CSCE 629 Analysis of Algorithms Course Project, Network Routing Protocol

Sanjay Nayak
Department of Computer Science & Engineering
Texas A & M University
College Station, TX, 77843
sanjaynayak@tamu.edu

Abstract

This course project implements the network routing protocol that is important in the area of network optimization. The data structures and algorithms used are what was studied during the course work, i.e. heap, graph, and tree. The main algorithms used in this project are modified versions of Dijkstra's algorithm and modified version of Kruskal algorithm that are used to solve the MAX-BANDWIDTH-PATH problem. The report gives a detailed analysis and procedure used for the implementations of the discussed algorithms in a simple, undirected, and weighted graph.

1 Introduction

MAX-BANDWIDTH-PATH problem can be formulated as a problem where the goal is to find the path with maximum bandwidth in a weighted graph provided two vertices s and t are given, where s denotes the start vertex and t denotes the target vertex. Here the above problem is solved using modified versions of Dijkstra and Kruskal algorithm which is discussed below.

1.1 Modified Dijkstra's Algorithm

$m \leftarrow$ number of edges, $n \leftarrow$ number of vertices

1.1.1 Theory

Dijkstra's algorithm is used for finding SHORTEST-PATH between two pair of vertices. With a little modification, it can be used for MAX-BANDWIDTH-PATH problem. With simple implementation of Dijkstra's algorithm using arrays or lists, the time complexity we get is that of $O(n^2)$. If we use heap data structure or 2-3 tree data structure to implement the Dijsktra, the time complexity can be reduced to $O((m+n)\ log\ n)$, where m indicates the number of edges and n indicates the number of vertices. In a dense graph, the time complexity is equivalent to $O(m\ log\ n)$ as here the number of edges is high compared with number of vertices.

1.1.2 Analysis, Modified Dijsktra's Algorithm without Heap

Algorithm 1 denotes the modified Dijkstra's algorithm without using the heap data structure. Here Step 1 takes $O(n)$ time. Step 11 takes $O(n^2)$ time, thus making the time complexity of the total algorithm as $O(n^2)$. Step 11 can be further optimized to reduce the time complexity from $O(n^2)$ to $O(n\ log\ n)$ which can be done if heap structure is implemented. Step 25 checks if the maximum BW path is present between the two vertices. If present, then using the dad array, the path can be traced back when started from vertex t and the path is stopped when vertex s is encountered.

As a result, in theory, the time taken by modified Dijkstra's algorithm if implemented without heap should take more time when compared with modified Dijsktra's algorithm if implemented with heap data structure for both sparse and dense graphs.

Algorithm 1 Modified Dijkstra's Algorithm without Heap for MAX-BANDWIDTH-PATH

Input: a graph network G, a source node *s* and a destination node *t*
Output: a path from *s* to *t* giving the maximum bandwidth

1: **for** each vertex v **do**
2: status$[v]$ = UNSEEN
3: **end for**
4: status$[s]$ = IN_TREE
5: **for** each edge $[s, w]$ **do**
6: status$[w]$ = FRINGE
7: dad$[w] = s$
8: bw$[w]$ = cap$[s, w]$ ▷ bw: Bandwidth
9: ▷ cap: Capacity of the edge/edge weight
10: **end for**
11: **while** there are FRINGEs **do**
12: $v \leftarrow$ max(F)
13: Delete(F,w)
14: status$[v]$ = IN_TREE
15: **for** each edge $[v, w]$ **do**
16: **if** status$[w]$ = UNSEEN **then**
17: status$[w]$ = FRINGE
18: dad$[w] = v$
19: bw$[w]$ = min{bw$[v]$, cap$[v, w]$}
20: **else if** status$[w]$ = FRINGE and (bw$[w]$ < min{bw$[v]$, cap$[v, w]$}) **then**
21: dad$[w]$ = v
22: bw$[w]$ = min{bw$[v]$, cap$[v, w]$}
23: **end if**
24: **end for**
25: **end while**
26: **if** status$[t] \neq$ IN_TREE **then**
27: return **No s-t max BW path**
28: **else**
29: max_bw_path $\leftarrow$ Stack
30: **while** s $\neq$ t **do**
31: max_bw_path $\leftarrow$ t
32: t = dad[t]
33: **end while**
34: return max_bw_path
35: **end if**

1.1.3 Analysis, Modified Dijkstra's Algorithm with Heap

Algorithm 2 represents the modified Dijkstra's algorithm when implemented with heap data structure. Here heap data structure is used to store the fringes that are visited in max heap order. Thus, retrieval of vertex that has maximum bandwidth among the fringes takes $O(1)$ time and the time taken to reset the heap when the maximum element is removed from the heap is $O(\ log\ n)$ i.e. the height of the tree.

