Johnson's algorithm

(Analysing time complexity using various heaps)

Aditi Aggarwal

21.12.2020

(2019CSB1063)

INTRODUCTION

Implementation of Johnson's algorithm which includes implementation of bellman ford algorithm and N times dijkstra which I implemented using various methods like arrays, binary heaps, binomial heaps and fibonacci heaps. And would also discuss the total time taken for all the heaps using some random graphs.

METHOD

Johnson's algorithm:- Johnson's algorithm is actually used to find all pairs' shortest path. So the one idea which came to mind is to implement the Dijkstra algorithm on all the nodes. But as we know the limitation of dijkstra that is it works only on positive edges. So we have to make all the edges positive in order to do this. So adding bias is a good option. So how to add a bias? Should I subtract the smallest negative number. But this wouldn't work. So in johnson's algorithm I first applied the Bellman Ford algorithm to make all the edges positive.

Bellman ford algorithm:- I added an extra vertex in my graph. And then draw the edge from that node to every node in the graph. And kept the weight of edges as 0. And after that applied bellman ford algorithm and find the distance of all the nodes from that extra added vertex. And give each node the value equal to distance calculated by bellman ford. And after that remove all the extra added edges and change the weight of other edges such that new weight of edge is equal to weight of old edge plus value of the node from

which the edge started minus value of the node where the edge ended.

After that by bellman ford properties the values of all the new edges are positive. So in the next step I implemented the dijkstra algorithm on each node using four different heaps.

1. Array based heaps
2. Binary heaps
3. Binomial heaps
4. Fibonacci heaps

1. ARRAY BASED HEAPS:-

In the first method I implemented the Dijkstra algorithm using heaps. My main idea was that I maintained an array in which I kept all the nodes on which I once applied relaxation operations. What I actually did was that I initially added vertex in my array. And after that removed the minimum element from the array using linear traversal and marked that minimum node. And place all the unmarked nodes back in the array which could be relaxed. And the nodes which were already in the array I changed their values again using linear traversal. And kept doing it until the array was not emptied. And then note down the distances of the nodes from that vertex and did that for all the other vertices. And in this way I implemented dijkstra for all the other vertices and calculated the shortest path for all the pairs.

2. BINARY HEAPS:-

Implemented binary heaps using arrays because binary heaps have a very good property to directly find their children and parent. Like indices of children are (2*index+1) and (2*index+2) where index is the index of that node. And the index of the parent is the floor of (index-1)/2. So I maintained a min binary heap in which the parent data is smaller than the child data so min value could be easily found in O(1) complexity. And after finding the minimum value just remove it and mark it and similarly add all the unmarked values in the heap where the edges could be relaxed. Added both the nodes which were already in the heap and unmarked nodes which were never in the heap. And maintained an array of all the nodes which were removed from the heap. And repeat this process until the array is emptied or the count of removed nodes from the heaps exceeds the total number of nodes. While inserting any node in a heap I inserted that

node at last and after that percolated it up in order to maintain the min binary heap. And while deleting any node I removed the top most node and replaced it with the last node of the binary heap and after that percolated it down in order to maintain the min heap property and in this way I implemented binary heaps.

3. BINOMIAL HEAPS:-

For the implementation of binomial heaps I used pointers. Firstly declared the structure of node as:-

Int data,int degree, int index, pointer to parent, sibling and the leftmost child

And maintained the head pointer and after that implemented dijkstra. To find the minimum node it requires log(n) time and after that to insert any node firstly inserted it as a single node in the heap and after that maintained the order of the heap that is in heap there is always one tree of 1 rank at a time so we maintained the order of trees in the heap. Combined all the trees of the same rank. And to perform decrease key operation and extract min operations we first have to find the minimum element which takes order logn and after that to delete that element we simply remove that element and all the trees which are child of that main tree we simply add it in a heap in order (degree) as we have to make the parent of each node as null. And after that I rearranged all the nodes in order that only one node of that particular rank should be present in a heap.

In Binomial heaps we have to maintain the order while inserting while deleting while decreasing any key every time we have to maintain order which says there should be only one tree of a particular rank in a heap. In this way I implemented binomial heaps.

4. FIBONACCI HEAPS :-

Implementation of fibonacci heaps was also done using pointers.The structure which I used during its implementation is :-

Int data,int rank, int index, pointer to parent, left sibling,right sibling and the leftmost child

And I also maintained the min pointer and the head pointer while implementing dijkstra. Main idea is that it is the most efficient and the fastest heap.To find the minimum node it requires only O(1) time and after that to insert any node insert it as a single node in the heap which again takes o(1) time. And to perform decrease key operation and extract min operations we first have to find the minimum element which takes order 1 and after that to delete that element we simply remove that element and mark the color of each node of the tree if it is black just make it a new tree and then all the trees which are child of that tree we simply add it in a heap in order (degree) as we have to make the parent of each node as null.else keep that node in the tree on whichever position it was. And after that rearranged all the nodes in order that only one node of that particular rank should be present in a heap.

