**Analysis of Algorithms**

**CSCE – 629 (Spring 2019)**

**Course Project Report**

Submitted By:

**Vansh Narula**

**UIN:** **128001615**

In this course project, I implemented a network routing protocol using the data structures and algorithms we have studied in class. This provided me with an opportunity to translate my theoretical understanding into a real-world practical computer program. Translating algorithmic ideas at a “higher level” of abstraction into real implementations in a programming language is not always trivial. The implementations forced me to work on more details of the algorithms, which indeed lead to much better understanding.

**1. Graph Generation:**

* We had to implement two graphs with 5000 vertices each– Sparse Graph with average Vertex degree of about 6 and a Dense Graph with an average vertex degree of 1000 (20%).
* I have created two classes, Node class and Edge Class. Node class has attribute as ‘node name’ and ‘adjacent list of edges’. Edge class has attribute as ‘source node’, ‘target node’ and ‘edge weight’. Edge weight is randomly generated between 1 – 9999.
* The Connection is done randomly. To implement this, I have randomly generated a number between 1 and 4999 for every possible neighbor vertex of a source vertex and if it less than 6 (for sparse graph) or 1000 (for dense graph), I have added an edge between the two vertices.
* To make sure that the generated graph is correct and follows the desired properties. I have defined a method to verify the graph. Another method outputs the average number of connections per node.

class Node:

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

self.name = name

self.adjacent\_list\_of\_edges = []

class Edge:

def \_\_init\_\_(self,source, target, weight):

self.source = source

self.target = target

self.weight = weight

import random

def generate\_graph(num\_nodes, probability):

graph = []

n\_edges = 0

for i in range(num\_nodes):

graph.append(Node(str(i)))

#making sure that graph is connected by making a cycle

for i in range(num\_nodes-1):

initial\_node = graph[i]

target\_node = graph[i+1]

edge\_weight = random.randint(1,10000)

initial\_node.adjacent\_list\_of\_edges.append(Edge(initial\_node,target\_node, edge\_weight))

target\_node.adjacent\_list\_of\_edges.append(Edge(target\_node, initial\_node, edge\_weight))

n\_edges+=1

edge\_weight = random.randint(1,10000)

graph[num\_nodes-1].adjacent\_list\_of\_edges.append(Edge(graph[num\_nodes-1], graph[0], edge\_weight))

graph[0].adjacent\_list\_of\_edges.append(Edge(graph[0], graph[num\_nodes-1], edge\_weight))

n\_edges+=1

#randomly adding other nodes in the graph

for i in range(num\_nodes):

initial\_node = graph[i]

for j in range(i+1,num\_nodes):

if(random.randint(0,num\_nodes) < (probability-1/num\_nodes) \* num\_nodes):

target\_node = graph[j]

edge\_weight = random.randint(1,10000)

initial\_node.adjacent\_list\_of\_edges.append(Edge(initial\_node, target\_node, edge\_weight))

target\_node.adjacent\_list\_of\_edges.append(Edge(target\_node, initial\_node, edge\_weight))

n\_edges+=1

return graph, n\_edges

Average number of edges per node for dense graph is 1000.5832

Average number of edges per node for sparse graph is

5.9992

**2. Heap Structure:**

I have implemented a heap Class following the instructions provided for heap. My heap is implemented as follows:

* The vertices of a graph are named by integers 0, 1,. . ., 4999;
* The heap is given by an array H[5000], where each element H[i] gives the name of a vertex in the graph. It is named as ‘heap’.
* The vertex “values” are given in another array D[5000]. Thus, to find the value of a vertex H[i] in the heap, we can use D[H[i]].
* In addition to above arrays, I have also kept an array node\_to\_heap\_index which stores the index of a node in the heap.

import math

import sys

class heap:

def \_\_init\_\_(self, max\_size):

self.max\_size = max\_size

self.heap = [-1] \* self.max\_size

self.D = [-1]\* self.max\_size

self.node\_to\_heap\_index = [-1] \* self.max\_size

self.size = 0

def Parent(self, i):

if i%2 !=0:

return math.floor((i-1)/2)

else :

return math.floor((i-2)/2)

def Left\_Child(self, i):

return (2 \* i + 1) if (2\* i + 1)< self.size else -1

def Right\_Child(self, i):

return (2 \* i + 2) if (2\* i + 2)< self.size else -1

def Swap(self,i,k):

self.heap[k],self.heap[i] = self.heap[i], self.heap[k]

self.node\_to\_heap\_index[self.heap[k]], self.node\_to\_heap\_index[self.heap[i]] = self.node\_to\_heap\_index[self.heap[i]], self.node\_to\_heap\_index[self.heap[k]]

def Max(self):

return self.heap[0]

def Fix\_Heap(self,i):

