CS201-DATA STRUCTURES AND ALGORITHMS

JOHNSON'S ALGORITHM-LAB06

By - Shanu Singroha(2019MEB1294)

Introduction

Johnson's Algorithm is used to find the shortest distance between any two nodes present in a graph. It is faster than the Floyd-Warshall algorithm which uses a dynamic programming approach.

Johnson's Algorithm uses Bellman-Ford and Dijkstra algorithms as its subparts. So the overall complexity reduces from O(V^3) to O(V^2logV + VE).

Basically, a graph may contain negative edges, and applying Dijkstra directly to a vertex may give misleading results as in Dijkstra all edges must be positive. So we need to reweight all the edges so that each of them is positive and then Dijkstra can be applied to each vertex for minimum distance between any two vertices.

We add a new node to the original graph and connect to each vertex of the original graph, and assigned 0 weight to each new edge added. And apply Bellman to the new vertex to get the weight of each vertex.

Finally, we will reweight each edge(u,v) as new weight = initial weight + mindistance[u] - mindistance[v].

Now all the edges are positive and we can apply Dijkstra.

And to get the minimum element we can apply different heaps structure and analyze each one of them separately.

1-> ARRAY

2-> BINARY HEAParray

3-> BINOMIAL HEAP

4-> FIBONACCI HEAP

ARRAY IMPLEMENTATION

For Array implementation of Dijkstra, we need operations to find the minimum element and extract it and decrease key function. For finding the minimum I did the O(n) implementation to look upon the complete array.

To decrease the key it was O(1) as I was storing the key minimum distance in a two-dimensional array. As the number of nodes is increasing we can see from the plot that the time taken to do Johnson's algorithm is increasing rapidly. As for large values of nodes in the graph, we may need to repeat the algorithm multiple times for each node thus increasing the overall complexity of the algorithm.

Time complexity Dijkstra -> N*(minextract) + E*(Decrease key)

= O(N^2)

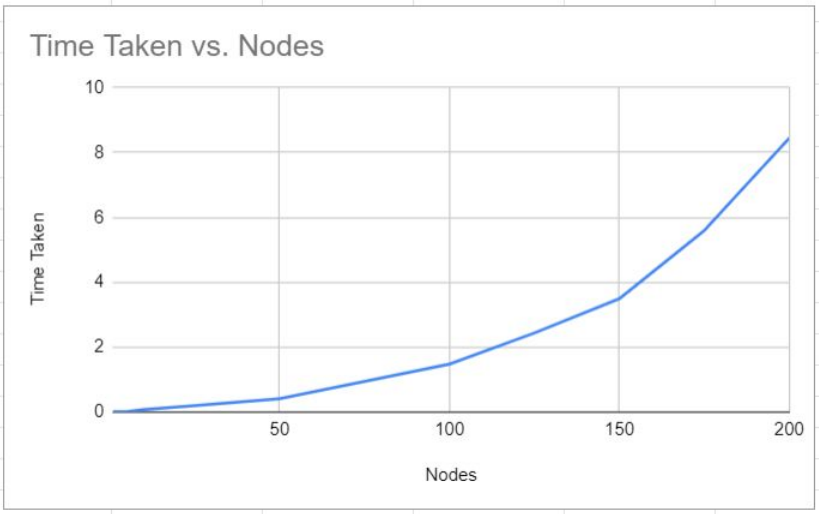
Thus we need some better implementation and for that, we will be using binary heaps.

We can observe from the plot that it is parabolic and it is correct as analysis gave complexity of O(N^2).

Array

Plot of Nodes vs Time Taken

Nodes	Time Taken
1	0.001
5	0.006
10	0.072
25	0.191
50	0.408
100	1.472
125	2.43
150	3.49
175	5.59
200	8.44



BINARY HEAP IMPLEMENTATION

For Binary Heap implementation of Dijkstra, we again need operations to find the minimum element and to decrease some particular key of node function. Finding the minimum binary heap takes O(1) whereas the extractmin and decreasekey function both take O(log n) time.

To decrease the key it is O(log n) as we need to percolate up the new key to the root of the heap. As the number of nodes is increasing we can see from the plot that the time taken to do Johnson's algorithm is increasing rapidly.