Unlike Binomial heaps we don't have to maintain the order while inserting.but while deleting while decreasing any key every time we have to maintain order which says there should be only one tree of a particular rank in a heap. In this way I implemented fibonacci heaps.

operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

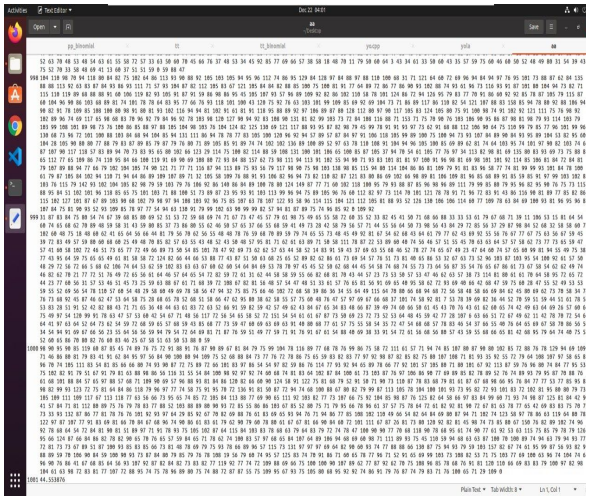
† amortized

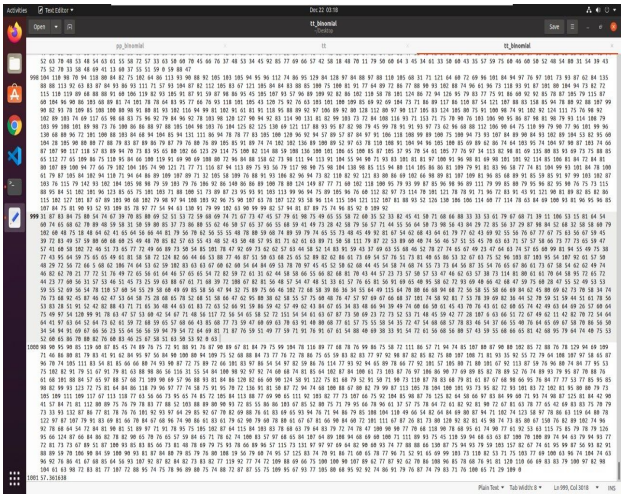
Table explaining all the time complexity of all the heaps.

I implemented my algorithm on a test case with 1000 nodes and many other cases.But I included only a small number of test cases in the table below. And below are the results which I got.

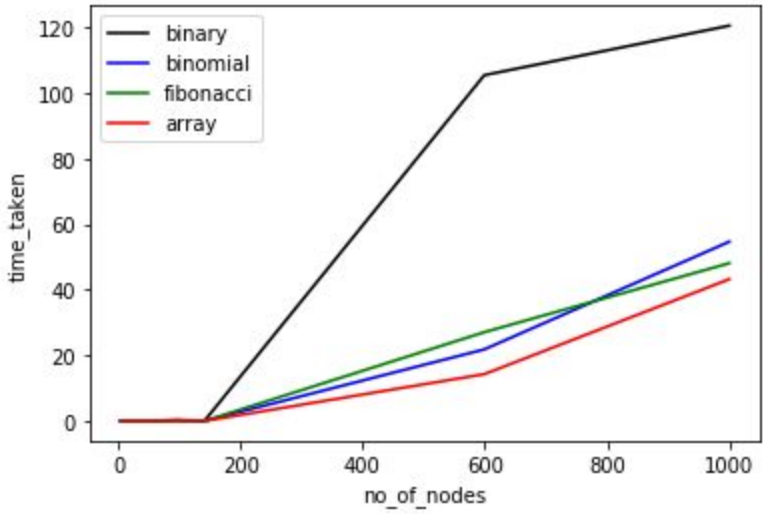
No. of nodes	Array based(Time taken)	Fibonacci heaps(Time taken)	Binary heaps(time taken)	Binomial heaps(Time taken)
4	0.000108	0.000182	0.000312	0.000189
20	0.000659	0.001245	0.003597	0.002218
40	0.007053	0.009223	0.029859	0.021676
96	0.112733	0.099040	0.343775	0.180775
141	0.000158	0.000129	0.000190	0.000247
599	14.289302	27.098948	105.485582	21.846670
1000	43.265268	48.138937	120.593202	54.705711

Here are some snapshots of my test cases:-





RESULTS



CONCLUSION

From the graph we can see as we increase the number of nodes the fibonacci heaps are the fastest. Initially almost all are the same. Binary heaps are the worst as we can see from here. Arrays and binomial are somewhat equal but as the number of nodes increases binomial keeps improving.

REFERENCES

1. Time complexity image from wiki.