Texas A&M University

Department of Computer Science and Engineering

CSCE 629-601 Analysis of Algorithms

Course Project Report

Ritika Nigam

UIN: 633002197

Majors: MSCS fall’22

1. Introduction

The aim of this project is to implement a network routing protocol using the data structures and algorithms learnt so far. Initially, the sparse graph and dense graph generation is done. Then, heap data structure is implemented with subroutines for MAXIMUM, INSERT and DELETE. Coming to the routing protocols algorithms for MAX-BANDWIDTH-PATH problem, there are 3 versions of implementations: Dijkstra's algorithm without using a heap structure, Dijkstra's algorithm using a heap structure for fringes and modifed Kruskal's algorithm in which edges are sorted by HeapSort.

1. Implementation
   1. Random Graph Generation

The classes for graph and vertex have been implemented. The graph class contains both the adjacency list and adjacency matrix, either of which can be used for the later algorithms to work on. To make sure that the graph is connected, initially a cycle is made by connecting all the vertices of the graphs, then later edges are added randomly to meet the requirement for the sparse and dense graphs. While adding the edges at random, the respective edge weight is also assigned at random within range 1 to 20.

* 1. Heap

The heap being implemented for this project is MaxHeap. The heap structure is used twice, one in Dijkstra with Heap Implementation to store the fringe nodes and in Kruskal’s Algorithm to HeapSort the edges for the generation of the Maximum Spanning Tree. In the Heap implementation, Maximum, Insert, Delete, Swap, fixHeap\_movUP, fixHeap\_movDOWN and empty subroutines are defined.

2.3 Routing Algorithm

The input to these algorithms would be the graph G, source vertex s and the destination vertex t. The output of this algorithm gives the maximum bandwidth path from s to t in G. m is the #edges and n is the #vertices. The psuedo code for the algorithms implemented are as follows:

* + 1. Dijkstra's algorithm without using the heap structure

The psuedo code is as follows:

1. for v=1to n do status[v]=unseen

2. status[s]=in-tree; bw[s] = +INF

3. for each edge [s,w] do

(a) status[s]=fringe

(b) bw[w]=weight[s,w]

(c) dad[w]=s

4. while status[t] != in-tree do:

(a) pick a fringe v of the max bw[v]

(b) status[v] = in-tree

(c) for each edge [v,w] do

i. if status[w] = un-seen then  
 A. status[w]=fringe  
 B. bw[w]=min{bw[v],weight[v,w]}

C. dad[w]=v

ii. else if status[w] = fringe and bw[w]<min{bw[v],weight[v,w]} then

A. bw[w]=min{bw[v],weight[v,w]}  
 B. dad[w]=v

(d) return dad[1...n]

2.3.2 Dijkstra’s algorithm using a heap structure for fringes

Modifications are done to the above algorithm to use a heap structure for fringes. The changes are done as follows:

• 4(a) v = Maximum(F); Delete(F,v);

• c(i)D Insert(F,w);  
• c(ii)C Insert(F,w);

* + 1. Modified Kruskal's algorithm for which edges are sorted by heap sort

The heap sort support for this algorithm has been added to the previous heap class

main()  
1. T = Kruskal(G) \\ construct a maxm spanning tree T for G

2. find the s-t path on T using BFS

3. return the path

Krukal(G)

1. p[1....n],rank[1...n] \\ p[i] is the parent of i, rank[i] is the rank of the sub-tree in which i is contained

2. sort edges e1,e2,...em in descending order using heap sort  
3. T = new graph with all vertices of G with no edges  
4. for each vertex v in T do MakeSet(v)

5. for each edge ei = (ui, vi) do

(a) r\_u=Find(ui); r\_v=Find(vi); \\ to find the roots of u and v in T

(b) if(r\_u != r\_v) then

i. add edge (ui, vi) in T;

ii. Union(r\_u,r\_v)

6. return T

MakeSet(v)

1. p[v]=0;

(a) rank[v]=0;

(b) p[v]=0;

Union(r1, r2)

1. if(rank[r1] < rank[r2])then p[r1] = r2  
2. else if(rank[r2] < rank[r1]) then p[r2] = r1

3. else then

(a) p[r1] = r2;

(b) rank[r2]++;

Find(v)

1. w = v;

2. while p[w]! = 0 do

(a) w = p[w]

3. return(w)

1. Testing

Graphical user interface, text

Description automatically generated The figure below represents the sample output of the code on testing:

3.1 Time Analysis for Sparse Graph

3.1.1 Dijkstra without Heap

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
| Graph1 | 0.020381 | 0.239962 | 0.152002 | 0.054939 | 0.199437 |
| Graph2 | 0.136164 | 0.174512 | 0.089222 | 0.119729 | 0.068480 |
| Graph3 | 0.075084 | 0.144127 | 0.026548 | 0.127149 | 0.024403 |
| Graph4 | 0.115871 | 0.069095 | 0.074183 | 0.157109 | 0.011613 |
| Graph5 | 0.133118 | 0.003670 | 0.152772 | 0.137317 | 0.075601 |

