Priority Queues: Performance Analysis

Ritik Garg

Overview and Purpose

The primary objective is to analyze the **time complexities** of heaps, implemented using different **data structures.** For this, **Johnson's algorithm for All Pair Shortest Path** is implemented using four different heap implementations viz., **array-based heaps, binary heaps, binomial heaps,** and **Fibonacci heaps.** The execution time for the algorithm is measured on different graph sizes for these different heap implementations.

Time complexity for Johnson's algorithm for APSP

The algorithm uses **Dijkstra** and **Bellman-Ford** algorithms for calculating single-source shortest path **(SSSP)** to determine shortest paths between all pairs. **Bellman-Ford is executed once on the graph and Dijkstra is run for every vertex.**

Considering a graph of **n** vertices and **m** edges, the time complexity for running Bellman-Ford is **O(mn)** and the time complexity for running Dijkstra is **n*O(extractMin)** + **m *O(decreaseKey)**, where decreaseKey = Operation of decreasing the value of a key in a heap and extractMin = Operation of extracting the minimum node from a heap. The time complexity of Johnson's algorithm for APSP is the **time complexity of Bellman ford** + **n * time complexity of Dijkstra**.

Implementations:

- ### Heaps using Array

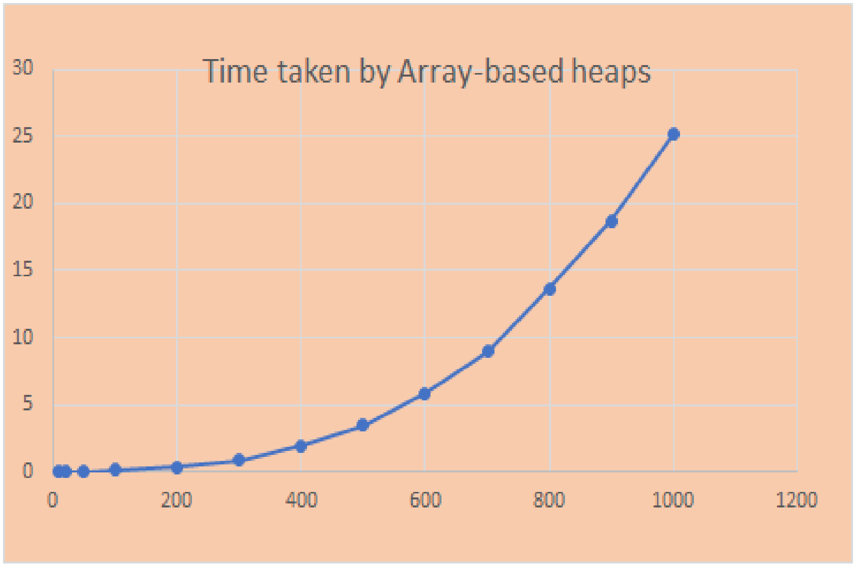
1) **makeHeap** - For making the initial heap, **heap[node]** = **initial distance** (0 or infinity). For deletion, **heap[node]** = **-1** (since all distance values in the heap for the updated graph are always >= **0**).
2) **extractMin** - Minimum is extracted by traversing through the array, accounting for a time complexity of **O(n)**.
3) **decreaseKey** - The key for a vertex is decreased by directly assigning the decreased value to the index, thus taking **O(1)** time.

Time complexity of **Dijkstra** = n *extractMin + m * decreaseKey = n*O(n) + m*O(1)

= **$O(n^2)$** (since, E = $O(n^2)$)

Time complexity of **Johnson's APSP** = $O(mn + n^3)$ = **$O(n^3)$**

Number of Nodes	Time taken by Array-based heaps
10	0.001
20	0.002
50	0.01
100	0.099
200	0.348
300	0.875
400	1.917
500	3.465
600	5.841
700	8.947
800	13.612
900	18.726
1000	25.142



Binary Heaps

1) makeHeap - The initial heap is made by assigning **heap[1]** = **pair (0, source node)** and **heap[other indices]** = **pair(999999, other nodes)** , using **‘1’-based** indexing.

2) extractMin - The minimum node is extracted from the first index of the heap. The value is replaced by the value at the last index and then the position of this node is adjusted by **percolating down** in **O(log n)**.

