Graphs- 2

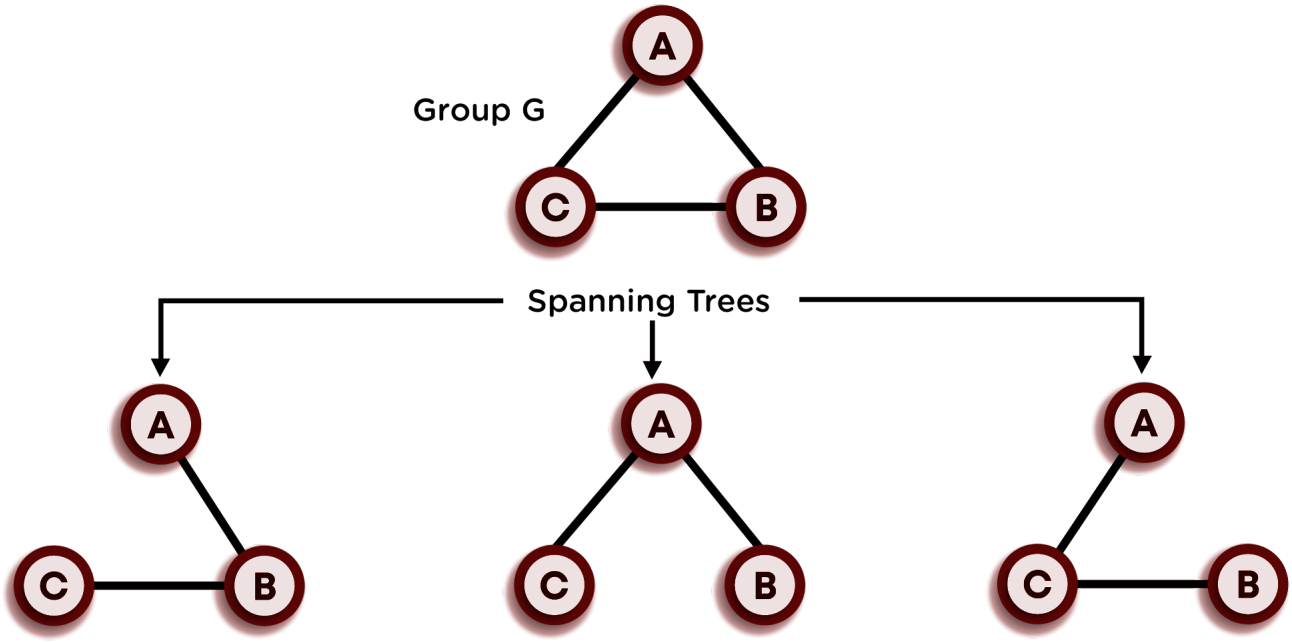
MST(Minimum Spanning Tree) & Kruskal's Algorithm

As discussed earlier, a tree is a graph, which:

- Is always connected.
- Contains no cycle.

If we are given an undirected and connected graph, a **spanning tree** means a tree that contains all the vertices of the same. For a given graph, we can have multiple spanning trees.

Refer to the example below for a better understanding of spanning trees.



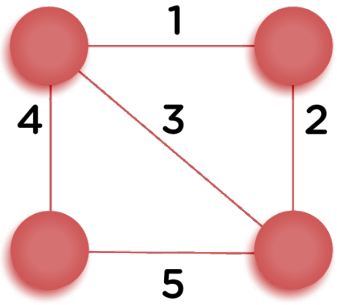
If there are *n* vertices and *e* edges in the graph, then any spanning tree corresponding to that graph contains *n* vertices and *n-1* edges.

Properties of spanning trees:

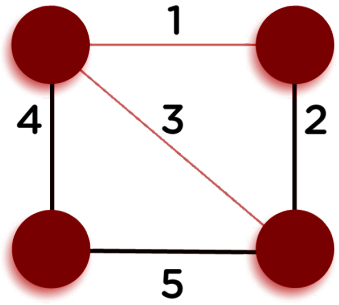
- A connected and undirected graph can have more than one spanning tree.
- The spanning tree is free of loops, i.e., it is acyclic.
- Removing any one of the edges will make the graph disconnected.
- Adding an extra edge to the spanning tree will create a loop in the graph.

Minimum Spanning Tree(MST) is a spanning tree with weighted edges.

In a weighted graph, the MST is a spanning tree with minimum weight than all other spanning trees of that graph. Refer to the image below for better understanding.

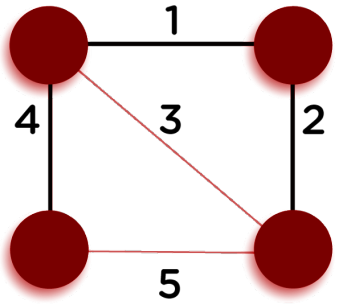


Undirected graph



Spanning tree

Cost=11(=4+5+2)



