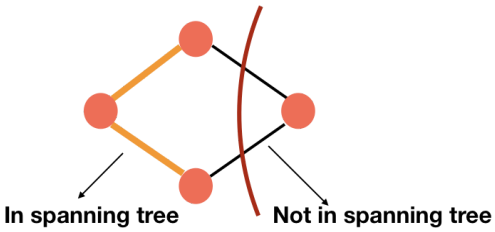
Minimum Spanning Tree | Prim's Algorithm

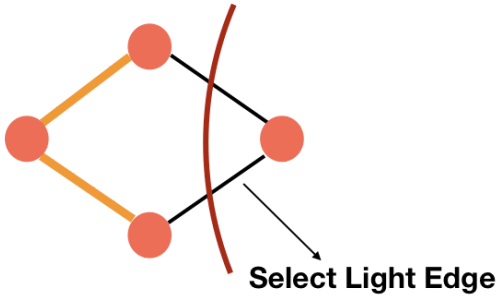
Like Kruskal's algorithm, Prim's algorithm is also used to find the minimum spanning tree from a graph and it also uses greedy technique to do so.

In prim's algorithm, we start growing a spanning tree from the starting position and then further grow the tree with each step. We divide the vertices into two parts, one contains the vertices which are in the growing tree and the other part has the rest of the vertices.



We mark the vertices which are in the growing tree differently to the vertices which are not in it (as we have used different colors in the above picture). This helps us to avoid loops in the tree.

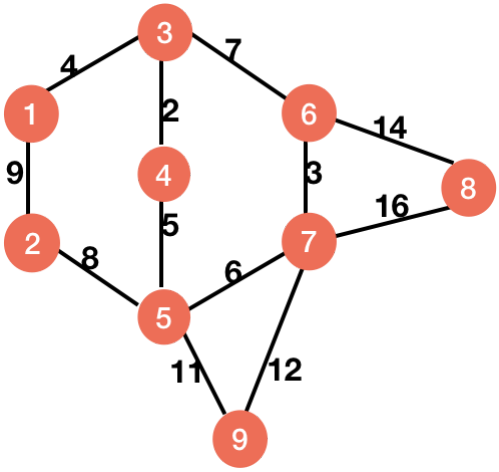
Now, we select the light edge each time and put the vertex at the end of the light edge into the growing tree.

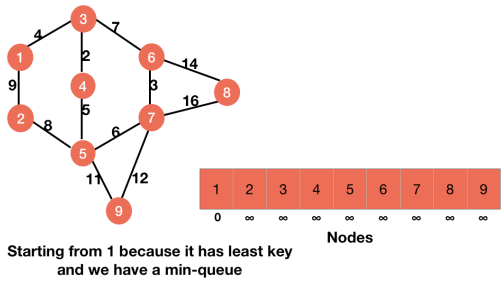


As we have already discussed that a light edge is a must have edge in a MST, so selecting a light edge every time gives us a minimum spanning tree.

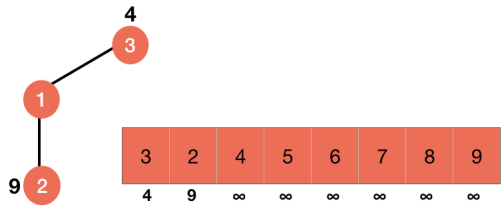
We are going to use a minimum priority queue (https://www.codesdope.com/blog/article/priority-queue-using-heap/) to store all the unvisited nodes. The reason for using the min-queue will be clear after going through the steps of the algorithm.

Initially, none of the nodes are visited. So, we put all the nodes in the queue with a key value of infinity for each node, except the node we are going to start our tree, we will make its key 0. We can arbitrarily choose any node to start with.

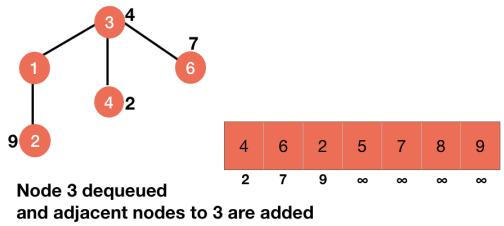




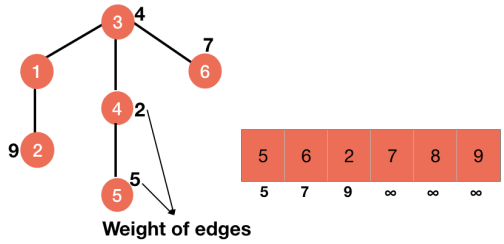
Since each node has a key of infinity except the node with which we are starting, so dequeuing the minimum priority queue will give us that node because its value of the key is minimum.



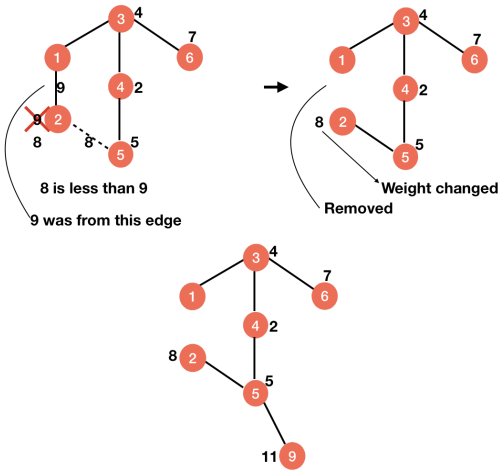
Now, we will visit each adjacent node of this node and change the key of each adjacent node to the weight of the edges going to them. So, now dequeue will give us the node with the edge with minimum weight and that's why we are using a minimum priority queue. Also we are dequeuing the queue in every step, so only the unvisited node will be in the queue.

