Implementing Johnson's Algorithm Using Different Heaps

Implementing and analyzing Johnson's algorithm to find all pairs of shortest paths in a given graph, where the following data structures are used for Dijkstra's algorithm:

- ➔ Array
- ➔ Binary Heap
- ➔ Binomial Heap
- ➔ Fibonacci Heap

Date: 22/12/2020

Author: Kshitiz Arora

Repository: https://github.com/kshitiz-arora/Fibonacci-Heap/

Objective:

Implementation of Johnson Algorithm using different types of heap data structures, and their time complexity analysis.

Explanation of Algorithm:

The algorithm has 2 parts, namely

1. Bellman-Ford Algorithm
2. Dijkstra's Algorithm

BELLMAN-FORD ALGORITHM:

- This is implemented to counter the inability of Dijkstra to work on the graph with negative edge weights.
- It changes the edge-weights by a value so that each edge would have positive weight.
- These values by which we need to change the edge weights are calculated by running the bellman-ford algorithm after introducing a new vertex that has an edge with 'zero' weight to all other vertices.
- Now, let us denote the distance of ith vertex calculated by running the bellman-ford algorithm by h[i].
- So, we update the weight of every edge from u to v as,
- weight (u, v) = weight (u, v) + h[u] - h[v]
- Time complexity of this algorithm is O(VE).

DIJKSTRA'S ALGORITHM:

- Now, we have a graph with non-negative edge weights. So, we can run Dijkstra from any vertex to find the shortest distances from that vertex to all other vertices.
- So, in order to find all pair shortest paths, we run Dijkstra from all vertices and hence, find the shortest paths from every vertex to all other vertices.
- In this assignment, we try to implement this algorithm using different types of heaps and see how it affects the running time of the program.

  The different types of implementations used are, namely:
  - Array based
  - Binary heap based
  - Binomial heap based
  - Fibonacci heap based

Explanation of Code:

JOHNSON'S:

- After taking adjacency matrix as input, we ensure that there are no self-loops.
- Then, we find the values by which we need to update edges by using the bellman-ford algorithm as explained above and store it in **vector<int> update_values**.
- If we find a negative cycle, we terminate the program and return **-1**.
- Now we update the edge weights and create a new adjacency matrix.
- After this, we implement the required Dijkstra's algorithm from every vertex

BELLMAN-FORD:

- We first add a new vertex to the graph (adjacency matrix) with edges from that vertex to all other vertices and zero edge-weight.
- Now iterate over the edges (E-1) times to get the shortest distances.
- Then we iterate once more to check if there is any update in distances. If yes, that means there is a negative cycle and we print **-1** and terminate the program.
- We return these distances as a vector of int (**distance**)**.**

DIJKSTRA (ARRAY):

- We initialize the distances to 999999.
- Then we update the distance of source to 0.
- We push the source to a vector/array.
- Now until all vertices (which are reachable) are explored, we search for the node with minimum distance in the array and select it. And update the distance of the adjacent nodes, to the minimum value.

>> In this method, searching/extracting the minimum distance node may take up to O(n) time complexity. So, to improve on this part, we use heaps instead of an array.

Before looking at heap-based implementation, we can take a look at the time complexities of certain operations in different types of heaps and expect how results should differ depending on which heap is used.

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

DIJKSTRA (BINARY HEAP):

- The basic implementation of Dijkstra's algorithm using binary heap is similar to array-based implementation, except for the storing of the nodes and their distances. Instead of using an array, we use a binomial heap (although it is implemented using an array).
- Binary Heap Implementation:
    - *Insert*: While inserting we just put the (vertex, distance) pair at an index in the list (corresponding to its position in the binary heap)
    - *Decrease Key*: First, we decrease the distance value of that particular node and then percolate it upwards.
    - *Extract Minimum*: The minimum element is the root element of the heap (property). So in order to extract it, we swap the root with the bottom-rightmost element of the heap. Now, we percolate down the root, to maintain the order property of the heap. Since the minimum element is placed at the last position of heap, we can assume that the size of heap is reduced (**activeNodes**) and we never access that part.
      Note: In order to access the nodes in O(1) time, we maintain a position vector, that stored the current location of a node in the heap-array.

DIJKSTRA (BINOMIAL HEAP):

- Similar to the binary heap, except for the implementation of **insert**, **decrease key**, and **extract min**.
- Binomial Heap Implementation:

  Note: We begin by declaring a root list (a doubly-linked list) that contains the root nodes and ranks of all the binomial trees of the heap. Also, we maintain a vector of pointers that contain the pointer corresponding to each node in the heap.

  - *Insert*: We push a 0 rank binomial tree corresponding to the new node, to the front of root list. Then we run the grouping operation that merges binomial trees of same rank in the root list to give an ideal binary heap.
  - *Decrease Key*: First, we decrease the distance value of that particular node and then percolate it upwards.
  - *Extract Minimum*: We know that minimum element is present in the root list. So we find the minimum element by traversing root list and remove it, while we add its children to the root list one by one, maintaining the structure property.

