Analysis of Algorithms for finding the Maximum Bandwidth Path between two nodes in a graph

Shruthi Sampathkumar

November 25, 2019

1 Abstract

Maximum bandwidth problem is an optimization problem which maximizes the minimum edge weight of a path from a source to destination. The formal definition of maximum bandwidth problem is as follows:

In graph theory, the graph bandwidth problem is to label the n vertices of v_i ***of a graph G with distinct integers f(***v_i***) so that the quantity max*** $(|f(v_i)$ ***- f(***v_j***)|:*** v_i***,*** $v_j \in E)$ ***is minimized (E is the edge set of G).***

We can apply the concept of maximum bandwidth in network routing techniques, computation of max flows, etc.

The analysis of performance on the various algorithms for finding the maximum bandwidth path is carried out using time taken by each of the algorithm to find the maximum bandwidth between two nodes in a given graph.

2 Methodology

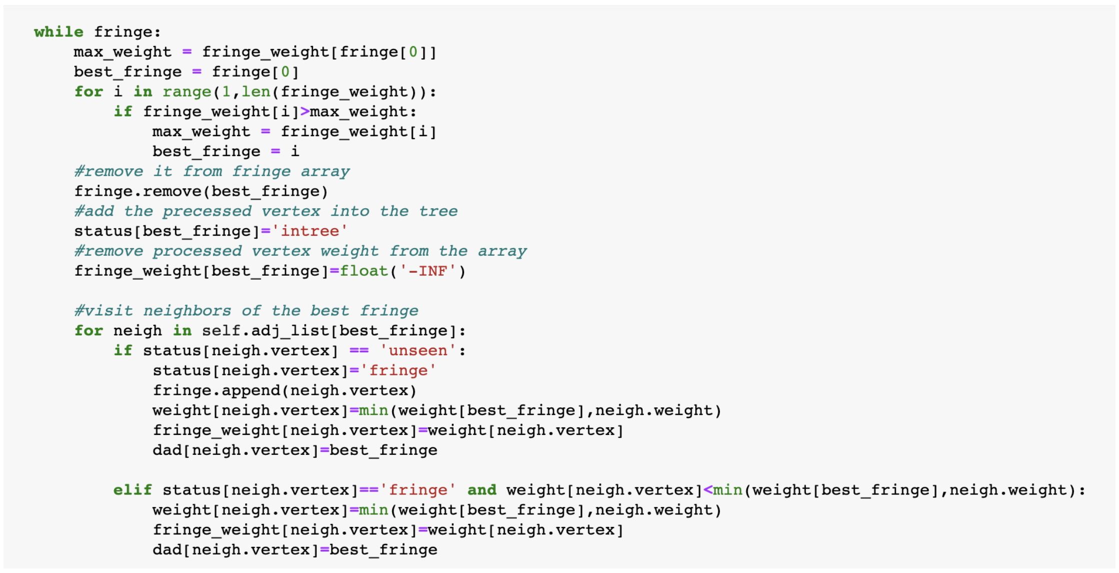
The analysis of performance of the following algorithms to find the maximum bandwidth path are discussed in this report :

- Modified Dijikstra's without using heaps
- Modified Dijikstra's using heaps
- Kruskal's algorithm

2.1 Modified Dijikstra's

Dijikstra's original algorithm to find the shortest path between source and all the other vertices is modified to meet the goal of finding maximum bandwidth path between source and the other vertices.

- The idea is to pick the fringe with the highest bandwidth at every iteration and add it to the tree.
- Following this step, just like in the original algorithm, we visit all the neighbors of the best fringe and update their maximum bandwidth and their parent vertex. For the neighbors which are not yet a fringe, we add them as a fringe and then do the above mentioned updates. For the neighbors, which are already a fringe, we check their current bandwidth with the minimum of best fringe's bandwidth and weight of the edge between best fringe and the respective neighbor. If this



```python
while fringe:
    max_weight = fringe_weight[fringe[0]]
    best_fringe = fringe[0]
    for i in range(1,len(fringe_weight)):
        if fringe_weight[i]>max_weight:
            max_weight = fringe_weight[i]
            best_fringe = i
    #remove it from fringe array
    fringe.remove(best_fringe)
    #add the precessed vertex into the tree
    status[best_fringe]='intree'
    #remove processed vertex weight from the array
    fringe_weight[best_fringe]=float('-INF')

    #visit neighbors of the best fringe
    for neigh in self.adj_list[best_fringe]:
        if status[neigh.vertex] == 'unseen':
            status[neigh.vertex]='fringe'
            fringe.append(neigh.vertex)
            weight[neigh.vertex]=min(weight[best_fringe],neigh.weight)
            fringe_weight[neigh.vertex]=weight[neigh.vertex]
            dad[neigh.vertex]=best_fringe

        elif status[neigh.vertex]=='fringe' and weight[neigh.vertex]<min(weight[best_fringe],neigh.weight):
            weight[neigh.vertex]=min(weight[best_fringe],neigh.weight)
            fringe_weight[neigh.vertex]=weight[neigh.vertex]
            dad[neigh.vertex]=best_fringe
```

