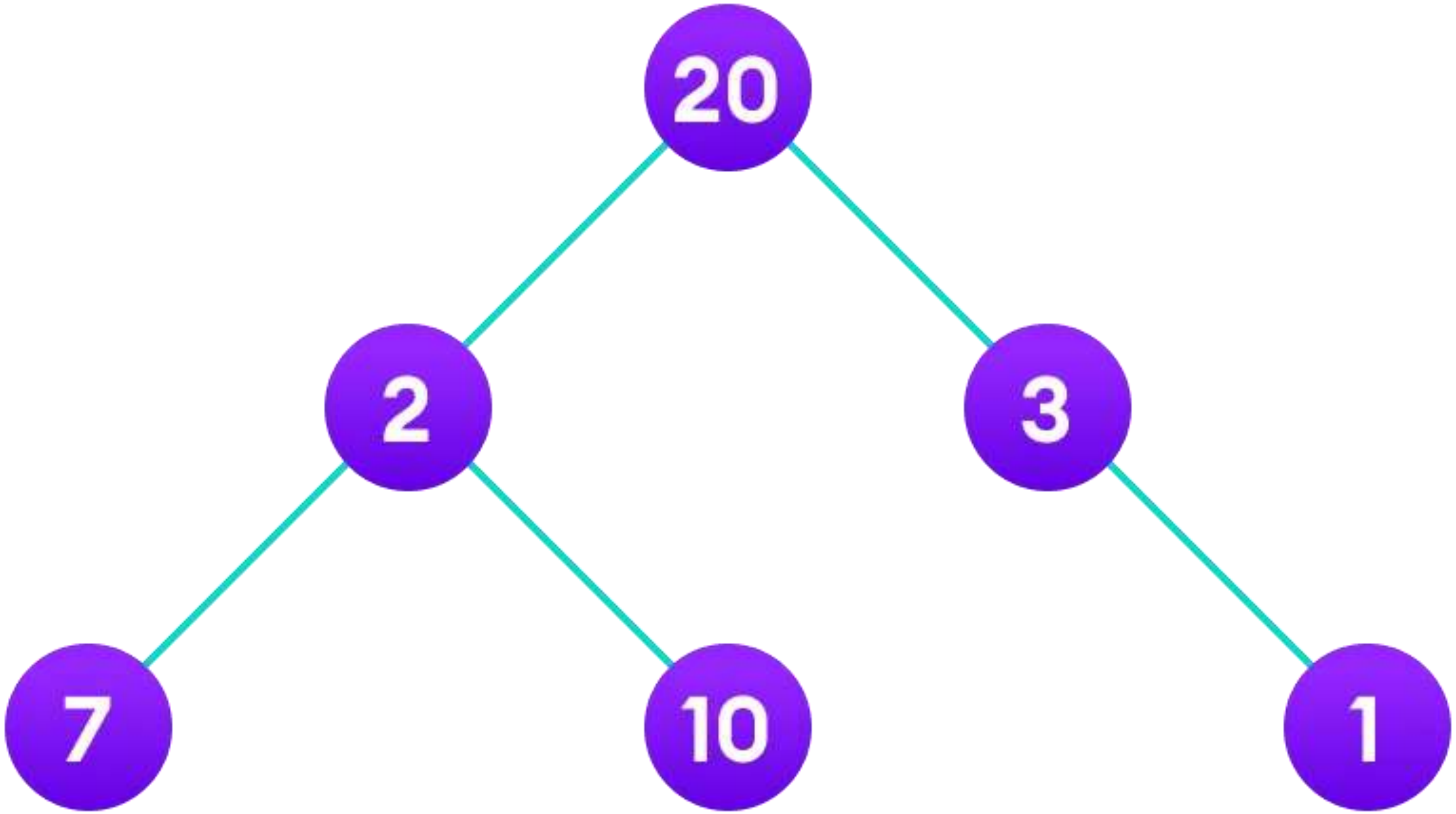
- *The algorithm is easier to describe.*
- *This algorithm can perform better than other algorithms (but, not in all cases).*

Drawback of Greedy Approach

- *Greedy algorithm doesn't always produce the optimal solution.*

For example, suppose we want to find the most weighted path in the graph below from root to leaf.

