CSCE 629-601 Analysis of Algorithms

Project

Student Id: 731008209 *Name: Sunanda Saha*

Overview

In this project we had to visualize the solution of the maximum bandwidth path problem in three different ways using two algorithms namely - Dijkstra's shortest path algorithm and Kruskal's maximum spanning tree algorithm.

Milestone 1:

The first milestone of the project was to generate connected and weighted random graphs. As suggested by professor, to make the graph connected I started with connecting all the nodes of the graph to one another and created a cycle of all 5000 nodes. Now we had to generate two kinds of such random graphs :

- The first kind (G1) needed to have the average vertex degree as 6 (sparse): As only the average overall degree needs to be 6 (irrespective of the degree of each node) and the total number of nodes is 5000, the total number of edges in the graph need to be exactly equal to (5000 * 6) / 2 = 15000.
  I wrote a function that takes the total number of edges and total number of nodes as input. It generates two random numbers each time and creates an edge between them. It also assigns a third random number as weight for the edge. Before appending this edge to an array of edges, it checks whether the same edge is already present. If present it generates 3 random numbers again as edge vertices and weights.

- The second kind (G2) needed to have each vertex to be adjacent to about 20% of the other vertices, which needs to be randomly chosen (dense): As 20% of the total 5000 nodes is 1000, I wrote a function which takes in the number of nodes as input. For each node it generates two random numbers - one number to serve as the other end of an edge and the second number as the weight of the edge. This is done a 1000 times for every node thus making sure that each node has a degree of 1000.

Once we have our array of edges, we create an adjacency list out of it.

Milestone 2:

The next milestone of the project was to generate a Max Heap structure as expected in the assignment.
The heap structure has one integer value size to keep a track of the number of elements in the heap and three arrays :

- The heap array given by an array H[5000]. Thus, H[i] gives the name of the vertex in the graph.

- The bandwidths are given in another array D[5000]. Thus, to find the bandwidth of a vertex H[i] in the heap, we can use D[H[i]].

- The position of a vertex in the heap in another array P. Thus, P[v] is the position of vertex v in the heap H[5000].

The heap was required to perform three main functions:

- Maximum : This extracts the root element of the heap.
- Insert : This inserts a new node into at the end of the heap and then heapifies up from the last node position to it's correct position according to it's bandwidth.
- Delete : This deletes a certain vertex by swapping it's position with the last node in the heap. The last node (now in the position of the deleted vertex) heapifies down to it's correct position according to it's bandwidth.

In both the cases of insert and delete, all the three arrays were needed to be modified at every heapify step.

Implementation Details

Approach1: Dijkstra's algorithm using arrays:
The first approach included Dijkstra's shortest path algorithm with arrays.
Dijkstra's algorithm finds the shortest path from a source s to all the vertices in a given graph. Here we modified it to give us the maximum bandwidth path from a source to a particular target node. We require three arrays for this :

- The status array stores the status of every vertex namely - unseen, in-tree, fringe.
- The bandwidth array that stores the minimum bandwidth (distance) till that index on the shortest path.
- The dad array that stores the predecessor of that index on the shortest path.

We at first change the status of all the neighbouring vertices of the source from unseen to fringe. Next we iteratively keep going through the status array to check which of the nodes are currently fringes and consecutively check their bandwidths. At the end of one such iteration, we have the node with the largest bandwidth. It's status is changed to in-tree and we repeat the same process with these nodes. In the end we check if the status of target node has become in tree or not. If it has not then there exists no maximum bandwidth path. In the other scenario we get our maximum bandwidth path by backtracking in the dad array.

Approach 2: Dijkstra's algorithm using heaps:
The second approach included Dijkstra's shortest path algorithm with heaps.
It is very similar to the first approach except I created a max heap to store the fringes. Hence every time I needed the fringe with maximum bandwidth, I no longer had to traverse the entire status array and find the fringes and then pick the one with maximum bandwidth. I could easily call the maximum function which fetches me the maximum bandwidth fringe in O(1) time. Using the heap has shown real improvement in run time.

Approach 3: Kruskal's algorithm using heapsort:
The third approach included Kruskal's maximum spanning tree algorithm with Heapsort.

