PROJECT REPORT ON

"Simulation of algorithms for finding Minimum Spanning Tree of a network"

Computer Communication Network

Department of ECE

PES University Bangalore

Faculty in charge

PROF. PRAJEESHA

Submitted by:

Sourabh T SRN: PES1UG20EC339

Pruthvik Kashyap S SRN: PES1UG20EC326

Semester: 5

Section: F

CONTENTS:

OBJECTIVE/ PROBLEM STATEMENT

To implement algorithms for finding the Minimum Spanning Tree of a given network.

INTRODUCTION

A **spanning tree** *T* of an undirected graph *G* is a subgraph that is a tree which includes all of the vertices of *G*.[1] In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree

A **minimum spanning tree (MST)** or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

Two of the important algorithms used for finding MST of a network are

1. Prim's Algorithm
2. Kruskal's Algorithm

PRIM'S ALGORITHM

Prim's algorithm (also known as Jarník's algorithm) is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. Therefore, it is also sometimes called the Jarník's algorithm, Prim–Jarník algorithm, Prim–Dijkstra algorithm or the DJP algorithm.

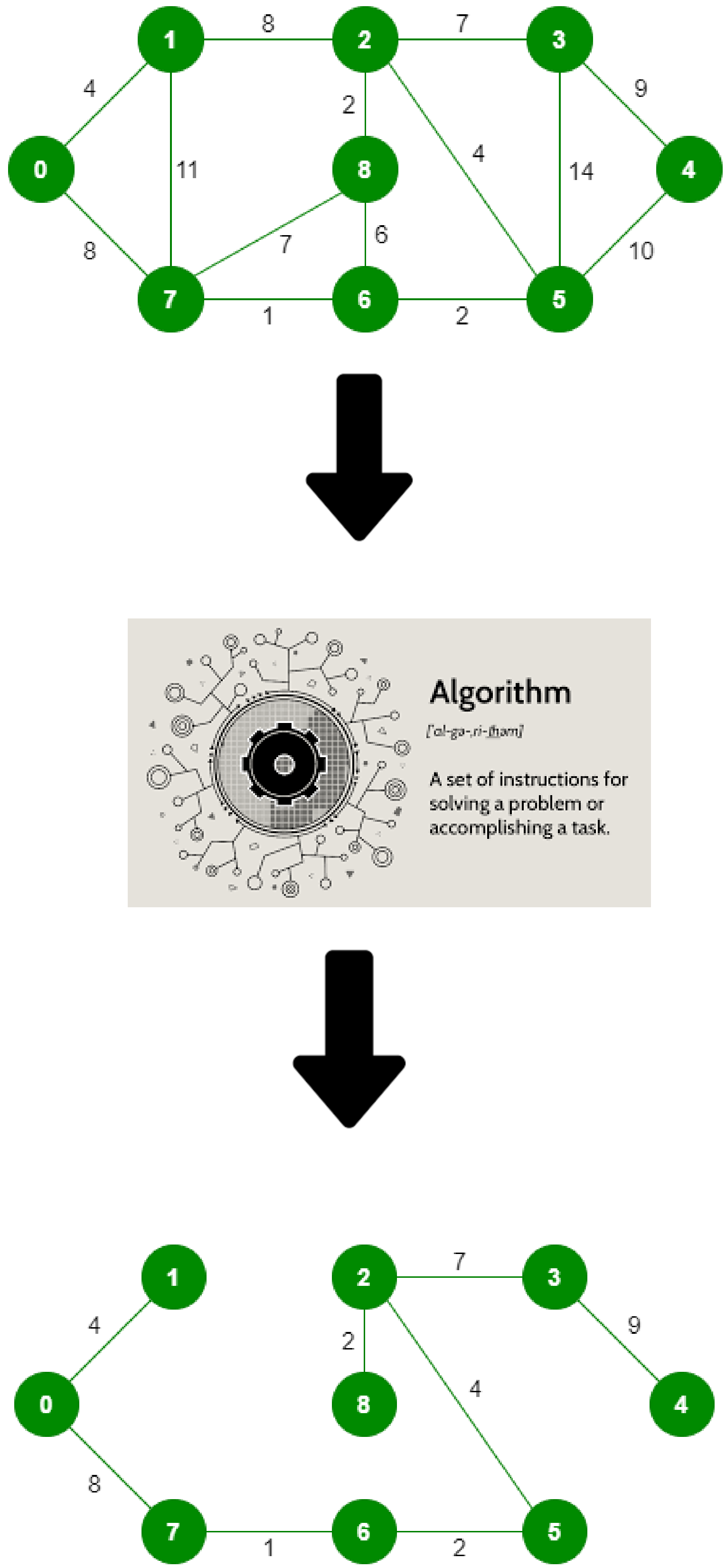
KRUSKAL'S ALGORITHM

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning

tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

This algorithm first appeared in *Proceedings of the American Mathematical Society*, in 1956, and was written by Joseph Kruskal. It was rediscovered by in 1957.

