Runtime Analysis of Johnson's Algorithm implemented using various data structures

Sagalpreet Singh
Indian Institute of Technology Ropar
Department of Computer Science
Email: 2019csb1113@iitrpr.ac.in

***Abstract*—The aim of this lab is to analyze the runtime performance of Johnson's algorithm when implemented using different data structures - arrays, binary heaps, binomial heaps and fibonacci heaps. Johnson's algorithm is used to compute all pair shortest paths in a graph. Johnson's algorithm uses Bellman Ford and Dijkstra as subroutines. Although Bellman Ford is implemented casually for all the four different implementations, Dijkstra's implementation differs. To store distance of nodes for the purpose of extracting minimum value, we require a priority queue, we use different data structures for this purpose in all four implementations and analyze the runtime in each case.**

I. Introduction

Johnson's algorithm is used in finding all pair shortest paths in a graph. A very naive algorithm for the purpose of finding all pair shortest paths could be to iterate over all nodes and apply single source shortest path individually to each of the nodes. As graph may contain negative edges, we will have to use Bellman Ford algorithm (an O(VE) algorithm for single source shortest path problem based on dynamic programming). This solution has a time complexity of O(V^2E). In worst case since E can itself be O(V^2), the complexity could be O(V^4). For modern machines, this algorithm is feasible to be applied only for graphs less than 100 vertices. Johnson's algorithm is a great improvement over this naive approach. The basic idea behind Johnson's algorithm is that if we somehow manage to modify the given graph so that all the edges are non-negative without affecting the relative distances between any two nodes through any path, we can use dijkstra instead of bellman ford.

Johnson's algorithm:
1) Add a new vertex v' to the given graph and insert an edge from v' to all the existing vertices with weight = 0
2) Apply bellman ford algorithm with v' as source and store distance of each node from v' as d[v]
3) Modify edge weight w'(u, v) = w(u, v) + d[u] - d[v]
4) Now, apply dijkstra for each node to obtain shortest path for each pair in the modified graph
5) To get shortest path between u and v in original graph add d[v] - d[u] to the shortest path distance between u and v in modified graph.

The key points that prove the correctness of algorithm are that adding d[u] - d[v] to w(u, v) makes edges non-negative without changing relative increase in path distance because for any path between u and v (d[u] - d[v]) is increased as intermediate changes are settled through successive cancellation. This also ensures that all edge weights are now non-negative, because if w'(u, v) were to be negative then d[v] > d[u] + w(u, v) which is a contradiction as d[v] is least possible distance from v'.

The time complexity of this algorithm is 1 times bellman ford + |V| times dijkstra. This is equal to O(VE + V^2log V) assuming that priority queue can perform Decrease Key in O(1) and Extract Min in O(log n). This is however not true for all the data structures that we use in experiment as for arrays extract min is O(n), and it is O(log n) for binary heaps and binomial heaps.

Thus all our implementations of Johnson's algorithm differ only in the dijkstra part.

II. experimentation

.
Following is the analysis and implementation of different implementations of priority queues.

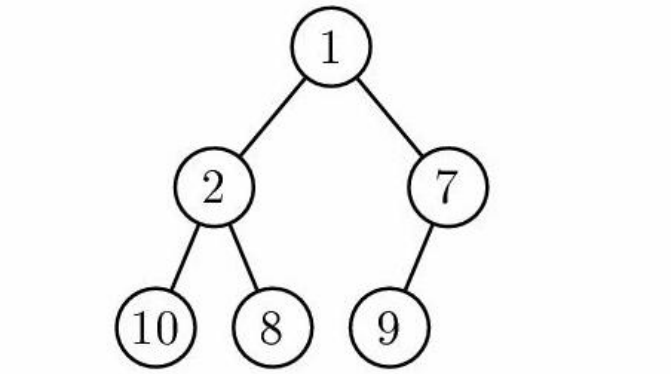
Implementation of Array Based Priority Queue

The values are stored in an array. Since we have random access to each index, Decrease Key works in O(1). While implementing dijkstra, we insert elements only once, that is at the starting. After that we only pop the minimum and never insert a new element. So, we do not remove the element from the array, instead we maintain another array of boolean values indicating whether an element still exists in the priority queue or stores a garbage value. So, an element can be popped in O(1) but to find the minimum, we need to traverse the entire array, which is O(n). Hence the overall time complexity for Extract Min would be O(n). Hence time complexity for Johnson's algorithm would be O(VE + EV^2) which is no better than the naive implementation.

