Minimum Spanning Tree

Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) It's an almost complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

A Binary Heap is implemented using a priority queue.

The most efficient way to implement the following algorithms is using a binary heap.

Minimum Spanning Tree

Links:

Website: https://techvidvan.com/tutorials/spanning-tree/

Youtube: https://www.youtube.com/watch?v=4ZlRH0eK-qQ

Spanning Tree

Every undirected and connected graph has a minimum of one spanning tree.

Consider a graph having V vertices and E number of edges. Then, we will represent the graph as **G(V, E)**.

Its spanning tree will be represented as **G'(V, E')** where **E'** $\subseteq$ **E** and the number of vertices remain the same.

So, a spanning tree G’ is a subgraph of G whose vertex set is the same but edges may be different.

For any complete graph, the number of spanning trees is **n^(n-2).** Thus, in the worst case, the number of spanning trees formed is of the order **O(n^2).**

General Properties of Spanning Trees:

- There can be more than one spanning tree possible for an undirected, connected graph.
- In the case of directed graphs, the minimum spanning tree is the one having minimum edge weight.
- All the possible spanning trees of a graph have the same number of edges and vertices.
- A spanning tree can never contain a cycle.
- Spanning tree is always **minimally connected** i.e. if we remove one edge from the spanning tree, it will become disconnected.
- A spanning tree is **maximally acyclic** i.e. if we add one edge to the spanning tree, it will create a cycle or a loop.
- It is possible to have more than one minimum spanning tree if the graph weights of some edges are the same.
- Any connected and undirected graph will always have at least one spanning tree.

For a simple connected graph, its spanning tree will have N-1 edges, where N is the number of vertices.

Minimum Spanning Tree- MST

A minimum spanning tree is defined for a weighted graph. A spanning tree having minimum weight is defined as a minimum spanning tree. This weight depends on the weight of the edges. In real-world applications, the weight could be the distance between two points, cost associated with the edges or simply an arbitrary value associated with the edges.

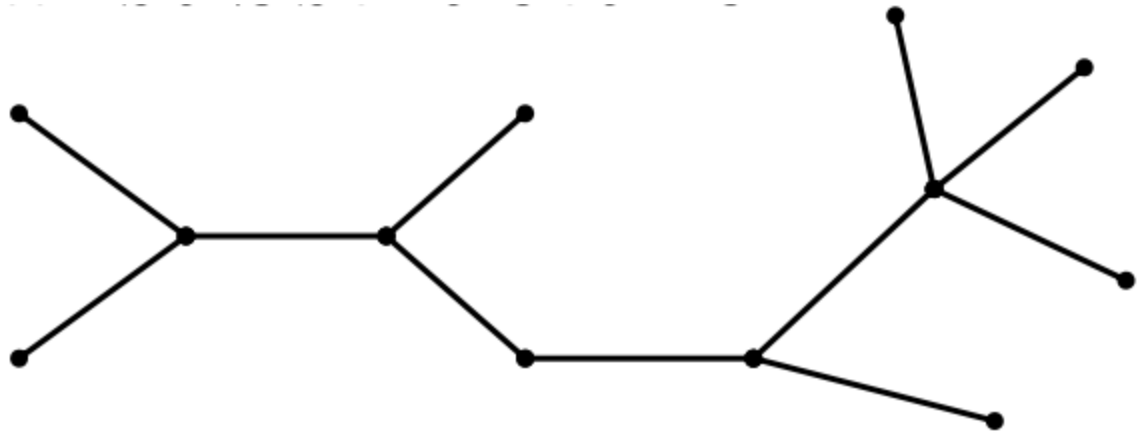
MST takes as input: A connected, undirected graph G = (V, E) with weight function w : E → R.

and outputs a spanning tree T of minimum weight: $w(T)=\sum w(u,v)$ where (u,v) belong to T.