Figure 2.1: Dijikstra's without heaps code snippet

minimum is greater than the current bandwidth of the neighbor, then the values such as bandwidth and parent vertex are updated.

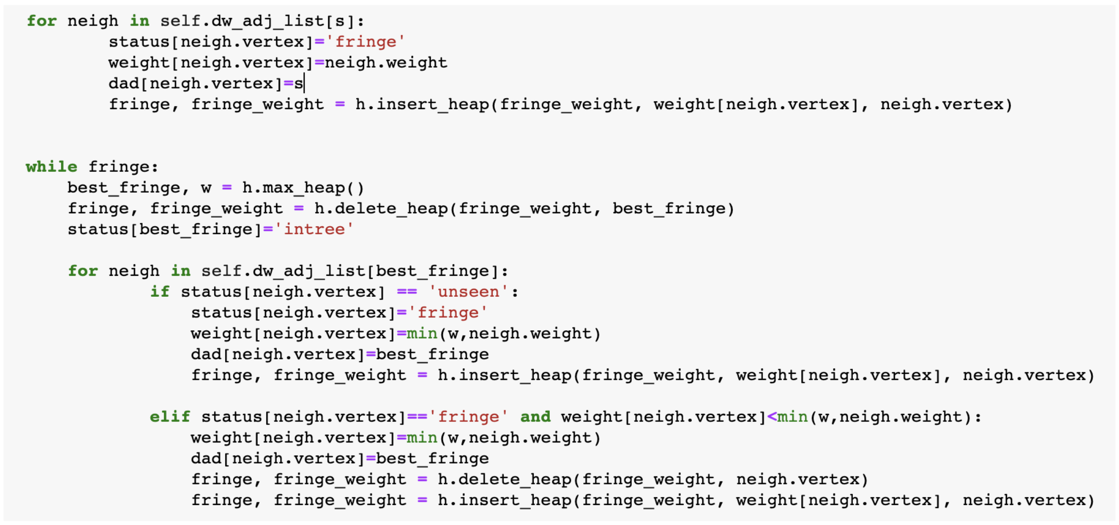
2.1.1 Modified Dijikstra's without using heaps

We can pick the best fringe by various algorithms. The **brute force** algorithm, as shown in Fig.2.1, to find the fringe with the highest bandwidth traverses the graph of fringes and finds the fringe with the highest bandwidth. Thus it's complexity is in the order of number of fringes. This is an inefficient method when we have a large graph with thousands of vertices as for every iteration it has to traverse through all the fringes to find the vertex with the highest bandwidth.

Time Complexity : To find the fringe with the highest bandwidth at every iteration, the algorithm traverses through the entire array of fringes and thus runs in the order of number of vertices in the graph which is given by */V/*. Number of iterations equals the number of vertices. Thus, the running time of Dijikstra's in the brute force approach is in the order of $O(|V|^2)$.