Overall Time Complexity -> N*(minextract) + E*(Decrease key)

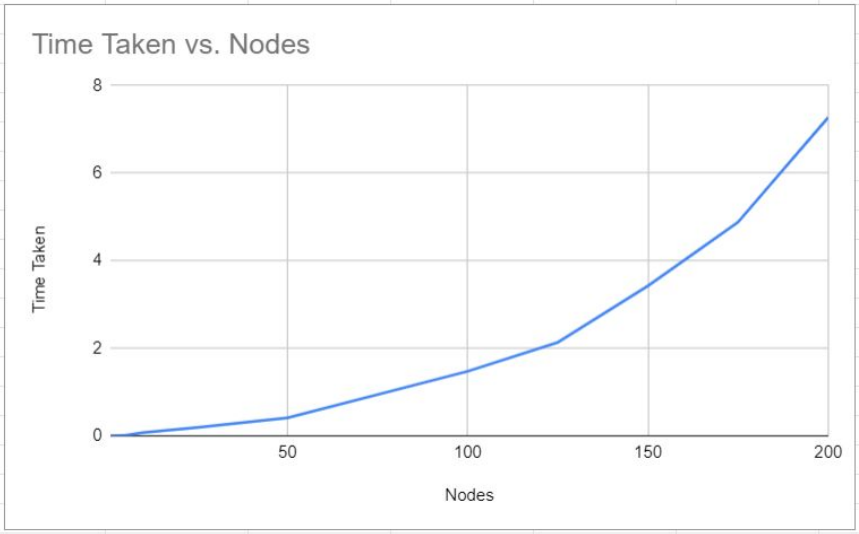
-> N*(O(log n)) + E*(O(log n))

Now depending upon whether the graph is sparse or dense we may have O(N*log N) in sparse and O(N^2*log N) in dense graphs.

Below we have a plot performed on my code of binary heap.

Next, we will use the binomial heap.

Binary Heap	
Plot of Nodes vs Time Taken	
Nodes	Time Taken
1	0.001
5	0.006
10	0.072
25	0.191
50	0.408
100	1.472
125	2.13
150	3.425
175	4.88
200	7.27



BINOMIAL HEAP IMPLEMENTATION

For Binomial Heap implementation of Dijkstra, we require operations to find the minimum element, to also extract it, and to decrease some particular key of node function. Finding the minimum binomial heap takes O(log n), the extractmin and decreasekey function also takes O(log n) time.

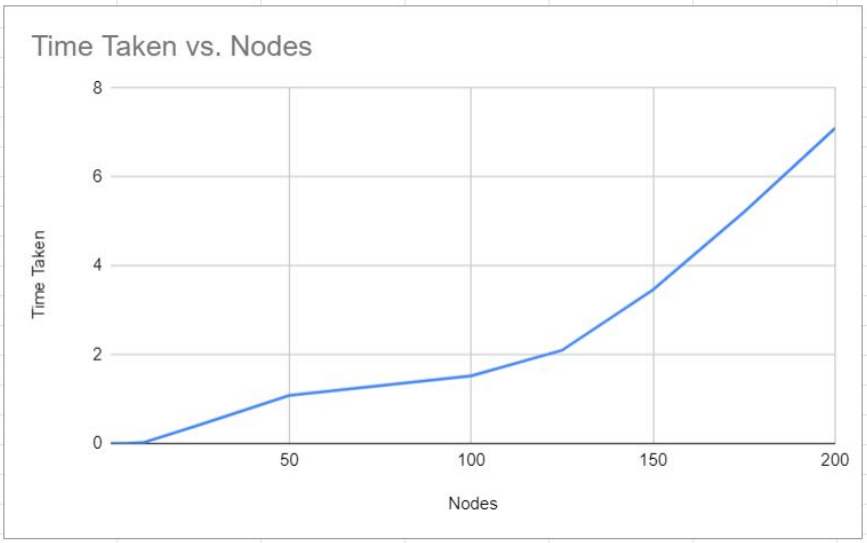
To decrease the key it is O(log n) as we need to percolate up the new key to the root of the heap. As the number of nodes is increasing we can see from the plot that the time taken to do Johnson's algorithm is increasing rapidly. For large values of nodes in the graph, we may need to repeat the algorithm multiple times for each node thus we need some better implementation.

Overall Time Complexity -> N*(minextract) + E*(Decrease key)

-> N*(O(log n)) + E*(O(log n))

Now depending upon whether the graph is sparse or dense we may have O(N*log N) in sparse and O(N^2*log N) in dense graphs. Below we have a plot performed on my code of binomial heap.