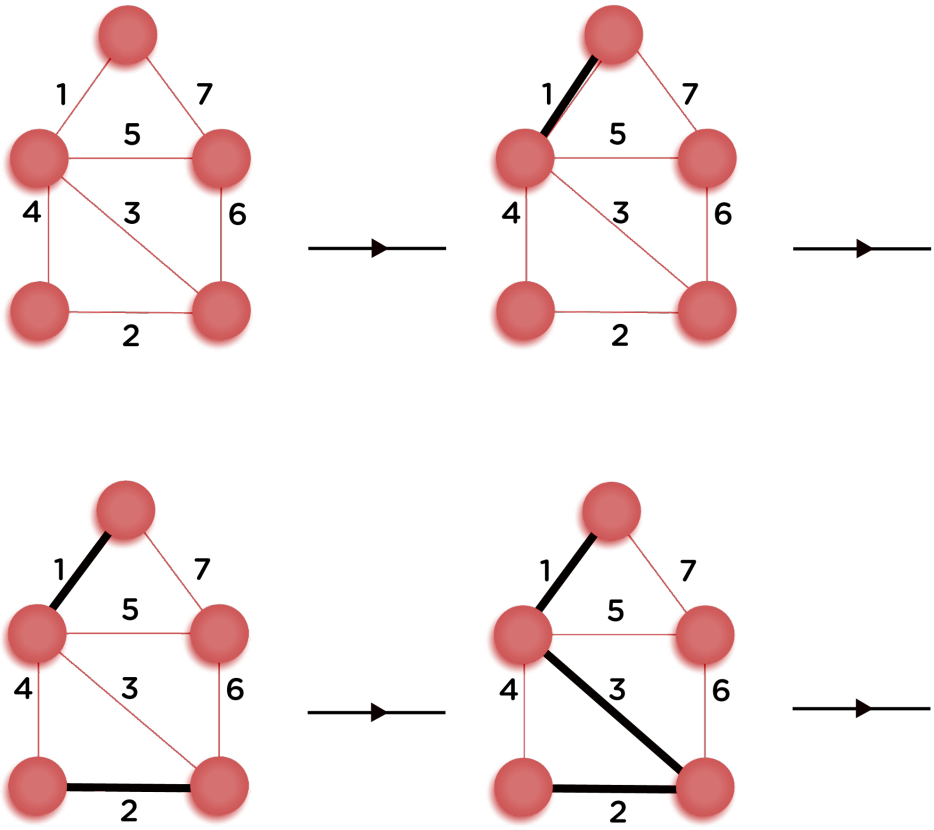
Minimum Spanning tree

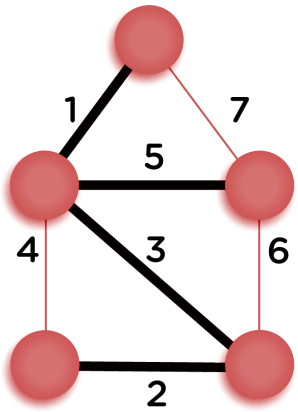
Cost=7(=4+1+2)

Kruskal's Algorithm:

This algorithm is used to find MST for the given graph. It builds the spanning tree by adding edges one at a time. We start by picking the edge with minimum weight, adding that edge into the MST, and increasing the count of edges in the spanning tree by one. Now, we will be picking the minimum weighted edge by excluding the already chosen ones and correspondingly increasing the count. While choosing the edge, we will also make sure that the graph remains acyclic after including the same. This process will continue until the count of edges in the MST reaches *n-1*. Ultimately, the graph obtained will be MST.

Refer to the example below for a better understanding of the same.





This is the final MST obtained using Kruskal's algorithm. It can be checked manually that the final graph is the MST for the given graph.

Cycle Detection

While inserting a new edge in the MST, we have to check if introducing that edge makes the MST cyclic or not. If not, then we can include that edge, otherwise not.

Now, let's figure out a way to detect the cycle in a graph. The following are the possible cases:

- By including an edge between the nodes A and B, if both nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.
- Out of two vertices, if any one of them has not been visited (or not present in the MST), then that vertex can also be included in the MST.
- If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can't use this method to detect the presence of the cycle.

Let's think of a better approach. We have already solved the **hasPath** question in the previous module, which returns true if there is a path present between two vertices v1 and v2, otherwise false.

Now, before adding an edge to the MST, we will check if a path between two vertices of that edge already exists in the MST or not. If not, then it is safe to add that edge to the MST.

As discussed in previous lectures, the time complexity of the **hasPath** function is O(E+V), where E is the number of edges in the graph and, V is the number of vertices. So, for (n-1) edges, this function will run (n-1) times, leading to bad time complexity, as in the worst case, $E = V^2$.

Now, moving on to a better approach for cycle detection in the graph.

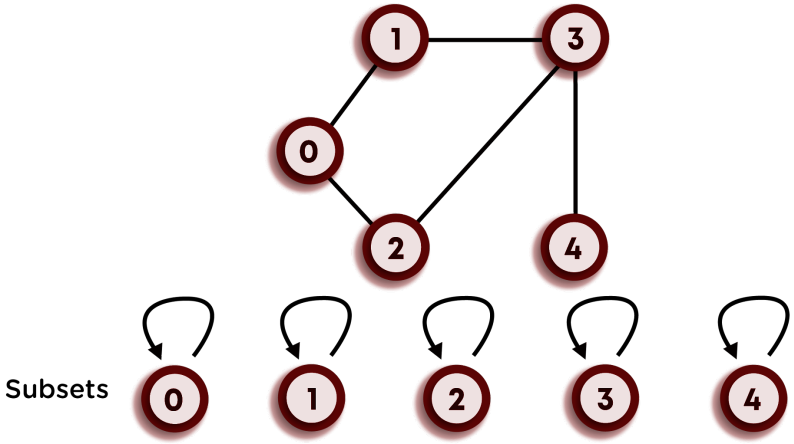
Union-Find Algorithm:

Before adding any edge to the graph, we will check if the two vertices of the edge lie in the same component of MST or not. If not, then it is safe to add that edge to the MST.

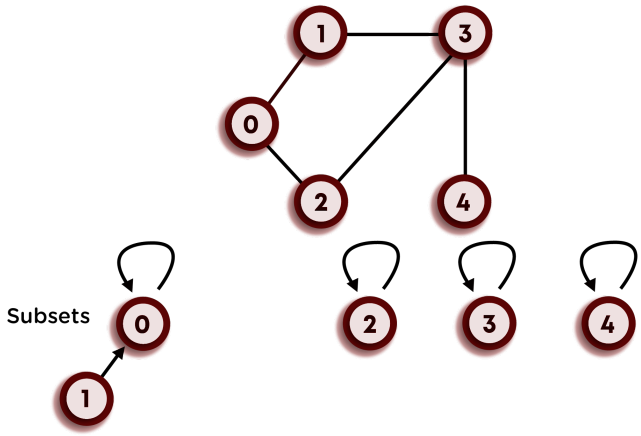
Following the steps of the algorithm:

- We will assume that initially, the total number of disjoint sets is equal to the number of vertices in the graph starting from 0 to n-1.
- We will maintain a parent array specifying the parent vertex of each of the vertex of the graph. Initially, as each vertex belongs to a different disjoint set (connected component), hence each vertex will be its parent.
- Now, before inserting any edge into the MST, we will check the parent of the vertices. If their parent vertices are equal, they belong to the same connected component; hence it is unsafe to add that edge.
- Otherwise, we can add that edge into the MST, and simultaneously update the parent array so that they belong to the same component(Refer to the code on how to do so).

