CS201 LAB-6 REPORT

Name : Chiranjiv Dutt
Entry Number : 2019CSB1083

❖ AIM:

To do analysis of implementations of four different types of heap data structure:

1. Array-based implementation
2. Binary heap
3. Binomial heap
4. Fibonacci heap

in Johnson's Algorithm for all pair shortest path, which is composed of two sub-components mainly: Bellman-Ford Algorithm and Dijkstra Algorithm for single source shortest path.

❖ JOHNSON'S ALGORITHM:

Johnson's Algorithm comprises of following steps :-

1. Add a new vertex to the given graph and add edges from new vertex to all existing vertices of given graph.
2. Run Bellman-Ford Algorithm on updated graph with newly added vertex taken as source. Return if negative-weight cycle is found.
3. Now reweight all edges of original graph as 'original weight + h[u] – h[v]' ; where h[s] is shortest distance of vertex s from new added vertex, and u is source vertex and v is destination vertex of that edge which is being considered.
4. Remove new added vertex and apply Dijkstra's Algorithm for all original vertices.

Runtime of algorithm :-

Runtime of Johnson's Algorithm is sum of runtime of Bellman-Ford Algorithm and N times runtime of Dijkstra's Algorithm; where N is number of vertices in the given graph. Runtime of Bellman-Ford Algorithm is O(MN); where M is number of edges and N is number of

vertices in the given graph. Therefore, total runtime of Johnson's Algorithm is O(MN + N*(runtime of Dijkstra)).

Runtime of Dijkstra depends upon the data structure which is used for its implementation.

❖ DIJKSTRA'S RUNTIME:

Runtime of Dijkstra's Algorithm is given by O(M*(decrease key operation / relax operation) + N*(extract min operation)) as all reachable edges are relaxed at least once and all nodes are extracted once to be included in the solution set, while determining single source shortest path.

1. ARRAY-BASED IMPLEMENTATION :
   decrease key operation : O(1) ; involves changing value at a particular index.
   extract min operation : O(N) ; as we have to traverse whole array of distances from source corresponding to every vertex to find minimum and extract it.
   Hence, Dijkstra's runtime is O(M + N^2) and correspondingly Johnson's Algorithm's runtime will be O(MN + N^3).

2. BINARY HEAP BASED IMPLEMENTATION :
   decrease key operation : O(log(N)) ; involves calling of percolate up function of binary heap.
   extract min operation : O(log(N)) ; involves calling of percolate down function of binary heap after extracting root node.
   Hence, Dijkstra's runtime is O(M*log(N) + N*log(N)) and correspondingly Johnson's Algorithm's runtime will be O(MN*log(N) + N^2*log(N)).

3. BINOMIAL HEAP BASED IMPLEMENTATION :
   decrease key operation : O(log(N)) ; involves going to a particular binomial tree of binomial heap and percolating up.
   extract min operation : O(log(N)) ; involves removal of minimum from root list, appending its children in another heap and taking union of the two heaps formed.
   Hence, Dijkstra's runtime is O(M*log(N) + N*log(N)) and correspondingly Johnson's Algorithm's runtime will be O(MN*log(N) + N^2*log(N)).

4. FIBONACCI HEAP BASED IMPLEMENTATION :
   decrease key operation : O(1) Amortized ; involves cutting of particular heap node and appending it into root list and recursively repeating the same for its parent depending upon its mark (uses cutting and recursive_cutting functions in implementation).
   extract min operation : O(log(N)) ; involves removal of minimum value node and appending its children into root list, and consolidating the heap which involves linking of nodes in root list having same rank.

   Hence, Dijkstra's runtime is O(M + N*log(N)) and correspondingly Johnson's Algorithm's runtime will be O(MN + N^2*log(N)).

