CS 478 PROJECT

M. Muwahid Asim 2019352

M. Saaim Qureshi 2019444

Table of Contents

**Introduction2**

**Implementation4**

**Experimental setup14**

**Results15**

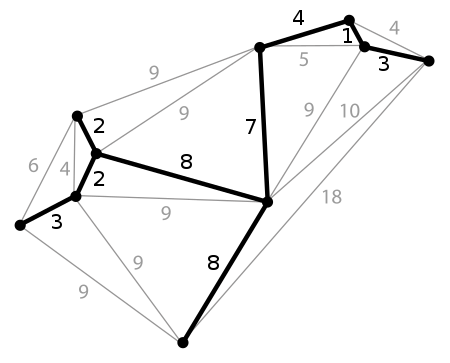
**Discussion21**

**References22**

**INTRODUCTION**

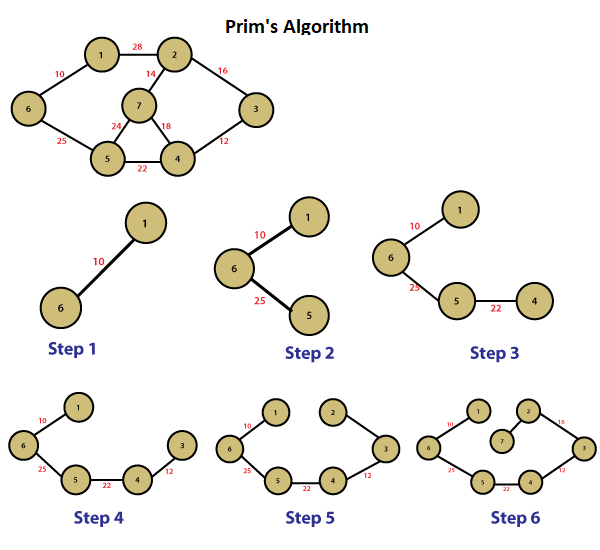
**Minimum Spanning Tree**

A minimum spanning tree 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.

****

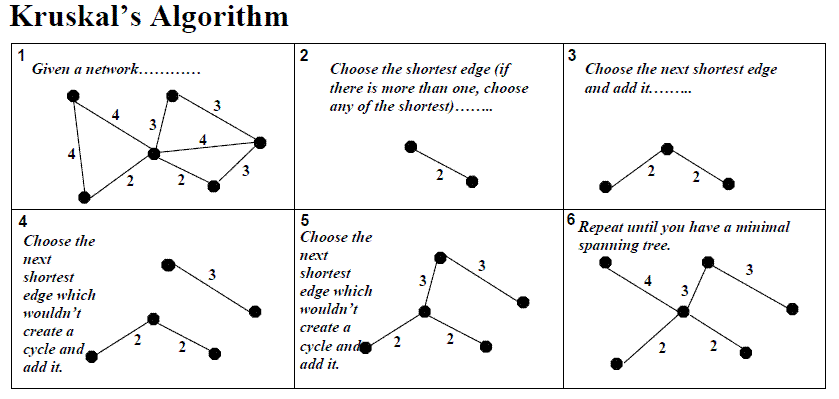
**Prim’s Algorithm**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), Prim's algorithm is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) that finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [weighted](https://en.wikipedia.org/wiki/Weighted_graph) [undirected graph](https://en.wikipedia.org/wiki/Undirected_graph). This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)) that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the [edges](https://en.wikipedia.org/wiki/Graph_theory) 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.



**Kruskal Algorithm**

Kruskal's algorithm finds a [minimum spanning forest](https://en.wikipedia.org/wiki/Minimum_spanning_tree) of an undirected [edge-weighted graph](https://en.wikipedia.org/wiki/Weighted_graph). If the graph is [connected](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)), it finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree). (A minimum spanning tree of a connected graph is a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a tree that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the sum of the [weights](https://en.wikipedia.org/wiki/Weighted_graph) 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](https://en.wikipedia.org/wiki/Connected_component_(graph_theory)).) It is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) in [graph theory](https://en.wikipedia.org/wiki/Graph_theory) as in each step it adds the next lowest-weight edge that will not form a [cycle](https://en.wikipedia.org/wiki/Cycle_(graph_theory)) to the minimum spanning forest.



