CS201 Final Lab Report

Entry Number- 2019CSB1085

Name- Ganesh Aggarwal

The objective of the lab was to implement the **Johnson's Algorithm** for finding shortest distance between all pair of nodes. The algorithm requires **Bellman Ford Algorithm** and **Dijkstra Algorithm** as its subparts. Dijkstra algorithm was implemented using 4 different kind of data structures for the **decrease key operation** and then the corresponding time complexities were analyzed. **Array, Binary heap, Binomial heap and Fibonacci heap** are the 4 different data structures used.

Johnson's Algorithm

The algorithm is designed to solve the problem of finding the shortest distance between all pair of nodes in a graph (directed or undirected), where each edge has some weight (positive or negative). The algorithm also reports if there is a negative weight cycle in the graph. The logic used is to use Dijkstra Algorithm N times, where N is the number of nodes in the graph but with some modification in the weight matrix before hand so that negative edge weights can be handled as Dijkstra Algorithm is only for positive edge weights.

In short, the algorithm can be described as:

1. Add a new node in the given graph and put **0 weight edges** from this node to all other nodes present in the graph. Now with this new node as the source apply **Bellman Ford Algorithm** and store the distances in a array named **'h'**.
2. Now, for all edges (x,y) that are present in the original graph (before adding the new node), assign the weight of edge (x,y) as follows: **w(x,y) = w(x,y) + h[x] -h[y].** Remove the newly added vertex and all the edges that start from it.
3. The above mentioned updating of edge weights will ensure that all edge weights are now positive. Now with each vertex as source, apply **Dijkstra Algorithm**. In this way, after considering each vertex as source we will have the shortest distance from each vertex to any other vertex. Let's assume it is stored in 2D matrix **'dist'**.
4. Now, in the end to nullify the effect of h, the distance from u to v is updated as **dist[u][v]=dist[u][v]+h[v]-h[u]**.

If V is the number of vertices and E is the number of edges, then time complexity for the above algorithm is **O (V*E + V*(time complexity of Dijkstra))**. **V*E** is for the bellman ford algorithm that is executed once. The **V*(time complexity of Dijkstra)** is because Dijkstra is executed V times. I have implemented Dijkstra using 4 different data structures so the total time complexity of the program will vary accordingly.

The 4 different implementations of Dijkstra are as follows:

Array Based: This is the most basic approach as it doesn't use any complex data structure. To find the vertex with the minimum distance, every time we iterate over the entire distance array and find the one with the minimum one. Also, to avoid taking a vertex twice, an array is used that keeps track of the vertices that have already been considered and while finding the minimum distance vertex if we come across such a vertex, we just skip it. The decrease key here is very simple as we just need to change the value in the distance array **(O(1)**). The extract min or more precisely find min in this case is done in **O(V)** as every time we are iterating over the entire distance array. The total time complexity for Dijkstra Algorithm in this is given by **O(V^2+E).**

Binary heap based: In this, initially all the nodes except source are inserted in the heap with distance value infinity and source is inserted with distance value 0. Then in each iteration, we extract the minimum distance node from the heap and then all edges that start from this node are relaxed if required using the decrease key operation. Complexity of extract min is **O(log V)** and decrease key is **O(log V)**. Total complexity of the Dijkstra algorithm in this approach comes out to be **O(V*logV+ E*logV).**

Binomial heap based: This approach is almost the same as binary heap, the only difference is in the way heap is stored which makes it a bit faster than the binary heap, as the union operation in binomial heap is relatively better than the one compared to binary heap. The complexity of extract min and decrease key remains the same as binary heap and hence, the total time complexity of the Dijkstra Algorithm in this case comes out to be **O(V*logV + E*logV)**.

Fibonacci heap based: In this approach, the main focus is to improve the complexity of the decrease min operation. To ensure that, the idea used is that consolidation is done only in the extract min operation. This reduces the complexity of decrease key as it can now be done in **O(1)** as we just have to make a few cuts now. This approach is referred to as the **lazy way** of doing operations. Thus, due to the improved time complexity of decrease min, the total complexity of Dijkstra Algorithm comes out to be **O(V*logV + E).**

Therefore, we can say that the theoretical time complexity of the Dijkstra Algorithm will be best in the **Fibonacci heap** based implementation as it provides very good time complexity for both the extract min and decrease key operation.

Execution Time Analysis

I tried to find out the average run time for different values of V, where V is the number of nodes in the graph. I have done this by setting up V at a particular value and then taking random graphs with a fixed value of density for edges and then taking the average of all the time used in each case. This way, I have found out the average runtime for a fixed V and fixed density of edges.

Implementation Type	Nodes/Probability	0.01	0.20	0.50	0.75
Array	**250**	0.04688	0.14688	0.21875	0.29063
Binary heap		0.02813	0.12500	0.18438	0.24688
Binomial heap		0.02813	0.11563	0.16875	0.23750
Fibonacci heap		0.04375	0.14063	0.18750	0.25625
Array	**600**	2.18750	5.43750	9.89062	14.45310
Binary heap		0.73438	3.84375	7.63542	11.82810
Binomial heap		0.65625	3.68750	7.31250	10.68230
Fibonacci heap		1.01562	4.04688	7.71354	10.61980
Array	**1000**	10.96350	24.34900	46.23440	71.38020
Binary heap		2.39583	14.12500	31.46880	56.32810
Binomial heap		2.25000	13.18750	30.39580	55.58330
Fibonacci heap		3.12500	15.80730	31.40620	56.77600

The above table shows the runtime (in seconds) for different values of V and density of edges. I have considered 3 different values of V (250, 600 and 1000) and 4 different values for density of edges (0.01, 0.2, 0.5, 0.75).

Observation

For small values of V(250), the difference between different types of implementation is not reflected much. All the data structures seem to take almost equal time with binomial heap being slightly better than the rest of all. As V is increased from 250 to 600, the time taken increases considerably for all data structures irrespective of the probability of edges. Now, the difference between the different ways of implementation seems to become significant. Array implementation takes much more time than the rest of the three. The other three of them take almost similar amount of time for all probabilities but Fibonacci heap takes a little more time than binary heap and binomial heap. The difference in time taken by binary heap and binomial heap is negligible and can be ignored. As we increase V from 600 to 1000, the time taken increases further more in all the cases. For such large graphs, the difference between array and the rest of the three is very significant. In this case also binomial heap outperforms all the other heaps.

Conclusion

We can conclude that array is the least effective as it is the slowest among all. This matches the theory as in theory the array implementation should be the slowest because of extract min which is O(V). The other three heaps take almost similar amount of time. Also, as the density of the graph increases, the time taken increases no matter what the value of V is and which implementation is used.

In practice Fibonacci heap turns out to be slower than binomial heap and binary heap although theoretically it should be faster than binomial heap. This can be due to the fact that although theoretical complexity for Fibonacci heap is less but the constant factors are very high because a lot of pointers are being changed in every operation. Since the constant factors in binomial heap and binary heap are comparatively small, they turn out to be slightly better than Fibonacci heap in practice.

Also, looking at the slightly better performance of binomial heap as compared to binary heap, we can infer that it is because of the fact that in decrease key operation while percolating up, the height would be much smaller in binomial heap for a lot of cases as compared to binary heap as the trees in binomial heap are divided where as in binary heap there is only one single tree. So, this might be the reason for more efficient performance of binomial heap.