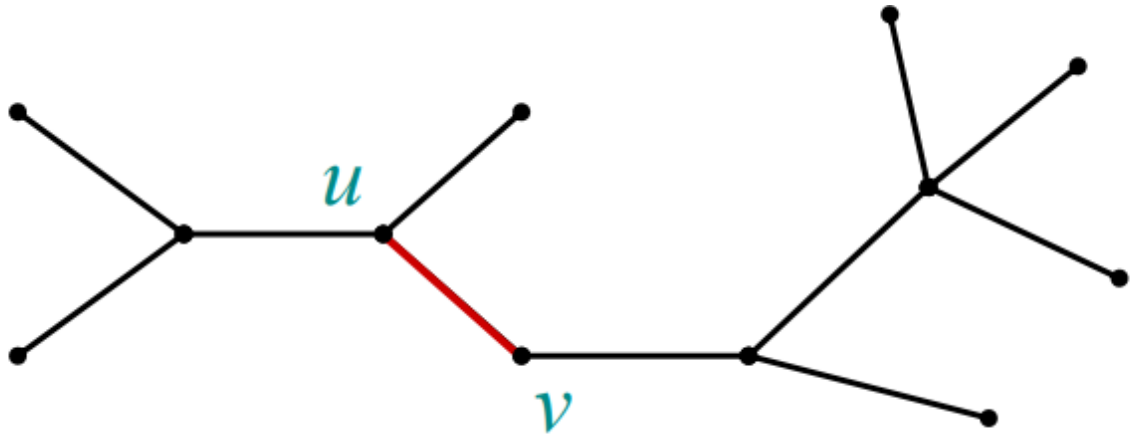
DYNAMIC PROGRAMMING

Optimal Substructure:

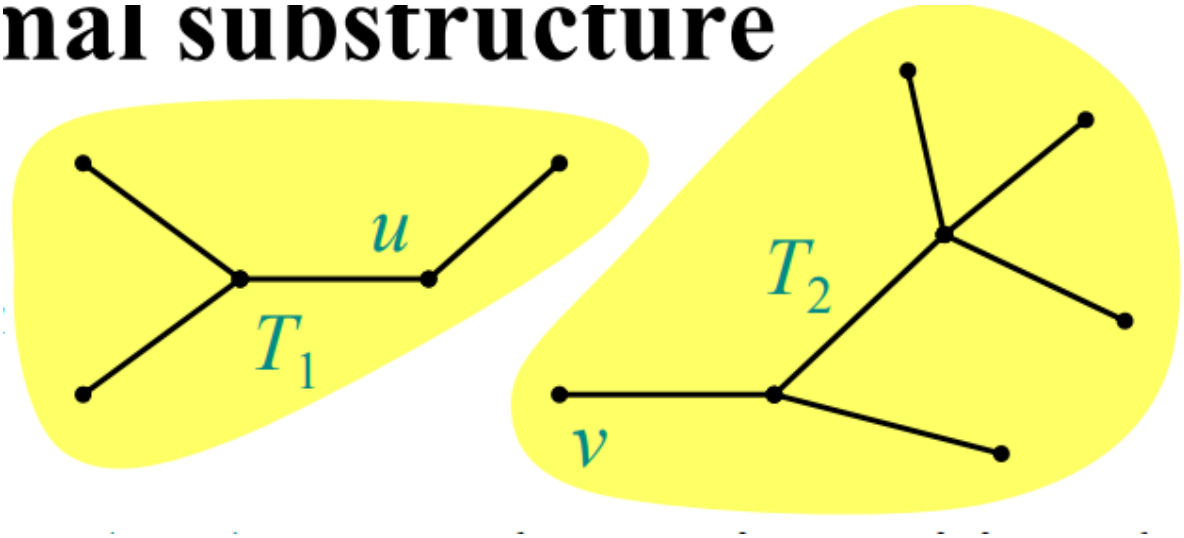
Given a minimum spanning tree T:



Remove any edge Remove any edge $(u, v) \in T$



Then, T is partitioned into two subtrees T1 and T2.



Theorem:

The theorem states that

the sub-tree T1 is an MST of G1 = (V1, E1),

the sub-graph of G induced by the vertices of T1:

V1=vertices of T1,

E1 = { (x, y) ∈ E : x, y ∈ V1 }.

and

the sub-tree T2 is an MST of G2 = (V2, E2),

the sub-graph of G induced by the vertices of T2:

V2=vertices of T2,

E2={(m, n) ∈ E : m, n ∈ V2 }.

Proof of theorem:

w(T) = w(u, v) + w(T1) + w(T2).

If T1′ were a lower-weight spanning tree than T1 for G1, then T′ = {(u, v)} ∪ T1′ ∪ T2 would be a lower-weight spanning tree than T for G.

We also have overlapping subproblems,

then Dynamic Programming can apply.

However it is more efficient to use a greedy method.

