Johnson's Algorithm Using Different Priority Queues

Shivang Datta

INTRODUCTION

The Johnson's algorithm is an algorithm that is used to find the shortest distance from all vertices to every other vertex. It uses two famous algorithms to do so, it first uses bellman ford algorithm only once on the entire graph by adding another new node in the graph (this is done because Dijkstra algorithm only works on positive edges, hence using this process each edge is made positive). After that we perform Dijkstra from each node to find the shortest distance from that particular node to every other node, hence Dijkstra is performed v (where v is the number of vertices in the given graph) times while bellman ford is performed only once.

The time complexity of Dijkstra depends heavily on the kind of priority que deployed in it, with the extensive use of decrease key operations (takes place e times where e is the number of edges in the given graph) and the extract min operations (takes place v times where v is the number of vertices in the given graph), one has to find and deploy a data structure in which the complexity of these given operations is quite low.

THEORETICAL ANALYSIS

Arrays: -

Theoretically before extracting the min (after all the decrease key operations have been performed) the array needs to be sorted and hence this leads to a sorting complexity of vlog(v) (before each extract min), hence running of one Dijkstra algorithm would cost us $v^2\log(v)$, this is because the extract min operation is performed v times in the Dijkstra algorithm. Here the decrease key operation takes place in O (1) since all we need to do is decrease the key of the element in the array (hence adding an extra e term in the complexity since decrease key is called e times). This makes the total complexity of the Dijkstra algorithm to be $v^2\log(v) + e$.

Binary Heap: -

This data structure is much more efficient than arrays and in both decrease key and extract min operations it takes a time of log(n) where n is the number of elements in the heap. Hence the Dijkstra algorithm complexity when using binary heaps is $(v+e)\log v$, this is because the operation extract min is called v times and the operation decrease key is called e times, where e and v have their usual meanings.

Binomial Heap: -

This is quite an intricate data structure, implementing it takes much more care than in binary heap or arrays, this however doesn't give gains in terms of complexity, however the implementation of this heap is very important to understand so as to completely analyse and understand the working of the Fibonacci heap. Hence its performance is very similar to that of the binary heap, only differing in the constants of multiplication-> (v+e)logv.

Fibonacci Heap: -

The amortized cost for extracting the minimum element in the Fibonacci heap is logn where n is the number of elements in the given heap. And the cost of decreasing the key of an element is O(1), this data structure improves the complexity by performing the clean up / consolidate operation only when the minimum in the heap is extracted and not in between, hence by performing consolidate not at each step , but only along with the extract min operation Fibonacci heap successfully reduces the amortized time complexity of different operations , hence the total complexity of the Dijkstra algorithm using amortized analysis is vlogv + e .This is because decrease key is called e times and extract min is called v times.

Comparing Fibonacci heaps with binomial and binary heaps , the performance boosts provided by the Fibonacci heaps would be more and more clearer when using dense graphs , that is to say graphs with more edges(larger e values) would be faster when Fibonacci heaps are used, this is because in the case of Fibonacci heaps the term with e is not multiplied with logv but that is not the case with binary and binomial heaps(the complexity of binomial heap (e+v)logv, the complexity of the Fibonacci heap vlogv + e)

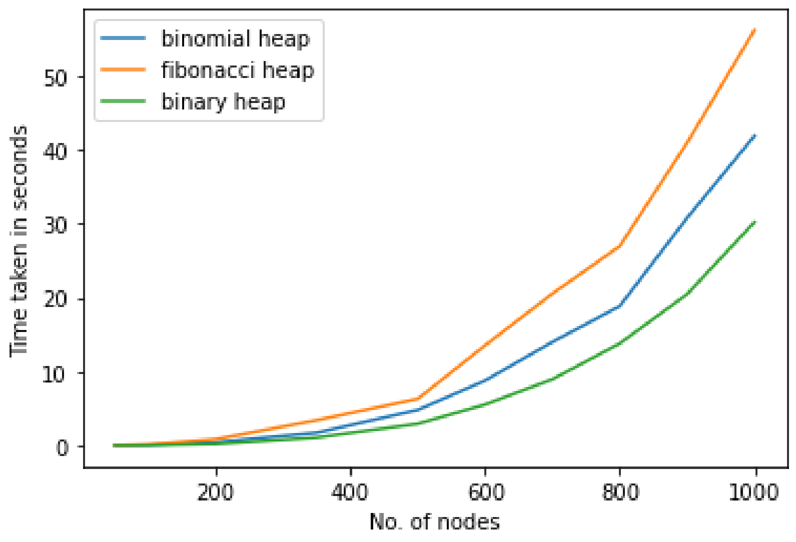
Note that the theoretical analysis done above doesn't guarantee that a particular priority que would work better for any given input, all it guarantees is that for extremely large inputs of the order of billions ,one would see that the heaps with the better complexity would tend to perform better , for smaller inputs this would not be the case because then the multiplication constants would dominate the expression of the complexity due to which even the algorithms with higher complexity would perform better. For example $(100*x)>x^2$ for smaller values of x, however this is not the case for extremely large values of x.

PRACTICAL ANALYSIS (using graphs, large and Random, up to 1000 nodes): -

All the random generated graphs used over here are non-negative so as to avoid any unwanted negative weighted cycle that may be introduced, because the shortest path is not defined for graphs containing negative weighted cycles.