**Time Complexity**

* The time complexity of Prim's algorithm is O(V^2).
* The time complexity of Kruskal's algorithm is O(E log V).

**IMPLEMENTATION**

**Prim’s Algorithm Implementation**

import sys

from random import randint as r

import numpy as np

import time

class Graph():

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printTree(self, parent):

print("Edge \tWeight")

for i in range(1, self.V):

print(parent[i], "-", i, "\t", self.graph[i][ parent[i] ])

def min\_Key(self, key, mstSet):

min = sys.maxsize

for v in range(self.V):

min\_index = 9

if key[v] < min and mstSet[v] == False:

min = key[v]

min\_index = v

return min\_index

def prim(self):

key = [sys.maxsize] \* self.V

parent = [None] \* self.V

key[0] = 0

mstSet = [False] \* self.V

parent[0] = -1

for cout in range(self.V):

u = self.min\_Key(key, mstSet)

mstSet[u] = True

for v in range(self.V):

if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:

key[v] = self.graph[u][v]

parent[v] = u

# self.printTree(parent)

g\_S\_10 = Graph(10)

g\_D\_10 = Graph(10)

g\_S\_50 = Graph(50)

g\_D\_50 = Graph(50)

g\_S\_100 = Graph(100)

g\_D\_100 = Graph(100)

g\_S\_200 = Graph(200)

g\_D\_200 = Graph(200)

g\_S\_250 = Graph(250)

g\_D\_250 = Graph(250)

A=10

B=50

C=100

D=200

E=250

g\_S\_10.graph=np.zeros((A,A),dtype=int)

g\_S\_10.graph=np.zeros((A,A),dtype=int)

g\_S\_50.graph=np.zeros((B,B),dtype=int)

g\_S\_50.graph=np.zeros((B,B),dtype=int)

g\_S\_100.graph=np.zeros((C,C),dtype=int)

g\_S\_100.graph=np.zeros((C,C),dtype=int)

g\_S\_200.graph=np.zeros((D,D),dtype=int)

g\_S\_200.graph=np.zeros((D,D),dtype=int)

g\_S\_200.graph=np.zeros((E,E),dtype=int)

g\_S\_200.graph=np.zeros((E,E),dtype=int)

for i in range(9):

weight = r(0,9)

g\_S\_10.graph[i][i+1] = weight

g\_S\_10.graph[i+1][i] = weight

g\_D\_10.graph[i][i+1] = weight

g\_D\_10.graph[i+1][i] = weight

num=0

while num < 5:

edge1 = r(0,9)

edge2 = r(0,9)

if (edge1 == edge2 or g\_D\_10.graph[edge1][edge2] != 0):

continue

weight = r(0,9)

g\_D\_10.graph[edge1][edge2] = weight

g\_D\_10.graph[edge2][edge1] = weight

num=num+1

for i in range(49):

weight = r(0,49)

g\_S\_50.graph[i][i+1] = weight

g\_S\_50.graph[i+1][i] = weight

g\_D\_50.graph[i][i+1] = weight

g\_D\_50.graph[i+1][i] = weight

num=0

while num < 25:

edge1 = r(0,49)

edge2 = r(0,49)

if (edge1 == edge2 or g\_D\_50.graph[edge1][edge2] != 0):

continue

weight = r(0,49)

g\_D\_50.graph[edge1][edge2] = weight

g\_D\_50.graph[edge2][edge1] = weight

num=num+1

for i in range(99):

weight = r(0,99)