Let's use the greedy algorithm here.



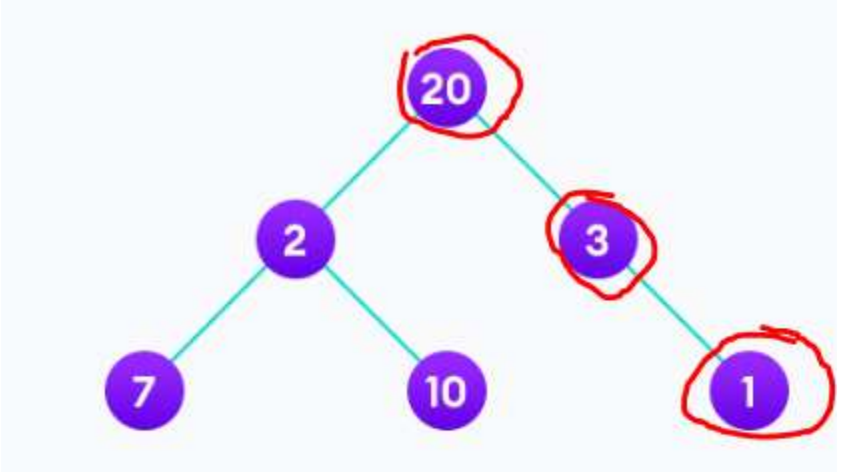
Greedy Approach

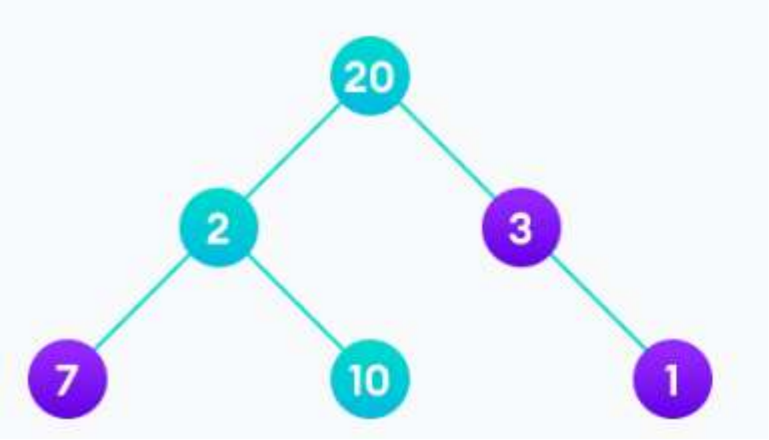
1. *Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is 2.*

2. *Our problem is to find the largest path. And the optimal solution at the moment is **3**. So, the greedy algorithm will choose 3.*

3. *Finally, the weight of an only child of 3 is **1**. This gives us our final result **20 + 3 + 1 = 24.** (as shown in the left side image below)*

*However, it is not the optimal solution. There is another path that carries more weight **20 + 2 + 10 = 32.** (as shown in the right-side image below)*





Let’s learn 3 greedy algorithms.

- ***Prims Algorithm***
  - *Used to find the minimum spanning tree of a graph*

- ***Kruskal's Algorithm***
  - *Used to find the minimum spanning tree of a graph*

- ***Dijkstra's Algorithm***
  - *Used to find the single source shortest path between source to any other vertices of a graph.*

Resources – GitHub

- *Minimum Spanning Tree - [DataStructures/spanningTree.md at main · PorkodiVenkatesh/DataStructures (github.com)]*
- *Prims Algorithm - [DataStructures/primsAlg.md at main · PorkodiVenkatesh/DataStructures (github.com)]*

