CS201 LAB 6

IMPLEMENTING JOHNSON'S ALGORITHM USING VARIOUS HEAPS

Aim: To Implement Johnson's Algorithm for finding All-pairs Shortest Paths for a Graph. While implementing the Dijkstra component of Johnson's, we use four kinds of heaps:

1)Array Based Implementation

2)Binary Heap

3)Binomial Heap

4)Fibonacci Heap

Submitted By: Vasundhara Pant

Entry Number: 2019CEB1035

Date: 22/12/2020

CONTENTS:

Basic Theory of Johnson's Algorithm

- Johnson's Algorithm is used to determine All pairs shortest paths (hereafter referred to as APSP) among all the vertices of a given graph. It employs Bellman Ford and Dijkstra's Algorithm as its subroutines.
- Since Dijkstra's algorithm can be used to determine shortest paths between a given source and all the other vertices in a better time complexity(O(VlogV)) than Bellman Ford(O(V*E)), it only follows that we should calculate APSP by executing Dijkstra's algorithm V times(V=number of vertices). But to do so, we first employ Bellman Ford Algo to make all edge weights positive, since Dijkstra only works on graphs with positive edge weights.

Bellman Ford:

- In Johnson's implementation of Bellman Ford, we first create a new node, add edges from this node to all the vertices of the graph, then treating this as source node run Bellman Ford algorithm on the graph.
- We obtain the shortest distances of the nodes from this source node(denoted by the vector h[]), which are then used to increase the edge weights. We add h[u]-h[v] to every edge going from u to v. Now, all edge weights become positive and Dijkstra can be implemented.

Dijkstra's Algorithm:

- We run Dijkstra's algo for all the V vertices, thereby obtaining APSP for the entire graph.
- In Dijkstra, we have to make use of heap to find the minimum distance, and also for decreasing key. It is here that we implement the four different types of heaps, denoted by 1, 2, 3 and 4.
- The complexity of Dijkstra's Algo (and thus Johnson's algo) is dependent on the type of heap used. We will analyze the different complexities later.

How I implemented the Code

- Bellman Ford is implemented only if negative weight is encountered in the graph. The algorithm was simply performed by adding a new node and calculating the shortest distances by considering this as source.
- We then update all the edge weights and run Dijkstra V times considering each node as source once.
- Initially in Dijkstra, all vertices are at Infinite distance and source is at distance 0.
- The two main operations in Dijkstra are Extract min and Decrease key. They are implemented in 4 different ways as follows:

1. Array Implementation:

- In this, a normal array is taken to store the distances from the source.
- Extract min takes O(N) time as the entire array needs to be traversed to find min node
- Decrease key take O(1) time.

2. Binary Heap

- This is implemented as a normal heap using a vector of pairs, pair.first stores the distance, and pair.second stores the node.
- Another array is created just to store the positions of the nodes in the heap. This is done so that decrease key may be implemented in O(log N) time, and we don't waste time on finding the node first.
- Extract min takes O(log N) time since after extracting min, the order property is restored, which take O(h=log N) time.
- Decrease key also take O(log N) time because of the same reason.

3. Binomial Heap

- This is implemented using a class, which represents a node of a binomial tree. It consists of data like node, degree and distance, and pointers to left child, right sibling and parent.
- The binomial Heap is represented by a list of root nodes, each of which are constructed using the class definition above.
- Again, I stored the references to each node in a vector, so that decrease key may be performed in O(log N) time.
- Extract min takes O(logN) time, since first we traverse the root list to find min, and then perform union on the min Node's children and the rest of the binomial trees(which again takes log N time)
- Decrease key is performed in O(log N) time, since that is the max time it takes to percolate up.

4. Fibonacci Heap

- I've implemented this using class too. The class Node consists of data like node, distance, degree, mark, and pointers to left sibling, right sibling, left child and parent node.
- The heap is represented by a list of root node, all of which are pointers to the class Node described above.
- This is a lazy version of binomial tree. In the beginning, the nodes are inserted one by one thereby forming an LL structure
- A pointer is maintained at all times to the minimum node of the heap.
- Extract min is performed in unit time since the minimum node is known, but then union operation is performed on the heap to maintain its structure. This union operation is done by maintaining an array to size Max degree, and then traversing the root list.
- I have taken the max degree to be 64, which will cover all large test cases. Therefore the overall complexity of Extract min becomes O(log N +64) which is equal to O(log N) for large values of N.
- The decrease key is performed in O(1) time by first decreasing the node's distance value, and if this new distance is less than its parent's distance, the node is cut off and added to the root list. Along with this, the parent is marked. If the parent had been marked before, the parent too is cut off and the process repeats.

Time Complexity Analysis

Expected Vs Observed

Johnson's Algorithm's time complexity= one time **Bellman Ford** + V times **Dijkstra**

= **O(|V|*|E|)** + **O(|V|^2 log |V|)**

(best case)

Inside Dijkstra, Expected complexities of various heaps used:

Operations->	**Extract Min**	**Decrease Key**
1)Array	O(n)	O(1)
2)Binary Heap	O(logn)	O(logn)
3)Binomial Heap	O(logn)	O(logn)
4)Fibonacci Heap	O(logn)	O(1)

OBSERVED:

In my implementation of the code, more or less the same time complexities are observed. In Fibonacci, the complexity during decrease key is increased from O(log N) to O(64) for small test cases. The rest of the complexities are same.

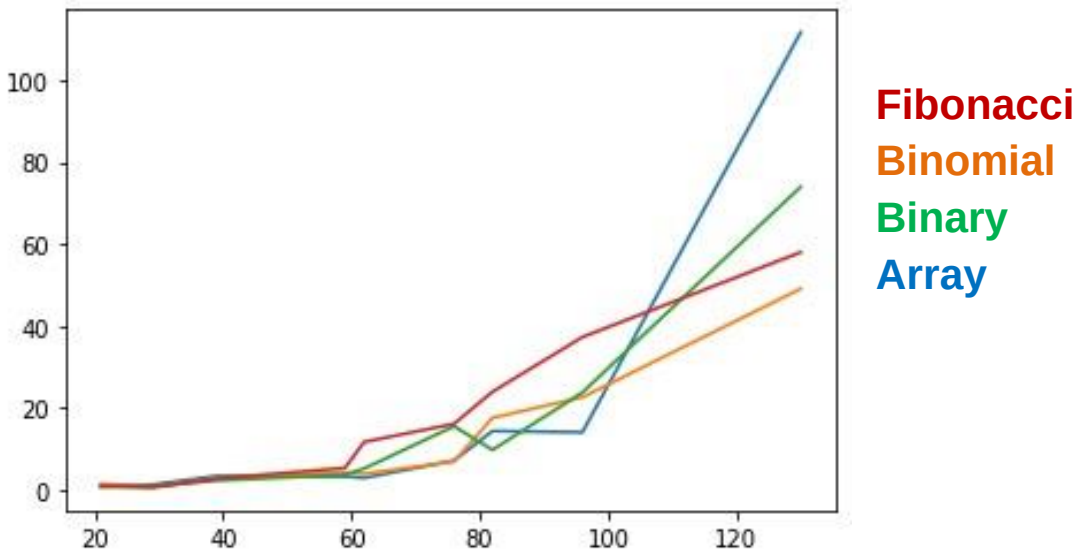
Some Observations:

- Array implementation works best for small test cases, since decrease key is done in O(1) time, and n is sufficiently small to have an effect on Extract min.
- Binary heap works best when the number of nodes starts to increase a little, since it is implemented statically using an array. It works better than array in all the nodes after a certain limit.

- Binomial heap takes a lot of time compared to array and binary heap for small test cases since it involves a lot of pointers and therefore many statements need to be executed even for small test cases, increasing the time. It starts to improve when the number of nodes increase and then becomes better than both binary and array based heaps. This is because of well distributed data in the binomial heap, which reduces the value of log N.
- Fibonacci heaps take the maximum time at the beginning, when number of nodes are less. This is again because of the large number of pointers(even more than binomial) which need to be changed for any operation. Therefore it is not recommended to use these for small inputs. However, when the number of nodes increase and the graph starts to become dense, the Fibonacci heap complexity increases exponentially and becomes the best among all the four heaps, since the decrease key operation, which is an integral operation of Dijkstra, is implemented in unit time using Fibonacci.

The graph for the observed time complexities of my code is attached below. I have taken random graphs which are both sparse and dense and have compared the time taken by each of the four heaps. (Note: it is not drawn to scale).

y->axis=Time taken



x->axis=Number of Nodes

Conclusion:

- The Fibonacci heap implementation works best for large test cases, and when the graph is extrapolated, it seems to work even better than binomial heap for sufficiently large test cases.
- Binary heap would be the most preferable choice of implementation for Dijkstra, if we know that the data isn't extremely large, and also not small enough to be implemented feasibly using an array.
- Binomial Heap works better than Binary Heap for large test cases, since the data is arranged quite efficiently in the form of binomial trees, which reduces the value of log N for large test cases.