The operation $Insert(F, w)$ in a max-heap can be done in $O(log\ n)$ time and getting the maximum element from a max-heap is done in $O(1)$ time. Deletion in max-heap will result in the fix-heap process which in turn takes $O(log\ n)$ time. Step 26 in Algorithm 2 resets the heap by performing the max heapify when the value of bandwidth representing the edge is changed to a different value. This indicates that a better bandwidth path is found. Step 6 takes $O(m\ log\ n)$ time and Step 13 takes $O(n\ log\ n)$ time. Thus the total time complexity of the Dijkstra's algorithm when heap data structure is used now becomes $O((m + n)\ log\ n)$.

Algorithm 2 Modified Dijkstra's Algorithm with Heap for MAX-BANDWIDTH-PATH

Input: a graph network G, a source node *s* and a destination node *t*
Output: a path from *s* to *t* giving the maximum bandwidth

for each vertex v **do**
    status$[v]$ = UNSEEN
end for
status$[s]$ = IN_TREE
F $\leftarrow$ heap of fringes
for each edge $[s, w]$ **do**
    Insert(F,w)
    status$[w]$ = FRINGE
    dad$[w] = s$
    bw$[w]$ = cap$[s, w]$ ▷ bw: Bandwidth
                ▷ cap: Capacity of the edge/edge weight
end for
while there are FRINGEs **do**
    $v \leftarrow$ max(F)
    status$[v]$ = IN_TREE
    Delete(F,w)
    for each edge $[v, w]$ **do**
        if status$[w]$ = UNSEEN **then**
            status$[w]$ = FRINGE
            dad$[w] = v$
            bw$[w]$ = min{bw$[v]$, cap$[v, w]$}
            Insert(F,w)
        else if status$[w]$ = FRINGE and (bw$[w]$ $<$ min{bw$[v]$, cap$[v, w]$}) **then**
            dad$[w]$ = v
            bw$[w]$ = min{bw$[v]$, cap$[v, w]$}
            reset_heap(F,w, bw$[w]$)
        end if
    end for
end while
if status$[t] \neq$ IN_TREE **then**
    return **No s-t max BW path**
else
    max_bw_path $\leftarrow$ Stack
    while s $\neq$ t **do**
        max_bw_path $\leftarrow$ t
        t = dad[t]
    end while
    return max_bw_path
end if

1.2 Modified Kruskal Algorithm

$m \leftarrow$ number of edges, $n \leftarrow$ number of vertices

1.2.1 Theory

Kruskal's Algorithm is originally created for creation of spanning tree from a graph. A spanning tree is a new graph that is created from the original graph by using all vertices and not all edges. A maximum (or minimum) spanning tree is a spanning tree that has the maximum (or minimum) weight over all the spanning trees in a graph.

The Kruskal algorithm can be modified so that it solves the MAX-BANDWIDTH-PATH problem. For this to happen, the maximum spanning tree needs to be obtained using the Kruskal's Algorithm. Then DFS can be used to find a path between the source (s) and target (t) vertices. Once the path is known, then the maximum bandwidth of that path can be obtained. This is possible through this algorithm as the algorithm provides us with a maximum spanning tree and by the definition of the

maximum spanning tree, we get a new graph with all vertices connected and having maximum weight among all the spanning trees.

The modified version of Kruskal's algorithm uses **Find**, **Union**, **DFS**, and original **Kruskal** algorithms to find the maximum bandwidth and its path.

1.2.2 Analysis

Algorithm 3 represents the Find algorithm that is used to find the rank of two vertices of an edge and then based on the rank, merging happens. It is done in $O(log\ n)$ time. Algorithm 4 represents the Union algorithm that merges the two vertices by updating the parents of the two vertices. It is done in $O(1)$ time. So Step 3 of Algorithm 6 takes time $O(m\ log\ n)$. Algorithm 6 is used to create maximum spanning tree of the given graph, which is then used in Algorithm 7 to find the MAX-BANDWIDTH-PATH. Algorithm 6 takes a total time of $O(m\ log\ n)$ to create the maximum spanning tree. The Algorithm 5 is used to find the path between the source (s) and target (t) vertices in the maximum spanning tree, which is then used to find the MAX-BANDWIDTH-PATH and the MAX-BANDWIDTH value and it takes $O(n + m)$ time. Thus, the main Algorithm 7 takes time $O(m\ log\ n)$ to get the maximum bandwidth path.