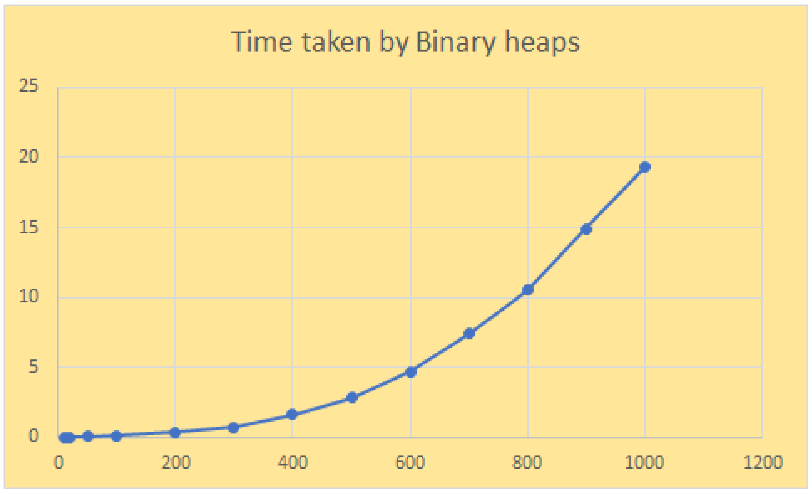
3) decreaseKey - The required key is decreased to the desired value. The position of the node is updated by **percolating up** in **O(log n)**.

Time complexity of **Dijkstra** = n * extractMin + m * decreaseKey

= n * O(log n) + m * O(log n) = **O(m log n)**

Time complexity of **Johnson’s APSP** = O(mn + mn log n) = **O(mn log n)**

Number of Nodes	Time taken by Binary heaps
10	0.001
20	0.003
50	0.021
100	0.091
200	0.316
300	0.729
400	1.59
500	2.789
600	4.656
700	7.372
800	10.494
900	14.892
1000	19.391



● Binomial Heaps

1) makeHeap - A new node in the binomial heap is inserted by creating a binomial heap on a **single** node and calling **unionBinomial()** to merge this new heap with the existing heap. To make the heap, the source node with initial **distance 0** and the other nodes with initial **distance 999999** are inserted using **insertBinomial()**.

2) extractMin - Removing the minimum node returned by **getMinBinomial(),** creates a new binomial heap of its child nodes and using **unionBinomial()** to merge the new binomial heap with the existing one. Time complexity is **O(log n).**

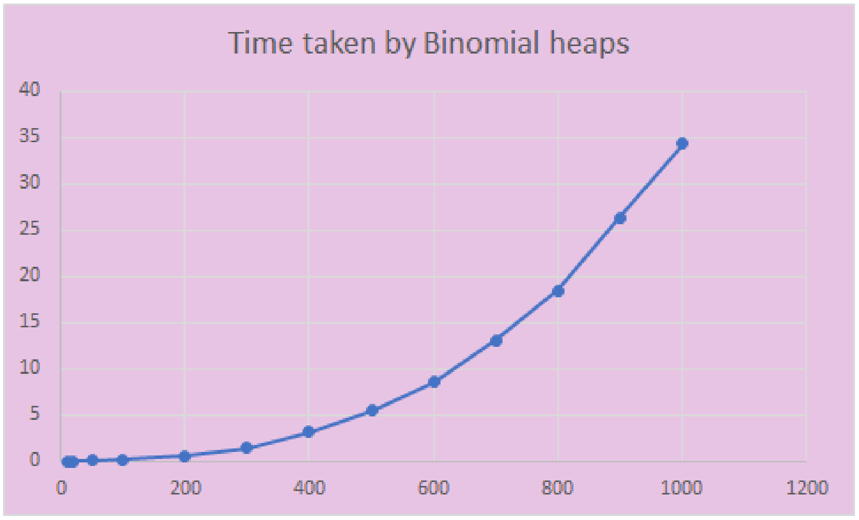
3) decreaseKey - The required key is decreased to the desired value and compared with the value of the parent. The values are swapped if the min-heap property is not satisfied. This is performed until we reach a node that has a parent of lower value or we reach a node in the root list. Time complexity is **O(log n)**