❖ RUNTIME ANALYSIS OF DIJKSTRA ON SOME RANDOM GRAPHS:

Some random graphs were generated with different number of edges and vertices on which Dijkstra algorithm is applied and its practical runtime is observed and analyzed. For different heap implementations its runtime (in seconds) is noted with changing number of vertices and edges in following tables:-

		Number of edges			
		2500	**5000**	**7500**	**10000**
Heap type	**Number of nodes**				
Array	**100**	0.0002	0.0002	0.0002	0.0002
Binary heap	**100**	0.001	0.0011	0.0012	0.0014
Binomial heap	**100**	0.0011	0.0012	0.0014	0.0016
Fibonacci heap	**100**	0.0002	0.0003	0.0003	0.0004
Array	**250**	0.0003	0.0004	0.0004	0.0004
Binary heap	**250**	0.0012	0.0015	0.0016	0.0018
Binomial heap	**250**	0.0014	0.0014	0.0017	0.0018
Fibonacci heap	**250**	0.0003	0.0003	0.0004	0.0005
Array	**500**	0.0012	0.0012	0.0014	0.0014
Binary heap	**500**	0.002	0.0021	0.0023	0.0024
Binomial heap	**500**	0.0022	0.0023	0.0024	0.0026
Fibonacci heap	**500**	0.0011	0.001	0.0014	0.0013

Table 1

		Number of edges			
		20000	25000	50000	70000
Heap type	Number of nodes				
Array	1000	0.0059	0.0062	0.0063	0.0063
Binary heap	1000	0.008	0.008	0.0096	0.0098
Binomial heap	1000	0.0085	0.0085	0.0092	0.0101
Fibonacci heap	1000	0.007	0.007	0.0075	0.0075
Array	2500	0.0209	0.021	0.023	0.0225
Binary heap	2500	0.023	0.022	0.027	0.028
Binomial heap	2500	0.025	0.0255	0.027	0.0281
Fibonacci heap	2500	0.0181	0.0182	0.0201	0.0205
Array	3000	0.0311	0.0322	0.0325	0.033
Binary heap	3000	0.031	0.031	0.033	0.033
Binomial heap	3000	0.032	0.0322	0.034	0.0335
Fibonacci heap	3000	0.0288	0.029	0.031	0.031

Table 2

//Comment for invigilator: My Fibonacci heap based implementation was not working or working partially in my submitted program. I made some minor changes in my code later. It was not still working satisfactorily but by observing its partial performance and runtime I filled the above two tables.

❖ OBSERVATIONS:

For graphs with smaller number of vertices, even if number of edges are comparatively higher, most efficient method seems to be **array-based implementation** of Dijkstra. This observation can be expected as array-based implementation of Dijkstra takes O(M+N^2) time and for small graphs M and N^2 are comparable. Also, if we change number of edges keeping number of vertices constant, a very minor change is observed in runtime of array-based implementation. But if we consider large graphs with more nodes and vertices, runtime of array-based implementation keeps on increasing rapidly with increase in number of nodes which makes this type of implementation to lag behind than rest of the implementations, making it worst for large graphs.

For small graphs, **Fibonacci heap based implementation** also seems to be working nicely, somewhat slower than array-based implementation but faster than other two heap based implementations. Since I have taken dense small graphs use of decrease key operation is

quite frequent. For both array-based implementation and Fibonacci heap based implementation decrease key operations takes O(1) time as discussed already, which makes it faster than other implementations. For larger graphs also, Fibonacci heap based implementation seems to be fastest with best running time of O(M + N*log(N)) amongst the four different types of implementations. In spite of its best complexity among the four implementations, its practical running time is not that much excellent due to high constant factor hidden in its runtime order.

For small graphs, **binary heap based implementation** and **binomial heap based implementation** are taking more time than that of using array or Fibonacci heap. For large graphs, both are taking more time than Fibonacci heap based implementation but lesser time than array-based implementation. Both of them take O(M*log(N) + N*log(N)) time.

❖ CONCLUSION:

For small graphs with less number of vertices, array-based implementation of Dijkstra and Johnson's Algorithm will be most suitable as it takes least runtime. If the graph is small as well as dense, this will perform much better than other implementations as decrease key operation is also cheaper.

For large graphs having a higher number of vertices, Fibonacci heap based implementation seems to be the most suitable with least runtime though practical performance is not as good as theoretically expected performance due to large constant factor hidden in its runtime order. If the graph is large as well as dense, this will perform much better than other implementations as decrease key operation is also cheaper.