Look at the following example, for better understanding:



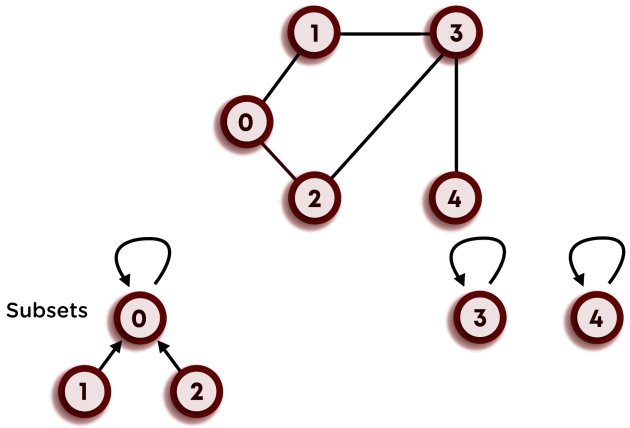
Initially all parent pointers are pointing to self
means only one element in each subset



Edge 0-1

Find: 0 belongs to subset 0 and 1 belongs to subset 1 so they are in different subsets.

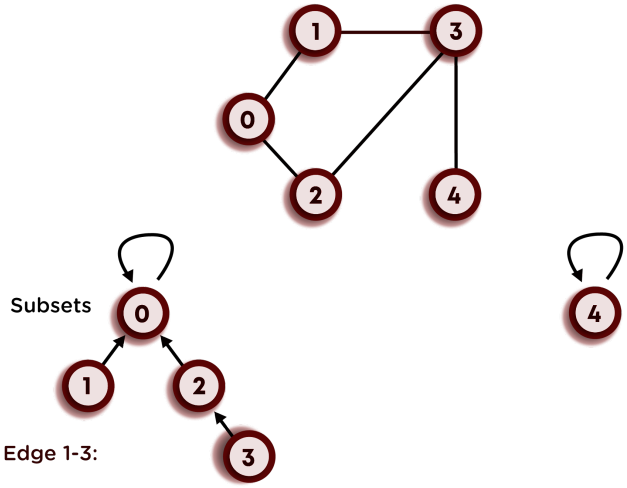
Union: Make 0 as the parent of 1, Updated set is {0,1}. 0 is the set representative since 0 is parent for itself.



Edge 0-2:

Find: 0 belongs to subset 0 and 2 belongs to subset 2 so they are in different subsets.

Union: Make 0 as the parent of 2, Updated set is {0,1,2}. 0 is the set representative since 0 is parent for itself.

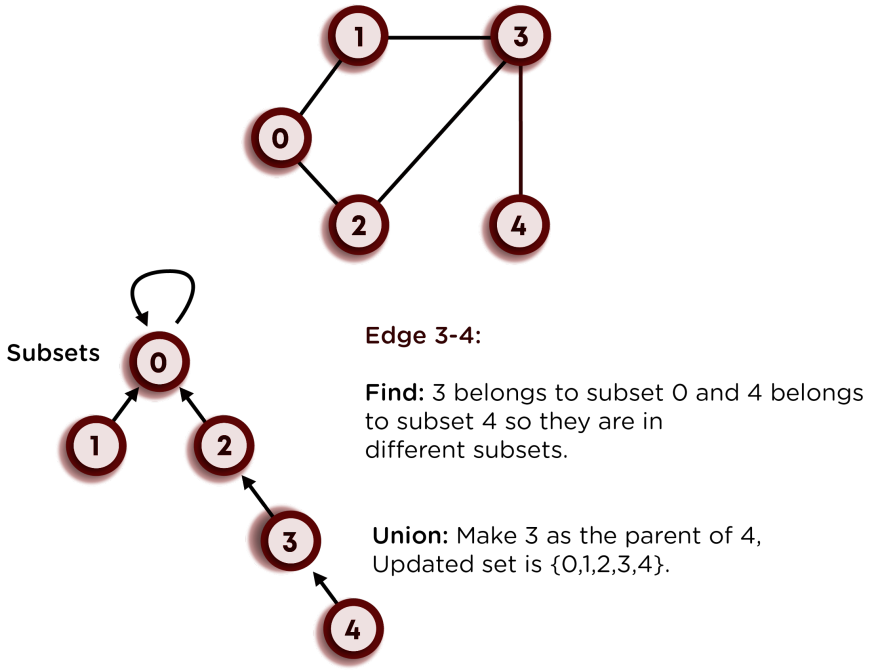


Edge 1-3:

Find: 1 belongs to subset 0 and 3 belongs to subset 3 so they are in different subsets.

Union: Make 1 as the parent of 3, Updated set is {0,1,2,3}. 0 is the set representative since 0 is parent for itself.

Note: While finding the parent of the vertex, we will be finding the topmost parent (Oldest Ancestor). For example: suppose, the vertex 0 and the vertex 1 were



connected, where the parent of 0 is 1, and the parent of 1 is 1. Now, while determining the parent of the vertex 0, we will visit the parent array and check the vertex at index 0. In our case, it is 1. Now we will go to index 1 and check the parent of index 1, which is also 1. Hence, we can't go any further as the index is the parent of itself. This way, we will be determining the parent of any vertex.

The time complexity of the union-find algorithm becomes O(V) for each vertex in the worst case due to skewed-tree formation, where V is the number of vertices in the graph. Here, we can see that time complexity for cycle detection has significantly improved compared to the previous approach.