g\_S\_100.graph[i][i+1] = weight

g\_S\_100.graph[i+1][i] = weight

g\_D\_100.graph[i][i+1] = weight

g\_D\_100.graph[i+1][i] = weight

num=0

while num < 50:

edge1 = r(0,99)

edge2 = r(0,99)

if (edge1 == edge2 or g\_D\_100.graph[edge1][edge2] != 0):

continue

weight = r(0,99)

g\_D\_100.graph[edge1][edge2] = weight

g\_D\_100.graph[edge2][edge1] = weight

num=num+1

for i in range(199):

weight = r(0,199)

g\_S\_200.graph[i][i+1] = weight

g\_S\_200.graph[i+1][i] = weight

g\_D\_200.graph[i][i+1] = weight

g\_D\_200.graph[i+1][i] = weight

num=0

while num < 100:

edge1 = r(0,199)

edge2 = r(0,199)

if (edge1 == edge2 or g\_D\_200.graph[edge1][edge2] != 0):

continue

weight = r(0,199)

g\_D\_200.graph[edge1][edge2] = weight

g\_D\_200.graph[edge2][edge1] = weight

num=num+1

for i in range(249):

weight = r(0,249)

g\_S\_250.graph[i][i+1] = weight

g\_S\_250.graph[i+1][i] = weight

g\_D\_250.graph[i][i+1] = weight

g\_D\_250.graph[i+1][i] = weight

num=0

while num < 130:

edge1 = r(0,249)

edge2 = r(0,249)

if (edge1 == edge2 or g\_D\_250.graph[edge1][edge2] != 0):

continue

weight = r(0,249)

g\_D\_250.graph[edge1][edge2] = weight

g\_D\_250.graph[edge2][edge1] = weight

num=num+1

# Function call

t0=time.time()

g\_S\_10.prim()

t1=time.time()

print("Time for Sparse graph with 10 vertices: ",t1-t0)

g\_D\_10.prim()

t2=time.time()

print("Time for Dense graph with 10 vertices: ",t2-t1)

g\_S\_50.prim()

t3=time.time()

print("Time for Sparse graph with 50 vertices: ",t3-t2)

g\_D\_50.prim()

t4=time.time()

print("Time for Dense graph with 50 vertices: ",t4-t3)

g\_S\_100.prim()

t5=time.time()

print("Time for Sparse graph with 100 vertices: ",t5-t4)

g\_D\_100.prim()

t6=time.time()

print("Time for Dense graph with 100 vertices: ",t6-t5)

g\_S\_200.prim()

t7=time.time()

print("Time for Sparse graph with 200 vertices: ",t7-t6)

g\_D\_200.prim()

t8=time.time()

print("Time for Dense graph with 200 vertices: ",t8-t7)

g\_S\_250.prim()

t9=time.time()

print("Time for Sparse graph with 250 vertices: ",t9-t8)

g\_D\_250.prim()

t10=time.time()

print("Time for Dense graph with 250 vertices: ",t10-t9)

**Kruskal Algorithm Implementation**

# Python program for Kruskal's algorithm to find

# Minimum Spanning Tree of a given connected,

# undirected and weighted graph

# Class to represent a graph

from random import randint as r

import time

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices # No. of vertices

self.graph = []

# to store graph

# function to add an edge to graph

def addEdge(self, u, v, w):

self.graph.append([u, v, w])

# A utility function to find set of an element i

# (truly uses path compression technique)

def find(self, parent, i):

if parent[i] != i:

# Reassignment of node's parent to root node as

# path compression requires

parent[i] = self.find(parent, parent[i])

return parent[i]

# A function that does union of two sets of x and y

# (uses union by rank)

def union(self, parent, rank, x, y):