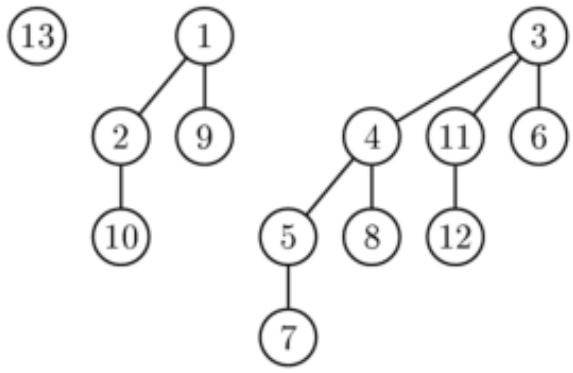
Implementation of Binary Heap Based Priority Queue

The binary heap is also implemented using an array but maintains heap property (each node's parent is less than or greater than the node itself except the root node) at all times due to which Extract Min is O(log n) operation. The Decrease Key operation is also O(log n). To convert into heap is O(log n) operation. Since, we are using binary heaps only for the

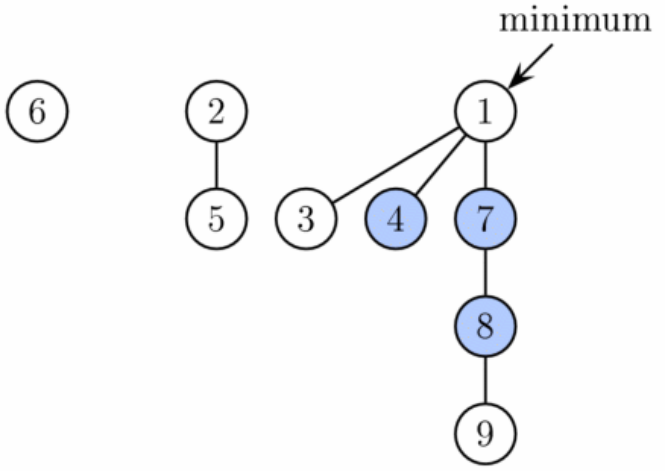
purpose of implementing dijkstra where initially all elements except one are infinite, we can do this in O(1) by putting the non infinite key at the index 0 and order for rest doesn't matter as all of those are infinite. Thus the time complexity of dijkstra algorithm is O(E logV) and Johnson's algorithm is O(VE + VElogV). This is better than the naive approach.



Implementation of Binomial Heap Based Priority Queue
The binomial heaps are implemented as a collection of binomial trees of unique rank. Extract Min and Decrease Key are O(log n) operations for this as well. Inserting a new key is O(log n) operation as trees need to be merged so that no two trees have same rank. Hence the same discussion as for binary heaps follow. Dijkstra algorithm is O(E logV) and Johnson's algorithm is O(VE + VElogV) using binomial heaps.



Implementation of Fibonacci Heap Based Priority Queue
Fibonacci Heaps is another implementation of priority queue where, data is stored in form of different trees connected though each other in form of doubly linked list. Insertion is done by simply appending the new node as a tree in the linked list. Decrease Key amortized is an O(1) operation as it is simply updated and the subtree appended to linked list. A node is also removed if two of its children have already been removed. Extract Min is O(log n) operation as on extracting a key, the subtrees are rearranged by merging so that each tree in the linked list now has a unique degree. Hence, overall dijkstra improves to O(VlogV) and Johnson's algorithm can be implemented in O(V^2logV + VE).