FLOW CHART


Algorithm
['al-gə-ˌri-t͟həm]
A set of instructions for solving a problem or accomplishing a task.

PROCEDURE:

Task: Step by Step

- Define a network in terms of routers and links between routers.
- Take the network as the input to the program and store it in most suitable data structure.
- Visualize the network taken as input.
- Run the algorithm of your choice to find the Minimum Spanning Tree of the network.
- Visualize the network taken as input.

SOFTWARE CODE

main.py :

```
import prims
import kruskal

selection = int(input("Please choose the algorithm: \n 1 : Prim's Algorithm \n 2
: Kruskal's Algorithm \n >>  "))

if selection==1 :
    network = {}
    # Network is stored in a dictonary where key is the source router and each
elements in the value list are the destination router

    router_count=0   # stores the number of routers in the network
    link_count=0    # stores the number of links in the network

    network, router_count, link_count = prims.takeNetworkInput(network,
router_count, link_count)


    prims.visualize(network) # Visulaize the MST of the network

    parent = [None]*router_count    # Used to store the parent of router
```

```
    key = [1000000]*router_count    # Used to store the link cost between the router and its parent
    mstSet = [False]*router_count   # Used to check is the router has been included into MST or not

   prims.prims(network, router_count, parent, key, mstSet) # Runs the Prim's Algorithm and stores the results into parent and key lists

    prims.visulaizeMST(parent, key, router_count)   # Visulaize the MST of the network



elif selection == 2:

    num_nodes=0   # stores the number of routers in the network
    num_links=0   # stores the number of links in the network
    links=[]    # Used to store the links in the network

    num_nodes, num_links = kruskal.takeNetworkInput(links) # Taking input

    kruskal.visualize(links)

    parent = [i for i in range(num_nodes)]
    rank = [0]*num_nodes
    cost=0

    mst = []

    cost = kruskal.kruskal(links, parent, mst, rank,cost)

    kruskal.visualizeMST(mst)
```

prims.py :

```
import heapq
from sys import argv
import matplotlib.pyplot as plt
import networkx as nx


def takeNetworkInput(network, router_count, link_count) :
  input_file = open(argv[1],'r')  # Opening the file which contains the network 
information
  Line = input_file.readline()
  Line = Line.strip().split(' ')

  router_count = int(Line[0]) # Total number of routers
  link_count = int(Line[1]) # Total number of links between routers


  for router in range(router_count):
    add_router(router, network, router_count)   # adding routers to the network


  links=[]
  for i in range(link_count) :
    line = input_file.readline()  # reading the links in the network line by line
    line = line.strip().split(' ')

    # Adding the links to the network by considering the router IDs and link cost
    add_link(int(line[0]),int(line[1]),int(line[2]),network)


  print_network(network)
  return network, router_count, link_count






# Add a Router to the dictionary
def add_router(v,network,router_count):
  if v in network:
    print("Router ", v, " already exists.")
  else:
    router_count = router_count + 1
    network[v] = []
```

```
# Add an link between router u and v with link cost e
def add_link(v1, v2, e,network):
  # Check if router u is a valid router
  if v1 not in network:
    print("Router ", v1, " does not exist.")
  # Check if router v is a valid router
  elif v2 not in network:
    print("Router ", v2, " does not exist.")
  else:
    temp = [v2, e]
    network[v1].append(temp)


# Print the network
def print_network(network):
  for vertex in network:
    for edges in network[vertex]:
      print(vertex, " -> ", edges[0], " link cost: ", edges[1])


def visualize(network):
            G = nx.Graph()
            for router_links in network:
                for dest in network[router_links]:
                     G.add_edge(router_links,dest[0],weight=dest[1])


            pos = nx.spring_layout(G, seed=7)  # positions for all nodes - seed
for reproducibility

            # routers
            nx.draw_networkx_nodes(G, pos, node_size=700)

            # links
            nx.draw_networkx_edges(G, pos)

            # router labels
            nx.draw_networkx_labels(G, pos, font_size=20, font_family="sans-
serif")

            # link cost labels
            edge_labels = nx.get_edge_attributes(G, "weight")
            nx.draw_networkx_edge_labels(G, pos, edge_labels)

            plt.title("Network")
```

```
            plt.show()


def visulaizeMST(parent,key,router_count):
            G = nx.Graph()
            for i in range(1,router_count):
                G.add_edge(parent[i],i,weight=key[i])


            pos = nx.spring_layout(G, seed=7)  # positions for all nodes - seed
for reproducibility

            # nodes
            nx.draw_networkx_nodes(G, pos, node_size=700)

            # edges
            nx.draw_networkx_edges(G, pos)

            # node labels
            nx.draw_networkx_labels(G, pos, font_size=20, font_family="sans-
serif")
            # edge weight labels
            edge_labels = nx.get_edge_attributes(G, "weight")

            nx.draw_networkx_edge_labels(G, pos, edge_labels)

            plt.title("Minimum Spanning Tree (MST)")
            plt.show()


def prims(network,router_count, parent, key ,mstSet):
  key[0]=0
  parent[0]=-1

  pq = []
  heapq.heapify(pq) # converting the list pq to a priority queue
  # A priority queue by default stores the entry with least priority at the front
of the queue

  heapq.heappush(pq,[0,0])

  for i in range(router_count-1):
      u = pq[0][1]
      heapq.heappop(pq)
```

```
    mstSet[u]=True

    for link in network[u]:
        v = link[0]
        cost = link[1]

        if (mstSet[v] == False) and (cost < key[v]):
            parent[v] = u
            key[v] = cost
            heapq.heappush(pq,[key[v],v])
```

