FIBONACCI HEAPS

IMPLEMENTATION AND ANALYSIS OF VARIOUS HEAPS

Harsh Priyadarshi | 2019CSB1088 | CS201

INTRODUCTION

1) Our main job was to implement Johnson's Algorithm.
2) Johnson's Algorithm uses Bellman Ford Algorithm and Dijkstra algorithm.
3) Time complexity of Bellman Ford algorithm is O(Edges*Vertices).
4) Time complexity of Dijkstra's Algorithm is O (V*(Extract min) +E*(Decrease Key)).
5) By using different Heap Structures one can improve the time complexity of Extract min and Decrease key and thus over all the total time complexity of Dijkstra's Algorithm.

THEORY

1) Johnson's Algorithm
    - The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative.
    - Johnson's algorithm uses both Dijkstra and Bellman-Ford as subroutines.
    - The idea of Johnson's algorithm is to re-weight all edges and make them all positive using Bellman Ford algorithm, then apply Dijkstra's algorithm for every vertex as Dijkstra can't be applied on a graph having negative edge weights.
    - The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is **O ($|V|^2$log |V|+|V||E|)**, the algorithm uses **O(|V||E|)** time for the Bellman–Ford stage of the algorithm, and **O (|V|log |V|+|E|)** for each of the |V| instantiations of Dijkstra's algorithm.

2) Bellman Ford Algorithm
    - A node is taken and outgoing edges are created of 0 weights.
    - Then we run bellman ford algorithm to find weights for each vertex.
    - Then each edge is modified as $D_{new}=D_{old}+V_{source}-V_{Dest.}$
    - This ensures the weights are all positive and function of source and destination node.
    - We can run Dijkstra's Algorithm on this to find the All Pairs of Shortest path and then reweight them to find answer corresponding to the original graph.

3) Dijkstra's Algorithm
    - Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph.
    - It can be implemented in O (V*(Extract min) +E*(Decrease Key)
    - The fastest known complexity is by using Fibonacci heaps i.e. **O (|V|log |V|+|E|).**

HEAP STRUCTURES

1) Array Based
   - We find min from the array for each extract min operation.
   - Time Complexity of Extract Min: - O(n)
   - Time Complexity of Decrease Key: - O(1)
2) Binary Heap
   - A Binary Heap is a Binary Tree which is complete and each parent is less than its child. (in case of min heap)
   - Time Complexity of Extract Min: - O(log(n)
   - Time Complexity of Decrease Key: - O(log(n)

3) Binomial Heap
   - A binomial heap is a data structure that acts as a priority queue but also allows pairs of heaps to be merged. It is important as an implementation of the mergeable heap abstract data type, which is a priority queue supporting merge operation. It is implemented as a heap similar to a binary heap but using a special tree structure that is different from the complete binary trees used by binary heaps.
   - Time Complexity of Extract Min: - O(log(n)
   - Time Complexity of Decrease Key: - O(log(n)

4) Fibonacci Heap
   - A Fibonacci heap is a data structure consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap.
   - Time Complexity of Extract Min: - O(log(n))
   - Time Complexity of Decrease Key: - O(1)

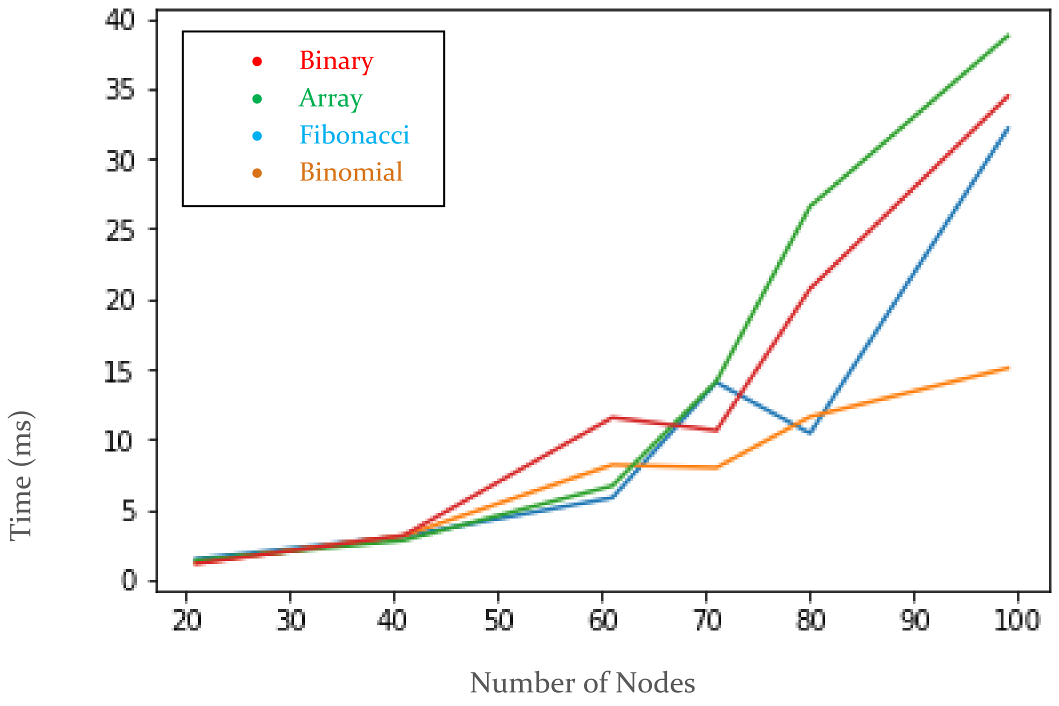
Operation	**Array**	**Binary**	**Binomial**	**Fibonacci**
Extract Min	O(n)	O(log(n))	O(log(n))	O(log(n))
Decrease Key	O(1)	O(log(n))	O(log(n))	O(1)