# Attach smaller rank tree under root of

# high rank tree (Union by Rank)

if rank[x] < rank[y]:

parent[x] = y

elif rank[x] > rank[y]:

parent[y] = x

# If ranks are same, then make one as root

# and increment its rank by one

else:

parent[y] = x

rank[x] += 1

# The main function to construct MST using Kruskal's

# algorithm

def KruskalMST(self):

result = [] # This will store the resultant MST

# An index variable, used for sorted edges

i = 0

# An index variable, used for result[]

e = 0

# Step 1: Sort all the edges in

# non-decreasing order of their

# weight. If we are not allowed to change the

# given graph, we can create a copy of graph

self.graph = sorted(self.graph,

key=lambda item: item[2])

parent = []

rank = []

# Create V subsets with single elements

for node in range(self.V):

parent.append(node)

rank.append(0)

# Number of edges to be taken is less than to V-1

while e < self.V - 1:

# Step 2: Pick the smallest edge and increment

# the index for next iteration

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

# If including this edge doesn't

# cause cycle, then include it in result

# and increment the index of result

# for next edge

if x != y:

e = e + 1

result.append([u, v, w])

self.union(parent, rank, x, y)

# Else discard the edge

minimumCost = 0

print("Edges in the constructed MST")

for u, v, weight in result:

minimumCost += weight

# print("%d -- %d == %d" % (u, v, weight))

print("Minimum Spanning Tree", minimumCost)

# Driver's code

if \_\_name\_\_ == '\_\_main\_\_':

g\_S\_10 = Graph(10)

g\_D\_10 = Graph(10)

g\_S\_50 = Graph(50)

g\_D\_50 = Graph(50)

g\_S\_100 = Graph(100)

g\_D\_100 = Graph(100)

g\_S\_200 = Graph(200)

g\_D\_200 = Graph(200)

g\_S\_250 = Graph(250)

g\_D\_250 = Graph(250)

for i in range(9):

weight = r(0,9)

g\_S\_10.addEdge(i,i+1,weight)

g\_D\_10.addEdge(i,i+1,weight)

for i in range(5):

edge1 = r(0,9)

edge2 = r(0,9)

weight = r(0,9)

while edge1 == edge2:

edge2 = r(0,9)

g\_D\_10.addEdge(edge1, edge2, weight)

for i in range(49):

weight = r(0,49)

g\_S\_50.addEdge(i,i+1,weight)

g\_D\_50.addEdge(i,i+1,weight)

for i in range(25):

edge1 = r(0,49)

edge2 = r(0,49)

weight = r(0,49)

while edge1 == edge2:

edge2 = r(0,49)

g\_D\_50.addEdge(edge1, edge2, weight)

for i in range(99):

weight = r(0,99)

g\_S\_100.addEdge(i,i+1,weight)

g\_D\_100.addEdge(i,i+1,weight)

for i in range(50):

edge1 = r(0,99)

edge2 = r(0,99)

weight = r(0,99)

while edge1 == edge2:

edge2 = r(0,99)

g\_D\_100.addEdge(edge1, edge2, weight)

for i in range(199):

weight = r(0,199)

g\_S\_200.addEdge(i,i+1,weight)

g\_D\_200.addEdge(i,i+1,weight)

for i in range(100):

edge1 = r(0,199)

edge2 = r(0,199)

weight = r(0,199)

while edge1 == edge2:

edge2 = r(0,199)

g\_D\_200.addEdge(edge1, edge2, weight)

for i in range(249):

weight = r(0,249)

g\_S\_250.addEdge(i,i+1,weight)

g\_D\_250.addEdge(i,i+1,weight)

for i in range(130):

edge1 = r(0,249)

edge2 = r(0,249)

weight = r(0,249)

while edge1 == edge2:

edge2 = r(0,249)

g\_D\_250.addEdge(edge1, edge2, weight)