The time complexities of modified Kruskal's Algorithm and the modified Dijkstra's Algorithm with heap is same in the worst case. But in practical scenarios, the actual time taken to find the MAX-BANDWIDTH-PATH will vary depending on the density of the graph. For a sparse graph, both algorithms should be at par with each other. But for a dense graph, Kruskal can take more time as it calculates the maximum spanning tree from the dense graph first, which can be more time consuming.

Further detailed analysis is done in the **Experiments** section for all the above algorithms.

Algorithm 3 Find Algorithm to find the root of a vertex

Input: a vertex v and parent array
Output: root of the vertex v

```
1: w = v
2: while parent[w] ≠ -1 do
3:     s ← w
4:     w = parent[w]
5: end while
6: while s ≠ ∅ do
7:     u ← s
8:     parent[u] = w
9: end while
10: return w
```

Algorithm 4 Union Algorithm to merge two different vertices under one root

Input: ranks of two vertices, r_1 and r_2, parent array, and rank array
Output: two different ranked vertices merged under a single root

```
1: if rank[r1] > rank[r2] then
2:     parent[r2] = r1
3: else if rank[r1] < rank[r2] then
4:     parent[r2] = r1
5: else
6:     parent[r1] = r2
7:     rank[r2] ++
8: end if
```

2 Implementation

The above three discussed algorithms for MAX-BANDWIDTH-PATH is implemented in Python3.8 without using any predefined libraries of Python except that of Queue, which is used during the Find operation during the creation of maximum spanning tree.

Algorithm 5 Depth First Search Algorithm to find the path between source and target

Input: a graph network G, a source node *s* and a destination node *t*
Output: a path from *s* to *t* giving the maximum bandwidth

```
color[v] = GRAY
for each edge [v, w] do
    if color[w] == WHITE then
        parent[w] = v
    end if
end for
color[v] = BLACK
```

Algorithm 6 Kruskal's Algorithm for Maximum Spanning Tree

Input: a graph network G,
Output: maximum spanning tree of the graph

```
edgeSort ← Sorted edges in non-increasing order using Heap Sort
mst ← empty graph
for all edges in edgeSort do
    e_i = [u_i, v_i]
    r_1 = Find(u_i)
    r_2 = Find(v_i)
    if r_1 ≠ r_2 then
        mst ← (u_i, v_i)
        Union(r_1, r_2)
    end if
end for
```

2.1 Graph Generator

A graph generator is implemented that generates two random, connected, and undirected graph of 5000 vertices. ***random*** function is used to generate the vertex numbers between 0 and 4999 with a condition that *no self loops are present* and *there are atmost one edge between any two vertices*. To make the graph connected, a cycle was created first and then the edges are connected randomly.

Sparse Graph G_1 is created with an average vertex degree of 6 and edge weight in the range $[1, 100]$. Dense Graph G_2 is created such that each edge vertex is adjacent to atleast 20% of the other vertices, i.e. 1000, considering the total vertex size as 5000 with edge weights in range of $[1, 10000]$. Average time taken for generation of Sparse Graph G_1 is 0.1 seconds, and Dense Graph G_2 is 158 seconds.

2.2 Heap Structure

Two different heap subroutines were implemented, one for the **modified Dijkstra's algorithm** that uses heap data structure and the other for the **modified Kruskal's algorithm** that uses HeapSort to sort the edges in non-increasing order.

Algorithm 7 Modified Kruskal's Algorithm for MAX-BANDWIDTH-PATH

Input: a graph network G, a source node *s* and a destination node *t*
Output: maximum bandwidth, a path from *s* to *t* giving the maximum bandwidth

```
mst ← max spanning tree using Algorithm 6
Apply DFS on mst to get the parent array between s and t
max_bw_path ← Stack, max_bw ← MAXINT
while s ≠ t do
    max_bw_path ← t
    max_bw = min(max_bw, mst.bw[t, parent[t]])
    t = dad[t]
end while
return max_bw_path, max_bw
```

For the heap subroutines used in the modified Dijkstra's algorithm, three different arrays (h, d, p) were used where h stores the heap and $h[i]$ gives the name of a vertex in the graph. d stores the values (maximum bandwidth values) and to find the value of $h[i]$, we can use $d[h[i]]$. p stores the position of each vertex in the heap h, where $p[i]$ returns the position of the vertex i in heap h. The heap concept used here is *max heap*. When any vertex is inserted into heap h, the max heap logic ensures that the heap formed follows the structure of max heap. Similarly, when a value is changed for any vertex in the heap, the max heap logic is executed that maintains the max heap structure. Max heap is used as we need to get the vertex with maximum bandwidth in Step 14 of Algorithm 2. A *reset heap* subroutine is created that will be called when Step 26 of Algorithm 2 is executed. It combines the operation of $Delete(F, w)$ to remove the old value of the vertex and $Insert(F, w)$ to insert the new value of the vertex into a single sub-routine. A print heap subroutine was also added that prints the heap which was used during test and debug sessions.