Kruskal's Algorithm: Implementation

Till now, we have studied the logic behind Kruskal's algorithm for finding MST. Now, let's discuss how to implement it in code.

Consider the code below and follow the comments for a better understanding.

```
class Edge:# Class that store values for each vertex
        def __init__ (self,src,dest,wt):
                self.src = src
                self.dest dest
                self.wt = wt

def getParent(v,parent): # Function to find the parent of a vertex
        if v == parent[v]: #When a vertex is parent of itself
                return v  # Recursively called to find the topmost parent
        return getParent(parent[v], parent)

def kruskal(edges,n,E):
        edges = sorted(edges,key = lambda edge:edge.wt) #Inbuilt sort
        output = []  # Array to store final edges of MST
        parent [i for i in range(n)]
        count = 0
        i = 0
        while count < (n-1):
                currentEdge = edges[i]
        # Figuring out the parent of each edge's vertices
                srcParent = getParent(currentEdge.src,parent)
                destParent = getParent(currentEdge.dest,parent)
        # Parents are not equal-> then add the edge to output
                if srcParent != destParent:
                        output.append(currentEdge)
                        count+=1  # Increased the count
                        parent[srcParent] = destParent #Update Parent array
                i+=1
        return output


li = [int(ele) for ele in input().split()]
```

```
n= li[0]
E= li[1]
edges = []
for i in range(E) :
        curr_input = [int(ele) for ele in input().split()]
        src = curr_input[0]
        dest = curr_input[1]
        wt = curr_input[2]
        edge = Edge(src,dest,wt)
        edges.append(edge)

output = kruskal(edges,n,E)
for ele in output:# Finally, printing the MST obtained.
        if(ele.src < ele.dest):
                print(str(ele.src) +" "+ str(ele.dest)+" "+ str(ele.wt)
        else:
                print(str(ele.dest) + " "+ str(ele.src)+ " "+ str(ele.wt)
```

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n, and the total number of edges is E)

- Take input in the array of size E.
- Sort the input array based on edge-weight. This step has the time complexity of O(E log(E)).
- Pick (n-1) edges and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle detection, in the worst-case time complexity of E edges will be O(E.n), as discussed earlier.

Hence, the total time complexity of Kruskal's algorithm becomes O(E log(E) + n.E). This time complexity is bad and needs to be improved.

We can't reduce the time taken for sorting, but the time taken for cycle detection can be improved using another algorithm named **Union by Rank and Path Compression**. You need to explore this on yourselves. The basic idea in these algorithms is that we will be

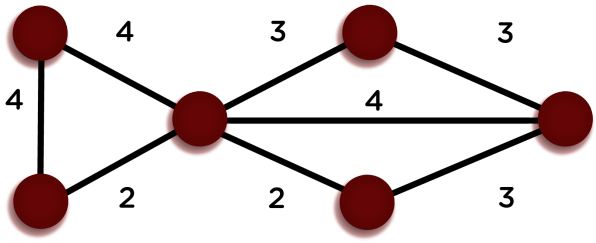
avoiding the formation of skewed-tree structure, which reduces the time complexity for each vertex to O(log(E)).

Prim's Algorithm

This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

In this algorithm, the MST is built by adding one edge at a time. In the beginning, the spanning tree consists of only one vertex, which is chosen arbitrarily from the set of all vertices of the graph. Then the minimum weighted edge, outgoing from this vertex, is selected and simultaneously inserted into the MST. Now, the tree contains two edges. Further, we will be selecting the edge with the minimum weight such that one end is already present there in the MST and the other one from the unselected set of vertices. This process is repeated until we have inserted a total of (n-1) edges in the MST.

Consider the following example for a better understanding.

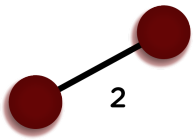


Step: 1

Start with a weighted graph

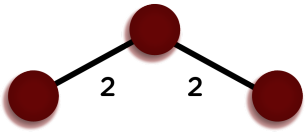
Step: 2

Choose a vertex



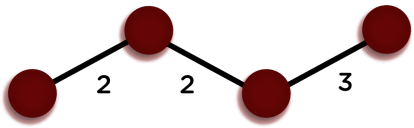
Step: 3

Choose the shortest edge from this vertex add it



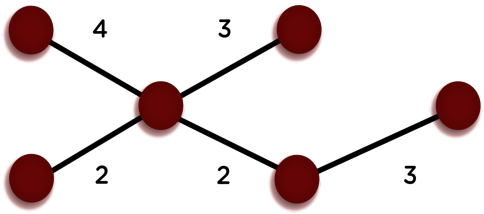
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Implementation:

- We are considering the starting vertex to be 0 with a parent equal to -1, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).
- The parent of all other vertices is assumed to be NIL, and the weight will be equal to infinity, which means that the vertex has not been visited yet.
- We will mark the vertex 0 as visited and the rest as unvisited. If we add any vertex to the MST, then that vertex will be shifted from the unvisited section to the visited section.
- Now, we will update the weights of direct neighbors of vertex 0 with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from vertex 0.
- This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.