kruskal.py :

```
import networkx as nx
import matplotlib.pyplot as plt
from sys import argv

class Link:
    def __init__(self, first,second,cost):
        self.u=first
        self.v=second
        self.cost=cost


def findPar(u,parent):
    if u==parent[u]:
        return u
    return findPar(parent[u],parent)

def unionn(u,v,parent,rank):
    u = findPar(u, parent)
    v = findPar(v, parent)

    if rank[u] < rank[v]:
        parent[u] = v
    elif rank[v] < rank[u]:
        parent[v] = u
    else:
        parent[v] = u
        rank[u] = rank[u]+1
```

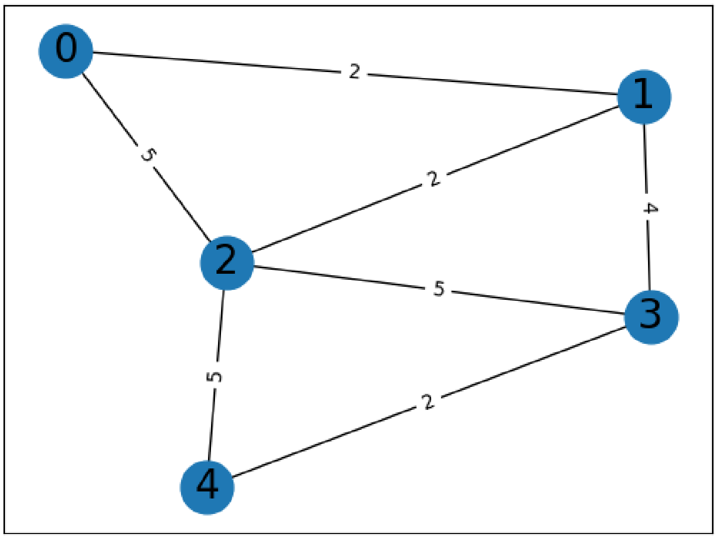
```python
def takeNetworkInput(links)
:
    input_file = open(argv[1],'r')
    Line = input_file.readline()
    Line = Line.strip().split(' ')

    num_nodes= int(Line[0])

    num_links = int(Line[1])

    for i in range(num_links) :
        line = input_file.readline()
        line = line.strip().split(' ')
        u = int(line[0])
        v = int(line[1])
        cost = int(line[2])
        links.append(Link(u,v,cost))

    links.sort(key  = lambda x: x.cost)

    for link in links:
        print(link.u,' -> ',link.v,' link cost: ',link.cost)
    print()

    return num_nodes, num_links

def visualize(links):
    G = nx.Graph()
    for router_links in links:
        G.add_edge(router_links.u,router_links.v,weight=router_links.cost)


    pos = nx.spring_layout(G, seed=7)

            # nodes
    nx.draw_networkx_nodes(G, pos, node_size=700)

    # routers
    nx.draw_networkx_edges(G, pos)

    # router labels
    nx.draw_networkx_labels(G, pos, font_size=20, font_family="sans-serif")
```

```
    # link weight labels
    edge_labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels)

    plt.title("Network")
    plt.show()


def visualizeMST(mst):
            G = nx.Graph()
            for router_links in mst:
                G.add_edge(router_links[0],router_links[1],weight=router_links[2]
)

            pos = nx.spring_layout(G, seed=7)  # positions for all nodes - seed
for reproducibility

            # nodes
            nx.draw_networkx_nodes(G, pos, node_size=700)

            # edges
            nx.draw_networkx_edges(G, pos)

            # node labels
            nx.draw_networkx_labels(G, pos, font_size=20, font_family="sans-
serif")
            # edge weight labels
            edge_labels = nx.get_edge_attributes(G, "weight")
            nx.draw_networkx_edge_labels(G, pos, edge_labels)


            plt.title("Minimum Spanning Tree (MST)")
            plt.show()

def kruskal(links, parent, mst, rank, costt):

    for link in links:
        if findPar(link.v, parent) != findPar(link.u, parent) :
            costt += link.cost
            mst.append((link.u,link.v,link.cost))
            unionn(link.u,link.v,parent,rank)

    for link in mst:
        print(link[0],' ',link[1],' ',link[2])
    return costt
```