Binomial Heap	
Plot of Nodes vs Time Taken	
Nodes	Time Taken
1	0
5	0.004
10	0.018
25	0.408
50	1.08
100	1.523
125	2.1
150	3.46
175	5.21
200	7.1



FIBONACCI HEAP IMPLEMENTATION

Now finally we will use Fibonacci heaps that cost O(1) for finding minimum and decreasing the key function.

For Fibonacci heap implementation of Dijkstra, we require operations to find the minimum element, to also extract it, and to decrease some particular key of node function. Finding the minimum in the Fibonacci heap takes O(1), the extractmin takes O(log N) and decreasekey function takes O(1) time.

To decrease the key it is O(1) as we need to change some pointers only. As the number of nodes is increasing we can see from the plot that the time taken to do Johnson's algorithm is increasing rapidly.

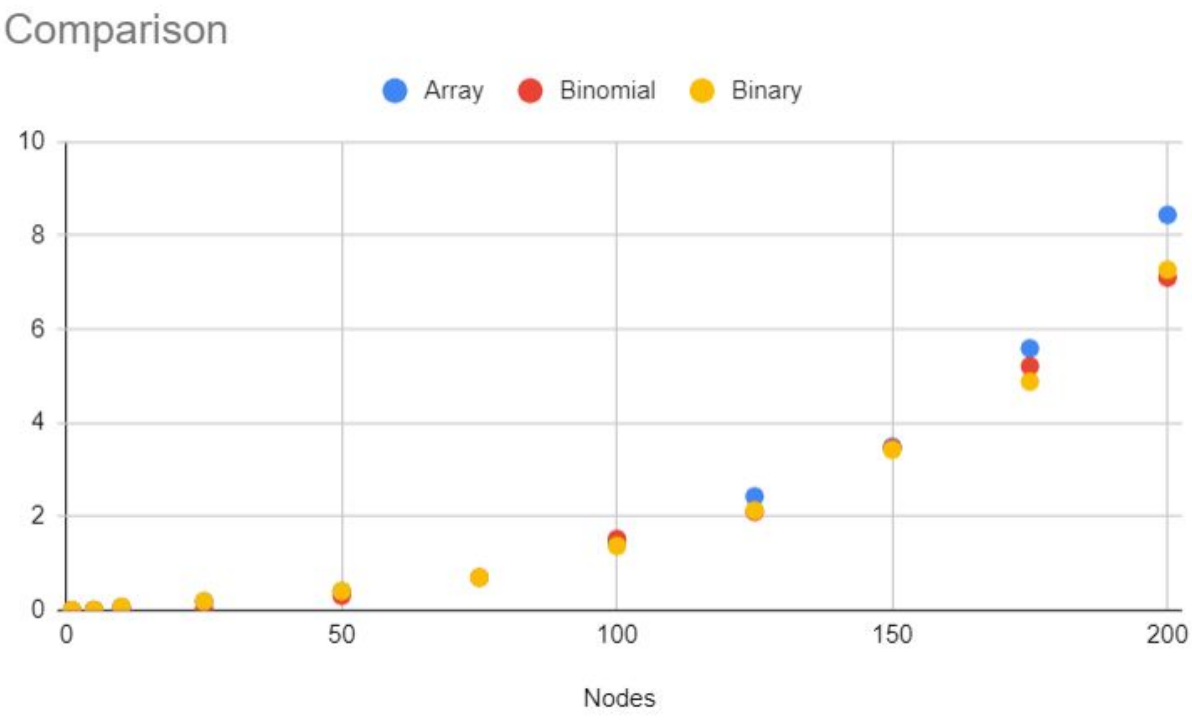
Time Complexity -> N*(minextraction) + E*(Decreasekey)

-> N*(O(log N)) + E *(O(1))

-> O(N^2) { in highly dense graph}

Now depending upon whether the graph is sparse or dense we may have O(N*log N) in sparse and O(N^2) in dense graphs.

In the case of graphs with a large number of nodes having a substantial number of edges the Fibonacci-based implementation may give a huge boost to the complexity as O(N*logN) is highly efficient.



We can observe that as the number of nodes in the graph increases the difference is clearly visible that the array is taking a lot more time than the other two heaps.

We can conclude that for graphs with large nodes time taken to execute the algorithm is very less for binomial and Fibonacci heaps compared to the time taken by arrays and binary heap.

Time taken to execute-> array > binary heap ~ binomial heap > fibonacci heap.

So we can conclude that array takes a lot of time in all cases whereas we prefer binomial heaps as it has amortized cost of insert O(1) and minimum element can be found in O(log N) time.