IMPLEMENTATION

1) Array (*djikstra_array*)
   - Values were stored in a vector of int.
   - For extract min, the program iterates through the array and then stores the value and makes the value *inf* so that it is not discovered further.
   - For decrease key, we can decrease the value of key by accessing the index of the node.
2) Binary Heaps (*djikstra_binary*)
   - Values are stores in a vector of int.
   - Root is stored in the 0th index and its children are stored in 2i+1 and 2i+2 index in the vector.
   - For extract min (*extract_key*), one can store the value of 0th index and replace index 0 by last index of the vector and then delete the last index as deleting from behind is an O(1) operation. We then percolate the root down to maintain heap property.
   - For decrease key (*decrease_key*) we first find the node we are looking for and then we change it's value and then percolate up to maintain the heap property.
3) Binomial Heaps (*djikstra_binomial*)
   - A structure *node* was made that stores the pointer to it's parent, child, right neighbor and left neighbor. Beside this it also stores it's node number, its degree and the shortest distance to that node in dijkstra's algorithm which is being found out.
   - Another structure of binomial tree was made that had a pointer to the min. value and it also stored address to each node and the degree of each subtree in binomial tree.
   - For extract min (*extract_binomial)*, we removed the min pointer and then added it's children to the main tree branch and then consolidated the tree i.e. (trees with same degree were merged).
   - For decrease key (*decrease_binomial*), we used the address vector in our main tree structure to locate the node and then percolated up in the same binomial tree branch.
4) Fibonacci Heaps (*djikstra_fibonaci*)
   - A structure *node* was made that stores the pointer to it's parent, child, right neighbor and left neighbor. Beside this it also stores it's node number, its degree and the shortest distance to that node in dijkstra's algorithm which is being found out.
   - Another structure of fibonaci tree was made that stores address of each node, a spare vector for consolidation, pointer to the min value, pointer to the head and a vector which shows whether the nodes is marked or not (for decrease key operation).
   - For extract min (*extract_fibonaci)* we removed the minimum and then added all its children to the root branch which is a double linked list. Then, we use our spare vector to consolidate(ith index holds the node of ith degree and thus if any other node of the same degree comes it merges them) the tree by merging trees with same degree. Then they are made into doubly linked list again and min is located and stored at the min pointer.
   - For decrease key (*decrease_fibonaci)* we get the address of the node from the address vector, and if it is not a parent node then we cut it from there and add it to the main branch of doubly linked list, and mark its parent, if the parent was already marker then we cut it as well and this process continues recursively until an unmarked node is reached. (parent node is always unmarked).

OBSERVATION

1) The values were same till 38 nodes or so, after that we can see a difference in the graphs of the time taken.
2) Array based heaps start taking longer time than other heaps as we reach higher values of other nodes.
3) Binomial heaps were initially performing slower than binary and Fibonacci but later prove to be better than the others.
4) Fibonacci Heaps show a big variation, the reasons can be
   i) Improper Implementation
   ii) Space complexity being high in my implementation
5) Fibonacci heaps still prove to be better than binary and array-based heaps.
6) Attached Graph



CONCLUSION

1) Initially performance of all the nodes were almost same.
2) Array based heap started taking longer time than other heaps.
3) Binomial Heaps proved to be the most time efficient.
4) Fibonacci heaps showed a large variation and proved to be better than array and binary.
5) Even though time complexity of Fibonacci is the best, still it was slower than binomial heaps, due to high space complexity and improper implementation.

REFERENCES

1) https://www.geeksforgeeks.org/binary-heap/
2) https://en.wikipedia.org/wiki/Binomial_heap
3) http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap21.htm
4) https://en.wikipedia.org/wiki/Johnson%27s_algorithm