Kruskal's algorithm finds the maximum spanning tree in a graph but here we modify it to give the maximum bandwidth path from a source to a particular target node.
At first, we sort the edges of the graph in descending order using heapsort. Then we keep picking one edge at a time so that no cycle is formed at any given moment. Once the maximum spanning tree is formed, we choose the path from the source node to the target node on the spanning tree and thus get our maximum bandwidth path.

Performance analysis of the algorithms

Approach 1:
We see that the time complexity is $\mathcal{O}(nm)$ where **n** refers to the number of nodes in the graph. This is because with arrays when we search for the fringe with highest bandwidth, we traverse the entire status array in O(n) and after getting the fringe node we traverse all of it's neighbours in O(m) which gives us the total time complexity of $\mathcal{O}(nm)$.
Hence, as expected this algorithm runs the slowest out of the 3 on the sparse graph.

Approach 2:
Here, the Time Complexity is $\mathcal{O}(mlog(n))$ where **m** refers to the number of edges and **n** refers to the number of nodes in the graph. This is because with heaps, the insertion and deletion in the heap takes $\mathcal{O}(logn)$. However, when we search for the fringe with highest bandwidth, we do so in O(1). After extracting the fringe node we traverse all of it's neighbours in O(m) which gives us the total time complexity of $\mathcal{O}(mlog(n))$.

Approach 3:
Here, the Time Complexity is $\mathcal{O}(mlog(n))$ where **m** refers to the number of edges and **n** refers to the number of nodes in the graph. This is because, heapsort in itself takes $\mathcal{O}(mlog(n))$ which is sort of the bottleneck for this algorithm. If given a sorted graph, then Kruskal's Time Complexity improves to $\mathcal{O}(mlog^*(n))$ where $log^*(n)$ is a much slower growing function as compared to $log(n)$ and in most practical cases can be bounded by 3, thereby bumping up the algorithm to approximately linear time complexity as discussed in [1].

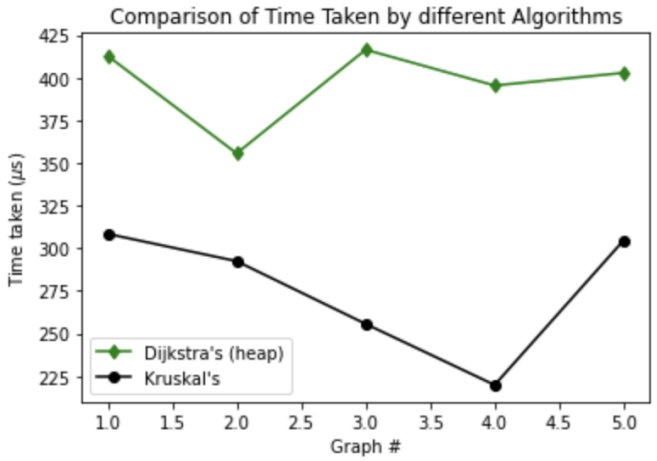
Analysis of the performance of the various algorithms

As can be seen from the tables, Kruskal's algorithm outperforms the Dijkstra's algorithm on all fronts. Also, as discussed in [1], more often than not, sorting takes approximately linear time, and since Kruskal's also has an approximately linear time complexity, the algorithm as a whole runs in approximate linear time i.e. $\mathcal{O}(m)$. On the other hand, Dijkstra's with heaps still requires $\mathcal{O}(mlog(n))$, hence Kruskal's performs better for all possible cases.

- Kruskal runs faster than Dijkstra when considered without the sorting time.
- Heapsort is a bottle neck and the implementation done here is not efficient. However, the inbuilt sorting functions are very efficient and hence when we look only at algorithm time instead of including sorting time, Kruskal performs best.