GREEDY

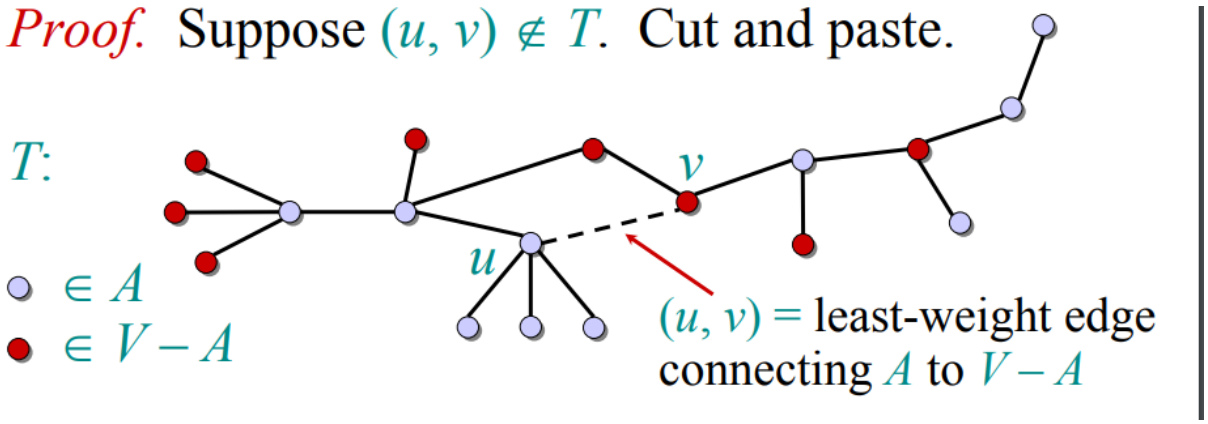
Theorem:

Let T be the MST of G = (V, E), and let A ⊆ V.

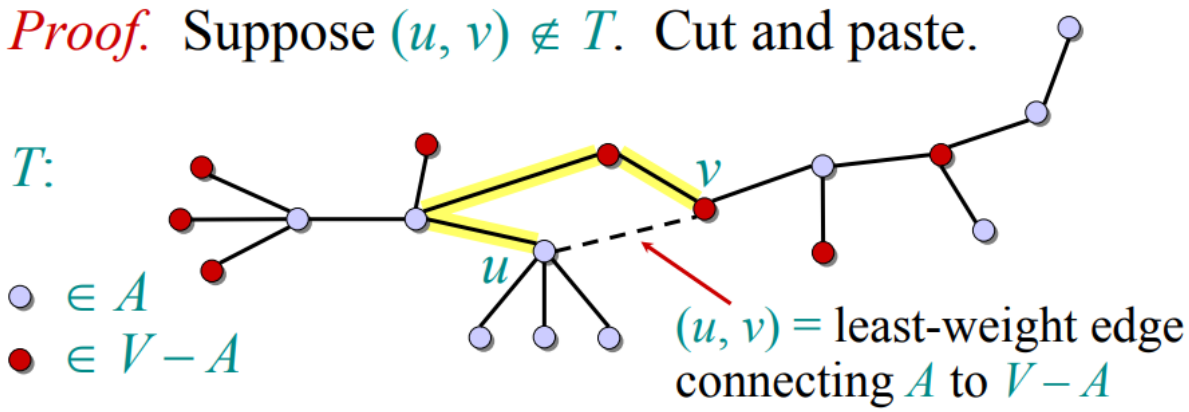
Suppose that (u, v) ∈ E is the least-weight edge connecting A to V – A.

Then, (u, v) ∈ T.

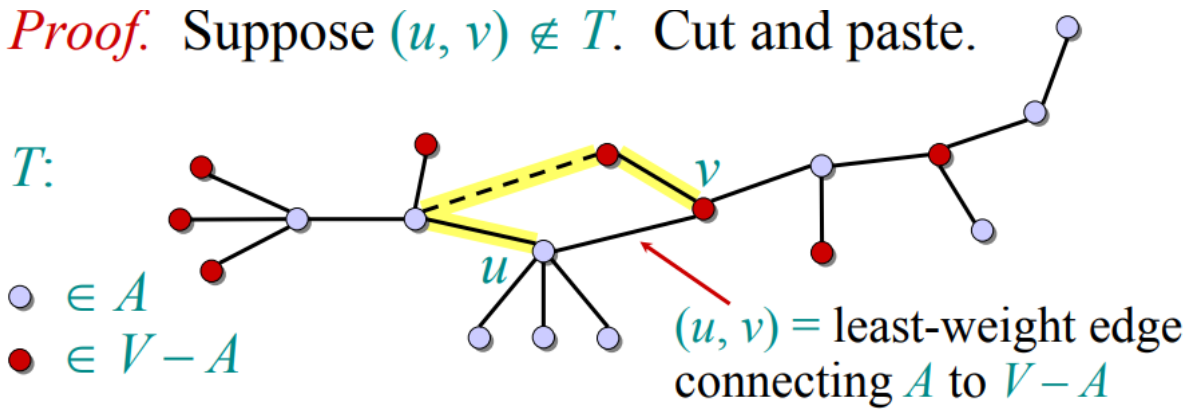
Proof:



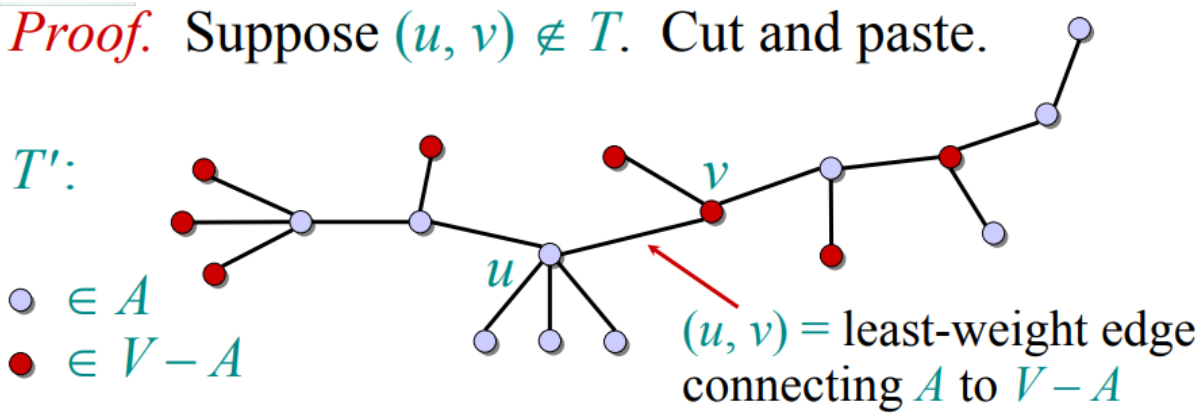
Consider the unique simple path from u to v in T:



Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in V – A:



A lighter-weight spanning tree than T results:



A minimum spanning tree can be solved using Prim's algorithms and Kruskal’s algorithm, which are both Greedy:

PRIM'S ALGORTHIM- GREEDY

Given an undirected, connected and weighted graph, find Minimum Spanning Tree (MST) of the graph using Prim’s algorithm.

Prim’s algorithm is a greedy algorithm.

It starts with an empty spanning tree.

The idea is to maintain two sets of vertices.

The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.

At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges.

After picking the edge, it moves the other endpoint of the edge to the set containing MST.

At every step of Prim’s algorithm, we find a cut of two sets, one contains the vertices already included in MST and other contains rest of the vertices,

pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim’s Algorithm Work? The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

Prim's algorithm is a MST algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex

- has the minimum sum of weights among all the trees that can be formed from the graph