L = self.Left\_Child(i)

R = self.Right\_Child(i)

maximum = i

if L != -1:

if L<= self.size-1 and self.D[self.heap[i]]< self.D[self.heap[L]]:

maximum = L

else:

maximum = i

if R!= -1:

if R<= self.size-1 and self.D[self.heap[maximum]]< self.D[self.heap[R]]:

maximum = R

if maximum != i:

self.Swap(i,maximum)

self.Fix\_Heap(maximum)

def Extract\_Max(self):

if self.size < 1:

print('error')

max\_element = self.heap[0], self.D[self.heap[0]]

self.Swap(0, self.size-1)

self.size= self.size - 1

self.Fix\_Heap(0)

return max\_element

def Insert(self, a, BW):

if self.size>= self.max\_size:

print('error is here')

self.heap[self.size] = int(a.name)

self.D[int(a.name)] = BW

i = self.size

self.size = self.size+1

self.node\_to\_heap\_index[int(a.name)] = i

while(i>0 and (self.D[self.heap[self.Parent(i)]] < self.D[self.heap[i]])):

self.Swap(i,self.Parent(i))

i = self.Parent(i)

def reset\_node(self, node\_number, value):

index = self.node\_to\_heap\_index[node\_number]

if(index == -1):

raise IndexError('Node not in Heap')

self.D[self.heap[index]] = value

parent = self.Parent(index)

while(self.D[self.heap[parent]]< self.D[self.heap[index]]):

self.Swap(index, parent)

index = parent

if(parent == 0):

break

parent = parent = self.Parent(index)

self.Fix\_Heap(index)

def Delete(self, i):

if i> self.size:

print('error in there')

self.D[self.heap[i]] = sys.maxsize

while(i>0 and (self.D[self.heap[self.Parent(i)]] < self.D[self.heap[i]])):

self.Swap(i, self.Parent(i))

i = self.Parent(i)

self.heap[0] = self.heap[self.size-1]

self.node\_to\_heap\_index[self.heap[0]] = 0

self.heap[self.size-1] = -1

self.size= self.size - 1

self.Fix\_Heap(0)

def print\_heap(self, node = 0, tab = ''):

if(node == -1):

return

print(tab,str(self.D[self.heap[node]]))

self.print\_heap(self.Left\_Child(node), tab + '\t')

self.print\_heap(self.Right\_Child(node), tab + '\t')

**3.Routing Algorithms:**

**General Algorithms:**

1) **Max Bandwidth path using Dijkstra’s algorithm:**

a) for each vertex v = 1 to n do

status[v] = unseen

b) status[s] = intree

d[s] = 0

dad[s] = -1

c) for each edge[s,v] do

status[v] = fringe

dad[v] = s

d[v] = wt(s,v)

d) while there are fringes do

let v be the fringe with max(d[v])

// For array it goes through all the d[v]

//To find out the max for heap it does extract\_max()

status[v] = intree

for each edge[v,t] do

if status[t] == unseen

then status[t] = fringe

dad[t] = v

d[t] = min(d[v], wt(v,t))

else if status[t] == fringe and d[t] < min(d[v], wt(v,t))

then d[t] = min(d[v], wt(v,t))

dad[t] = v

2) **Max Bandwidth path using Kruskal’s Algorithm:**

a) Sort all edges in decreasing order

b) T = φ

c) for each edge ei = [vi, wi] do

r1 = find(vi)

r2 = find(wi)

if r1 != r2

then T = T + ei

Union(r1, r2)

d) return T

3) **Union-Find**

Union(s1, s2):

if rank[s1] > rank[s2]

then dad[s2] = s1

rank[s1] += rank[s2]

else

then dad[s1] = s2

rank[s2] += rank[s1]

Find(i):

a) s = queue()

b) while (i != dad[i]) do

s.push(i)

i = dad[i]

c) while (s not empty) do

dad[s.pop()] = i

d) return i

**Time Complexity:**

**A**. **MaxBandwidthPath using Dijkstra’s Algorithm(array implementation)** : **O(n2+m)**

The inner loop to find max runs n times and the outer loop can also run

maximum of (n-1) times. Hence, O(n2). The inner for loop runs for total of m times

in whole program. Hence, O(m). Thus overall time complexity is O(n2+m)

**B**. **MaxBandwidthPath using Dijkstra’s Algorithm (heap implementation) :** **O(mlogn + nlogn)**

The inner for loop in whole program runs for total number of edges, m, and the

insert in the heap takes logn times. Hence, O(mlogn). The outer loop can run

maximum of (n-1) times and the delete in heap takes O(nlogn). Hence O(nlogn).