2.1.2 Modified Dijikstra's using heaps

When we use a **max heap** to store the fringes, at every iteration, we can extract the fringe with the highest bandwidth in constant time and remove it from the heap in the order of log of number of fringes as shown in Fig.2.2. Thus, the efficiency of the algorithm greatly increases. Also, in order to update the bandwidth of the existing fringe, we can



```
for neigh in self.dw_adj_list[s]:
        status[neigh.vertex]='fringe'
        weight[neigh.vertex]=neigh.weight
        dad[neigh.vertex]=s
        fringe, fringe_weight = h.insert_heap(fringe_weight, weight[neigh.vertex], neigh.vertex)


while fringe:
    best_fringe, w = h.max_heap()
    fringe, fringe_weight = h.delete_heap(fringe_weight, best_fringe)
    status[best_fringe]='intree'

    for neigh in self.dw_adj_list[best_fringe]:
            if status[neigh.vertex] == 'unseen':
                status[neigh.vertex]='fringe'
                weight[neigh.vertex]=min(w,neigh.weight)
                dad[neigh.vertex]=best_fringe
                fringe, fringe_weight = h.insert_heap(fringe_weight, weight[neigh.vertex], neigh.vertex)

            elif status[neigh.vertex]=='fringe' and weight[neigh.vertex]<min(w,neigh.weight):
                weight[neigh.vertex]=min(w,neigh.weight)
                dad[neigh.vertex]=best_fringe
                fringe, fringe_weight = h.delete_heap(fringe_weight, neigh.vertex)
                fringe, fringe_weight = h.insert_heap(fringe_weight, weight[neigh.vertex], neigh.vertex)
```

Figure 2.2: Dijikstra's with Heap code snippet

remove the vertex from the heap and insert the vertex with the updated weight into the heap.

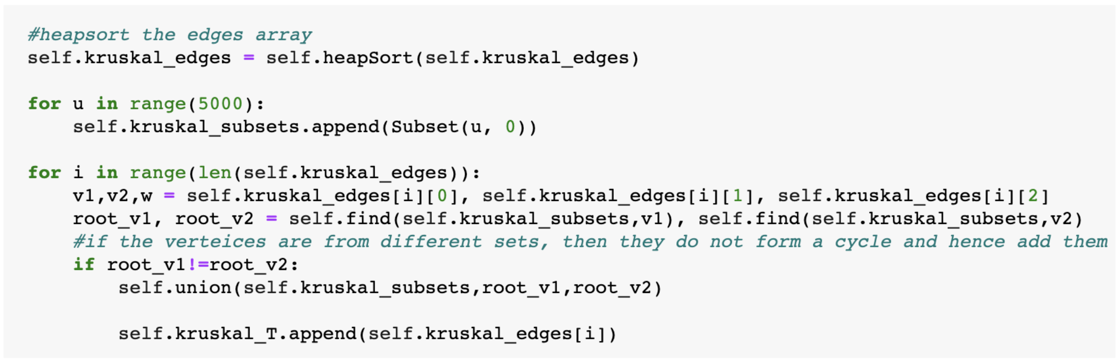
Time Complexity : Since a max heap data structure is maintained to store the fringes, at every iteration, the best fringe is chosen in *O(1)* time. To remove fringes from the heap and to update the fringe weights, it takes *O(log|V|)* time where V is the number of vertices in the graph. The maximum number of iterations possible equals twice the number of edges in the graph which is given by |E|. Thus, with the incorporation of heap structure in Dijikstra's to find the maximum path, we can see that the algorithm runs in ***O(|E|*log |V|)*** time.

2.2 Kruskal's algorithm

The idea is to add the edge with the highest bandwidth at every iteration to the maximum spanning tree. While adding edges in this manner, we check if the edge currently being processed creates a cycle in the maximum spanning tree. If it does, it is not added to the tree and if it doesn't, it is added to the maximum spanning tree. This is shown in Fig.2.3. Since at every iteration we pick the edge with the maximum edge weight, the resulting spanning tree will give the maximum spanning tree of the given graph.

In order to find the edge with the highest bandwidth at every iteration, we use **heap sort** to sort the edges according to their weights.

Using the sorted edges, at every iteration, we pick the maximum weighted edge. We **find the root** of the two vertices which forms the end points of the chosen edge. If



```python
#heapsort the edges array
self.kruskal_edges = self.heapSort(self.kruskal_edges)

for u in range(5000):
    self.kruskal_subsets.append(Subset(u, 0))

for i in range(len(self.kruskal_edges)):
    v1,v2,w = self.kruskal_edges[i][0], self.kruskal_edges[i][1], self.kruskal_edges[i][2]
    root_v1, root_v2 = self.find(self.kruskal_subsets,v1), self.find(self.kruskal_subsets,v2)
    #if the verteices are from different sets, then they do not form a cycle and hence add them
    if root_v1!=root_v2:
        self.union(self.kruskal_subsets,root_v1,root_v2)

        self.kruskal_T.append(self.kruskal_edges[i])
```

Figure 2.3: Kruskal's code snippet

their roots are the same, then we can infer that they would create a cycle when added to the spanning tree. Thus we ignore the edge. If their roots are different, we add them to the tree. The find technique that we use for this purpose is made efficient using **path compression**.