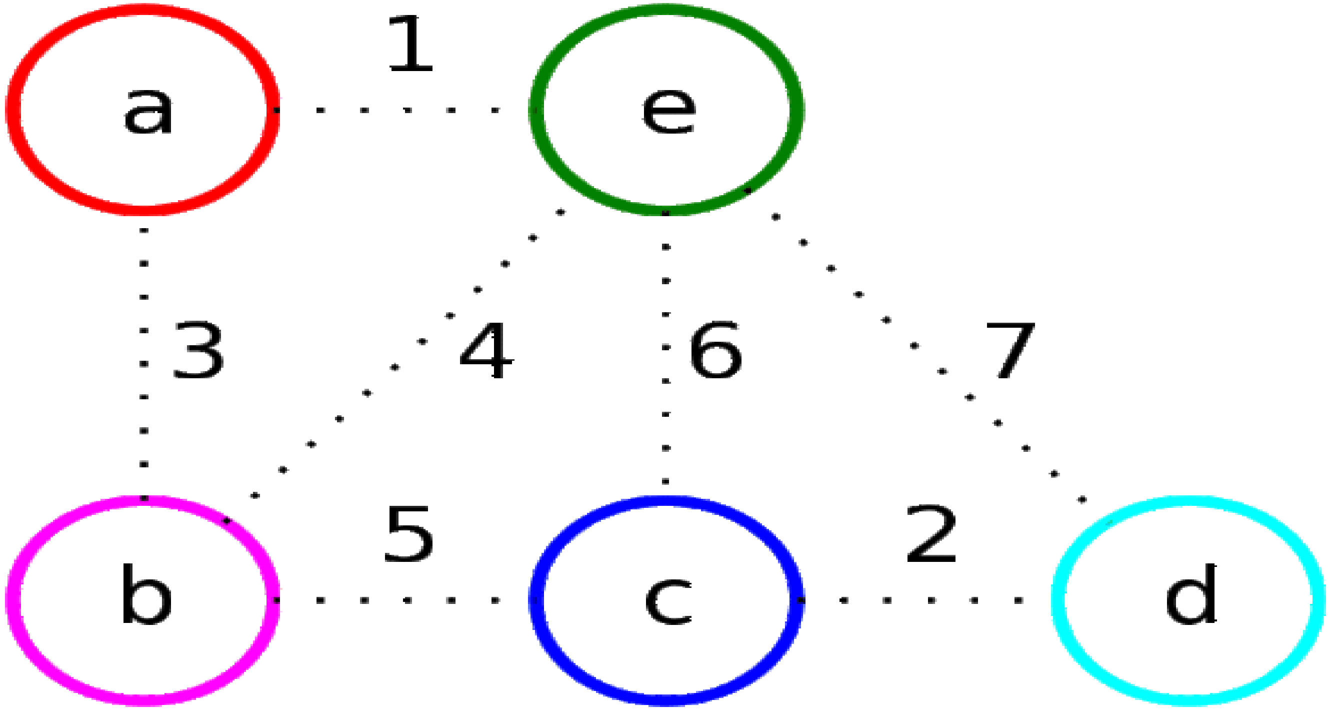
Kruskal Algorithm

- *To find the minimum spanning tree of the graph by traversing the edges of the graph*
- *This algorithm finds an optimum solution at every stage rather than finding a global optimum.*