To analyze the performance of all 4 types of priority queues by using them in implementing dijsktra which in turn was used to implement Johnson's algorithm, the code was written in C++ and tested over large graphs, directed and undirected and the runtime was calculated for each case. The details of implementation are as follows:
1) The graph was taken as input in the form of weight matrix but stored in the form of adjacency list.
2) The graph was modified by inserting a new node and putting edges from it to all the existing nodes with weight 0.
3) Bellman Ford was applied with newly added vertex as source and distances from it to all other vertices were stored in distance array. If negative weight cycle is encountered, -1 is printed and program moves to next test case
4) The edge weights for all edges was modified by adding distance[u] - distance[v] to edge (u, v).
5) The additionally added node and edges were removed.
6) Dijsktra was run on the graph for each vertex to obtain all pair shortest paths in modified graph.
7) The all pair shortest paths are printed after adding distance[v] - distance[u] to each value.
8) Finally the time taken is also printed for each test case.

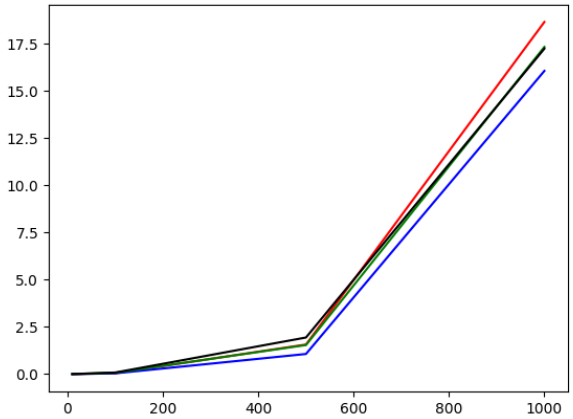
Array based priority queue is implemented in place in the program. However, binary heap, binomial heap and fibonacci heap have been implemented as individual structs with own methods and properties. To maintain uniformity, the method names have been kept constant for all three of them.
The graphs for testing were generated through a python script by creating a matrix of dimensions (n X n) whose each entry is initialized to 999,999 which represented infinity and assigning edge weight randomly between -999 to 999 by some probability ensuring that there is no negative edge cycle present in the graph. To remove possible bias and variations, for each n numerous graphs were created and running time was reported as average of time taken over all cases.

III. Results

The experiment was done in two parts, the first one to analyze average running time for each implementation, and

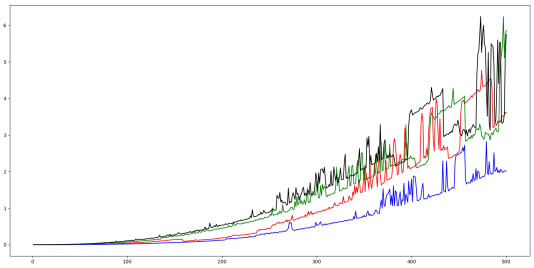
the second one to analyze the time taken for algorithm to run over graphs with specific edge density over a range of 0 to 500 nodes graph.
The first part of experiment was run on 300 different graphs with following specifications:
1) 100 graphs had 10 vertices each and an edge probability ranging uniformly between 0 and 1.
2) 100 graphs had 100 vertices each and an edge probability ranging uniformly between 0 and 1.
3) 100 graphs had 1000 vertices each and an edge probability ranging uniformly between 0 and 1.
These graphs were given as input to all four implementations of Johnson's algorithm and running time plotted.

Following were the average times taken by these implementations.
1) For 10 vertices:
Array Based: 0.00024898 seconds
Binary Heap Based : 0.00032898 seconds
Binomial Heap Based : 0.00093335 seconds
Fibonacci Heap Based : 0.00080896 seconds
2) For 100 vertices:
Array Based: 0.04521615 seconds
Binary Heap Based : 0.03832048 seconds
Binomial Heap Based : 0.06980382 seconds
Fibonacci Heap Based : 0.08137712 seconds
3) For 500 vertices:
Array Based: 1.56739745 seconds
Binary Heap Based : 1.065494 seconds
Binomial Heap Based : 1.53894453 seconds
Fibonacci Heap Based : 1.93903084 seconds
4) For 1000 vertices:
Array Based: 18.672187 seconds
Binary Heap Based : 16.0771801 seconds
Binomial Heap Based : 17.3477766 seconds
Fibonacci Heap Based : 17.2568706 seconds

