CSCE-629 Analysis of Algorithms Fall 2017
Course Project Report

Venkata Satya K.S. Bondapalli
UIN:725006670

November 30, 2017

0. 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 modified Kruskal's algorithm in which edges are sorted by HeapSort.

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.

1.1 Graph G_1, the average vertex degree is 8

The average vertex degree implies that on an average the no. of edges at a vertex is 8. Since, it is an undirected graph, it means that #Edges = 4 * #Vertices. After the initial cycle is made, from all the possible leftover combinations of an edge between any 2 vertices, the edges are picked at random such that the #Edges count is equal to 4 * #Vertices.

1.2 Graph G_2, each vertex is adjacent to about 20% of the other vertices, which are randomly chosen

After ensuring that the graph is connected by making an initial cycle, each pair of vertex (u,v) is chosen at random and the two vertices are connected with a probability of 20%. u is iterated from (1 to 5000) and v is iterated from (u+2 to 5000).

2. Heap Structure

The heap class to be used in the later algorithms has been defined and implemented with support for the following operations: MAXIMUM, INSERT and DELETE. The max heap has been implemented by closing following the solution for HW#1 Problem#5. I have taken care to cover the corner cases in HeapFy functions in which if the node has one child only.

3. Routing Algorithms

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:

3.1 Dijkstra's algorithm without using the heap structure

The psuedo code is as follows:

1. for v = 1 to 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]

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);

3.3 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 e_1,e_2,...e_m 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 $e_i = (u_i, v_i)$ do
   (a) r_u=Find(u_i); r_v=Find(v_i); \\ to find the roots of u and v in T
   (b) if(r_u != r_v) then
      i. add edge (u_i, v_i) 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(r_1, r_2)

1. if($rank[r_1] < rank[r_2]$)then $p[r_1] = r_2$
2. else if($rank[r_2] < rank[r_1]$) then $p[r_2] = r_1$
3. else then
   (a) $p[r_1] = r_2$;
   (b) $rank[r_2] + +$;

Find(v)

1. $w = v$;
2. while $p[w]! = 0$ do
   (a) $w = p[w]$
3. return(w)

Testing

1. Results

	A	B	C	D	E	F	G
1	Sparse				Dense		
2	D w/o	D w/	Kruskal		D w/o	D w/	Kruskal
3	0.438965797	0.065084934	1.34053278		4.098696947	2.319338083	265.9338083
4	2.127019882	0.152341127	1.266756058		2.736368895	1.952710867	229.2710867
5	2.157790899	0.026474953	1.349528074		4.467850924	2.621898174	296.1898174
6	2.330387115	0.113482952	1.399105072		2.527981997	2.600128174	294.0128174
7	2.307390928	0.147476912	1.356145144		3.102673054	2.336758852	267.6758852
8	0.98697114	0.086804867	1.349916935		9.098633051	5.232757807	557.2757807
9	1.885935068	0.075392008	1.347283125		12.38700008	4.607435942	494.7435942
10	2.060549021	0.124944925	1.331882954		10.43403006	3.888566971	422.8566971
11	1.725136995	0.106219053	1.366964102		4.650633097	1.748023987	208.8023987
12	0.970037937	0.06592989	1.371529102		2.304172039	1.985723019	232.5723019
13	2.725991964	0.197583914	1.694638968		12.23395801	6.105847836	644.5847836
14	0.936516047	0.063691139	1.342182159		4.885140181	2.447189093	278.7189093
15	1.713850021	0.065428972	1.34790206		0.986170053	5.262888908	560.2888908
16	0.735516071	0.058706999	1.364675045		5.294090986	2.650042057	299.0042057
17	2.480933905	0.222988844	1.357161045		2.87968111	2.078603029	241.8603029
18	2.123551846	0.388748884	2.301985979		10.28424096	3.627834082	396.7834082
19	0.303891182	0.041445971	2.187153101		9.117419958	4.981694937	532.1694937
20	1.234011889	0.132335901	2.216558933		2.093497992	3.380849838	372.0849838
21	2.956043005	0.200447083	2.176569939		5.519816875	0.682512045	102.2512045
22	3.0921278	0.198565006	2.184798956		4.687436104	2.493043184	283.3043184
23	2.178842068	0.111434937	2.163595915		7.668256044	3.217218161	355.7218161
24	4.167832136	0.263730049	2.126040936		2.09038496	4.843599081	518.3599081
25	1.715300083	0.198004007	2.169471025		10.74491501	6.75880003	709.880003
26	4.361372948	0.289366007	2.167378902		12.50089788	6.431226015	677.1226015
27	2.625841856	0.137895107	2.16869998		11.41771293	5.496892929	583.6892929
28							
29	Sum						
30	50.3418076	3.534524441	42.44845629	0	158.2116592	89.7515831	9825.15831
31	Avg						
32	2.013672304	0.141380978	1.697938251	0	6.328466368	3.590063324	393.0063324

	Sparse	Dense
Dijkstra's without heap	2.0136723042	6.3284663677
Dijkstra's with heap	0.1413809776	3.590063324
Kruskal's	1.6979382515	393.0063323976

Note: The units for the above observations is seconds.

Analysis

- On a Sparse graph G_1: Performance of Dijkstra's with heap $>$ Kruskal's $>$ Dijkstra's without heap
- On a Dense graph G_2: Performance of Dijkstra's with heap $>$ Dijkstra's without heap$>$ Kruskal's

Performance: inversely proportional to the running time of the algorithm