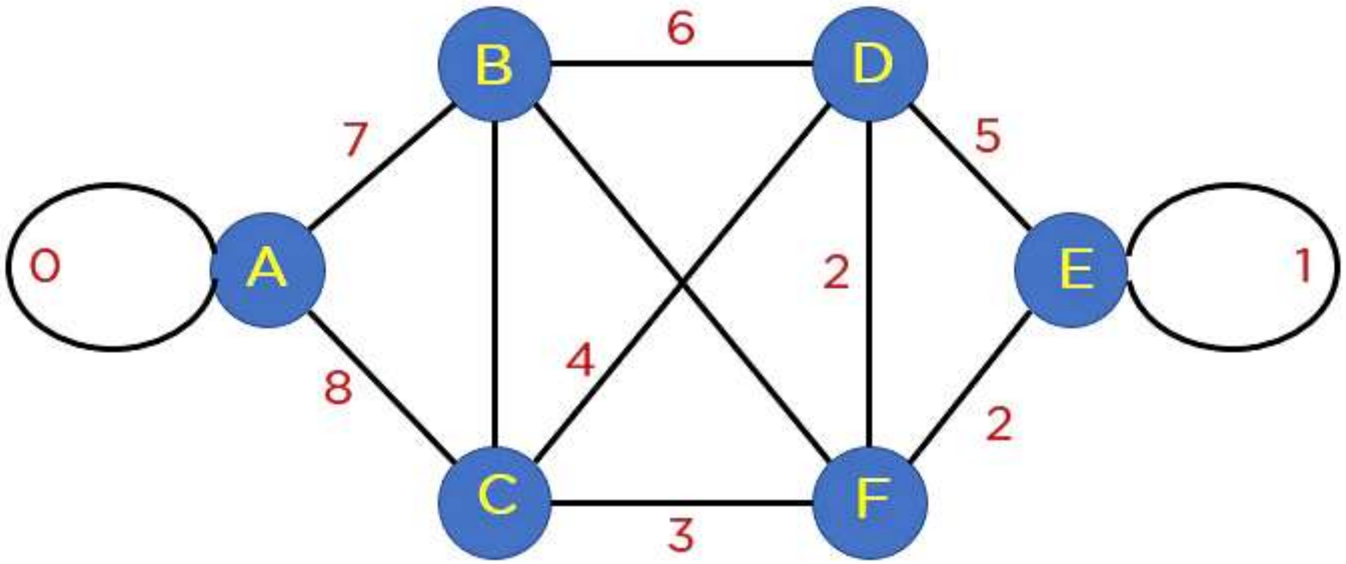
Steps:

- ***Step 1:*** *Sort all edges in increasing order of their edge weights.*
- ***Step 2:*** *Pick the smallest edge.*
- ***Step 3:*** *Check if the new edge creates a cycle or loop in a spanning tree.*
- ***Step 4:*** *If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.*
- ***Step 5:*** *Repeat from step 2 until it includes |V| - 1 edges in MST.*

Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



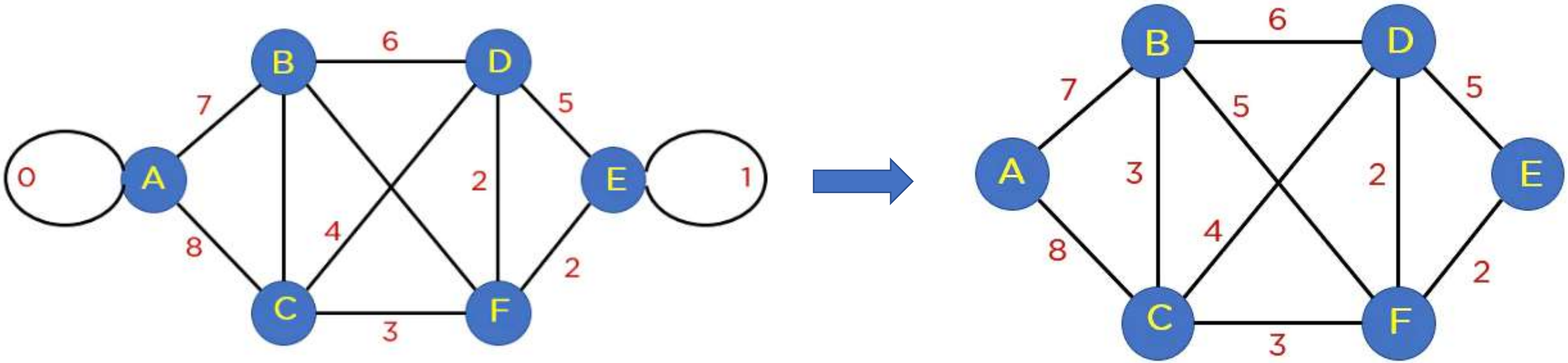
Let’s Try!!