Time complexity of **Dijkstra** = n * extractMin + m * decreaseKey

= n * O(log n) + m * O(log n) = **O(m log n)**

Time complexity of **Johnson's APSP** = O(mn + mn log n) = **O(mn log n)**

Number of Nodes	Time taken by Binomial heaps
10	0.004
20	0.004
50	0.056
100	0.164
200	0.59
300	1.399
400	3.099
500	5.419
600	8.542
700	13.083
800	18.43
900	26.398
1000	34.365



• Fibonacci Heaps

1) makeHeap - insertFibonacci() creates a new node and inserts it in the root list in O(1) time. To make the heap, the source node with initial **distance 0** and the other nodes with initial **distance 999999** are inserted using **insertFibonacci()**.

2) extractMin - The minimum node is extracted using **extractMinFibonacci(),** which deletes the minimum node from the heap, pushes its children into the root list, and performs **consolidation** to merge nodes of the **same rank** together. Time complexity is **O(log n)**.

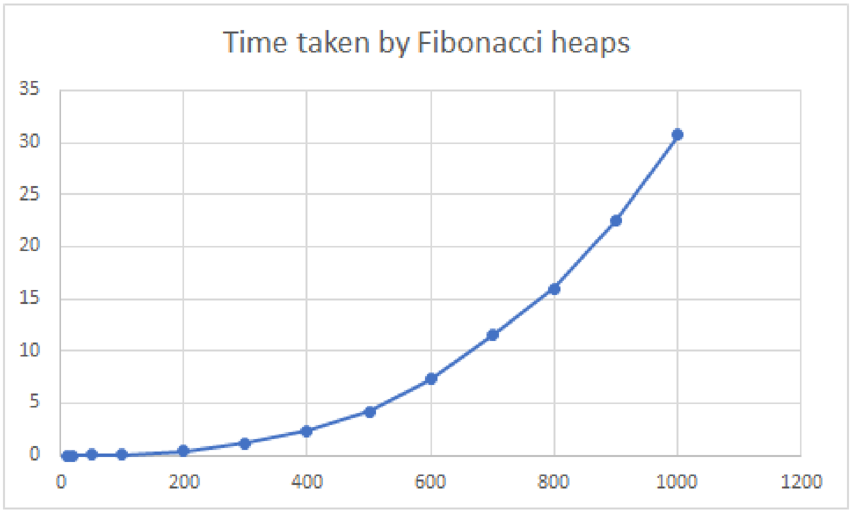
3) decreaseKey - The key is decreased to the desired value In case, the **min-heap** property at this node is violated, the node is cut and inserted into the root list. Based on the **unmarked/marked** status of the parent node, **cascade** cuts are performed. Amortized time complexity is **O(1)**.