The heap subroutine used in the modified Kruskal's algorithm was used to sort the edges in non-increasing order. This subroutine has similar functionalities as the above heap subroutine for the Dijkstra with minor differences in the implementation. Here also, the max heap concept is used in order to sort the edges of the graph. Three different arrays (s, t, d) were used, where s stores the source vertex of an edge, t stores the target vertex of the same edge, and d stores the bandwidth value/ capacity of that edge. Since this edge heap is used to sort the edges only, the delete or the reset functionality was not implemented. The insert subroutine inserts each element such that the property of max heap is followed.

For all the heap structure implementation, the heap position of parent and child nodes are given by $(i-1)/2$ for the parent node, $(2*i+1)$ for left child, and $(2*i+2)$ for the right child, and the position of the root node starts from 0.

2.3 Routing Algorithm

The routing algorithms used in the analysis included the three different algorithms discussed above i.e. modified Dijkstra without heap, modified Dijkstra with heap, and modified Kruskal. The implementations of the above three algorithms are done using the algorithms described in section 1 of this report. Algorithm 1 is used to implement the **modified Dijkstra's algorithm without heap** data structure, Algorithm 2 is used for **modified Dijkstra's algorithm with heap** data structure implementation, and Algorithm 7 is used for **modified Kruskal's algorithm** implementation that uses HeapSort for sorting of the graph's edges.

3 Experiments

For the analysis of the performance between the three discussed algorithms, 5 pairs of Sparse G_1 graphs and Dense G_2 graphs were generated. For each of the graph generated, 5 pairs of randomly selected source-destination vertices were used and their running time was recorded. In order to calculate the running time of each algorithm, *datetime* module of Python was used. The current time before and after the main algorithm function call was recorded and the time difference between the two times was calculated in seconds, which was used for the analysis.

For the experiments, 5 pairs of source and target vertices are randomly generated. For each of the source and target vertices generated, 5 different Sparse Graph G_1 and 5 different Dense Graph G_2 are generated and the running time for each of the 3 different algorithms is calculated. Each of the three algorithms runs for 50 times.

4 Results

The analysis can be found in Table 1 - 5. Each of the analysis table contains analysis for a pair of randomly generated graphs, sparse and dense graphs. The *Source-Target Pair* column indicates the five different pair of source and target pairs generated randomly. The same five pairs of vertices were used in all graph analysis.

The results indicate that **modified Dijkstra's algorithm with heap** data structure performs the best for both sparse and dense graphs. It is also noticed that modified Kruskal takes more time for execution than modified Dijkstra with heap even though the worst case time complexity is

Table 1: Modified Routing Algorithms Run time analysis for Graph Pair 1

		Modified Routing Algorithms Time (seconds)		
Graph Type	Source-Target Pair	Dijkstra (Without Heap)	Dijkstra (With Heap)	Kruskal
Sparse Graph	Pair 1	8.78586	0.219532	2.133431
	Pair 2	6.07144	0.251202	2.167773
	Pair 3	2.787788	0.125793	2.148171
	Pair 4	10.98102	0.423455	2.168562
	Pair 5	10.396027	0.392208	2.148973
Dense Graph	Pair 1	6.958942	2.799269	157.928913
	Pair 2	3.653271	1.666443	156.790039
	Pair 3	5.629876	2.512138	155.99107
	Pair 4	5.023674	2.424562	157.962531
	Pair 5	5.769135	0.686122	157.226297