In addition to adding the chosen edge to the tree, we also merge their roots by using the **Union** technique. This is done so that in the following iterations, we can avoid creating a cycle by adding edge that belongs to the same root.

Once the maximum spanning tree is constructed in the discussed manner, we can find the path from the given source to the destination using a depth first search starting at the source.

Time Complexity : Sorting the edge weights takes *O(|E|*log |E|)* time where |E| is the number of edges in the graph. Union-Find algorithm with the incorporation of path compression mechanism takes *O(|E|*log |E|)* time. Finally, after forming the maximum spanning tree, the depth first search to find the path between source and destination takes O(|E|) time and thus the running time of Kruskal's algorithm with the mentioned modifications is in the order of ***O(|E|*log |E|)***.

3 Implementation

In order to implement the algorithms and to test them, there were supporting algorithms included in the project which are as follows:

3.1 Graph Generation:

There were two types of graphs, each of 5000 vertices, that were generated in order to test the algorithms. One of them was a sparse graph which was which had an average degree of 6. The other one was a dense graph in which each vertex was connected to an average of 20% of the total number of vertices in the graph. The result of the graph generation algorithm is the adjacency list for each of the vertices in the graph.

We use a Node object that contains vertex value and edge weight value. For every edge $e_{i,j}$ of weight $w_{i,j}$ that is being generated between two vertices v_i and v_j, we add a node with the vertex number i and edge weight $w_{i,j}$ to the adjacency list of the vertex j and vertex number j and edge weight $w_{i,j}$ to the adjacency list of the vertex i.

We start with connecting the vertices in the graph with a cycle as follows: Vertex 0 is connected to vertex 1, vertex 1 is connected to vertex 2 and so on until vertex 4999 is connected to vertex 0. Thus we make sure that the graph is connected.

We use handshake lemma which is given by the equation in 3.1 to calculate the number of edges required in sparse and dense graph respectively.

$$average\_number\_of\_edges\_undirected\_Graph = (number\_of\_edges*2) \div number\_of\_vertices \tag{3.1}$$

With this information, we calculate the number of edges required in each of the sparse and dense graph with since we know the average number of edges in the respective graphs.

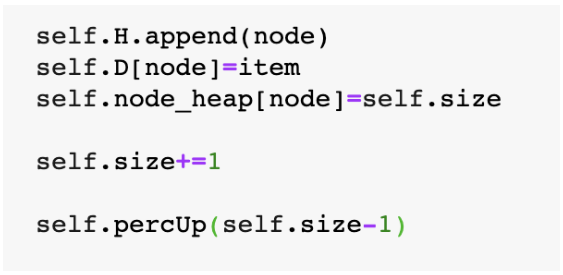
Followed by this, we randomly generate two vertices and a weight which would be the edge weight between the two vertices and include it in the graph if the two vertices are not already connected. This is done by checking the adjacency list of the two vertices. Also, self loops are avoided by making sure that the randomly generated two vertices are not the same.

3.2 Max Heap

In order to make the Dijikstra's modified algorithm to find the maximum bandwidth path more efficient, we incorporate it with heap data structure which reduces the running time greatly.

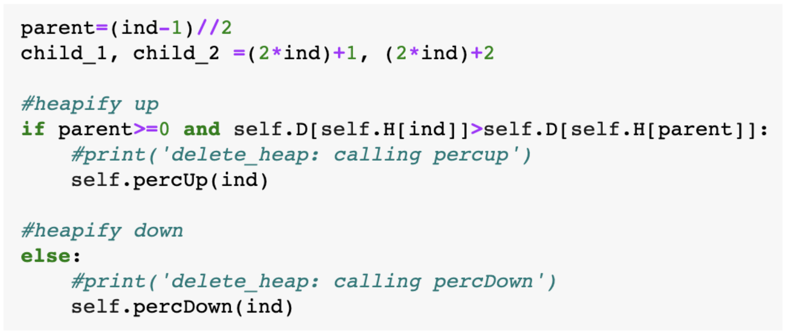
The heap class contains functions such as percolate up and percolate down which are used while inserting a vertex into the heap and deleting a vertex from the heap. Insertion of a vertex occurs when we visit a vertex that has an *unseen* status. Deletion of a vertex happens when we find the vertex with the maximum bandwidth at every iteration and while updating the bandwidth of a vertex which is already a fringe.