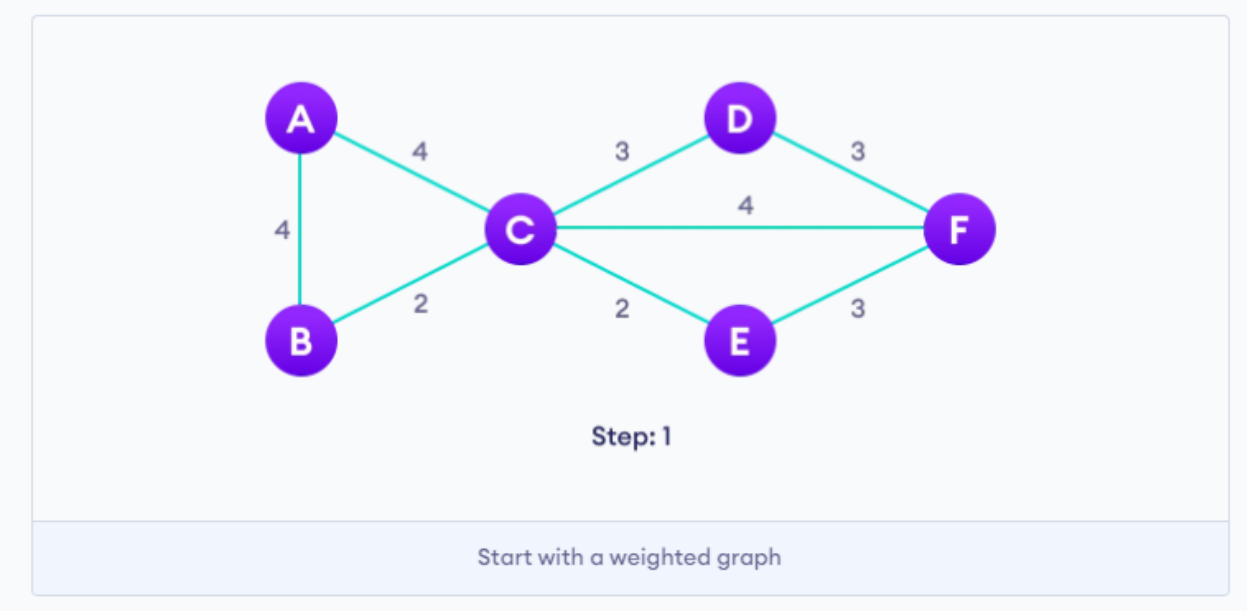
How Prim's algorithm works

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

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

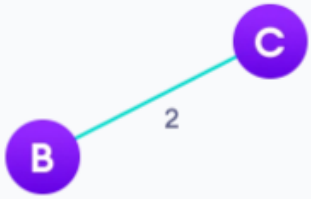
1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

STEPS:



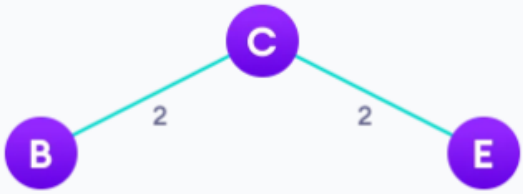
Step: 2

Choose a vertex



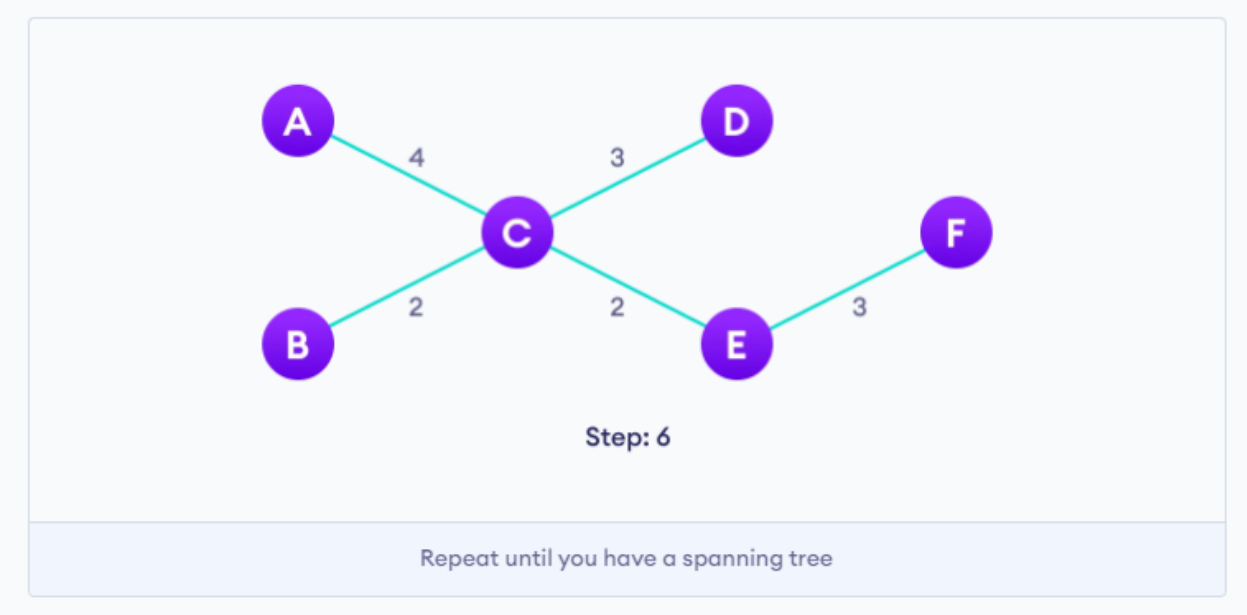
Step: 3

Choose the shortest edge from this vertex and add it



Step: 4

Choose the nearest vertex not yet in the solution



PSEDU CODE:

IDEA: Maintain V – A as a priority queue Q. Key each vertex in Q with the weight of the least weight edge connecting it to a vertex in A.