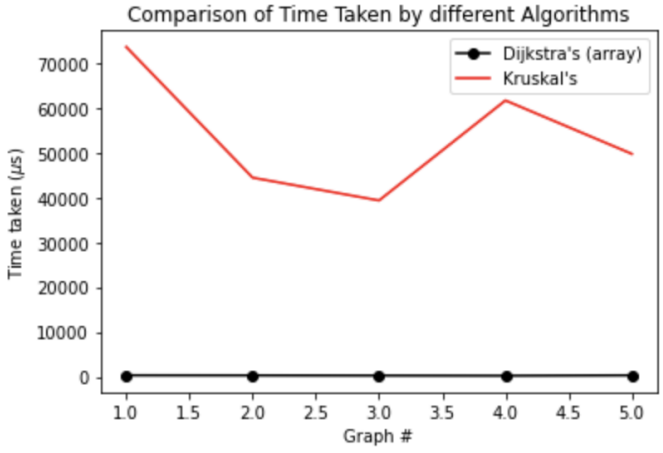
s-t Node	Dijkstra's using Array	Dijkstra's using Heap	Kruskal's	Kruskal's without sort
3548-1736	325546	2742	4796	1482
3419-4362	94037	1517	4994	1798
3696-3061	166327	1702	3063	1062
4021-3538	322198	2803	2674	1035
1487-4717	389110	1799	3390	1284
14-4096	262071	1986	3672	1671
927-2368	168107	1357	2823	1143
355-2505	115083	1913	2856	1298
433-1412	367567	1084	3616	1623
780-41	217863	2071	2875	1233
4509-684	248597	1907	3887	1478
1135-2327	154665	1511	2307	1058
908-1664	196580	3344	3592	1453
1900-461	185999	1956	2864	1131
383-1642	134597	1585	2879	1127
2395-774	414706	3396	4984	1977
4061-1652	329610	435	2774	1071
632-3621	206018	1967	2667	1012
4882-1654	356546	1738	3717	1400
3062-4813	310196	2115	2619	1019
685-714	292947	1938	3956	1332
4525-2503	314535	723	2642	1398
927-1019	122535	2360	4100	1815
286-3879	64679	989	2632	1004
54-1334	308177	1988	2587	978

Table 1: Time Analysis for Sparse Graph G1 (time is in microseconds)



s-t Node	Dijkstra's using Array	Dijkstra's using Heap	Kruskal's	Kruskal's without sort
286-3879	54.2812	0.6281	1.2872	0.5891
4061-1652	68.9673	0.8354	1.1534	0.7102
4302 - 37	35.4682	0.9345	1.6532	0.8734
355-2505	42.5204	0.3114	0.7294	0.4281
54-1334	33.9083	0.2634	0.6463	0.1795
419-4362	48.5676	0.09872	0.5487	0.1376
927-2368	65.5212	0.5689	1.6342	0.5425
1135-2327	52.7653	0.8345	1.8974	0.7099
780-41	49.8782	0.7622	1.0865	0.4019
433-1412	73.3642	0.9326	1.4742	0.7754
982 - 3921	70.3891	0.8732	1.8363	0.3718
632-3621	32.1837	0.1973	0.4329	0.0929
583 - 844	49.7362	0.7482	1.4220	0.6931
2395-774	81.0375	1.0213	1.2864	0.7936
902 - 2383	33.9583	0.3874	0.7392	0.3012
3038 - 389	57.7631	0.9744	1.5734	0.7622
402 - 3224	53.2913	0.8641	1.2301	0.4291
109 - 3628	21.2672	0.8523	1.2821	0.7673
238 - 243	47.6627	0.9875	1.9853	0.8972
192 - 4024	32.7682	0.7672	1.6863	0.7564
4892 - 3434	28.3072	0.7382	0.4718	0.9203
2 - 362	44. 0824	0.6481	1.0363	0.7293
5 - 720	68.3083	0.3427	1.3974	0.5836
309 - 4824	38.9391	0.5032	1.0205	0.4294
1028 - 428	37.3937	0.4091	1.4892	0.5829

Table 2: Time Analysis for Dense Graphs in seconds.



Potential Improvements in performance

Using Fibonacci heaps, we can reduce the time complexity of Dijkstra's to $\mathcal{O}(m + nlog(n))$ and for Kruskal the time complexity reduces to $\mathcal{O}(m+nlog(n))$. It is important to note that, Fibonacci heaps perform deletion in $\mathcal{O}(log(n))$ and other operations in $\mathcal{O}(1)$. However both these are amortized times as mentioned in [2] which means that only in very rare occurrences, does the algorithm perform worse than the above mentioned times for the respective operations. Thus, looking at the bigger picture, even though there's some complications associated with using Fibonacci heaps it might be worthwhile to try incorporating it.

1 References

1. A note on practical construction of maximum bandwidth paths. Inf. Process. Lett. 83(3): 175-180 (2002)

2. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms," 25th Annual Symposium onFoundations of Computer Science, 1984., 1984, pp. 338-346, doi: 10.1109/SFCS.1984.715934.