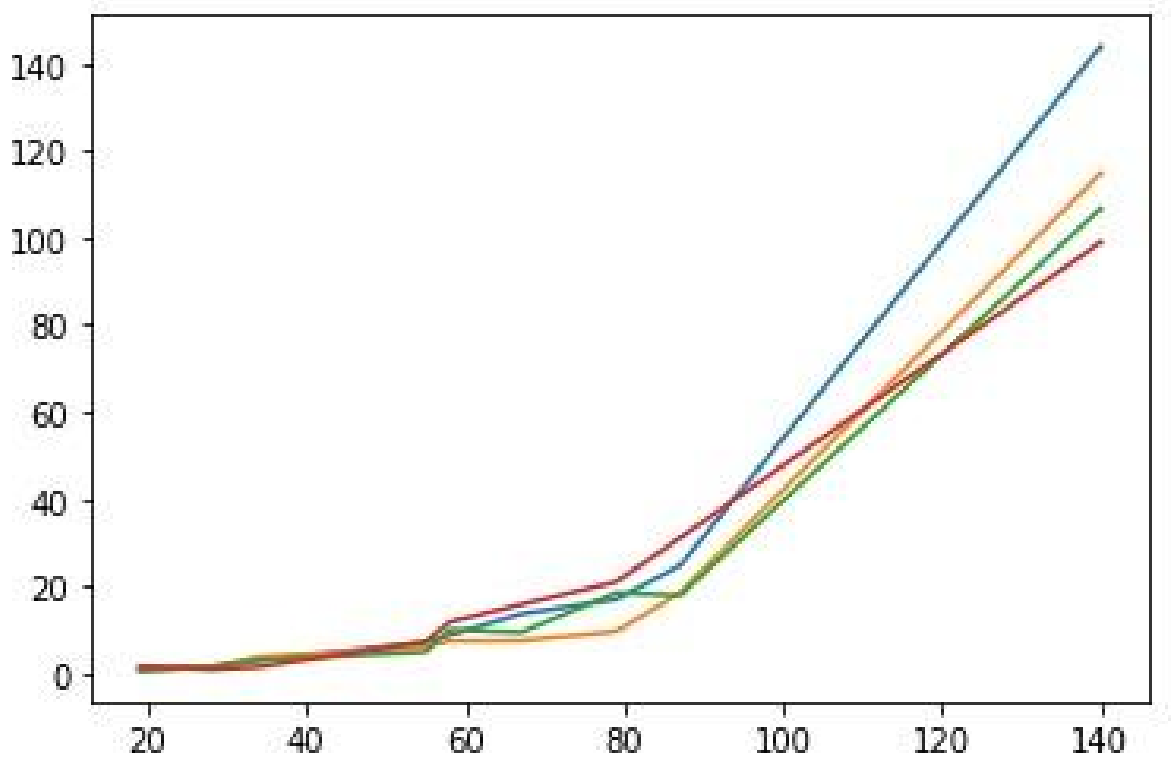
DIJKSTRA (FIBONACCI HEAP):

- Similar to the binary and binomial heap, except for the implementation of **insert**, **decrease key**, and **extract min**.
- Binomial Heap Implementation:

  Note: Here, instead of a root list, we use a circular doubly linked list, and maintain a min pointer. Also, we maintain a vector of pointers that contain the pointer corresponding to each node in the heap.

  - *Insert*: We add a 0 rank binomial tree (one node) corresponding to the new node, to the circular doubly linked list. We don't run the operation to group nodes yet.
  - *Decrease Key*: First, we decrease the distance value of that particular node. Then if its value remains greater than its parent's, we don't do anything. Else, we remove it from its parent node and add it to the root list (circular dll).
    If the parent was unmarked, we mark it. Else, we remove the parent also, and perform this process recursively taking parent as node.
    We also update min pointer when adding any node to root list.
  - *Extract Minimum*: The minimum element is the one corresponding to **min pointer**. We remove the min pointer from root list (to be returned from function) and add the children of min pointer to root list. We then update min pointer to the min value node in root list.
    Now, we group the nodes based on rank similar to how we did in binomial heap decrease key and insert operations.

Time vs Number of Nodes Analysis:



Description:
X-axis: Number of nodes (not to scale)
Y-axis: Time (not to scale)

Array Based **Binomial Heap** **Binary Heap** **Fibonacci Heap**

Inference:

- Initially, i.e, for small number of nodes (<20), the execution times of all the implementations were same.
- Around 50 nodes (39 and 61), array, binomial heap, and binary heap worked equally well, and fibonacci was a bit slow (< 1 sec difference)
- As number of nodes started increasing (> 500), array based results were considerably slower.
- Also, binomial-heap's result were also not as expected. It performed a bit slower probably because during decrease key and extract min, when we had to add the children of node to root list, I added them one by one, instead of creating a new list. This may be the reason for slower binomial heap. This is a conjecture, not confirmed.
- And Fibonacci performed better for higher nodes, equal to or a bit better than binary.