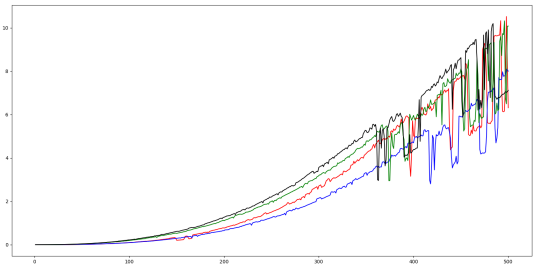


(a) Red: Array | Blue: Binary | Green: Binomial | Black: Fibonacci

The above data clearly indicates that distinction in time taken is very less for graphs with lesser number of nodes. Array Based implementation is most effective for smaller graphs. Binary Heap based implementation takes slightly more time. Fibonacci and Binomial Heaps take 4 times more time as compared to array based implementation for smaller graphs.
For slightly larger graphs (n = 100), binary heaps perform the best followed by array based implementation. Binomial Heaps perform not so good and fibonacci perform the worst although the difference is not very significant. An important thing to note here is that with vertices getting 10 times the previous ones, time for binary increased by approximately 130 times, for binomial by 90 times, for fibonacci by 100 times but for arrays it increased by 200 times.
For even larger graphs (n = 1000) binary heaps perform the best. Fibonacci heaps show a lot of improvement and perform nearly as good as binomial heaps but arrays perform the poorest of all.
The above discussion analyzes average performance for each implementation, further we look into cases, where we analyze performance for specific edge density of graphs. We analyze performance for two different sets of graphs, first ones are graphs of nodes varying from 1 to 500 with edge density 0.25 and second with edge density 0.70.



(b) Edge Density = 0.25



(c) Edge Density = 0.70

Plot b indicates that binary heaps perform better than other data structures for sparse graphs. Binomial heaps and Fibonacci heaps do not perform very well and show great variation in run time, so on analyzing their performance from average run times we observe they may be good in some cases but very bad in others. Also it can be seen that

for larger graphs, performance of arrays gets equally bad as binomial and fibonacci heaps.
Plot c indicates that at significantly large graphs (close to 500 nodes) and edge density of about 0.70, fibonacci heaps perform the best amongst all possible choices. Arrays perform very bad, performance of binary heaps and binomial heaps is equally good.
Comparing plot b and c, it is also seen that on increasing edge density, more time is taken. This is obvious as more edge density would mean more edges implying larger graph.

IV. Conclusion

.
It is quite evident that the theory and practical results are in accordance with each other a lot except for a few seemingly contradictory observations for which we put forward the possible reasons.
For smaller graphs, arrays are performing better even though they have the worst time complexity because binary heap tries to maintain heap property for which it spends some time. Since the number of vertices is small, finding minimum is anyways very easy and array based implementation doesn't require building heap which improves its running time. Fibonacci and Binomial heaps perform poorly because they involve a lot of pointer updates in implementation and for smaller graphs, this takes up most of the time.
With increasing size of graph, arrays start performing poorly as expected but fibonacci heaps perform not so well. This is primarily because it involves a lot of pointer updates. Even for very large graphs fibonacci heaps do not perform the best because the Decrease Key is actually O(1) amortized but the constants involved are actually very high. However, with increasing size of graph fibonacci heaps donot get as worse as others which indicates that for sufficiently large graphs, fibonacci heaps will actually perform better than other implementations of priority queues. This is evident from two facts: The slope of graph is least for fibonacci heaps and when tested specifically for graphs with high edge density it is seen that fibonacci heaps perform the best among all possible options as is evident in plot c.

V. Discussion

.
Johnson's algorithm and Dijkstra's algorithm can thus be implemented using Fibonacci Heap for the best of all discussed worst case time complexity of $O(V^2 \log V + VE)$. This is good asymptotically but may not be a good choice for real life graphs of sizes upto a few thousands or millions vertices because of the high constants involved in its amortized analysis and so many pointer updates. For such purposes, binary heaps offer a better running time in almost all scenarios unless the number of nodes or edge density isn't very high. For those cases it is better to use fibonacci heaps.

Acknowledgment

.
I would like to thank Dr. Puneet Goyal because of whom this experiment could be carried out successfully. His instructions for the course CS201 led to this experiment and its result.

References

.

1) Introduction to Algorithms by Thoms H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
2) GeeksForGeeks articles
3) CS201 Notes by Dr. Puneet Goyal