Thus overall time complexity comes to O(nlogn + mlogn)

**C. MaxBandwidthPath using Kruskal Algorithm :** **O(mlog\*n + mlogm)**

Sorting m edges takes time O(mlogm) and m find operations with path

compression in the for loop takes O(mlog\*n) time. Hence, O(mlogm + mlog\*n)

**4.Implementation:**

1. **Dijkstra’s without heap:**

* This follows the normal Dijkstra’s implementation where the largest fringe is chosen as the largest fringe element in the array.
* This requires traversing through entire array each time.
* Function max\_BW\_fringe calculates the largest fringe and return its index.
* ‘BW’, ‘status’ and ‘dad’ array stores the Bandwidth, status ( 0 : Unseen, 1: fringe, 2: intree) and Dad of a particular vertex indexed using its number.

def apply\_Without\_Heap(self):

fringes\_count = 0

BW = [-1] \* len(self.G)

status = [-1] \* len(self.G)

dad = [-1] \* len(self.G

for node in self.G:

status[int(node.name)] = 0 # 0 --> Unseen

status[int(self.S.name)] = 2 # 2 --> Intree

BW[int(self.S.name)] = float('inf')

for edge in self.G[int(self.S.name)].adjacent\_list\_of\_edges:

BW[int(edge.target.name)] = int(edge.weight)

status[int(edge.target.name)] = 1 # 1 --> Fringe

dad[int(edge.target.name)] = int(self.S.name)

fringes\_count += 1

maximum\_bw = sys.maxsize

while(fringes\_count > 0):

current\_max = self.max\_BW\_fringe(self.G, status, BW, len(self.G))

status[current\_max] = 2

if(current\_max == int(self.T.name)):

maximum\_bw = BW[current\_max]

return maximum\_bw, dad

fringes\_count -= 1

for edge in self.G[current\_max].adjacent\_list\_of\_edges:

w = int(edge.target.name)

if(status[w] == 0):

status[w] = 1

fringes\_count += 1

dad[w] = current\_max

BW[w] = min(BW[current\_max], int(edge.weight))

elif(status[w] == 1 and BW[w] < min(BW[current\_max], int(edge.weight))):

dad[w] = current\_max

BW[w] = min(BW[current\_max], int(edge.weight))

def max\_BW\_fringe(self, graph, status\_array, bandwidth\_array, N):

index = -1

max\_fringe\_bw = -sys.maxsize - 1

for i in range(0, N):

if((status\_array[i] == 1) and (bandwidth\_array[i] >= max\_fringe\_bw)):

index = i

max\_fringe\_bw = bandwidth\_array[i]

return index

1. **Dijkstra’s with heap:**

* This uses the heap class defined in step2.
* The largest fringe is calculated through heap operation – extract\_max.
* Whenever a new node is identified as fringe, it is inserted in the heap.
* Whenever a new BW value for a fringe is generated, older fringe is deleted, and new fringe is inserted in heap using the reset\_node command.

def apply\_With\_Heap(self):

BW = [-1] \* len(self.G)

status = [-1] \* len(self.G)

dad = [-1] \* len(self.G)

self.max\_heap = heap(len(self.G))

for node in self.G:

status[int(node.name)] = 0 # 0 --> Unseen

status[int(self.S.name)] = 2 # 2 --> Intree

BW[int(self.S.name)] = float('inf')

for edge in self.G[int(self.S.name)].adjacent\_list\_of\_edges:

status[int(edge.target.name)] = 1 # 1 --> Fringe

BW[int(edge.target.name)] = int(edge.weight)

self.max\_heap.Insert(edge.target, BW[int(edge.target.name)])

dad[int(edge.target.name)] = int(self.S.name)

while (self.max\_heap.size != 0):

max\_element = self.max\_heap.Extract\_Max()

current\_max = max\_element[0]

status[current\_max] = 2

if current\_max == int(self.T.name):

return dad, BW[current\_max]

for edge in self.G[current\_max].adjacent\_list\_of\_edges:

w = int(edge.target.name)

if status[w] == 0:

status[w] = 1

BW[w] = min(BW[current\_max], int(edge.weight))

self.max\_heap.Insert(edge.target, BW[w])

dad[w] = current\_max

elif (status[w] == 1 and BW[w] < min(BW[current\_max], int(edge.weight)) ):

dad[w] = current\_max

BW[w] = min(BW[current\_max], int(edge.weight))

self.max\_heap.reset\_node(w, BW[w])