RESULT:

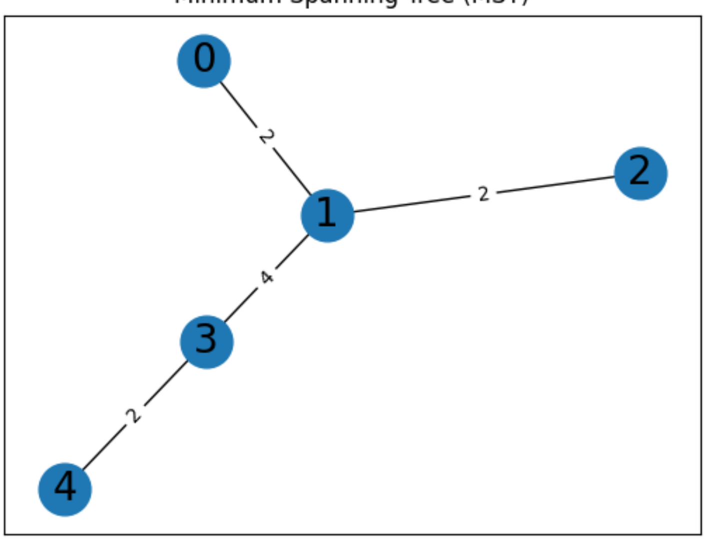
Example 1:
The network given as input.

