# CS201 Lab- 6 report

**Name:** **Hardik kapoor**

**Entry number:** **2019MCB1220**

**Aim:** To implement Johnson's algorithm (in which Bellman Ford and Dijkstra are used as subroutines) with implementing Dijkstra in 4 ways that are array, binary heap, binomial heap and Fibonacci heaps.

**Theory:**

## Algorithm description

- Firstly, a new node is added to the graph, and connected a 0 weight edge to all the vertices.
- We apply Bellman Ford algorithm from this new node, and calculate an array, let us say h for all the nodes (which will be equal to the shortest path calculated from bellman ford from this new node to all the nodes).
- We create a new graph in which weight of all the vertices are w'(u,v)=w(u,v)+h[u]-h[v]. This new weight will always be greater than zero, as h[v]<=w(u,v)+h[u].
- Hence, we get a graph which has nonnegative weights. We can apply Dijkstra, making each node of the graph as source and create the final all pair shortest path array.
- We can reverse the effect of h, by adding h[v]-h[u] to all distances D(u,v).

The runtime of this algorithm will be O(runtime of bellman ford + V*(runtime of Dijkstra)), where V is the total number of nodes in this graph. The runtime of Bellman Ford is always O(V*E) (E is the number of edges), hence the total runtime of this algorithm will be O(V*E+V*(runtime of Dijkstra)).

The runtime of Dijkstra is O(V*(extract min) + E* (Decrease key))

The runtime of this algorithm will vary due to the four different types of implementation of Dijkstra, which are

- Array
- Binary Heap
- Binomial Heap
- Fibonacci Heap

## Implementation

Here is a brief description of all the implementations:

1) Array based implementation: In array based implementation, the path array is used, along with a Boolean array which tells us if the current vertex is in the solution set or not. We find the minimum weight path not in the solution set, and relax the edges changing the path array (single source shortest path) alongside. Hence, the decrease key function can be though of as relaxing the edges and can only be done in O(1) (changing the value in the array), and find-min (or extract min) is iterating over the whole array, and finding the minimum which is not already in the solution set which can be done in O(V).
   Hence, the final theoretical complexity of array based implementation of Dijkstra comes out to be **O(V*V+E) =O(V^2+E).**

2) Binary heap based implementation: In binary heap based implementation, all the nodes along with its path value from the source is added to the binary heap, and then in every iteration, the minimum of the binary heap is extracted (done in O(log V)) and edges are relaxed from this using decrease key (done in O(log V)).
   Hence, the final theoretical complexity of Binary heap based Dijkstra comes out to be **O(V*log(V)+E*log(V)).**

3) Binomial heap based implementation: In binomial heap based implementation, all the nodes are inserted in a binomial heap, and then in every iteration, the minimum of the binomial heap is extracted (done in O(log V)) and edges are relaxed from this using decrease key (done in O(log V)). Functions like union (union of two binomial heaps, such that no two trees of same degree exist) are used here, in which can do union of two binomial heaps in O(log(V))
   Hence, the final theoretical complexity of Binomial heap based Dijkstra comes out to be **O(V*log(V)+E*log(V)).**

4) Fibonacci heap based implementation: In Fibonacci heap based implementation, we lazily do the consolidation (remaking the heap such that no two trees of same rank exist) to decrease the complexity of decrease key. In Fibonacci heap based implementation, the extract min is O(log(V)) just as in the other cases, but the decrease key comes out to be O(1) (due to the consolidation done lazily).
   Hence, the final theoretical complexity of Fibonacci heap based Dijkstra comes out to be **O(V*log(V)+E),** which is the lowest of all the cases.

The main difference between all the implementations is in decrease key and extract min functions. The decrease key works fastest in array based and Fibonacci heap based implementation, which is in O(1) but works in O(log(V)) in binomial and binary heap based implementations. The extract min is the slowest in array implementation (O(V)) but is identical in terms of complexities in all the heap based implementation (O(log(V)).

Hence, theoretically, Fibonacci heap must perform fastest whereas array implementation must perform slowest for large graphs.

## Runtime Analysis

I have done runtime analysis for two orders of V and varying degree of density. These are:

- V of order 400
- V of order 1000

Larger V than this is not possible to perform in practice, because it will take a lot of time to perform the operations in Johnson's algorithm.