1. **Kruskal’s using Heap Sort:**

* To implement heap sort for getting the edge with highest weight each time. I have implemented another heap specific to edges. It contains 3 arrays: ‘S’, ‘T’ and ‘D’. S array contains the source of edge, T array contains the target of edge and D contains the value of edge weight. It implements similar procedure like the Dijkstra’s heap to implement heap sort.
* This new heap gives out an iterable number of extract\_max operations (using Yield Operation). Thus, allowing us to receive all edges in non-increasing order.
* After that we do number of union-find operations following the below mentioned implementations.

from queue import Queue

import sys

class Kruskal:

def \_\_init\_\_(self, Graph, Source, target, n\_edges):

self.G = Graph

self.S = Source

self.T = target

self.num\_of\_edges = n\_edges

def sort\_and\_iterate\_edges(self):

edge\_heap = EdgeHeap(self.num\_of\_edges)

for i in range(0, len(self.G)):

for edge in self.G[i].adjacent\_list\_of\_edges:

if(int(edge.source.name) < int(edge.target.name)):

edge\_heap.Insert(int(edge.source.name), int(edge.target.name), int(edge.weight))

for \_ in range(0, self.num\_of\_edges):

yield edge\_heap.extract\_max()

def get\_maximum\_spanning\_tree(self):

N= len(self.G)

maximum\_spanning\_tree = []

for i in range(0, N):

maximum\_spanning\_tree.append(Node(i))

dad\_array = [-1] \* N

rank\_array = [0] \* N

for edge\_source, edge\_target, edge\_weight in self.sort\_and\_iterate\_edges():

r1 = self.find(edge\_source, dad\_array)

r2 = self.find(edge\_target, dad\_array)

if(r1 != r2):

maximum\_spanning\_tree[edge\_source].adjacent\_list\_of\_edges.append(Edge(maximum\_spanning\_tree[edge\_source], maximum\_spanning\_tree[edge\_target], int(edge\_weight)))

maximum\_spanning\_tree[edge\_target].adjacent\_list\_of\_edges.append(Edge(maximum\_spanning\_tree[edge\_target], maximum\_spanning\_tree[edge\_source], int(edge\_weight)))

self.union(r1, r2, rank\_array, dad\_array)

return maximum\_spanning\_tree

def union(self, rank1, rank2, rank\_array, dad\_array):

if(rank\_array[rank1] > rank\_array[rank2]):

dad\_array[rank2] = rank1

elif (rank\_array[rank1] < rank\_array[rank2]):

dad\_array[rank1] = rank2

else:

dad\_array[rank1] = rank2

rank\_array[rank2] += 1

def find(self, v, dad\_array):

w = v

q = Queue()

while(dad\_array[w] != -1):

q.put(w)

w = dad\_array[w]

while not q.empty():

dad\_array[q.get()] = w

return w

def get(self,maximum\_spanning\_tree, i, j):

for edge in maximum\_spanning\_tree[i].adjacent\_list\_of\_edges:

if(int(edge.target.name) == j):

return int(edge.weight)

return -1

def apply\_dfs(self, graph, node\_number, color\_array, path\_array, target):

if (node\_number == target):

return True

found = False

color\_array[node\_number] = 2

for edge in graph[node\_number].adjacent\_list\_of\_edges:

if(color\_array[int(edge.target.name)] == 1):

path\_array[int(edge.target.name)] = int(edge.source.name)

found = self.apply\_dfs(graph, int(edge.target.name), color\_array, path\_array, target)

if found:

break

color\_array[node\_number] = 3

return found

def get\_maximum\_bandwidth(self, maximum\_spanning\_tree, source, target, N):

color\_array = [1] \* N # 1 -->White

path\_array = [-1] \* N

self.apply\_dfs(maximum\_spanning\_tree, source, color\_array, path\_array, target)

path = str(target)

k = target

maximum\_bandwith = sys.maxsize

while(k != source):