Network

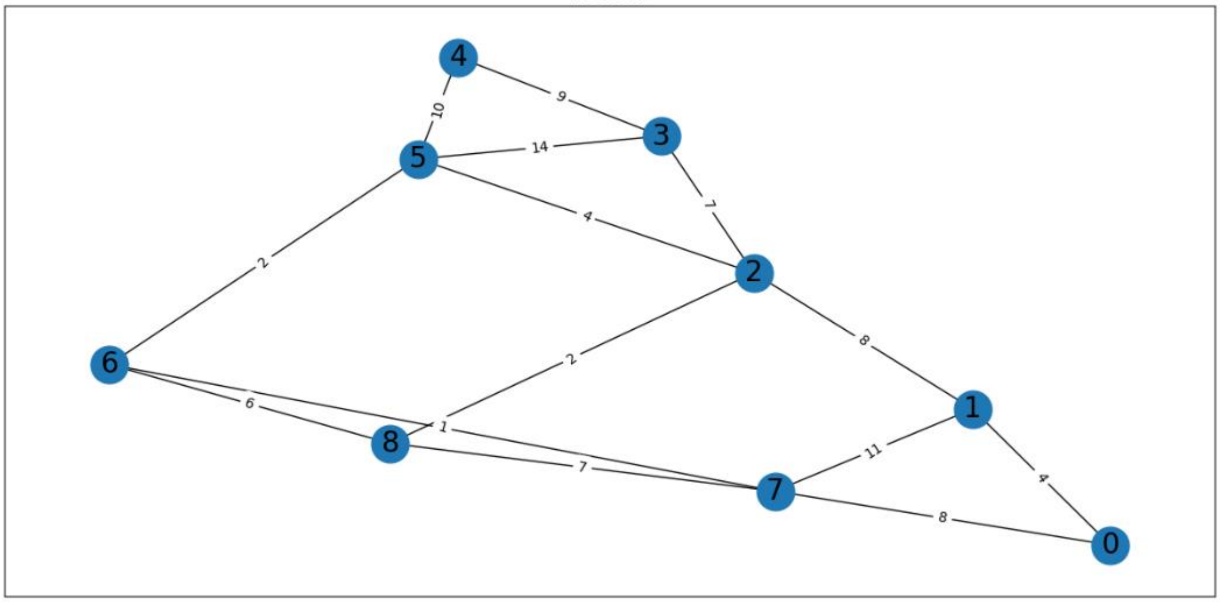


The corresponding minimum spanning tree.

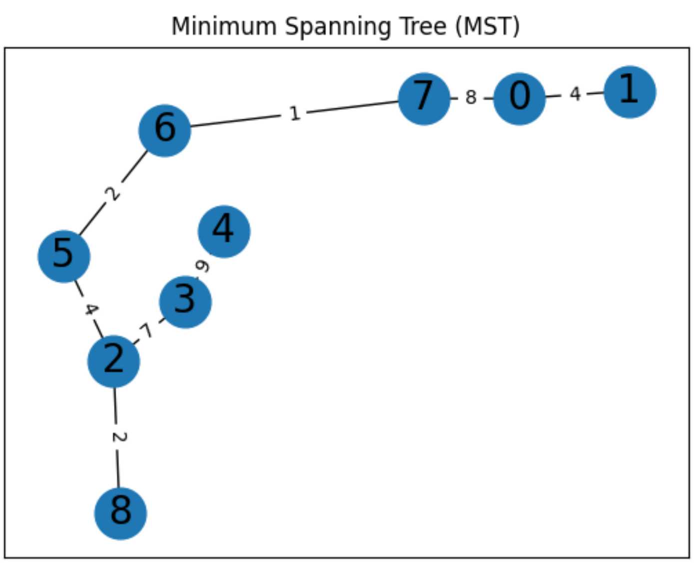
Minimum Spanning Tree (MST)



Example 2:
The network given as input.



The corresponding minimum spanning tree.



CONCLUSION:

Prim's algorithm:

The time complexity of Prim's algorithm is O(E + N*logN) where N is the number of routers and E is the number of links and the space complexity is O(N + E).
This time and space complexities are achieved as we are using a adjacency list to store the network and a minimum priority queue to store the links that are to be considered next while running the algorithm.

Kruskal's algorithm:

The time complexity of Kruskal's algorithm is nearly equal to O(E*logN) and space complexity is O(E) +O(N) + O(N) which can be approximated to O(E).
Here we are using an array of Python objects to store the links of the network and a Disjoint Set data structure to find the MST.

Comparison of the two algorithms –

	Kruskal	**Prim**
Multiple MSTs	Offers a good control over the resulting MST	Controlling the MST might be a little harder
Implementation	Easier to implement	Harder to implement
Requirements	Disjoint set	Priority queue
Time Complexity	$O(E \cdot log(V))$	$O(E + V \cdot log(V))$

REFERENCES:

Prim's Algorithm
https://en.wikipedia.org/wiki/Prim%27s_algorithm

Kruskal's Algorithm
https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Prim's v/s Kruskal's article
https://www.baeldung.com/cs/kruskals-vs-prims-algorithm#:~:text=The%20advantage%20of%20Prim%27s%20algorithm,with%20the%20same%20weight%20occur.

Networkx Python module
https://networkx.org/documentation/stable/tutorial.html

Matplotlib Python module
https://matplotlib.org/

APPENDIX

Data for networks used during demonstration

Example 1:
5 7
0 1 2
0 2 5
1 2 2
1 3 4
2 3 5
2 4 5
3 4 2

Example 2:
9 14
0 1 4
0 7 8
1 2 8
1 7 11
2 3 7
2 8 2
2 5 4
3 4 9
3 5 14
4 5 10
5 6 2
6 7 1
6 8 6
7 8 7