3.1.2 Dijkstra With Heap

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
| Graph1 | 0.005617 | 0.007773 | 0.007299 | 0.011196 | 0.005971 |
| Graph2 | 0.005396 | 0.007782 | 0.006687 | 0.006434 | 0.012245 |
| Graph3 | 0.004863 | 0.007113 | 0.006208 | 0.003971 | 0.007070 |
| Graph4 | 0.005568 | 0.004322 | 0.005493 | 0.011364 | 0.008101 |
| Graph5 | 0.007049 | 0.002186 | 0.008693 | 0.009819 | 0.009515 |

3.1.3 Kruskal Using heap

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
| Graph1 | 0.026352 | 0.051074 | 0.074641 | 0.101274 | 0.125808 |
| Graph2 | 0.026417 | 0.050657 | 0.075455 | 0.101422 | 0.125670 |
| Graph3 | 0.026186 | 0.051202 | 0.074218 | 0.102293 | 0.125657 |
| Graph4 | 0.026480 | 0.050873 | 0.074500 | 0.101923 | 0.125649 |
| Graph5 | 0.026709 | 0.050694 | 0.074365 | 0.101893 | 0.126354 |

3.2. Time Analysis for Dense Graph

3.2.1 Dijkstra without Heap

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
| Graph1 | 0.311510 | 0.376062 | 0.552437 | 0.393297 | 0.667290 |
| Graph2 | 0.366778 | 0.239818 | 0.737220 | 0.633232 | 0.651465 |
| Graph3 | 0.268601 | 0.524881 | 0.375233 | 0.370478 | 0.585401 |
| Graph4 | 0.310243 | 0.538868 | 0.711802 | 0.437294 | 0.932026 |
| Graph5 | 0.185270 | 0.403850 | 0.544134 | 0.358456 | 1.004209 |

3.2.2 Dijkstra With Heap

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
| Graph1 | 0.174779 | 0.253587 | 0.288909 | 0.478773 | 0.646504 |
| Graph2 | 0.224554 | 0.194826 | 0.573944 | 0.480757 | 0.995911 |
| Graph3 | 0.200529 | 0.193329 | 0.379460 | 0.415270 | 0.979749 |
| Graph4 | 0.089711 | 0.391066 | 0.522209 | 0.546090 | 0.676692 |
| Graph5 | 0.163358 | 0.257275 | 0.521679 | 0.415153 | 0.96404 |

3.2.3 Kruskal Using heap

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
| Graph1 | 4.377486 | 9.426243 | 14.877192 | 21.188873 | 26.928918 |
| Graph2 | 4.235023 | 9.586739 | 15.140027 | 21.449246 | 27.110798 |
| Graph3 | 4.218216 | 9.503941 | 15.132410 | 21.230857 | 27.031536 |
| Graph4 | 4.221260 | 9.488899 | 15.127328 | 21.294997 | 27.569386 |
| Graph5 | 4.232025 | 9.446883 | 15.239198 | 21.239360 | 27.191944 |

1. Analysis on the Output
   1. Theoretical Time Complexity

|  |  |
| --- | --- |
| Algorithm | Time Complexity |
| Dijkstra Without Heap | O(n2) |
| Dijkstra with Heap | O((n + m)logn) |
| Maximum Spanning Tree Build Time | O(mlogn) |
| Path Find Time | O(m) |

4.2 Dijkstra’s Algorithm

From the theoretical Complexities, we can observe that Dijkstra Implementation with Heap is faster than that without Heap and this can also be seen in the output table. This is because insertion and deletion in Heap takes O(logn) time. In non-heap implementation, it takes O(n) time to find the fringe vertex with the maximum bandwidth. Therefore, the performance of Dijkstra with heap is better than Dijkstra without heap. This result holds true for both sparse as well as dense random graphs. We can observe that the trend remains the same as the number of nodes increases the running time increases. This is because we have more number of edges to be examined now.

* 1. Kruskal’s Algorithm

Maximum Spanning Tree build time is O(mlogn) as per the theory. Since building of an MST involves many operations like to find the edges in decreasing order and creating an tree, we can see that the time to build a MST is larger. And the time to find the source to destination path is very less than the build time as it just involves going through the parent array got from DFS which takes linear time. We can observe than Kruskal is faster than Dijkstra with Heap for graph with 6 degrees. And, for the dense graph, Dijkstra with Heap is faster than that of Kruskal Algorithm. The reason behind this might be that since a greater number of edges are to be inserted in the heap and are to be sorted each time in the case of Kruskal Algorithm.

1. Conclusion

From the outputs, we can observe that although the theoretical time complexity are the same, the running time for each algorithm differs for each case. This is because the graphs are random in nature and each graph won’t let the algorithm reach its worst case scenario which is being represented by the theoretical complexities. Dijkstra performs better when we need to find a single path between two vertices. Kruskal algorithm gives us an advantage of generating a given MST once and determining the path based on that MST in linear time.