# Function call

t0=time.time()

g\_S\_10.KruskalMST()

t1=time.time()

print("Time for Sparse graph with 10 vertices: ",t1-t0)

g\_D\_10.KruskalMST()

t2=time.time()

print("Time for Dense graph with 10 vertices: ",t2-t1)

g\_S\_50.KruskalMST()

t3=time.time()

print("Time for Sparse graph with 50 vertices: ",t3-t2)

g\_D\_50.KruskalMST()

t4=time.time()

print("Time for Dense graph with 50 vertices: ",t4-t3)

g\_S\_100.KruskalMST()

t5=time.time()

print("Time for Sparse graph with 100 vertices: ",t5-t4)

g\_D\_100.KruskalMST()

t6=time.time()

print("Time for Dense graph with 100 vertices: ",t6-t5)

g\_S\_200.KruskalMST()

t7=time.time()

print("Time for Sparse graph with 200 vertices: ",t7-t6)

g\_D\_200.KruskalMST()

t8=time.time()

print("Time for Dense graph with 200 vertices: ",t8-t7)

g\_S\_250.KruskalMST()

t9=time.time()

print("Time for Sparse graph with 250 vertices: ",t9-t8)

g\_D\_250.KruskalMST()

t10=time.time()

print("Time for Dense graph with 250 vertices: ",t10-t9)

**EXPERIMENTAL SETUP**

**Approach**

* 5 dense and 5 sparse graphs with 10, 50, 100, 200, 250 vertices respectively were taken.
* Random weights assigned in the graph for dense graph random vertices were generated for example for a graph of 250 vertices first 249 vertices were created to make a sparse graph and then another 130 vertices were created to create cycles and a dense graph

**Language**

* Python is used as the programming language for the implementation

**Platform**

* Visual Studio Code is used as the platform to run the code

**System Specification**

* Processor: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 Mhz, 6 Core(s), 12 Logical Processor(s)
* System Type: x64-based PC
* Installed Physical Memory (RAM): 8.00 GB

**RESULTS**

* 5 samples were for each code

**Prim’s Algorithm**

* **Iteration 1**

Time for Sparse graph with 10 vertices: 0.0

Time for Dense graph with 10 vertices: 0.001001596450805664

Time for Sparse graph with 50 vertices: 0.0009562969207763672

Time for Dense graph with 50 vertices: 0.0003666877746582031

Time for Sparse graph with 100 vertices: 0.0035271644592285156

Time for Dense graph with 100 vertices: 0.0015232563018798828

Time for Sparse graph with 200 vertices: 0.0167849063873291

Time for Dense graph with 200 vertices: 0.0043735504150390625

Time for Sparse graph with 250 vertices: 0.008565902709960938

Time for Dense graph with 250 vertices: 0.006738901138305664

* **Iteration 2**

Time for Sparse graph with 10 vertices: 0.0

Time for Dense graph with 10 vertices: 0.0009341239929199219

Time for Sparse graph with 50 vertices: 0.0010406970977783203

Time for Dense graph with 50 vertices: 0.0

Time for Sparse graph with 100 vertices: 0.003985881805419922

Time for Dense graph with 100 vertices: 0.0009601116180419922

Time for Sparse graph with 200 vertices: 0.015224695205688477

Time for Dense graph with 200 vertices: 0.005986928939819336

Time for Sparse graph with 250 vertices: 0.007077693939208984

Time for Dense graph with 250 vertices: 0.007086992263793945

* **Iteration 3**

Time for Sparse graph with 10 vertices: 0.0

Time for Dense graph with 10 vertices: 0.0

Time for Sparse graph with 50 vertices: 0.001657247543334961

Time for Dense graph with 50 vertices: 0.0

Time for Sparse graph with 100 vertices: 0.003951311111450195

Time for Dense graph with 100 vertices: 0.0009982585906982422

Time for Sparse graph with 200 vertices: 0.015784740447998047

Time for Dense graph with 200 vertices: 0.004949092864990234

Time for Sparse graph with 250 vertices: 0.0071184635162353516

Time for Dense graph with 250 vertices: 0.007668256759643555

* **Iteration 4**

Time for Sparse graph with 10 vertices: 0.0

Time for Dense graph with 10 vertices: 0.0007424354553222656

Time for Sparse graph with 50 vertices: 0.0010004043579101562

Time for Dense graph with 50 vertices: 0.0

Time for Sparse graph with 100 vertices: 0.004024028778076172

Time for Dense graph with 100 vertices: 0.00231170654296875

Time for Sparse graph with 200 vertices: 0.015573978424072266

Time for Dense graph with 200 vertices: 0.003534078598022461

Time for Sparse graph with 250 vertices: 0.00848698616027832

Time for Dense graph with 250 vertices: 0.007058382034301758

* **Iteration 5**

Time for Sparse graph with 10 vertices: 0.0

Time for Dense graph with 10 vertices: 0.0

Time for Sparse graph with 50 vertices: 0.001453399658203125

Time for Dense graph with 50 vertices: 0.0

Time for Sparse graph with 100 vertices: 0.00401616096496582

Time for Dense graph with 100 vertices: 0.0019690990447998047

Time for Sparse graph with 200 vertices: 0.013189315795898438

Time for Dense graph with 200 vertices: 0.006292581558227539

Time for Sparse graph with 250 vertices: 0.006487846374511719

Time for Dense graph with 250 vertices: 0.007272243499755859

**Kruskal Algorithm**

* **Iteration 1**

Minimum Spanning Tree 31

Time for Sparse graph with 10 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 22

Time for Dense graph with 10 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 1123

Time for Sparse graph with 50 vertices: 0.0009617805480957031

Edges in the constructed MST

Minimum Spanning Tree 713

Time for Dense graph with 50 vertices: 0.0007359981536865234

Edges in the constructed MST

Minimum Spanning Tree 4548

Time for Sparse graph with 100 vertices: 0.00046443939208984375

Edges in the constructed MST

Minimum Spanning Tree 3227

Time for Dense graph with 100 vertices: 0.0004584789276123047

Edges in the constructed MST

Minimum Spanning Tree 20233

Time for Sparse graph with 200 vertices: 0.00010156631469726562

Edges in the constructed MST

Minimum Spanning Tree 14917

Time for Dense graph with 200 vertices: 0.00046753883361816406

Edges in the constructed MST

Minimum Spanning Tree 30029

Time for Sparse graph with 250 vertices: 0.001001119613647461

Edges in the constructed MST

Minimum Spanning Tree 20872

Time for Dense graph with 250 vertices: 0.0009963512420654297

* **Iteration 2**

Minimum Spanning Tree 41

Time for Sparse graph with 10 vertices: 0.0011548995971679688

Edges in the constructed MST

Minimum Spanning Tree 23

Time for Dense graph with 10 vertices: 0.0005049705505371094

Edges in the constructed MST

Minimum Spanning Tree 1062

Time for Sparse graph with 50 vertices: 0.0010006427764892578

Edges in the constructed MST

Minimum Spanning Tree 741

Time for Dense graph with 50 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 4749

Time for Sparse graph with 100 vertices: 0.0009963512420654297

Edges in the constructed MST

Minimum Spanning Tree 3165

Time for Dense graph with 100 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 19721

Time for Sparse graph with 200 vertices: 0.0009984970092773438

Edges in the constructed MST

Minimum Spanning Tree 13782

Time for Dense graph with 200 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 32299

Time for Sparse graph with 250 vertices: 0.000997781753540039

Edges in the constructed MST

Minimum Spanning Tree 22841

Time for Dense graph with 250 vertices: 0.0009970664978027344

* **Iteration 3**

Minimum Spanning Tree 33

Time for Sparse graph with 10 vertices: 0.000997781753540039

Edges in the constructed MST

Minimum Spanning Tree 20

Time for Dense graph with 10 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 1275

Time for Sparse graph with 50 vertices: 0.0009987354278564453

Edges in the constructed MST

Minimum Spanning Tree 859

Time for Dense graph with 50 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 4724