Graph G(V, E)

If you observe this graph, you'll find ***two looping edges*** *connecting the same node to itself again. And you know that the tree structure can never include a loop or parallel edge.*

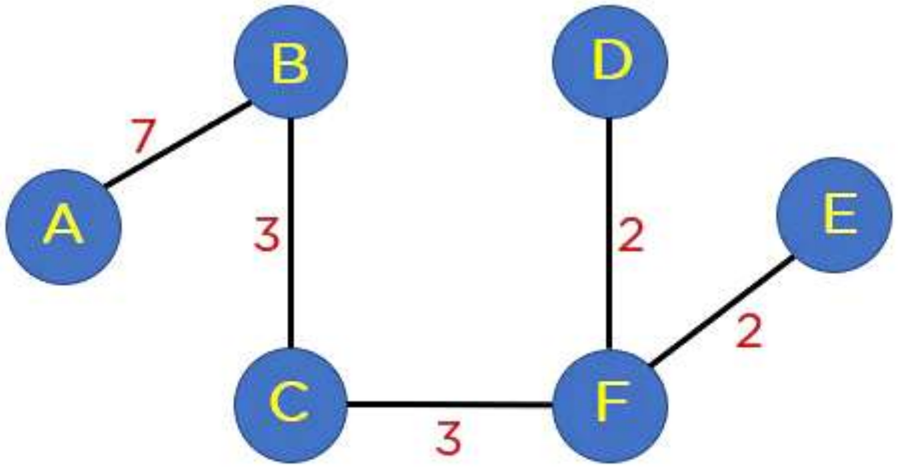
Hence, primarily you will need to remove these edges from the graph structure.



The next step that you will proceed with is arranging all edges in a sorted list by their edge weights.

Source Vertex	Destination Vertex	Weight
E	F	2
F	D	2
B	C	3
C	F	3
C	D	4
B	F	5
B	D	6
A	B	7
A	C	8

The summation of all the edge weights in MST T(V', E') is equal to 17, which is the least possible edge weight for any possible spanning tree structure for this graph.



Minimum Spanning Tree.

Implementation of Kruskal Algorithm

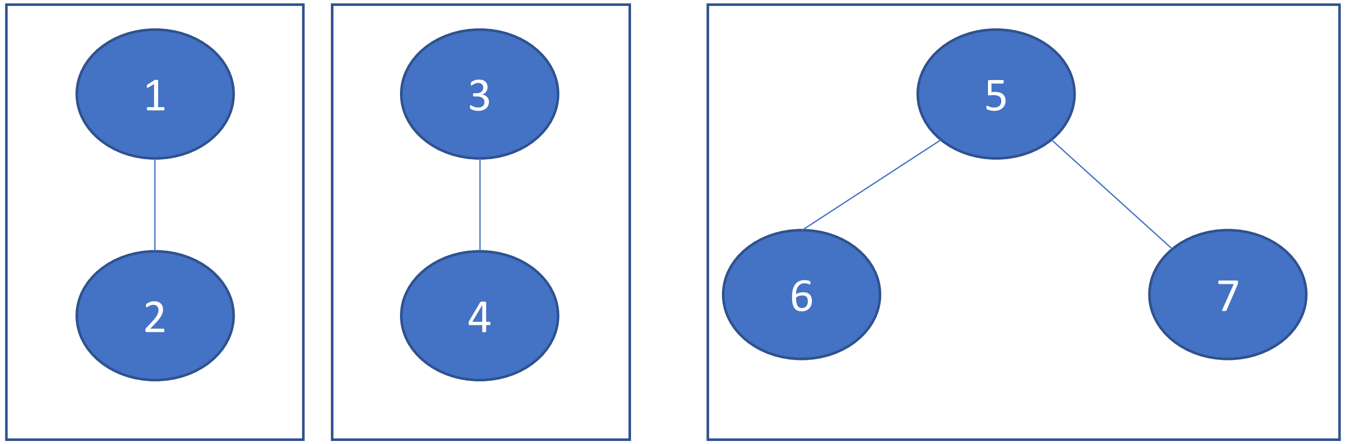
- *This algorithm revolves around determining whether adding an edge would result in a cycle or not.*
- *Disjoint sets used to detect the cycle in the graph.*