path = str(path\_array[k]) + "->" + path

maximum\_bandwith= min(maximum\_bandwith, self.get(maximum\_spanning\_tree, k, path\_array[k]))

k = path\_array[k]

return maximum\_bandwith, path

def apply\_Kruskal(self):

maximum\_spanning\_tree = self.get\_maximum\_spanning\_tree()

max\_BW, path = self.get\_maximum\_bandwidth(maximum\_spanning\_tree, self.S, self.T, len(self.G))

return max\_BW, path

**5.Testing:**

* For testing, we must generate 5 different graphs (both sparse and dense). For each graph we must randomly generate 5 different Source Target Pair.
* We then apply all three algorithms on each S-T pair.
* We calculate the time taken to generate the path by each algorithm.
* We then evaluate and compare the results.
* Following testing is done using the under mentioned code.

import time

def apply\_different\_algorithms(graph, n\_times):

for \_ in range(n\_times):

source,target = get\_random\_ST\_pair(graph[0])

MWP = Dijkstra(graph[0], source, target)

MBW = Kruskal(graph[0], int(source.name), int(target.name), graph[1])

start\_with\_heap=time.time()

value = MWP.apply\_With\_Heap()

end\_with\_heap = time.time()

start\_without\_heap = time.time()

value2 = MWP.apply\_Without\_Heap()

end\_without\_heap = time.time()

start\_kruskal = time.time()

value3 = MBW.apply\_Kruskal()

end\_kruskal = time.time()

path = str(target.name)

k = int(target.name)

while(k != int(source.name)):

path = str(value[0][k]) + "->" + path

k = value[0][k]

print(path)

print('max BW with heap=', value[1])

print('time taken with heap=', (end\_with\_heap-start\_with\_heap))

without\_heap\_path = str(target.name)

k1 = int(target.name)

while(k1 != int(source.name)):

without\_heap\_path = str(value2[1][k1]) + "->" + without\_heap\_path

k1 = value2[1][k1]

print(without\_heap\_path)

print('Max BW without heap =', value2[0])

print('time taken without heap=', (end\_without\_heap-start\_without\_heap))

print(value3[1])

print('max BW path using Kruskal=', value3[0])

print('time taken with kruskal=', (end\_kruskal-start\_kruskal))

for i in range(5):

sparse\_graph = generate\_graph(5000, float(6/5000))

print('applying on sparse graph---------------------------------------------------- /n -----------------------------------------')

apply\_different\_algorithms(sparse\_graph,5)

dense\_graph = generate\_graph(5000, 0.20)

print('applying on dense graph---------------------------------------------------- /n -----------------------------------------')

apply\_different\_algorithms(dense\_graph,5)

**6.Results:**

**The following example depicts the procedure using one example. The rest of the results will be depicted using the table provided.**

**Source = 406; Target = 4593 (Sparse Graph 1)**

Path = 406->1277->3171->1888->75->74->3082->3083->124->4725->1556->1358->3617->4304->1875->1876->1350->3001->3000->4196->4197->2225->1364->455->2457->777->778->779->1522->1521->2560->2548->2549->2550->656->1748->1373->4410->4409->4408->4407->1433->4939->4940->4941->2435->1289->1290->3946->3947->3401->3402->1363->4028->1960->1961->3197->337->1513->161->2792->4419->3431->3432->953->4311->4593

**Max BW without heap** = 6832

**time taken without heap**= 1.4002 secs

Path = 406->1277->3171->1888->75->74->3082->3083->124->4725->1556->1358->3617->4304->1875->1876->1350->3001->3000->4196->2454->4346->143->3676->4013->1461->1078->4580->4579->3790->3789->3788->3787->4130->2687->2688->2980->351->350->4938->4939->4940->4941->2435->1289->1290->3946->3947->3401->3402->1363->4028->1960->1961->3197->337->1513->161->2792->4419->3431->3432->953->4311->4593

**max BW with heap**= 6832

**time taken with heap**= 0.1745 secs

Path = 406->1277->3171->1888->75->74->3082->3083->124->4725->1556->1358->3617->4304->1875->1876->1350->3001->3000->4196->4197->2225->1364->455->11->1343->2596->2040->3735->3736->1074->1576->298->297->2367->824->2624->465->2580->1348->993->3675->3674->4908->2464->2365->753->1458->2577->2674->2864->351->350->4938->4939->4940->4941->2435->1289->1290->3946->3947->3401->3402->1363->4028->1960->1961->3197->337->1513->161->2792->4419->3431->3432->953->4311->4593