In all the analysis done, I have made the graph of varying degrees of density, considering edges made with probability 1/50 up till made with probability 1.

## Small V (of order 400)

| | 1 | 0.5 | 0.1 | 0.04 | 0.02 | |
|---|---|---|---|---|---|---|
| **300** | 0.656 | 0.453 | 0.375 | 0.203 | 0.203 | **Array** |
| | 0.546 | 0.328 | 0.125 | 0.09375 | 0.09375 | **Binary** |
| | 0.593 | 0.296 | 0.109 | 0.0781 | 0.0781 | **Binomial** |
| | 0.578 | 0.312 | 0.14 | 0.1098 | 0.1098 | **Fibonacci** |
| **500** | 3 | 2.04 | 1.281 | 1.03 | 1 | **Array** |
| | 2.406 | 1.64 | 0.468 | 0.359 | 0.234 | **Binary** |
| | 2.234 | 1.42 | 0.484 | 0.406 | 0.281 | **Binomial** |
| | 2.343 | 1.54 | 0.562 | 0.421 | 0.296 | **Fibonacci** |

In this table, the probability is on the columns and the number of vertices in rows.

This is the trend that is followed for graphs of order 300 and 500. As we can see, array performs worst, due to the extract min being of O(V), binary and binomial heap perform comparatively and Fibonacci lags just a little bit. Also, as the probability is increased, the gap between array and heaps is decreased because array implementation decrease key is fast, as we are just changing the value in the array, and even though the theoretical complexity of decrease key of array and Fibonacci heap is the same, array will perform much better (in terms of decrease key) due to very fewer constant factors.

Binary and binomial heaps are identical, with binary heap performing faster in some cases and binomial in other cases. Their complexities are similar theoretically, but in implementation, binomial heap is faster in many cases.

## Large V (of order 1000)

| | 1 | 0.5 | 0.1 | 0.04 | 0.02 | |
|---|---|---|---|---|---|---|
| **800** | 12.07 | 7.921 | 5.234 | 4.406 | 4.375 | **Array** |
| | 9.015 | 5.109 | 1.75 | 1.109 | 0.781 | **Binary** |
| | 9.203 | 5.015 | 1.89 | 1.171 | 0.875 | **Binomial** |
| | 8.921 | 4.968 | 1.92 | 1.281 | 1.031 | **Fibonacci** |
| **1200** | 41.343 | 27.421 | 17.187 | 14.79 | 14.687 | **Array** |
| | 30.828 | 17.281 | 5.265 | 2.912 | 2.906 | **Binary** |
| | 29.109 | 15.56 | 4.437 | 2.912 | 2.25 | **Binomial** |
| | 29.468 | 15.61 | 4.907 | 3.281 | 2.828 | **Fibonacci** |

In this table, the probability is on the columns and the number of vertices in rows.

This table follows the trend of graphs with graphs of 800 and 1200 vertices with varying probability of edges. In this trend, the difference between arrays and all the other heaps is considerable, due to V being increased. Also, here as we increase the density, the gap decreases, due to faster decrease key in array than in all the heaps.

Here, binomial heap works faster than binary heap in most of the cases, in big and dense graphs (1200 nodes with probability mode than 0.1). I think, this is because as the trees are divided nicely, the decrease key operation does less operations (as percolate up is done much lesser times for each node).

Fibonacci heap performs slower in some cases and fast in some cases, slower especially in sparse graph as the constants are high (as we are changing the pointers many times in the linked list). Also, in extract min, binding the two trees has a huge constant, and we do that many times in the consolidate operation.

## Result / Conclusion

The array based implementation consistently performs slower than all the other implementations, due to the high complexity of extract min operation. This gap is decreased, as the edges are more because the decrease key operation is much faster in array than in all the others, due to low constant and O(1) complexity, which matches the theory.

Binary and Binomial heap implementations perform identical, with binary outperforming binomial in some low V cases, and binomial performing much faster in other cases. Both have similar theoretical complexity, but in practice, binomial performs much faster in many cases.

Fibonacci heaps, even though it has an O(1) theoretical complexity, does not perform faster than binomial and binary heaps in many cases, due to a large constant.