Disjoint Sets?

Disjoints Sets

Disjoints Sets has 2 operations

1. Find
2. Union



S1 S2 S3

Undirected disconnected graph

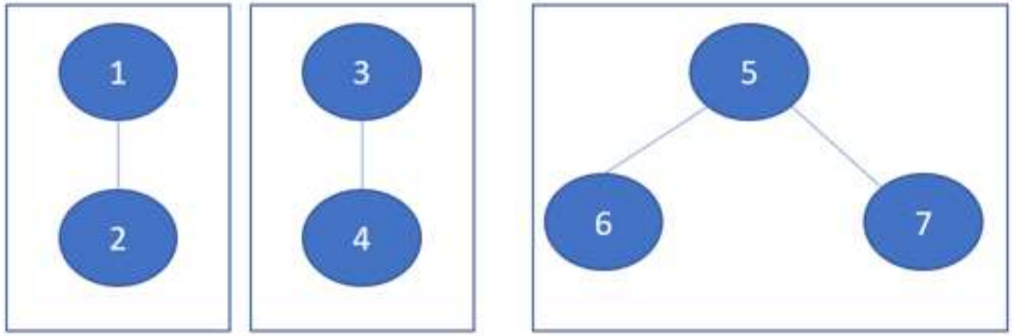
<u>Find:</u>

determines which set the particular element is in.

Find(1) = S1
Find(6) = S3
Find(7) = s3

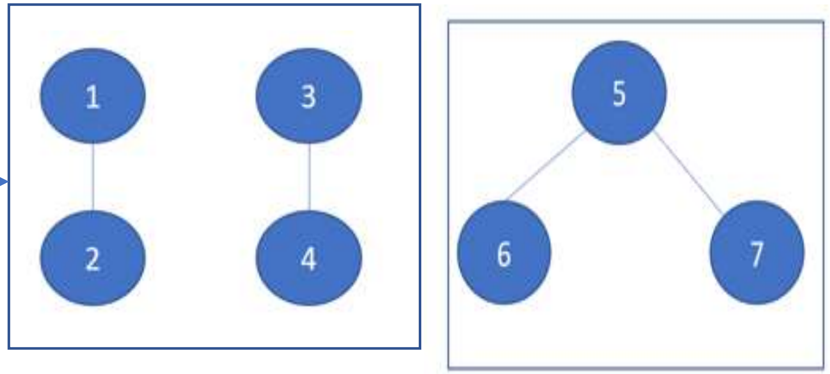
<u>Union:</u>

merges the two disjoint set



S1 S2 S3

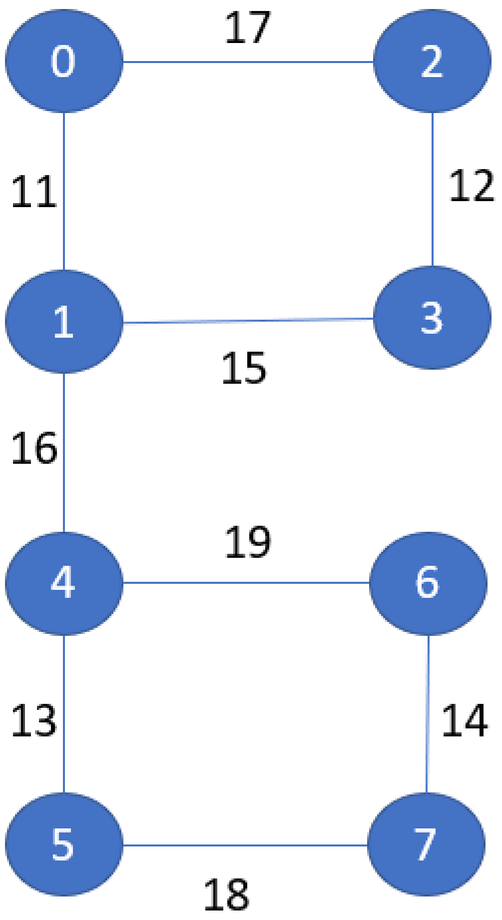
Union(S1, S2)



S4 S3

Now, lets detect the cycle using Disjoint Sets Or Union Find Algorithm

Let's consider below graph,   V = { 1, 2, 3, 4, 5, 6, 7}



source	0	2	4	6	1	1	1	5	4
destination	1	3	5	7	3	4	2	7	6
weight	11	12	13	14	15	16	17	18	19

Initially, we can consider each element as set.
S_0 = {0} , S_1 = {1}, S_2 = {2} , S_3 = {3}, S_4 = {4}, S_5 = {5}, S_6 = {6}, S_7 = {7}

Initially, we will maintain a parent array (initialized with -1) which keep track of edges which doesn't forms cycle (MST edges)

parent

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

indices -> Vertices

After that we'll be taking the edges (u , v) and forming sets one by one.
Take Edge(u , v):

1. *Find(u) -> Set of u*
2. *Find(v) -> Set of v*
3. *If u and v belongs to the different set, then union(S_u , S_v)*
    - *Update parent[u] = v*
4. *If u and v belongs to the same set, then that edge forms a cycle*

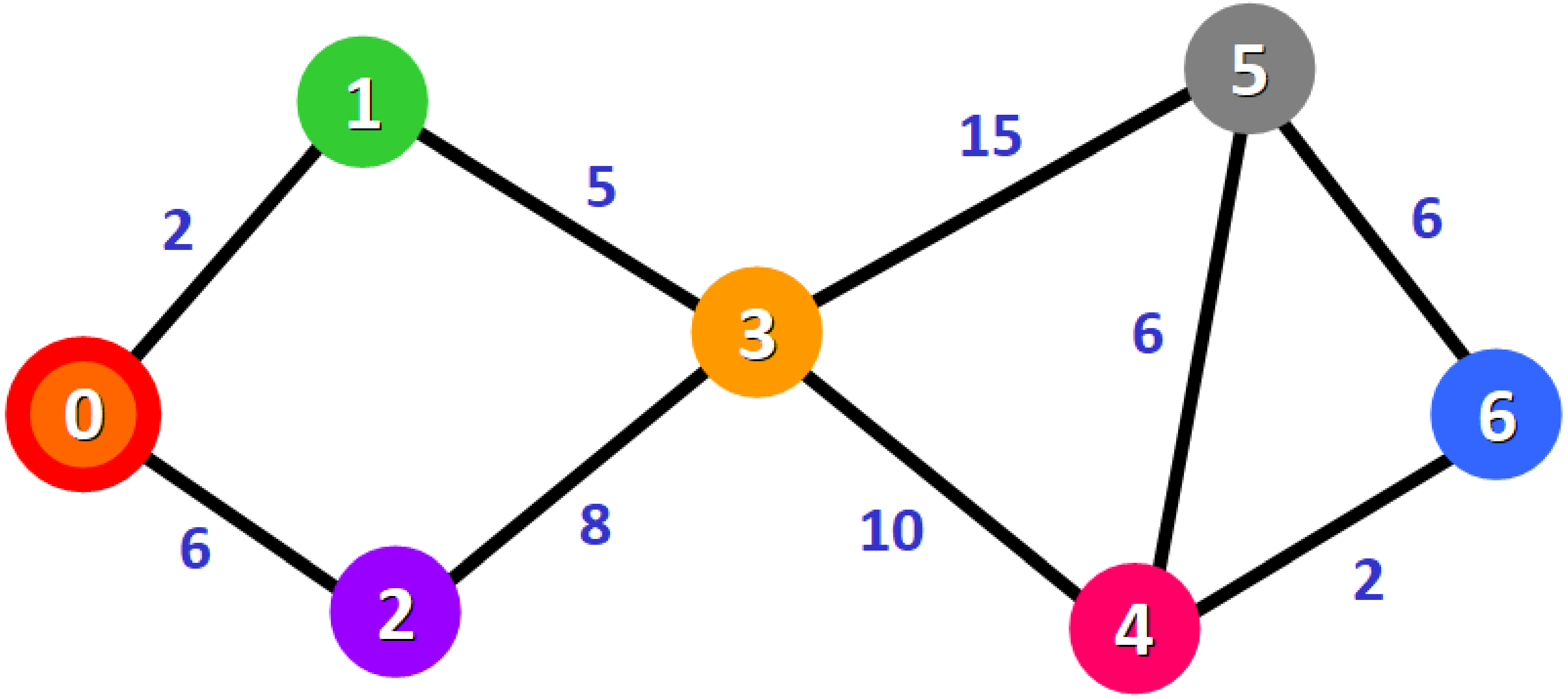
Difference between Prims and Kruskal Algorithm