Let's look at the code now:

```
class Graph:

        def __init__ (self,nVertices):
                self.nVertices = nVertices
                self.adjMatrix = [[0 for 1 in range(nVertices)] for j in range(nVertices)]

        def addEdge(self,v1,v2,wt):
                self.adjMatrix[v1][v2] = wt
                self.adjMatrix[v2][v1] =  wt

        def __getMinVertex(self,visited,weight):
                minVertex = -1  # Initialized to -1 means there is no vertex
                for i in range(self.nVertices):
        # Conditions :
        #the vertex must be unvisited and either minVertex value is -1
        #or if minVertex has some vertex to it, then weight of
        #currentVertex should be less than the weight of the minVertex.
            a = (minVertex == -1 or (weight[minVertex] > weight[i]))
                        if(visited[i] is False and a:
                                minVertex = i
                return minVertex

        def prims(self):
        # Initially, the visited array is assigned to false
        #and weights array is set to infinity.
                visited = [False for i in range(self.nVertices)]
                parent = [-1 for i in range(self.nVertices)]
                weight = [sys.maxsize for i in range(self.nVertices)]

            for i in range(self.nVertices - 1):
        # Find min vertex
                minVertex = self.__getMinVertex(visited,weight)
                visited[minVertex] = True
        # Explore unvisited neighbors
                for j in range(self.nVertices):
                    if(self.adjMatrix[minVertex][j]>0 and visited[j] is False):
                                if(weight[j] > self.adjMatrix[minVertex][j]):
                                        weight[j] = self.adjMatrix[minVertex][j]
                                        parent[j] = minVertex
        # Final MST printed
            for i in range(1,self.nVertices):
                    if parent[i] > i:
```

```
                a = str(i) +" "+ str(parent[i])+" "+ str(weight[i])
                        print(a)
                else:
                a = str(parent[i]) +" "+ str(i) +" "+ str(weight[i])
                        print(a)

        def removeEdge(self,v1,v2):
                if not self.containsEdge(v1,v2):
                        return
                self.adjMatrix[v1][v2] = 0
                self.adjMatrix[v2][v2] = 0

        def containsEdge(self,v1,v2):
                return True if self.adjMatrix[v1][v2] > 0 else False

li = [int(ele) for ele in input().split()]
n = li[0]
E= li[1]
g= Graph(n) #Inbuilt Graph class
for in range(E):
        curr_edge = [int(ele) for ele in input().split()]
        g.addEdge(curr_edge[0], curr_edge[1], curr_edge[2])
g.prims ()
```

Time Complexity of Prim's Algorithm:

Here, n is the number of vertices, and E is the number of edges.

- The time complexity for finding the minimum weighted vertex is O(n) for each iteration. So for (n-1) edges, it becomes $O(n^2)$.
- Similarly, for exploring the neighbor vertices, the time taken is $O(n^2)$.

It means the time complexity of Prim's algorithm is $O(n^2)$. We can improve this in the following ways:

- For exploring neighbors, we are required to visit every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.

- Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of $O(n^2)$. Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a **priority queue** where the priority will be taken as weights of the vertices. This will take O(log(n)) time complexity to remove a vertex from the priority queue.

These optimizations can lead us to the time complexity of O((n+E)log(n)), which is much better than the earlier one. Try to write the optimized code by yourself.

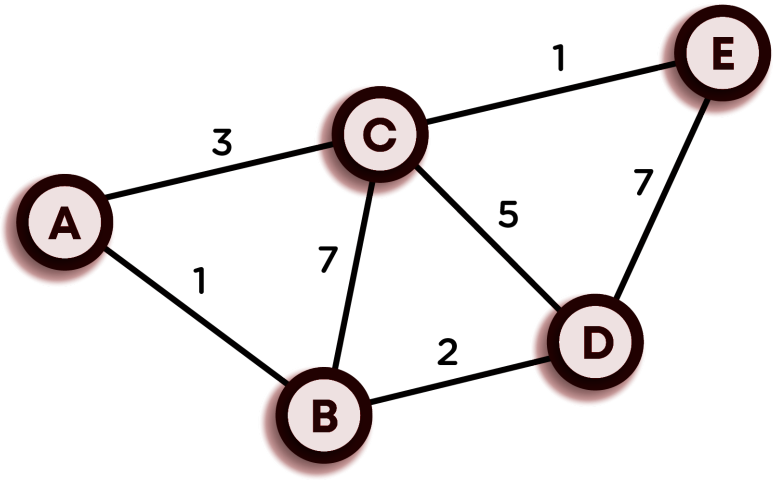
Dijkstra's Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

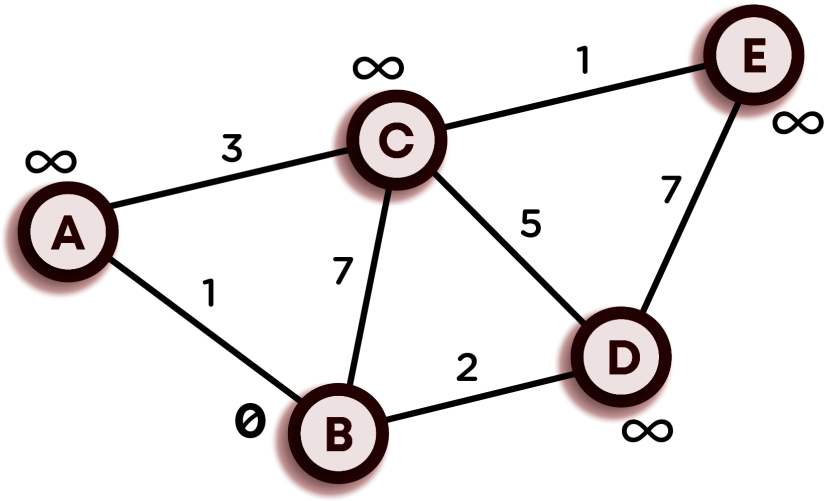
Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular source vertex.

Let's consider the algorithm with an example:

1. We want to calculate the shortest path between the source vertex C and all other vertices in the following graph.

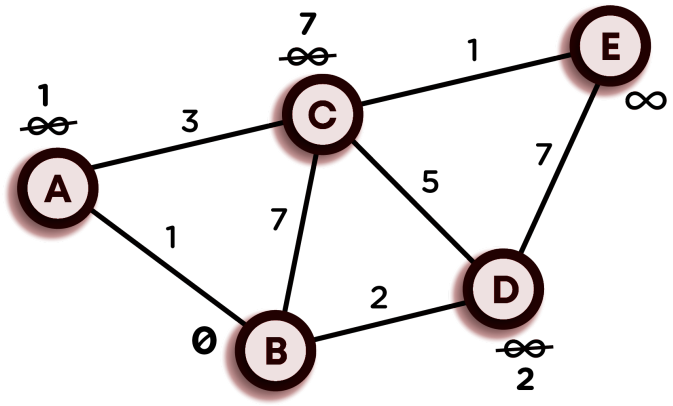


2. While executing the algorithm, we will mark every node with its **minimum distance** to the selected node, which is C in our case. Obviously, for node C itself, this distance will be 0, and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.