Table 2: Modified Routing Algorithms Run time analysis for Graph Pair 2

		Modified Routing Algorithms Time (seconds)		
Graph Type	Source-Target Pair	Dijkstra (Without Heap)	Dijkstra (With Heap)	Kruskal
Sparse Graph	Pair 1	5.333736	0.172668	2.102072
	Pair 2	10.76015	0.188278	2.133747
	Pair 3	6.46185	0.250813	2.087148
	Pair 4	1.756675	0.083616	2.104227
	Pair 5	10.940876	0.392203	2.132653
Dense Graph	Pair 1	5.380384	2.371208	153.824817
	Pair 2	5.617764	0.386698	153.301247
	Pair 3	5.615207	2.606953	150.859299
	Pair 4	2.79186	1.719751	154.996065
	Pair 5	6.037179	2.68212	152.823553

similar. This can be attributed to the fact that modified Kruskal uses the maximum spanning tree to find the MAX-BANDWIDTH-PATH. The generation of maximum spanning tree and the different extra algorithms used in the modified Kruskal algorithm must be contributing to the difference in time. For the sparse graph, modified Dijkstra without heap performs the worst among the three algorithms, which can be attributed to the fact that it takes $O(n^2)$ time for its execution which is higher. Therefore, **for the sparse graph, modified Dijkstra with heap performs the best followed by modified Kruskal, which is followed by modified Dijkstra without heap**.

Analysis of the dense graph leads to an interesting observation. Modified Kruskal algorithm takes much longer time than modified Dijkstra without heap algorithm even though the time complexity of modified Kruskal is $O(m\ log\ n)$ and that of modified Dijkstr without heap is $O(n^2)$. This can be attributed to the density of the graph. As density increases, the time taken to parse all edges and vertices to find the maximum spanning tree also increases, thus increasing the total time. Also modified Kruskal is a combination of several algorithms like Find, Union, Maximum Spanning Tree generation, and DFS. They contribute to the time complexity which becomes more evident in higher density graphs. This is not the case for modified Dijkstra's algorithms where sparse and dense graphs have small time difference. Thus, **for the dense graphs, modified Dijkstra with heap performs the best followed by modified Dijkstra without heap, and then modified Kruskal algorithm**.

5 Discussions

This analysis provided us with an idea as to how different algorithms for MAX-BANDWIDTH-PATH behave under different circumstances. Further study by using other sorting algorithms like merge sort, bucket sort can be used in modified Kruskal algorithm instead of HeapSort for edge sorting to check if there is any improvement in the performance. Studies using graphs with varying density,

Table 3: Modified Routing Algorithms Run time analysis for Graph Pair 3

		Modified Routing Algorithms Time (seconds)		
Graph Type	Source-Target Pair	Dijkstra (Without Heap)	Dijkstra (With Heap)	Kruskal
Sparse Graph	Pair 1	4.139353	0.22655	2.118579
	Pair 2	4.79019	0.062526	2.14365
	Pair 3	0.815087	0.219558	2.103018
	Pair 4	2.597377	0.235161	2.15027
	Pair 5	8.049785	0.329327	2.110833
Dense Graph	Pair 1	3.927576	1.791143	161.236165
	Pair 2	4.962728	2.239058	162.129965
	Pair 3	2.278558	1.092389	162.675604
	Pair 4	2.878749	2.17491	163.108738
	Pair 5	5.039692	2.339857	160.642527

Table 4: Modified Routing Algorithms Run time analysis for Graph Pair 4

		Modified Routing Algorithms Time (seconds)		
Graph Type	Source-Target Pair	Dijkstra (Without Heap)	Dijkstra (With Heap)	Kruskal
Sparse Graph	Pair 1	8.202809	0.282455	2.102307
	Pair 2	3.217176	0.141033	2.134128
	Pair 3	1.488742	0.204916	2.118426
	Pair 4	3.327559	0.298872	2.133287
	Pair 5	9.851329	0.376173	2.086468
Dense Graph	Pair 1	5.973813	2.014162	162.418815
	Pair 2	3.280477	1.166658	165.223988
	Pair 3	5.451575	0.439604	161.514115
	Pair 4	5.281074	2.451101	155.982767
	Pair 5	5.802677	2.540168	154.770631

Table 5: Modified Routing Algorithms Run time analysis for Graph Pair 5

		Modified Routing Algorithms Time (seconds)		
Graph Type	Source-Target Pair	Dijkstra (Without Heap)	Dijkstra (With Heap)	Kruskal
Sparse Graph	Pair 1	5.054609	0.201262	1.08184
	Pair 2	5.129849	0.18786	1.112263
	Pair 3	1.115975	0.125427	1.098328
	Pair 4	1.646237	0.021129	1.09132
	Pair 5	2.765538	0.039747	1.105425
Dense Graph	Pair 1	6.604804	3.043002	153.97047
	Pair 2	5.051771	0.517812	153.086116
	Pair 3	2.462413	0.88256	153.980051
	Pair 4	2.913837	1.350188	153.018756
	Pair 5	1.608949	0.70944	156.252153

number of vertices, and use of map data structure instead of arrays to save space can also be used in the future for more analysis.