For this purpose, in our implementation, we maintain three lists that are represented



```
self.H.append(node)
self.D[node]=item
self.node_heap[node]=self.size

self.size+=1

self.percUp(self.size-1)
```

Figure 3.1: Insertion in Heap code snippet



```
parent=(ind-1)//2
child_1, child_2 =(2*ind)+1, (2*ind)+2

#heapify up
if parent>=0 and self.D[self.H[ind]]>self.D[self.H[parent]]:
    #print('delete_heap: calling percup')
    self.percUp(ind)

#heapify down
else:
    #print('delete_heap: calling percDown')
    self.percDown(ind)
```

Figure 3.2: Deletion in Heap code snippet

by **H, D and node_heap**. They correspond to a heap that stores the vertex numbers, a list that stores the weights of existing fringes and a map to the node index to heap index respectively. The list H, which denoted the heap, expands and contracts as and when we add elements to the heap and delete from it. The order of vertices in H denotes the vertex order in the heap. The list D is used to sort and arrange the vertices in H. The index i in D list contains the weight of the vertex v_i if v_i is a fringe, otherwise it contains negative infinity. The value at index i in node_heap list represents the index of vertex v_i in the heap H if v_i is a fringe, otherwise it contains *None* as it's value at i.

Insertion in heap always appends the new vertex at the end of the heap and percolates up as shown in Fig.3.1. **Deletion** in heap always moves the vertex at the end of the heap to the index where deletion occurs and percolates down or percolates up depending on the heap property violation as shown in Fig.3.2.

3.3 Union & Find

Find technique is tailored with path compression mechanism to increase the efficiency of Kruskal's algorithm. When we find the root of a vertex, we compress it's path and it's ancestors' path from the root vertex and make their immediate parent as the root vertex. Thus, when we try to find any of these vertices the next time, we would be able to find the root vertex faster. In the union method, we merge the roots of the two vertices that forms the end points of an edge. To make union run faster, we attach the root with smaller depth to the root with the larger depth so that the modification of parent information is minimized.

3.4 Testing

For testing purpose, to make sure that all corner cases are met, we run the algorithms on 5 sparse graphs ad 5 dense graphs. Also, for each of the random graphs, 5 pairs of randomly chosen source and destination is given as input to all the 3 algorithms along with the adjacency list of the graphs. Time is recorded starting from the call to each of the algorithms till the determination of the maximum bandwidth paths by each of the algorithms. Results of the testing on sparse graphs and dense graphs are shown in the figure 3.3 and figure 3.4 respectively.

