CS201 LAB-6 REPORT

Name: Vanshal Singh

Entry Number: 2019CSB1129

The report is based on analysis of **Johnsons's Algorithm**, in which we use **Bellman-Ford** and **Dijkstra's Algorithm** as subroutine, and implement Dijkstra using 4 different data structures which are:-

- Arrays
- Binary Heap
- Binomial Heap
- Fibonacci Heap

Johnson's Algorithm

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in an edge weighted, directed graph. It can handle negative edge weights, but negative weight cycles must not exist. We use Bellman-Ford Algorithm to reweight the edges such that all the edge-weights become non-negative, and then implement Dijkstra's Algorithm on the modified graph.

A small step-by-step description of the algorithm is as follows:-

1. We add a new node '*n*' to the graph, and connect it to all the other nodes in the graph with zero edge weights.

2. Now, we will use Bellman-Ford Algorithm, and find the minimum weight *h[v]* for a path from node '*n*' to a node '*v*'. If a negative cycle is detected, the program will be terminated.

3. Now, the values of the edge-weights will be recomputed using the values calculated in Bellman-Ford Algorithm: an edge from *u* to *v*, having edge weight *w(u,v)*, is given the new edge-weight *w(u,v) + h(u) - h(v)*.

4. Finally, the node '*n*' is removed, and Dijkstra's Algorithm is used to find the shortest paths from each node '*s'* to every other vertex in the modified graph. The distance in the original graph is then computed for each distance *D(u,v),* by adding *h(u)-h(v)* to the distance returned by Dijkstra's Algorithm.

Time Complexity: *V*(Time Complexity of Dijkstra) + (Time Complexity of Bellman Ford)*

Data Structures use in Dijkstra's Algorithm

Given below is a brief description of the implementations of the data structures used in Dijkstra's Algorithm:-

1. **Arrays:** Here, a path array is used, along with a boolean visited array, to check if the current node is the the solution set or not. Now, we find the minimum weight path not in the solution set, and we relax the edges, and update the path array. We can see the updating of the distance array analogous to the decrease key function, which is done in *O(1)* time, and find min can be viewed analogous to finding the minimum non-visited element in the array, which is *O(V).* Hence, the final time complexity for array implementation of Dijkstra is ***O(V^2 + E)***.
2. **Binary Heap:** Here, we add all the nodes in the binary heap with their path value, and in every iteration, we use extract min to find the minimum unvisited element from the heap, which is done in *O(log V),* and we relax the edges using decrease key, which is done in *O(logV)*. Hence, the final time complexity for binary heap implementation of Dijkstra is ***O(V*log(V) + E*log(V))***.
3. **Binomial Heap:** Here, we add all the nodes in the binomial heap with their path value, and in every iteration, we use extract min to find the minimum unvisited element from the heap, which is done in *O(log V),* and we relax the edges using decrease key, which is done in *O(logV)*. Hence, the final time complexity for binary heap implementation of Dijkstra is ***O(V*log(V) + E*log(V))***.
4. **Fibonacci Heap**: Here, we do consolidation (changing the heap such that no two trees of the same rank will exist), which will decrease the complexity of the heap. The time complexity of extract min is *O(log(V)),* and relaxation of edges using decrease key will take *O(1)* time. Hence, the final time complexity for fibonacci heap implementation of Dijkstra is ***O(V*log(V) + E)***.

Analysis of Time Complexities

Here is a table showing the time complexities for different operations on the three heap data structures we are observing:-

OPERATION	BINARY HEAP	BINOMIAL HEAP	FIBONACCI HEAP
Insert	*O(log N)*	*O(log N)*	*O(1)*
Find-min	*O(1)*	*O(log N)*	*O(1)*
Delete	*O(log N)*	*O(log N)*	*O(log N)*
Decrease-key	*O(log N)*	*O(log N)*	*O(1)*
Union	*O(N)*	*O(log N)*	*O(1)*

Analysis of implementation of different data structures in Dijkstra's Algorithm on Graphs

Here is a table consisting of time taken(in seconds) by the four data structures for the graphs consisting of combinations of (150,250,400) nodes and (2500,5000,7500,10000) edges **(12 Graphs)**.

Nodes/Edges	10000	7500	5000	2500	
150	0.41613	0.340606	0.25781	0.147142	**Array**
	5.92314	5.95726	5.97912	6.03408	**Binary**
	0.21266	0.16714	0.128629	0.0809	**Binomial**
	0.064782	0.05287	0.034477	0.019179	**Fibonacci**
250	1.17334	0.960471	0.7004	0.384885	**Array**
	10.4017	10.2621	10.73	10.3686	**Binary**
	0.370583	0.311341	0.241173	0.165957	**Binomial**
	0.073697	0.05877	0.040177	0.024854	**Fibonacci**
400	3.10537	2.52779	1.7403	0.870083	**Array**
	16.6939	16.5789	16.6774	16.3328	**Binary**
	0.725699	0.587648	0.483339	0.329323	**Binomial**
	0.08933	0.074934	0.05643	0.038482	**Fibonacci**

Here, we can see that for small graphs (150 to 400) nodes, there is not much difference in the time taken by array and binomial are close, but binomial is more efficient than array, which is expected. Fibonacci Heap was not working properly, which is reflected here in the values of time, and binary heap is taking unexpected time.

So, we can conclude that binomial heap is better than array, and as the size and density of the graph increases, the difference becomes more and more prominent.

For Binary heap and fibonacci heap, we are unable to conclude results due to unexpected ambiguities in the code.

Here are the values of time taken for a large graph, i.e it has 1000 nodes and 40000 edges.

Array: 61.825 seconds

Binary Heap: 48.555 seconds

Binomial Heap: 6.369 seconds

Fibonacci Heap: 0.432 seconds

Here also, we can see that fibonacci heap is not working properly, hence giving ambiguous time values, and binary heap is taking little more time than expected, but it is still faster than the array implementation of Dijkstra.

Binomial heap is performing very well as compared to array, as discussed above.

These kind of values are obtained because for Array, the time complexity of Dijkstra is *O(V^2 + E)* for the binary heap and binomial heap is *O(V*log(V) + E*log(v))*, so for small graphs, we can say that V^2 ~ VlogV , so the time taken for the array will be similar to these two heaps. But as the size of graph increases, V^2 >> V*logV, and thus we get such prominent difference in time. In case of heaps, binomial heap is faster than binary heap, as there is better division of trees as binomial trees in binomial heap, the percolate up will be faster than that of binary heap. Fibonacci Heap will be the fastest, as the theoretical complexity of fibonacci heap is *O(V*log(V) + E),* which is the best among all.

Conclusion

So we can see that for small graph, the time taken for the data structures is almost same, but as we increase the number of nodes and the density of the graph, the difference becomes more and more prominent.

Using the theoretical evidences, we can still claim that if the fibonacci heap would have been implemented properly, it may have given us the best time taken for the above graphs, as it the most efficient.

Therefore, we can conclude that the efficiency of these data structures is not much different for small graphs, but for large graphs, the efficiency order can be claimed as:-

Fibonacci Heap > Binomial Heap > Binary Heap > Arrays