6 Conclusion

After the current analysis, it can be concluded that the best algorithm is found to be modified Dijkstra's algorithm with heap data structure. Modified Kruskal algorithm was found to be better than modified Dijkstra's algorithm without using heap data structure in sparse graphs but performed worse in dense graphs. Modified Kruskal algorithm uses four different algorithms in order to find the MAX-BANDWIDTH-PATH. This can be a reason for its worse performance among the three in dense graph scenario.

7 Appendix

The below results include the output from the code to generate the MAX-BANDWIDTH-PATH and perform running time analysis.

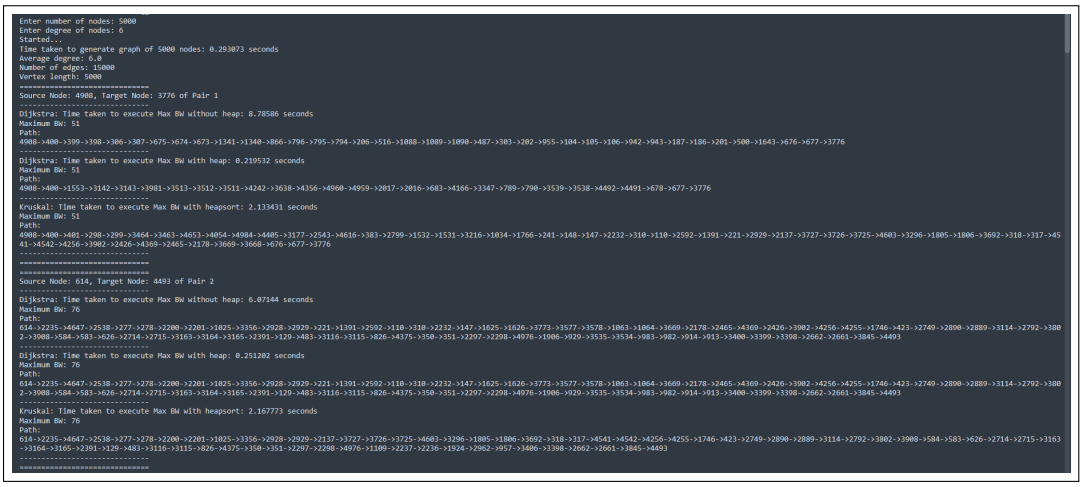

Figure 1: Sparse Graph Output, Source-Target Pair 1 and Source-Target Pair 2

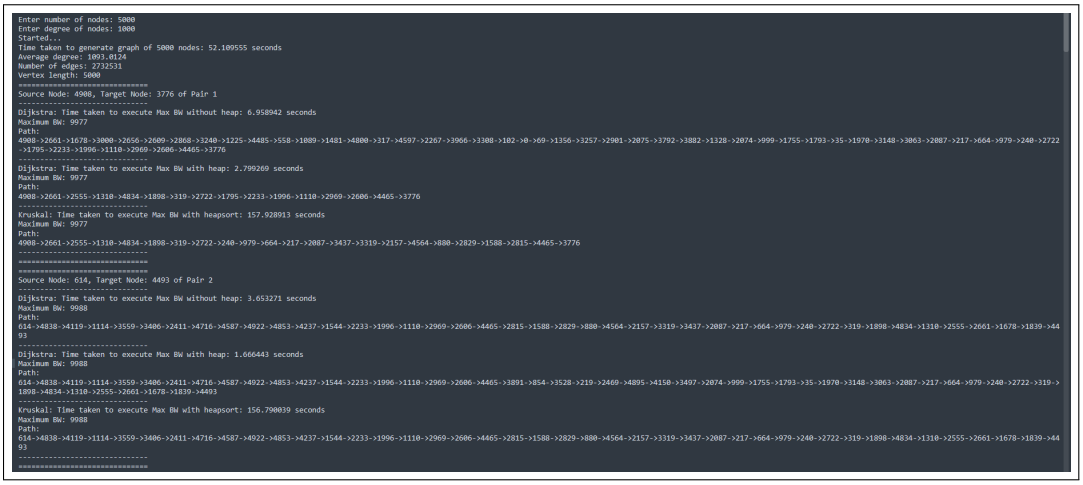

Figure 2: Dense Graph Output, Source-Target Pair 1 and Source-Target Pair 2