4 Analysis

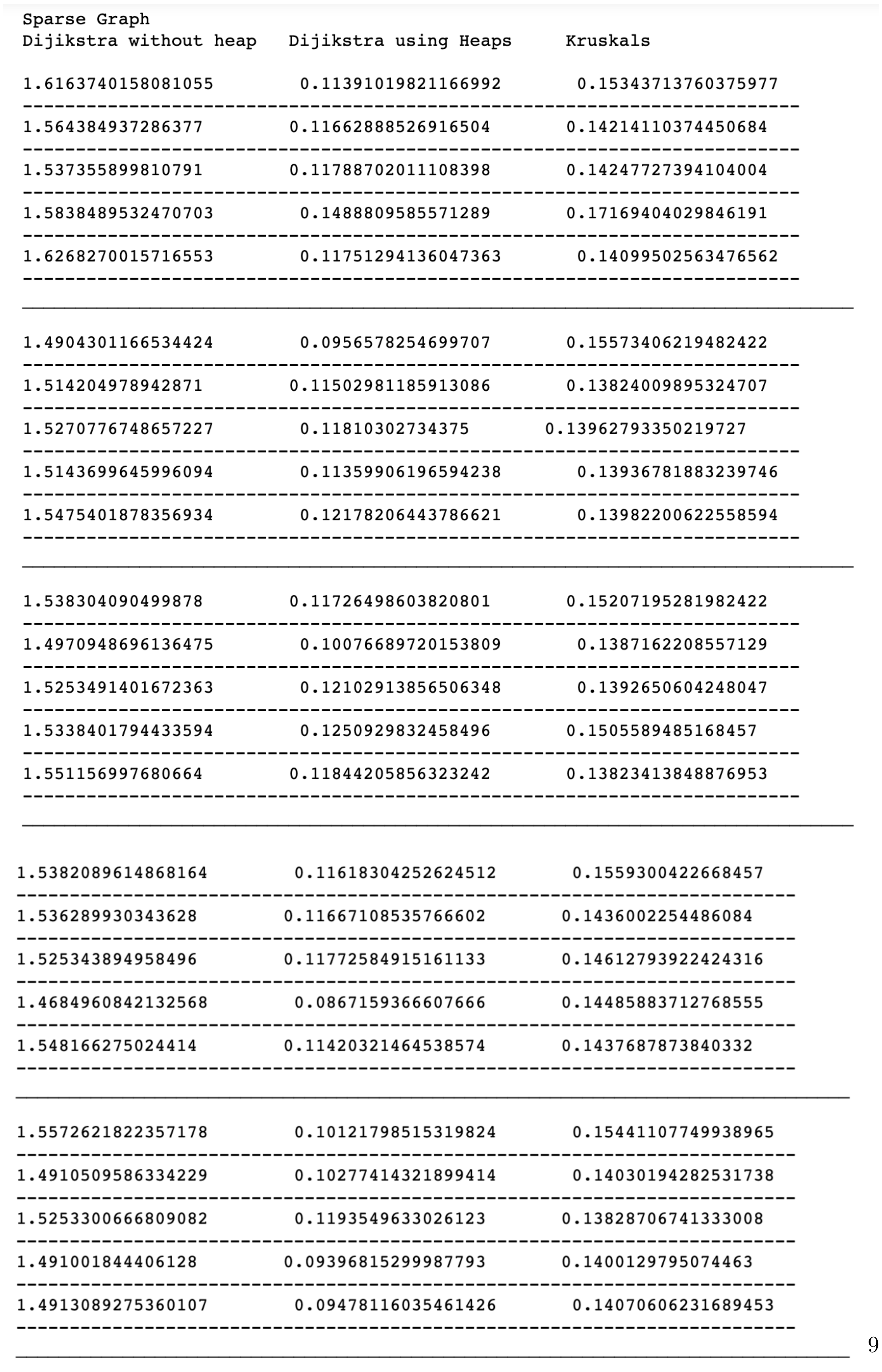
We can infer from the results that in sparse graphs, dijikstra's without heap is the slowest as it has to traverse through every vertex to find the vertex with the highest fringe for every iteration. The inefficiency of dijikstra's without heap algorithm comes from this traversal. Since $|V|^2$ is greater than ***O(|E|*log |V|)*** and ***O(|E|*log |E|)*** as the number of vertices is highly greater than the number of edges in a sparse graph, the running time is given by the order,

Dijikstra's with heap $<$ ***Kruskal's*** $<$ ***Dijikstra's without heap***

In dense graphs, as the number of edges (|E|) is very much greater than the number of vertices (|V|), we can infer from the results that ***O(|E|*log |V|)*** is faster than $|V|^2$ which is faster than ***O(|E|*log |E|)***. Thus the running time of the algorithms is in the order,