Q ← V

key[v] ← ∞ for all v ∈ V

key[s] ← 0 for some arbitrary s ∈ V

while Q ≠ ∅

    do u ← EXTRACT-MIN(Q)

for each v ∈ Adj[u]

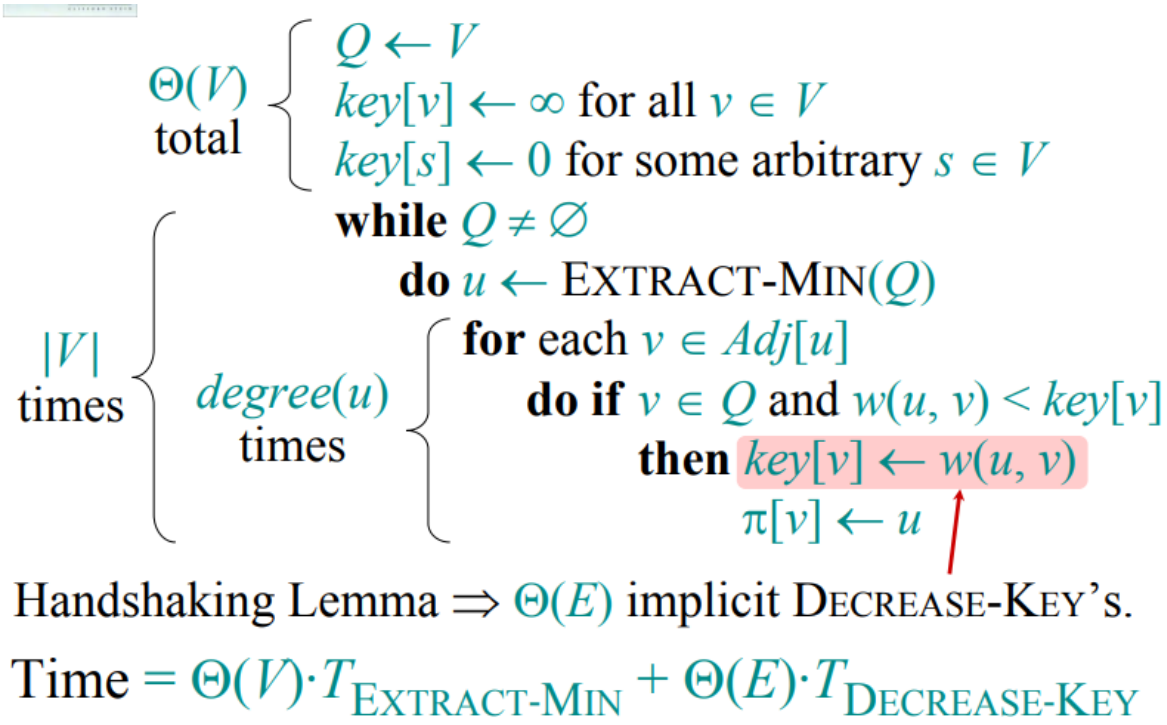
do if v ∈ Q and w(u, v) < key[v]

then key[v] ← w(u, v) ▷ DECREASE-KEY

π[v] ← u

At the end, {(v, π[v])} forms the MST.

ANAYLSIS OF PRIM:



$$\text{Time} = \Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$

Time Complexity= `O(E log V)`

PRIM CODE

```
import java.util.*;
```

```
public class PRIM {
```

```
public static class Edge {
int source;
int destination;
int weight;

public Edge(int source, int destination, int weight) {
   this.source = source;
   this.destination = destination;
   this.weight = weight;
 }
}
```

```
public static class HeapNode{
int vertex;
int key;
}
```

```
public static class ResultSet {
int parent;
int weight;
}
```

```
public static class Graph {
int vertices;
LinkedList<Edge>[] adjacencylist;
@SuppressWarnings("unchecked")
Graph(int vertices) {
   this.vertices = vertices;
   adjacencylist = new LinkedList[vertices];

   for (int i = 0; i <vertices ; i++)
     adjacencylist[i] = new LinkedList<>();

 }

public void addEgde(int source, int destination, int weight) {
Edge edge = new Edge(source, destination, weight);
adjacencylist[source].addFirst(edge);
edge = new Edge(destination, source, weight);
adjacencylist[destination].addFirst(edge);
}
```

```
public void primMST(){
 boolean[] inPriorityQueue = new boolean[vertices];
 ResultSet[] resultSet = new ResultSet[vertices];
 int [] key = new int[vertices];

 HeapNode [] heapNodes = new HeapNode[vertices];
 for (int i = 0; i <vertices ; i++) {
   heapNodes[i] = new HeapNode();
   heapNodes[i].vertex = i;
   heapNodes[i].key = Integer.MAX_VALUE;
   resultSet[i] = new ResultSet();
   resultSet[i].parent = -1;
   inPriorityQueue[i] = true;
   key[i] = Integer.MAX_VALUE;
 }


 heapNodes[0].key = 0;


 PriorityQueue<HeapNode> pq = new PriorityQueue<>(vertices,new Comparator<HeapNode>() {
```

```
    @Override
    public int compare(HeapNode o1, HeapNode o2) {
      return o1.key -o2.key;
    }
  });

  for (int i = 0; i <vertices ; i++) {
    pq.offer(heapNodes[i]);
  }

  while(!pq.isEmpty()){
    HeapNode extractedNode = pq.poll();
    int extractedVertex = extractedNode.vertex;
    inPriorityQueue[extractedVertex] = false;


    LinkedList<Edge> list = adjacencylist[extractedVertex];
    for (int i = 0; i <list.size() ; i++) {
      Edge edge = list.get(i);

      if(inPriorityQueue[edge.destination]) {
        int destination = edge.destination;
        int newKey = edge.weight;
        if(key[destination]>newKey) {
          decreaseKey(pq, newKey, destination);
          resultSet[destination].parent = extractedVertex;
          resultSet[destination].weight = newKey;
          key[destination] = newKey;
        }
      }
    }
  }
  printMST(resultSet);
}
```

```
public void decreaseKey(PriorityQueue<HeapNode> pq, int newKey, int vertex){

  @SuppressWarnings("rawtypes")
  Iterator it = pq.iterator();
  while (it.hasNext()) {
    HeapNode heapNode = (HeapNode) it.next();
    if(heapNode.vertex==vertex) {
      pq.remove(heapNode);
      heapNode.key = newKey;
      pq.offer(heapNode);
      break;
    }
  }
}

public void printMST(ResultSet[] resultSet){
```

```
  int total_min_weight = 0;
  System.out.println("Minimum Spanning Tree: ");
  for (int i = 1; i <vertices ; i++) {
    System.out.println("Edge: " + i + " – " + resultSet[i].parent +" key: " + resultSet
[i].weight);
    total_min_weight += resultSet[i].weight;
  }
  System.out.println("Total minimum key: " + total_min_weight);
}
}
```

```
public static void main(String[] args) {
int vertices=8; //Edges+1
Graph graph = new Graph(vertices);
graph.addEgde(1, 6, 10);
graph.addEgde(5, 6, 25);
graph.addEgde(5, 7, 24);
graph.addEgde(4, 5, 22);
graph.addEgde(4, 7, 18);
graph.addEgde(1, 2, 28);
graph.addEgde(2, 7, 14);
graph.addEgde(2, 3, 16);
graph.addEgde(3, 4, 12);
graph.primMST();
}
}
```

KRUSKAL'S ALGORITHM- GREEDY

Link: website: https://www.programiz.com/dsa/kruskal-algorithm

Kruskal's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from a vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex

- has the minimum sum of weights among all the trees that can be formed from the graph