Prims Algorithm	Kruskal Algorithm
Here we starts to build the Minimum Spanning Tree from **any of the node in the graph**	Here we starts to build the Minimum Spanning Tree from **minimum weighted edge in the graph**
Here we traverse the **one node several time** in order to get it minimum distance	Here we traverse the **edge only once** and based on cycle it will either reject or accept it
Prim's algorithm's time complexity is O(V^2), V being the number of vertices and can be improved up to O(E log V) using Fibonacci heap.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
It works only on connected graph.	It can work on connected as well as disconnected graph
Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.

Dijkstra's Algorithm

- *With Dijkstra's Algorithm, you can find the shortest path from a source vertex to all other vertices in the graph.*
- *Single Source Shortest Path Algorithm*
- *This algorithm is used in **GPS devices** to find the shortest path between the current location and the destination.*
- *It has broad applications in industry, specially in domains that require modeling networks.*

Let’s consider below graph



- *Consider source vertex as 0*
- *Dijkstra's algorithm finds the shortest path from source vertex 0 to all the other vertices in the graph.*

We will have the shortest path from vertex 0 to vertex 1, from vertex 0 to vertex 2, from vertex 0 to vertex 3, and so on for every vertex in the graph.

You need three arrays

1. *distance[] – stores the shortest distance from source to that vertex*
2. *visited[] – keeps track whether we visited the array or not*
3. *parent[] - keeps track of the parent of each vertex in the shortest path*

Note: Size of all these arrays would the number of vertices in the graph

Initialize all with

- *distance[i] = ∞*
- *visited [i] = false*
- *parent[i] = ∞*

Get the sourceVertex and then update

- *distance[sourceVertex] = 0*
- *parent[sourceVertex] = -1*

Steps

1. *Find the min value in the distance array, considering only the non visited vertices i.e., only including vertices v, where visited[v] = false*
    - *Then assign u as the vertex which has min value*

2. *Find all the direct edges from vertex u to the non visited vertex v i.e., graph[u][v] != 0 and visited[v] = false*
    - *For each edge (u, v)*

```
if (distance[u] + graph[u][v]) < distance[v])
    distance[v] = distance[u] + graph[u][v]
    parent[v] = u
```

3. *Mark visited[u] = true*

4. *Repeat the above 3 steps until you visit all the vertices in the graph*