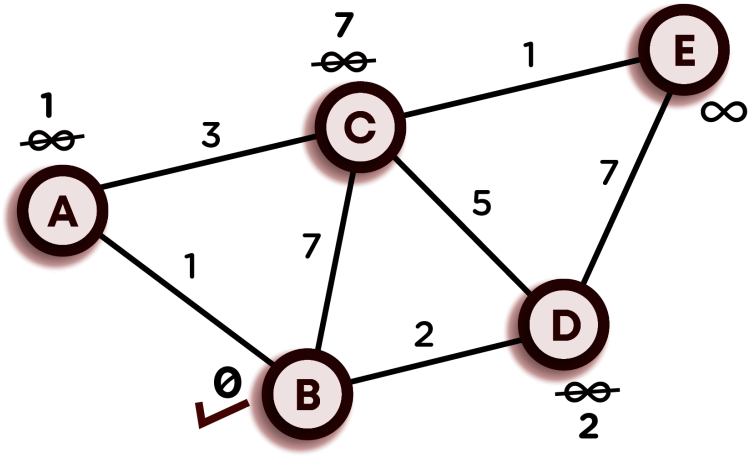


3. Now, we will check for the neighbors of the current node, which are A, B, and D. Now, we will add the minimum cost of the current node to the weight of the edge

connecting the current node and the particular neighbor node. For example, for node B, it's weight will become minimum(infinity, 0+7) = 7. This same process is repeated for other neighbor nodes.

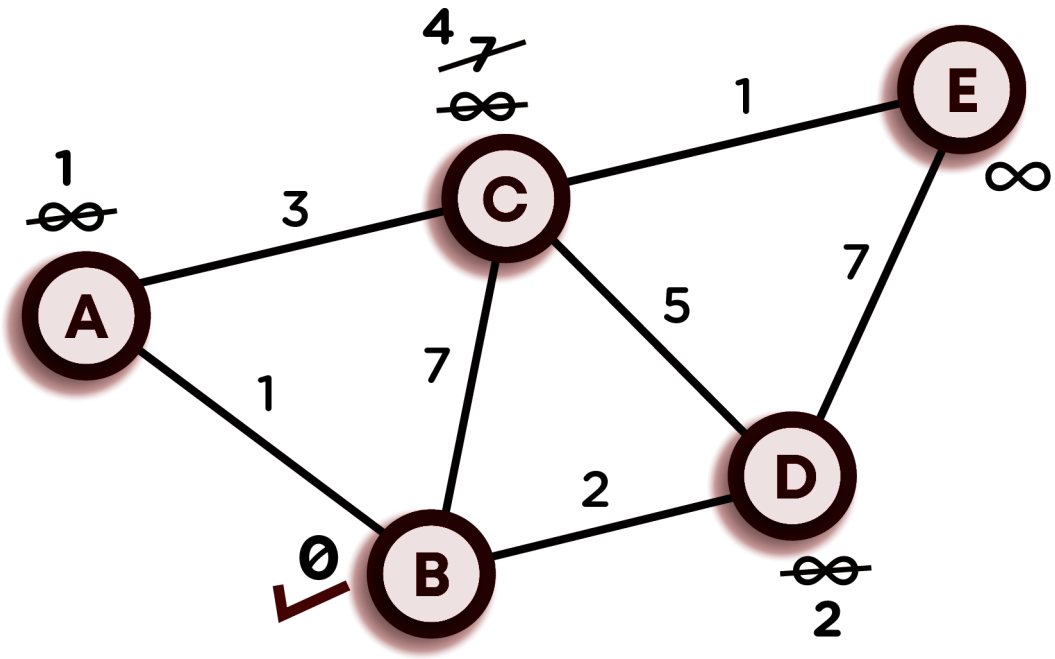


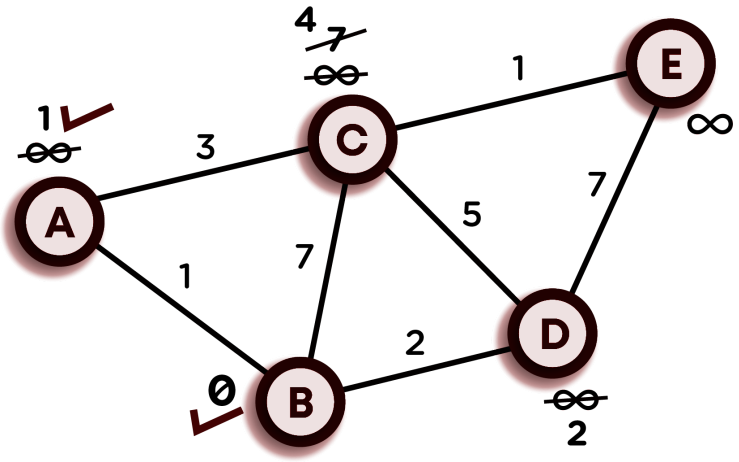
4. Now, as we have updated the distance of all the neighbor nodes of the current node, we will mark the current node as visited.