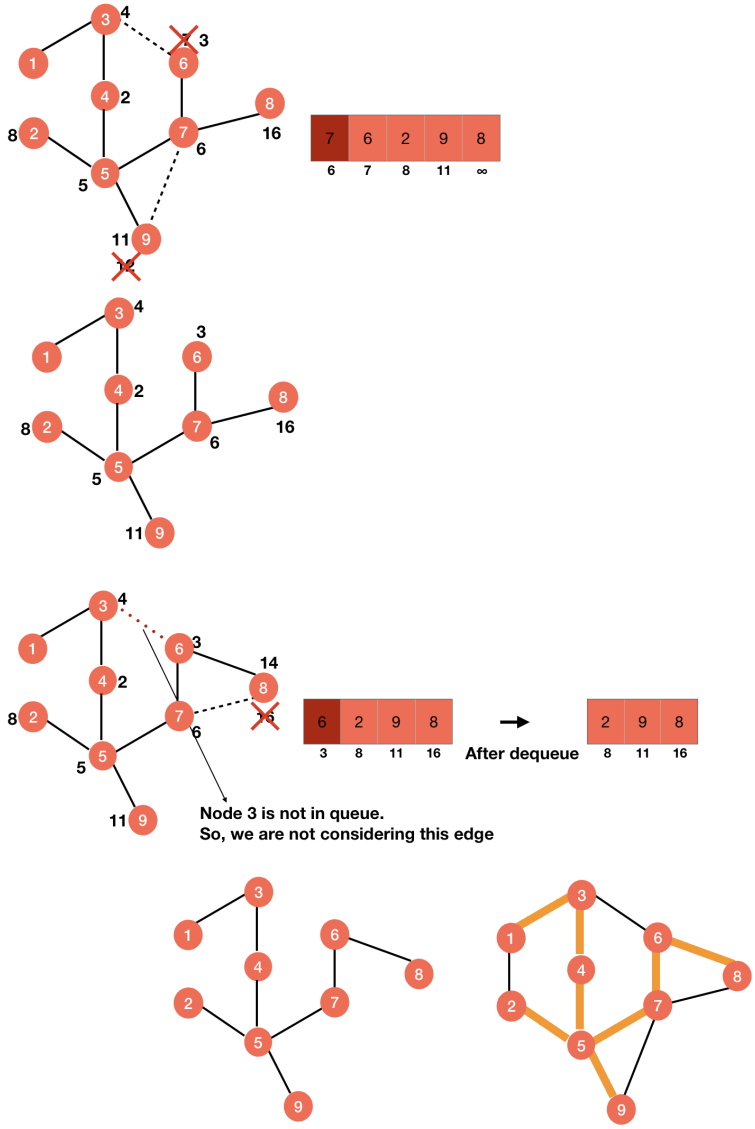


We will dequeue again and get the edge with the minimum weight.



Now, we will visit the adjacent nodes of this node and assign key equal to the weights of the edges to them. If the node has already a key assigned to it, we will select the minimum key among the two.





Code for Prim's Algorithm

From the steps given above, it must be clear that our first task is to make the key of each node infinite except the node with which we are starting (make it 0) and put them in a minimum priority queue. We also need to keep the track of the nodes or the edges we are selecting in each step. We can either make a queue and put these nodes in that in the sequence we are selecting them and return at the last or we can keep the record of the parent of each node to trace back the MST.

We are going to use a queue in this chapter.

```
for i in G.V
   i.key = infinite

Select a node n and make its key 0 n.key=0

min_queue = G.V
q = NULL // queue to return
```

Now, we need to iterate until the `min_queue` is empty and check the adjacent elements of the node dequeued from it.

```
while !min_queue.is_empty
   u = min_queue.dequeue()
   for i in G.Adj[u]
```

The for loop is checking all the adjacent nodes of u, and so we will first check if the node is in queue or not. If it is not, then the node has already been taken and taking it again will make a loop. We will also compare the key already assigned to the node with the key of the current edge going to it and assign least of these two.

```
for i in G.Adj[u]
   if i in min_queue and weight(u, i) < i.key
     i.key = weight(u, i)
```

At last, we need to push this element in the queue which we are going to return.

```
while !min_queue.is_empty
   ...
   for i in G.Adj[u]
     if i in min_queue and weight(u, i) < i.key
        ...
        q.enqueue(i)
return q
```

```
PRIM(G)
  for i in G.V
    i.key = infinite

  Select a node n and make its key 0 n.key=0

  min_queue = G.V
  q = NULL // queue to return

  while !min_queue.is_empty
    u = min_queue.dequeue()
    for i in G.Adj[u]
      if i in min_queue and weight(u, i) < i.key
        i.key = weight(u, i)
        q.enqueue(i)
  return q
```

Analysis of Prim's Algorithm

We are going to do this analysis assuming that the minimum priority queue (https://www.codesdope.com/blog/article/priority-queue-using-heap/) is implemented using a binary heap (https://www.codesdope.com/blog/article/heap-binary-heap/). So, we can make a heap in $O(V)$ time. Also, the dequeuing the min-queue will take $O(\lg V)$ time and it is under the while loop which is repeating V times, so this process will take $O(V \lg V)$ time. Since there are $O(E)$ edges, so the for loop will be executed $O(E)$ times and inside it, we can decrease the key in $O(\lg V)$ time, so it will take $O(E \lg V)$ time (q.enqueue(i) will take constant time). Thus, the total time taken will be $O(V \lg V + E \lg V) = O(E \lg V)$.

> “ Intelligence is the ability to adapt to change. ”
>
> - Stephen Hawking

PREV (/course/algorithms-minimum-spanning-tree/)