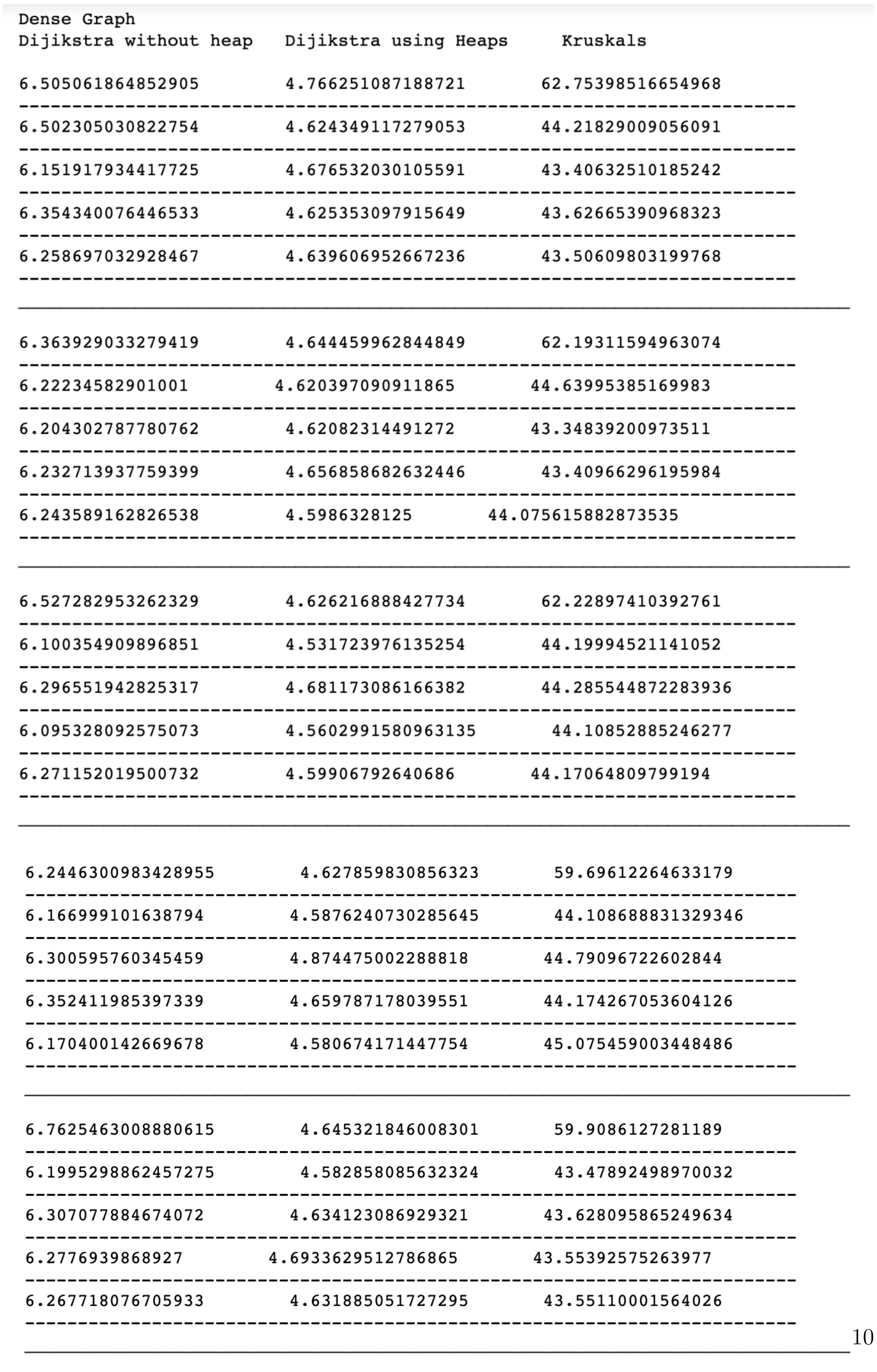
Dijikstra's with heap $<$ ***Dijikstra's without heap*** $<$ ***Kruskal's***