Not dense graphs: -

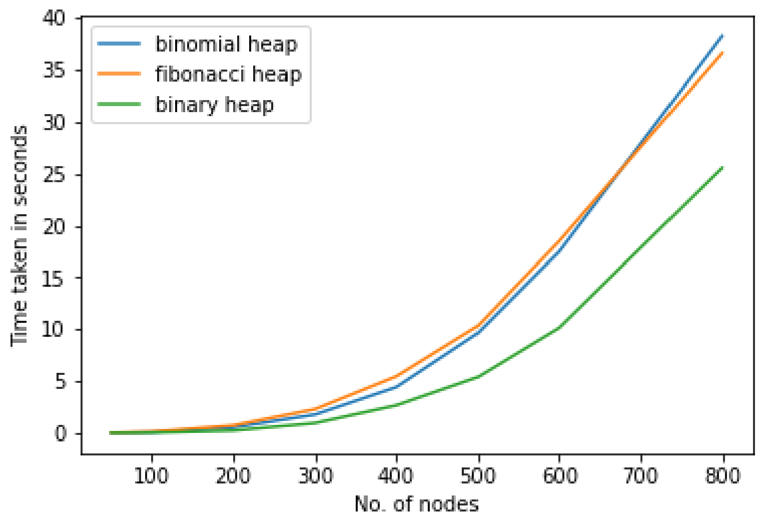
Shown below is the curve between the number of nodes and the time taken for the code to completely run, here the graph used is not so dense (a 1/2 probability whether there is an edge between 2 given vertices or not). A python code written by me was used to generate a 2d adjacency matrix of the graphs and then those were given as input and another python code was use to plot the graphs below.



And as predicted in the theoretical discussion performed above, due to large constants the Fibonacci heap takes more time to run than the binomial and the binary heap, since the graph is not dense at all and the term e is small, the vlogv term is the dominant one (in the entire expression of e + vlogv) and hence the shape of the three graphs is quite similar.

Dense Graphs: -

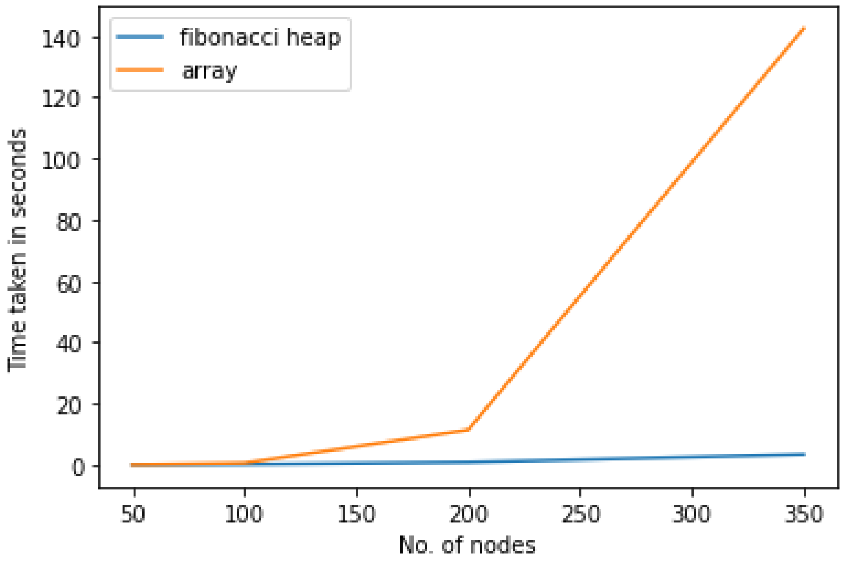
Shown below is the curve between the number of nodes and the time taken for the code to completely run, here the graph used is relatively denser than the one used previously, this would surely lead to more time taken by all the different heaps but would help us compare the performance of Fibonacci heap and binomial heap, because theoretically it was analysed that for more dense graphs Fibonacci would tend to outperform the other heaps.



Here we can see that the binary heap performs better than both the other heaps but this is mainly because of the smaller constants in the case of binary heap, when we compare the binomial and the Fibonacci heap ,we get extremely interesting results , we see when the number of nodes are smaller the time taken by the binomial heap is lesser than the Fibonacci heap (this is again due to difference in the multiplication constants), however as the number of nodes get larger the elogv term in the complexity of binomial heap becomes more dominant and the time taken by the binomial heap increases slightly than the Fibonacci heap, as correctly analysed in the theoretical part. If the number of nodes is further increased (towards millions), the green graph or the graph of the binary heap would also surely cross over the orange graph or the graph of the Fibonacci heap.

Practically Analysing Arrays: -

Given below is the graph of the array and the Fibonacci heap between the number of nodes and the time taken.



It is quite easily scene from the graph above that the performance of the array is the worst among all the different data structures used. This is in line with the theoretical analysis made.

CONCLUSION: -

Hence summarising the analysis made above. For smaller number of nodes and less dense graphs it may prove to be beneficial to use binary heap above Fibonacci heaps, this is because of larger constants involved in the case of Fibonacci heaps and also because the graph is less dense and hence decrease key operation would be called much lesser and the main benefit for us in Fibonacci heaps is the decrease key which can be performed in O (1).

However, for denser graphs it can be observed that Fibonacci heap starts performing better and this is mainly due to the fact that in this case the decrease key operation is called many times (due to larger value of e), and hence we get the benefit of using the Fibonacci heaps because here the decrease key takes place in O (1) time. But the clear benefit would be observed for extremely large number of nodes, of the order of billions, because at such large values the constants wouldn't really matter and the Fibonacci heap would clearly outperform both, the binomial and binary heap.