**max BW path using Kruskal**= 6832

**time taken with kruskal**= 0.8916 secs

**Final Results: -**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sparse Graph** | | | **Dense Graph** | | |
| **Dijkstra’s with Heap (s)** | **Dijkstra’s without Heap (s)** | **Kruskal’s with Heap Sort (s)** | **Dijkstra’s with Heap (s)** | **Dijkstra’s without Heap (s)** | **Kruskal’s with Heap Sort (s)** |
| 0.1745  0.0528  0.09678  0.1286  0.0159  0.1655  0.0030  0.1785  0.1775  0.1336  0.1725  0.1545  0.1276  0.1077  0.1047  0.1186  0.0917  0.1685  0.0977  0.1126  0.1017  0.0678  0.1805  0.0279  0.1156 | 1.4002  0.3879  0.7988  0.8447  0.0249  1.3294  0.0149  1.4770  1.4341  1.0404  1.4780  1.2137  0.9265  0.8561  0.2323  0.8686  0.6343  1.3823  0.6522  1.1012  0.7379  0.4587  1.4421  1.0113  1.1530 | 0.8916  0.8696  0.8935  0.8856  0.8765  0.8297  0.8407  0.8308  0.8317  0.8397  0.8148  0.8178  0.8178  0.8198  0.8178  0.8207  0.8277  0.8218  0.8267  0.8237  0.8258  0.8337  0.8387  0.8267  0.8258 | 1.6914  0.7061  1.6834  3.266  1.4980  2.5791  2.1255  3.1965  3.3061  2.2280  0.0837  0.8498  2.4344  0.3850  0.9096  1.8699  0.4677  3.3121  1.6735  1.6495  1.5240  3.3560  0.6782  3.4198  2.2718 | 2.3587  2.0694  1.4561  4.0900  1.7702  1.6406  2.8942  4.5697  4.7054  3.2543  2.2938  3.5784  2.2590  2.4354  3.6541  2.6877  0.5615  4.6904  3.4258  0.9195  4.6665  4.8161  1.6086  4.8290  3.1765 | 158.8112  159.8894  160.5163  154.6394  154.7042  154.0190  154.4708  154.699  153.6011  154.5047  154.3382  153.8811  154.9146  155.7454  155.0713  154.6155  154.3073  154.5208  155.4981  153.8216  154.9186  153.7528  154.0022  154.4908  154.6923 |

**7.Conclusion:**

* We can see that the Dijkstra’s with heap outperforms the performance of Dijkstra’s without heap and Kruskal’s for both sparse graph as well as dense graph. This is because it sorts using heap sort and it stops as soon as we find the target fringe.
* Kruskal’s performance is good for sparse graph as the maximum spanning tree is also sparse and the number of edges is less. That is why, it is feasible to run kruskal’s in sparse graph. It runs faster than Dijkstra’s without heap on an average. This makes sense as the run time for Kruskal is dependent on sorting which is O(mlogm). In sparse graph, the ‘m’ is comparatively small and thus, the sorting is fast. Where as for dense graph, we can see that the performance of Kruskal degrades a lot. This is because ‘m’ is inherently large for dense graph.
* It does make sense to run Kruskal once to get the maximum spanning tree if you have a static graph and we have a lot of S-T pair. Once we have the maximum spanning tree in hand, we can calculate the Maximum Bandwidth Path between any two vertices using linear time algorithm such as DFS (O(M +E)). But as we know that these algorithms are designed for dynamic graph used in wireless communication systems, it is safe to assume that Kruskal is not feasible for dense dynamic graph.
* The performance of Dijkstra’s algorithm without heap is average for both sparse as well as dense graph.
* **For sparse graph**:

Time taken by all 3 algorithms are as follows:

**Dijkstra’s with heap <<<< Kruskal’s with heap sort <<<< Dijkstra’s without heap (Average case).**

* **For Dense Graph:**

Time taken by all 3 algorithms are as follows:

**Dijkstra’s with heap <<<< Dijkstra’s without heap (Average case) <<<< Kruskal’s with heap sort.**

**8.Future Improvement:**

Merge sort can be used instead of heap sort because Merge sort on arrays has considerably better data cache performance, often outperforming heapsort on modern desktop computers because merge sort frequently accesses contiguous memory locations (good locality of reference); heapsort references are spread throughout the heap.