CS201 LAB-6 REPORT

Analysing running time of Johnson's algorithm using four different heaps

GURKIRPAL SINGH

2019CSB1087

22.12.2020

INTRODUCTION

I implemented Johnson's algorithm using four different heaps(array based, binary heap, binomial heap, fibonacci heap) to determine their running time. The difference in their running time is due to the dijkstra part of the algorithm. As in dijkstra we need to find and delete the minimum element and decrease their value, we use heaps. Decrease key operation is the one which makes the difference. As it is fastest in fibonacci heaps, fibonacci heaps implementation is expected to be fastest.

ALGORITHMS

Johnson's Algorithm:

It is an algorithm used to find All Pair shortest paths for negative edge weights. It involves running Bellman Ford algorithm once to make all weights non negative and then running Dijkstra n(number of nodes) times taking each node as source vertex once. So, its time complexity is the sum of time complexity of Bellman Ford and n times the time complexity of Dijkstra $O(V.E + V^2 logV)$. It is much better as compared to the naive approach of running Bellman ford n times from all the vertices once(V^2E).

Dijkstra Algorithm:

It is used to find the shortest path of all vertices from a given source vertex. In this the adjacent edges are relaxed as a node is added in the visited cloud and the nodes in the visited cloud are the ones selected from minimum heaps.It also involves decreasing the value if found less after relaxing an edge.The time complexity of Dijkstra algorithm is O(E.logV) for Binary heap implementation and optimised O(E + VlogV) for fibonacci heap implementation.

Bellman Ford Algorithm:

It is just relaxing all edges V-1 times. So time complexity-O(V.E). But it can be optimised in many ways like just relaxing the edges adjacent to the one selected.

IMPLEMENTATION

I implemented all the functions in C++ making use of its standard Template library for vectors. Bellman and Relax operation is a common function for all cases as it is independent of the type of heap used. For dijkstra, initially all the nodes are inserted in the heap and then the one with minimum distance is chosen and put in the visited cloud and its adjacent nodes are relaxed and their values are updated in the heap using decrease key operation. A map is used to store the pointer to the node for a particular

vertex present in the heap to access the required node in O(1). It can be thought of as a similar operation to BFS. Hence its time complexity can be **[Relax*Extract_Minimum*DecreaseKey].**

Array Based Implementation:

Array based implementation is simple requiring no extra function as it chooses minimum by traversing through all the nodes. It is very slow as it takes O(V) just to find the minimum edge. Here in dynamic array implementation, Insertion is amortised O(2). Delete minimum is **O(n)** as all the nodes are transverse and decrease key is **O(1)**.

Binary Heap Implementation:

It involves functions for insertion, deletion and decrease key.

Insertion [bin_insert()] in binary heaps is O(logN) though adding in the heap is **O(1)** because after addition it is required to be percolated up to make it reach its position.

Similarly ExtractMin is also **O(logN)** as it requires replacing of head node by last node and then percolating the head node downwards to put it in its correct position. Here FindMinimum is **O(1)** as it is always the head node.

For decrease key, index of the node is stored in a lookup table to access the node in **O(1)** as and when required. So, decrease key is also **O(logN)** involving decreasing the value and then percolating it up for its correct position.

Binomial Heap Implementation:

Union of two binomial heaps is **O(logN)** as it includes merging and grouping of heaps with same degree. Most of the operations of binomial heaps revolve around the union operation.

Insertion is making a new heap of single element and its union with the existing one. Making a new heap is O(1) and Union is **O(logN),** hence insertion is **O(logN)**.

ExtractMin is deleting the node and making a different heap of its children and

then the union of two heaps. It is also based on union operation ,hence it is also **O(logN).**

For Decrease key, a lookup table is maintained to store the pointer to the node of a particular vertex which is then used to access the node in O(1) while decrease key. Decrease Key in binomial heaps is very similar to that in binary heaps involving changing the value and then percolating up to maintain heap property.Hence , percolate up is O(logN) which gives us **O(logN)** for Decrease Key.

Fibonacci Heap Implementation

ExtractMin in fibonacci heaps is O(logN) as it normally deletes the node and meld its children to the head heap which makes it amortised O(1). and also consolidates the heap to maintain heap property which makes it O(logN).

Insertion is O(1) as it just includes adding a new node in the main linked list of the heap.

Decrease Key is also O(1) as it also makes the node separate adding it in the main linked list.

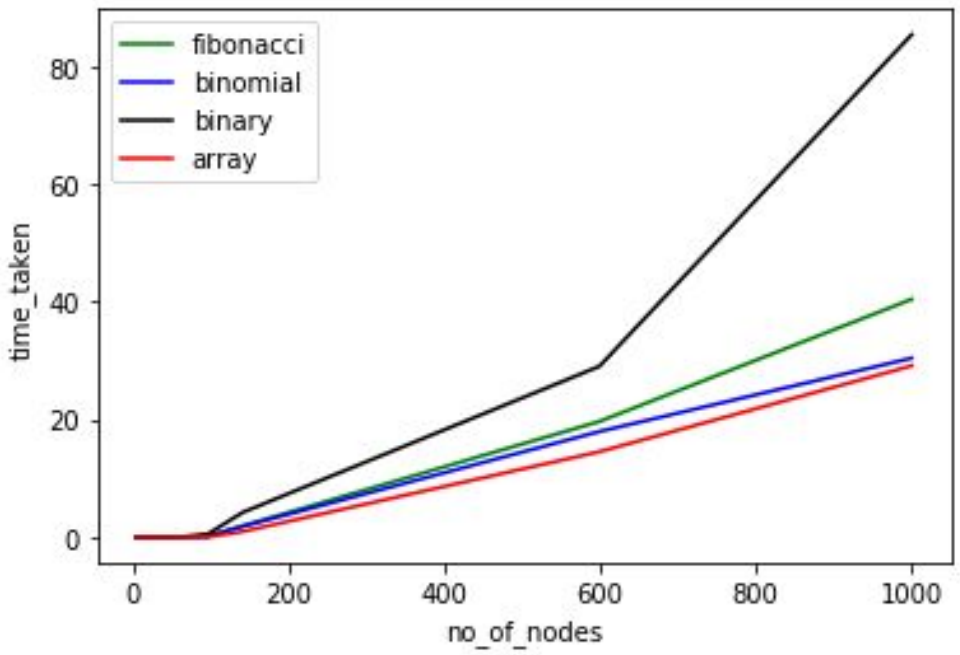
Due to very efficient decrease key and extractMIn operations, Fibonacci haaps are very efficient. But because of high space usage, it becomes a little slow sometimes than other array based and binomial heaps.

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

DATA (Obtained by running code)

Number of Nodes	Array Based	Binary Heap	Binomial Heap	Fibonacci Heap
4	0.000124	0.000250	0.000348	0.000228
6	0.000129	0.000395	0.000624	0.000376
20	0.000656	0.006814	0.003743	0.002426
30	0.000478	0.000703	0.000593	0.000631
40	0.004945	0.047550	0.020965	0.018083
100	0.060156	0.550919	0.162321	0.197056
150	0.986453	4.250081	1.863392	1.969410
600	14.544615	29.063469	17.977701	19.691984
1000	29.154712	85.414742	30.445751	40.454288



Graph from data

CONCLUSION

From the graph and the data obtained from the code, it can be observed that fibonacci and binomial heaps remain efficient throughout the code because of low complexity of ExtractMin and decrease key . Because the number of nodes in the graphs is not much hence array based implementation is also working perfectly. As the number of nodes increase and graph becomes more and more dense, binary and array become ineffective while fibonacci becomes effective there.

REFERENCES

1. Class Notes