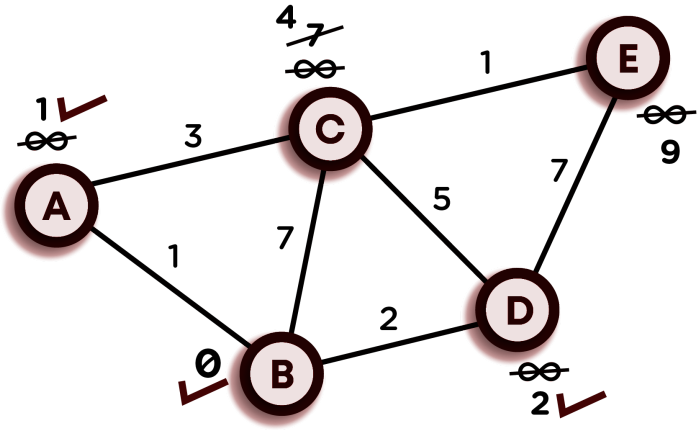


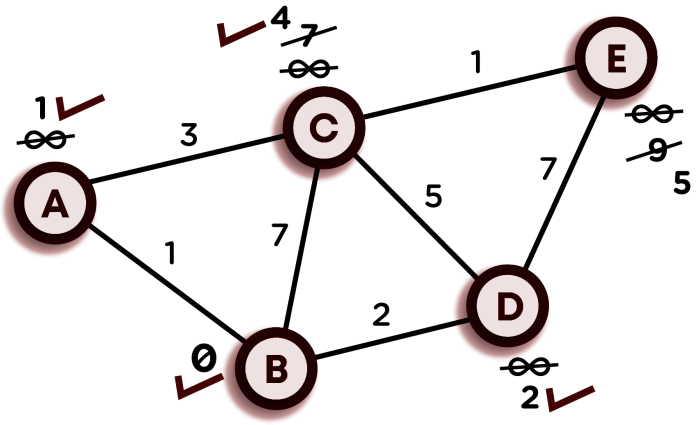
5. After this, we will be selecting the minimum weighted node among the remaining vertices. In this case, it is node A. Take this node as the current node.

6. Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:

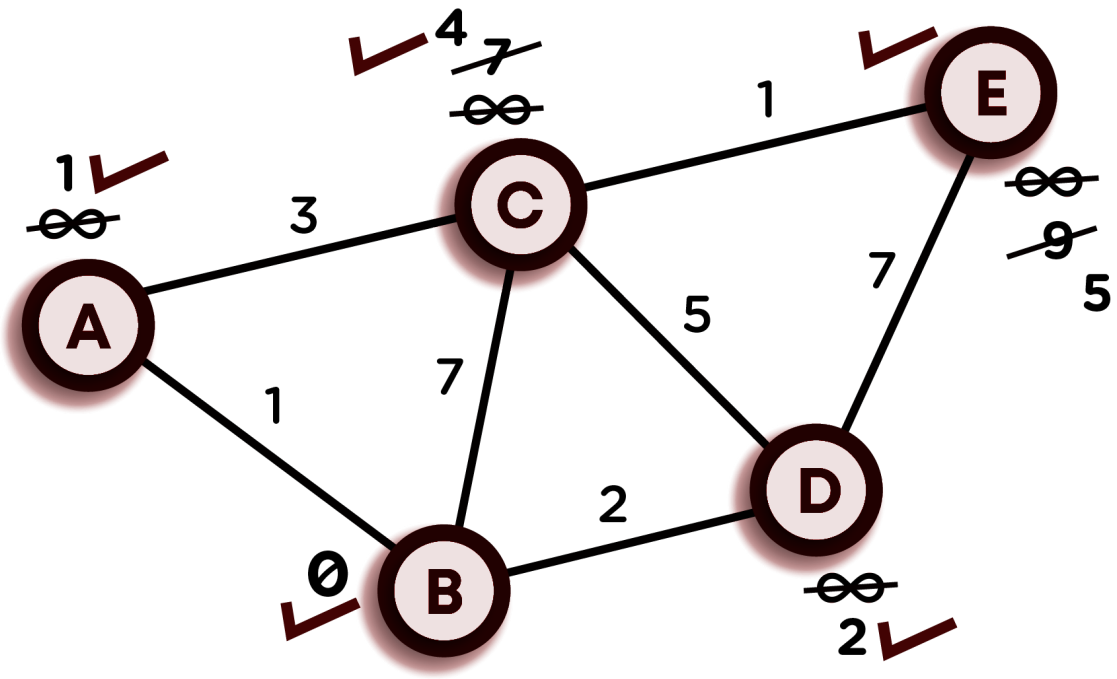








7. Finally, we will get the graph as follows:



The distances finally marked at each node are minimum from node C.

Implementation:

Let's look at the code below for a better explanation:(Code is nearly same as that of Prim's algorithm, just a change while updating the distance)

```
class Graph:

        def __init__ (self,nVertices):
                self.nVertices = nVertices
                self.adjMatrix = [[0 for 1 in range(nVertices)] for j in range(nVertices)]

        def addEdge(self,v1,v2,wt):
                self.adjMatrix[v1][v2] = wt
                self.adjMatrix[v2][v1] =  wt

        def __getMinVertexD(self,visited,weight):
                minVertex = -1  # Initialized to -1 means there is no vertex
                for i in range(self.nVertices):
              a = (minVertex == -1 or (weight[minVertex] > weight[i]))
                        if(visited[i] is False and a:
                                minVertex = i
                return minVertex

        def djikstra(self):
         # Initially, the visited array is assigned to false
         #and weights array is set to infinity.
                visited = [False for i in range(self.nVertices)]
                dist = [sys.maxsize for i in range(self.nVertices)]
         dist[0]=0
             for i in range(self.nVertices - 1):
              # Find min vertex
                        minVertex = self.__getMinVertexD(visited,weight)
                        visited[minVertex] = True
              # Explore unvisited neighbors
                        for j in range(self.nVertices):
                             if(self.adjMatrix[minVertex][j]>0 and visited[j] is False):
                                        if(dist[j] > dist[minVertex]+ self.adjMatrix[minVertex][j]):
```

```
                                    dist[j] = self.adjMatrix[minVertex][j] +dist[minVertex]
                                    parent[j] = minVertex

        # Final MST printed
            for i in range(self.nVertices):
                    print(str(i)+" " + str(dist[i]))


li = [int(ele) for ele in input().split()]
n = li[0]
E= li[1]
g= Graph(n) #Inbuilt Graph class
for in range(E):
        curr_edge = [int(ele) for ele in input().split()]
        g.addEdge(curr_edge[0], curr_edge[1], curr_edge[2])
g.djikstra()
```

Time Complexity of Dijkstra's algorithm:

The time complexity is also the same as that of Prim's algorithm, i.e., **$O(n^2)$**. This can be reduced by using the same approaches as discussed in Prim's algorithm's content.