Time for Sparse graph with 100 vertices: 0.0009980201721191406

Edges in the constructed MST

Minimum Spanning Tree 3090

Time for Dense graph with 100 vertices: 0.001003265380859375

Edges in the constructed MST

Minimum Spanning Tree 19663

Time for Sparse graph with 200 vertices: 0.001987457275390625

Edges in the constructed MST

Minimum Spanning Tree 13825

Time for Dense graph with 200 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 30803

Time for Sparse graph with 250 vertices: 0.0009980201721191406

Edges in the constructed MST

Minimum Spanning Tree 21912

Time for Dense graph with 250 vertices: 0.0009965896606445312

* **Iteration 4**

Minimum Spanning Tree 32

Time for Sparse graph with 10 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 24

Time for Dense graph with 10 vertices: 0.0009980201721191406

Edges in the constructed MST

Minimum Spanning Tree 1412

Time for Sparse graph with 50 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 1009

Time for Dense graph with 50 vertices: 0.0009984970092773438

Edges in the constructed MST

Minimum Spanning Tree 4966

Time for Sparse graph with 100 vertices: 0.000997304916381836

Edges in the constructed MST

Minimum Spanning Tree 3500

Time for Dense graph with 100 vertices: 0.0010013580322265625

Edges in the constructed MST

Minimum Spanning Tree 20042

Time for Sparse graph with 200 vertices: 0.0010020732879638672

Edges in the constructed MST

Minimum Spanning Tree 14319

Time for Dense graph with 200 vertices: 0.0009875297546386719

Edges in the constructed MST

Minimum Spanning Tree 30036

Time for Sparse graph with 250 vertices: 0.0009984970092773438

Edges in the constructed MST

Minimum Spanning Tree 21474

Time for Dense graph with 250 vertices: 0.0009963512420654297

* **Iteration 5**

Minimum Spanning Tree 47

Time for Sparse graph with 10 vertices: 0.0011820793151855469

Edges in the constructed MST

Minimum Spanning Tree 40

Time for Dense graph with 10 vertices: 0.0010004043579101562

Edges in the constructed MST

Minimum Spanning Tree 1311

Time for Sparse graph with 50 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 857

Time for Dense graph with 50 vertices: 0.0009968280792236328

Edges in the constructed MST

Minimum Spanning Tree 4551

Time for Sparse graph with 100 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 3311

Time for Dense graph with 100 vertices: 0.000997304916381836

Edges in the constructed MST

Minimum Spanning Tree 20252

Time for Sparse graph with 200 vertices: 0.0

Edges in the constructed MST

Minimum Spanning Tree 13847

Time for Dense graph with 200 vertices: 0.000997304916381836

Edges in the constructed MST

Minimum Spanning Tree 30297

Time for Sparse graph with 250 vertices: 0.0010058879852294922

Edges in the constructed MST

Minimum Spanning Tree 22003

Time for Dense graph with 250 vertices: 0.0

**DISCUSSION**

As we can see in the results above that on average the time taken for Kruskal algorithm is less than time taken for Prim’s Algorithm for both sparse and dense graphs. For smaller number of vertices this difference is quite small and at some instances not noticeable but as the number of vertices increase the time difference between the two algorithm increases. This proves that the time complexity of Prim’s Algorithm is greater than Kruskal algorithm and so Kruskal algorithm is more efficient.

**REFERENCES**

<https://en.wikipedia.org/wiki/Prim's_algorithm>

<https://en.wikipedia.org/wiki/Minimum_spanning_tree>

<https://en.wikipedia.org/wiki/Kruskal's_algorithm>

[Prim's minimum spanning tree algorithm in Python - CodeSpeedy](https://www.codespeedy.com/demonstrate-prims-minimum-spanning-tree-algorithm-in-python/)

[Kruskal’s Minimum Spanning Tree Algorithm | Greedy Algo-2 - GeeksforGeeks](https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/)