An important observation made from the experiment is that the margin of inefficiency of Dijikstra's without heaps algorithm on sparse graph is smaller when compared to the inefficiency of Kruskal's algorithm on dense graph. This helps us draw a conclusion that Dijikstra's algorithm could be a better choice, and in particular Dijikstra's algorithm using heap, for finding maximum bandwidth in network optimization problems.



```
Sparse Graph
Dijikstra without heap   Dijikstra using Heaps     Kruskals

1.6163740158081055        0.11391019821166992       0.15343713760375977
-------------------------------------------------------------------------
1.564384937286377       0.11662888526916504       0.14214110374450684
-------------------------------------------------------------------------
1.537355899810791       0.11788702011108398       0.14247727394104004
-------------------------------------------------------------------------
1.5838489532470703        0.1488809585571289       0.17169404029846191
-------------------------------------------------------------------------
1.6268270015716553        0.11751294136047363       0.14099502563476562
-------------------------------------------------------------------------
______________________________________________________________________________

1.4904301166534424        0.0956578254699707       0.15573406219482422
-------------------------------------------------------------------------
1.514204978942871       0.11502981185913086       0.13824009895324707
-------------------------------------------------------------------------
1.5270776748657227        0.11810302734375       0.13962793350219727
-------------------------------------------------------------------------
1.5143699645996094        0.11359906196594238       0.13936781883239746
-------------------------------------------------------------------------
1.5475401878356934        0.12178206443786621       0.13982200622558594
-------------------------------------------------------------------------
______________________________________________________________________________

1.538304090499878       0.11726498603820801       0.15207195281982422
-------------------------------------------------------------------------
1.4970948696136475        0.10076689720153809       0.1387162208557129
-------------------------------------------------------------------------
1.5253491401672363        0.12102913856506348       0.1392650604248047
-------------------------------------------------------------------------
1.5338401794433594        0.1250929832458496       0.1505589485168457
-------------------------------------------------------------------------
1.551156997680664       0.11844205856323242       0.13823413848876953
-------------------------------------------------------------------------
______________________________________________________________________________

1.5382089614868164        0.11618304252624512       0.1559300422668457
-------------------------------------------------------------------------
1.536289930343628       0.11667108535766602       0.1436002254486084
-------------------------------------------------------------------------
1.525343894958496       0.11772584915161133       0.14612793922424316
-------------------------------------------------------------------------
1.4684960842132568        0.0867159366607666       0.14485883712768555
-------------------------------------------------------------------------
1.548166275024414       0.11420321464538574       0.1437687873840332
-------------------------------------------------------------------------
______________________________________________________________________________

1.5572621822357178        0.10121798515319824       0.15441107749938965
-------------------------------------------------------------------------
1.4910509586334229        0.10277414321899414       0.14030194282531738
-------------------------------------------------------------------------
1.5253300666809082        0.1193549633026123       0.13828706741333008
-------------------------------------------------------------------------
1.491001844406128       0.09396815299987793       0.1400129795074463
-------------------------------------------------------------------------
1.4913089275360107        0.09478116035461426       0.14070606231689453
-------------------------------------------------------------------------
______________________________________________________________________________
```

Figure 3.3: Sparse graph results



```
Dense Graph
Dijikstra without heap   Dijikstra using Heaps     Kruskals

6.505061864852905        4.766251087188721       62.75398516654968
-------------------------------------------------------------------------
6.502305030822754        4.624349117279053       44.21829009056091
-------------------------------------------------------------------------
6.151917934417725        4.676532030105591       43.40632510185242
-------------------------------------------------------------------------
6.354340076446533        4.625353097915649       43.62665390968323
-------------------------------------------------------------------------
6.258697032928467        4.639606952667236       43.50609803199768
-------------------------------------------------------------------------
_____________________________________________________________________________

6.363929033279419        4.644459962844849       62.19311594963074
-------------------------------------------------------------------------
6.22234582901001        4.620397090911865       44.63995385169983
-------------------------------------------------------------------------
6.204302787780762        4.62082314491272       43.34839200973511
-------------------------------------------------------------------------
6.232713937759399        4.656858682632446       43.40966296195984
-------------------------------------------------------------------------
6.243589162826538        4.5986328125       44.075615882873535
-------------------------------------------------------------------------
_____________________________________________________________________________

6.527282953262329        4.626216888427734       62.22897410392761
-------------------------------------------------------------------------
6.100354909896851        4.531723976135254       44.19994521141052
-------------------------------------------------------------------------
6.296551942825317        4.681173086166382       44.285544872283936
-------------------------------------------------------------------------
6.095328092575073        4.5602991580963135       44.10852885246277
-------------------------------------------------------------------------
6.271152019500732        4.59906792640686       44.17064809799194
-------------------------------------------------------------------------
_____________________________________________________________________________

6.2446300983428955        4.627859830856323       59.69612264633179
-------------------------------------------------------------------------
6.166999101638794        4.5876240730285645       44.108688831329346
-------------------------------------------------------------------------
6.300595760345459        4.874475002288818       44.79096722602844
-------------------------------------------------------------------------
6.352411985397339        4.659787178039551       44.174267053604126
-------------------------------------------------------------------------
6.170400142669678        4.580674171447754       45.075459003448486
-------------------------------------------------------------------------
_____________________________________________________________________________

6.7625463008880615        4.645321846008301       59.9086127281189
-------------------------------------------------------------------------
6.1995298862457275        4.582858085632324       43.47892498970032
-------------------------------------------------------------------------
6.307077884674072        4.634123086929321       43.628095865249634
-------------------------------------------------------------------------
6.2776939868927        4.6933629512786865       43.55392575263977
-------------------------------------------------------------------------
6.267718076705933        4.631885051727295       43.55110001564026
-------------------------------------------------------------------------
_____________________________________________________________________________
```

Figure 3.4: Dense graph results