Topological sorting

Topological sorting is a technique used to linearly order the vertices of a directed acyclic graph (DAG) in such a way that for every directed edge (u, v), vertex u comes before vertex v in the ordering. It is typically used to solve problems that involve dependencies between tasks or activities.

Here's the pseudocode for topological sorting in Python:

```
function topologicalSort(graph):
   # Initialize an empty list to store the topological ordering.
   topologicalOrder = []

   # Create a dictionary to keep track of visited vertices.
   visited = {}
```

```
# Helper function to perform depth-first search (DFS).
function dfs(v):
    # Mark the current vertex as visited.
    visited[v] = True

    # Recursively visit all adjacent vertices.
    for neighbor in graph[v]:
        if neighbor not in visited:
            dfs(neighbor)

    # Add the current vertex to the topological order.
    topologicalOrder.append(v)

# Iterate through all vertices in the graph.
for vertex in graph:
    if vertex not in visited:
        # Start DFS from unvisited vertices.
        dfs(vertex)

# Reverse the order to get the topological sort.
topologicalOrder.reverse()

return topologicalOrder
```

Applications of Topological sorting :

1. Task Scheduling: Topological sorting is often used in task scheduling to determine the order in which tasks or jobs should be executed based on their dependencies. For example, in a build system like Make or in a task scheduler, tasks may have dependencies on other tasks, and topological sorting can help ensure that tasks are executed in the correct order.
2. Course Prerequisites: In educational institutions, course prerequisites form a directed acyclic graph, where courses depend on other courses. Topological sorting can be used to determine the order in which courses should be taken to ensure that all prerequisites are met.

3. Dependency Resolution: In software development, libraries and modules can have dependencies on other libraries or modules. Topological sorting can help resolve these dependencies and ensure that modules are loaded in the correct order.
4. Package Management: Package managers like npm (Node Package Manager) and pip (Python Package Installer) use topological sorting to install packages and their dependencies in the correct order.
5. Project Scheduling: In project management, tasks in a project can have dependencies on each other, and topological sorting can help create a project schedule that respects these dependencies.

In summary, topological sorting is a versatile algorithm used in various domains to handle situations where there are dependencies between elements, and it ensures that these dependencies are satisfied in a systematic and efficient manner.

Kahn's Algorithm for Topological Sorting

Kahn's algorithm is a popular method for topological sorting of directed acyclic graphs (DAGs). It is based on the concept of repeatedly removing vertices with no incoming edges.

Algorithm Steps:

1. Initialize:
   - Create an empty list to store the topological order.
   - Calculate the in-degree (number of incoming edges) for each vertex in the graph.
   - Create a queue (or a similar data structure) to store vertices with an in-degree of 0

2. Enqueue:
   - Enqueue all vertices with an in-degree of 0 into the queue.

3. Process:
   - While the queue is not empty, do the following:
     - Dequeue a vertex from the queue.
     - Add the vertex to the topological order.
4. Update In-degrees:

- For each neighbour of the dequeued vertex, decrement its in-degree by 1.
- If the neighbour's in-degree becomes 0, enqueue it.

5. Repeat:
   - Repeat steps 3 and 4 until the queue is empty.

6. Result:
   - If the topological order contains all vertices (the graph is a DAG), return the topological order.
   - If not all vertices are included in the topological order (the graph has cycles), it is not a valid DAG.

Python Implementation:

Here's a Python implementation of Kahn's algorithm for topological sorting:

```
from collections import defaultdict, deque

def topologicalSort(graph):
    # Calculate in-degrees for each vertex.
    in_degree = defaultdict(int)
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    # Initialise a queue with vertices having in-degree 0.
    queue = deque([u for u in graph if in_degree[u] == 0])

    # Initialise the topological order.
    topological_order = []

    # Perform Kahn's algorithm.
    while queue:
        u = queue.popleft()
        topological_order.append(u)
        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
```

```
            queue.append(v)

    # Check if all vertices are included in the topological order.
    if len(topological_order) == len(graph):
        return topological_order
    else:
        return []  # Graph contains cycles.

# Example usage:
graph = {
    1: [2, 3],
    2: [4],
    3: [4, 5],
    4: [5],
    5: []
}

result = topologicalSort(graph)
if result:
    print("Topological Order:", result)
else:
    print("The graph has cycles.")
```

This Python code demonstrates Kahn's algorithm to find the topological order of a DAG represented as a dictionary of vertices and their outgoing edges. It handles both cases where a valid topological order exists and where the graph contains cycles.

Applications of Kahn's algorithm

1. Compiler Optimizations: Compilers use topological sorting to optimise the order in which code is executed or compiled. This can result in more efficient code generation and reduced compilation times.
2. Database Query Optimization: Query planners in database systems may use topological sorting to optimise the order in which database tables are joined, reducing query execution time.

3. Workflow Management: In workflow automation and data processing systems, tasks often have dependencies on the output of other tasks. Kahn's algorithm can be used to determine the execution order of these tasks.
4. Critical Path Analysis: In project management, Kahn's algorithm can be employed to identify the critical path in a project schedule. This path represents the sequence of tasks that, if delayed, would cause the project's completion date to be delayed as well.
5. Event Scheduling: Event-driven systems, such as event-driven simulations or event processing engines, can benefit from Kahn's algorithm to determine the order in which events should be processed based on event dependencies.
6. Parallel Execution: In parallel computing, Kahn's algorithm can help determine the execution order of parallel tasks while respecting dependencies, leading to efficient parallel processing.