Time complexity of **Dijkstra** = n * extractMin + m * decreaseKey

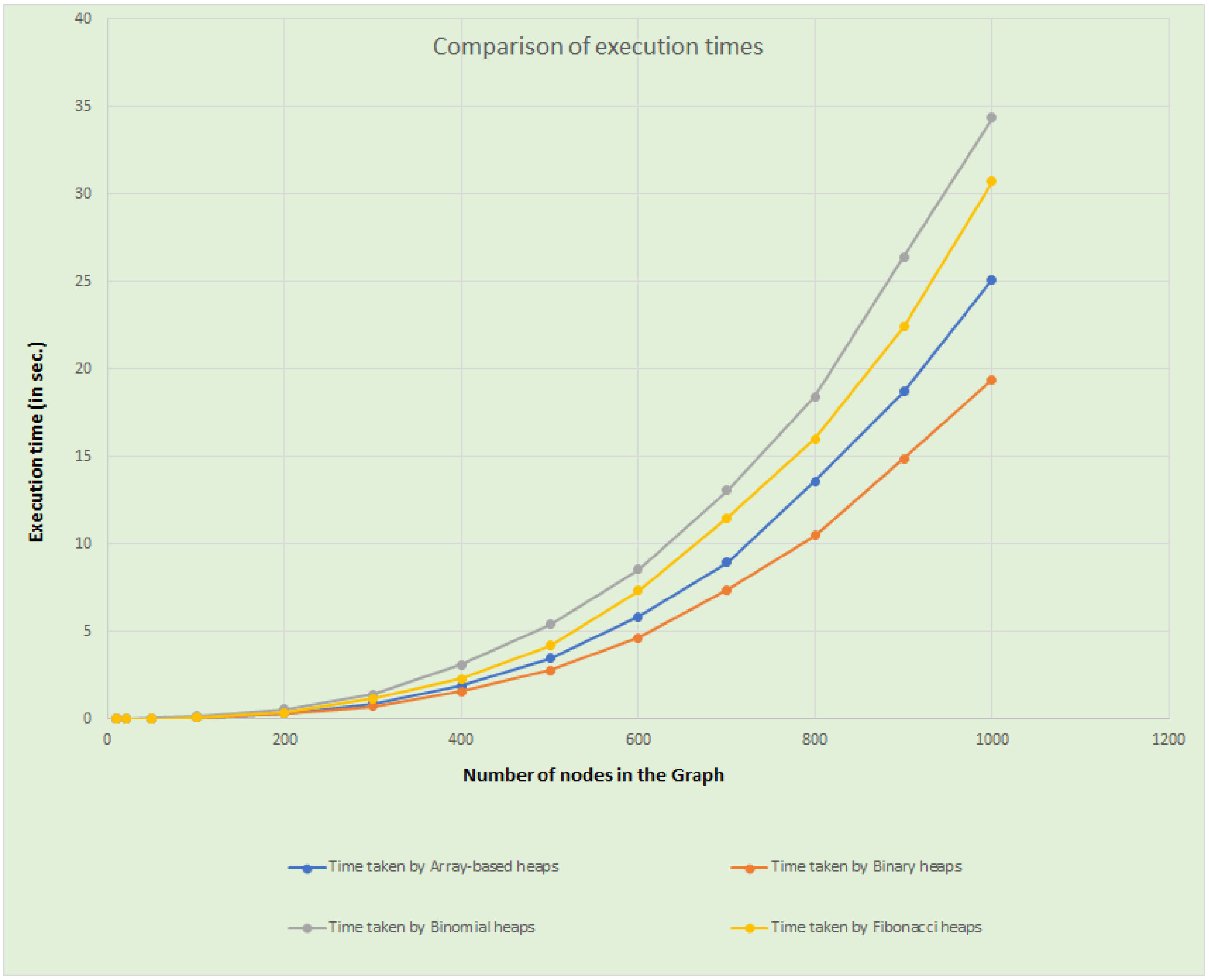
= n * O(log n) + m * O(1) = **O(m + n log n)**

Time complexity of **Johnson's APSP** = **O(mn + n^2 log n)**

Number of Nodes	Time taken by Fibonacci heaps
10	0.002
20	0.003
50	0.016
100	0.083
200	0.38
300	1.159
400	2.303
500	4.215
600	7.324
700	11.498
800	15.995
900	22.437
1000	30.732



Combined Analysis:



Observation :

From the above graph which compares the execution time of Johnson's algorithm for All Pair Shortest Path executed using heaps of different data structure implementations, it can be observed that the execution time is minimum for **binary heaps**, followed by **array-based heaps**, followed by **Fibonacci heaps** and maximum for **binomial heaps**. This contradicts our theoretical analysis but leads us to some important conclusions.

Conclusion :

The Fibonacci heap has a higher execution time than expected because of high values of **constant factors** (such as maintaining circular linked lists, assigning and updating pointers, etc.) neglected during complexity analysis. Consequently, it has a higher time of execution than binary heaps but still lesser than binomial heaps.

Also, binomial heaps having a higher execution time than binary heaps follows from the same analysis that theoretically they have equivalent time complexities but due to the additional work of **assigning and updating pointers**, it has a higher execution time in a real-world scenario.

Another unexpected observation is the execution time of the array, but this can be attributed to the **O(n) implementation of extractMin** and **O(1) implementation of decreaseKey.** The array-based heap could also have been implemented by keeping the complexity of extractMin as O(1) and decreaseKey as O(n), making the theoretical time complexity to **$O(mn^2)$** where **m** = **$O(n^2)$**, instead of **$O(n^3)$** as calculated above.