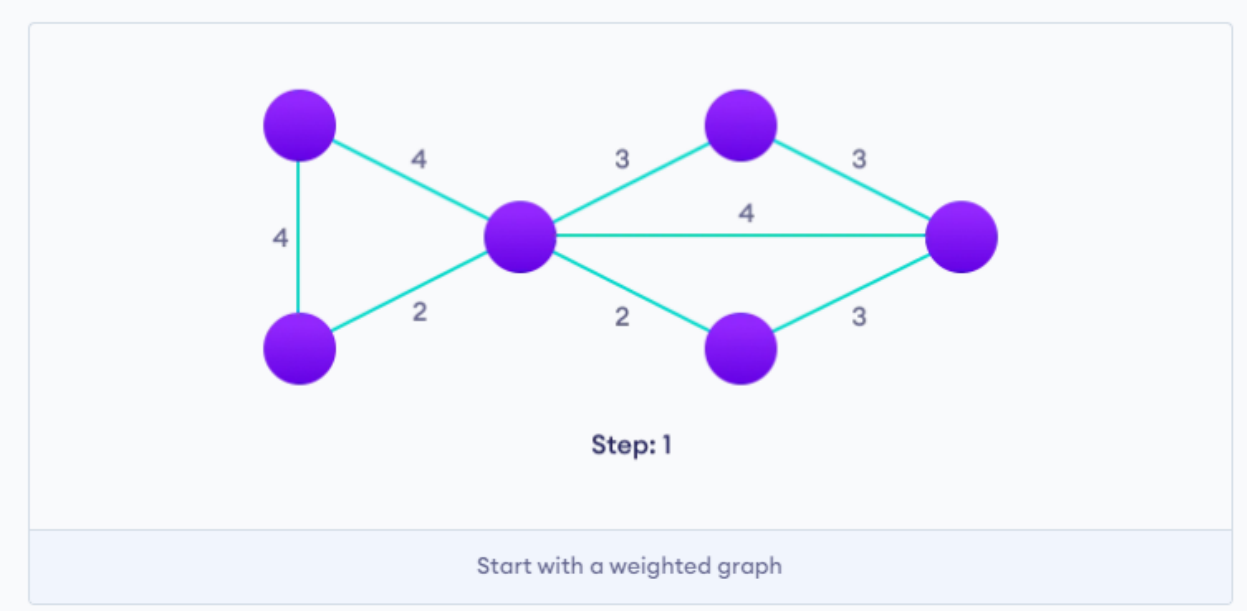
How Kruskal's algorithm works

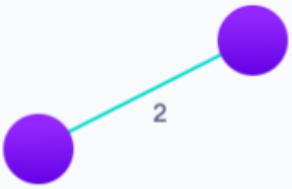
We start from the edges with the lowest weight and keep adding edges until we reach our goal.

1. The steps for implementing Kruskal's algorithm are as follows:
2. Sort all the edges from low weight to high
3. 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.
4. Keep adding edges until we reach all vertices.

Running time = O(E log V).

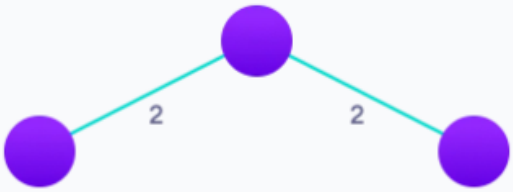
STEPS:





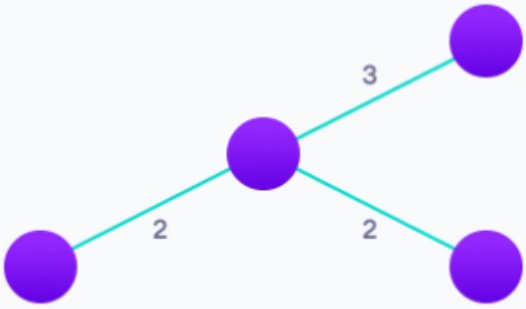
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



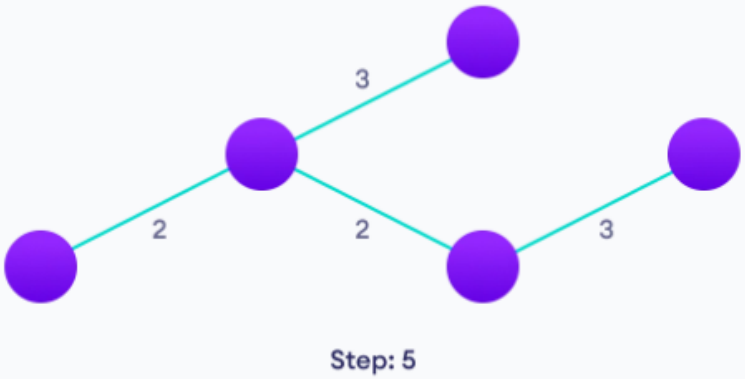
Step: 3

Choose the next shortest edge and add it



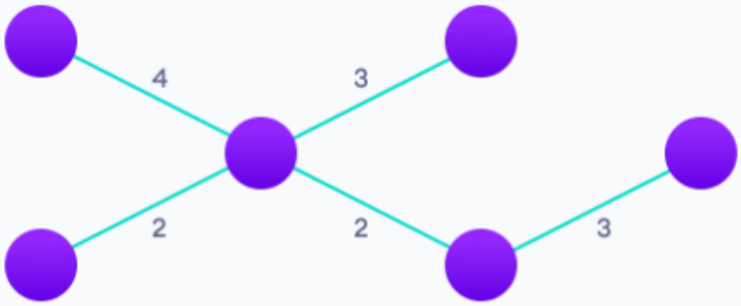
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree