CS201 : Lab 06 Report

Johnson's Algorithm Using Different Data Structures

Vikram Setty - 2018MED1010
Indian Institute of Technology Ropar

Overview

The analysis explained is with reference to performing Johnson's Algorithm on a directed/undirected graph by representation using an adjacency list (as per the code implemented).

Using notation :

- E : Number of Edges in the Graph
- V : Number of Vertices in the Graph

Johnson's Algorithm : (Bellman Ford's Algorithm) + V*(Dijkstra's Algorithm)

Time Complexity of Bellman Ford's Algorithm : O(EV)

Time Complexity of Dijkstra's Algorithm is not so straight forward. It depends on the time taken for decrease_key & extract_min operation.
decrease_key : O(V) & extract_min : O(V) : Complexity O(V^2 + EV)
decrease_key : O(log(V)) & extract_min : O(log(V)) : Complexity O(Elog(V))
decrease_key : O(1) (amortized) & extract_min : O(log(V)) : Complexity O(Vlog(V) + E)

All these are theoretical findings, and they would be further looked at in detail, and an effort to verify these by using the time taken during execution under different large (may/may not be dense) graphs by the developed code would ve done.

Though no mathematical or sufficient results to prove the theoretical analysis is done here, a fair visualization of the same is however carried out.

Time Complexity Analysis (Theoretical)

Array Implementation :
Decrease Min Operation : O(V)
Extract Min Operation : O(V)
Overall Time Complexity of Johnson's Algorithm : $O(EV + V^3)$

Binary Heap Implementation :
Decrease Min Operation : O(log(V))
Extract Min Operation : O(log(V))
Overall Time Complexity of Johnson's Algorithm : O(EV + EVlog(V))

Binomial Heap Implementation :
Decrease Min Operation : O(log(V))
Extract Min Operation : O(log(V))
Overall Time Complexity of Johnson's Algorithm : O(EV + EVlog(V))

Fibonacci Heap Implementation :
Decrease Min Operation : O(1) (Amortized)
Extract Min Operation : O(log(V))
Overall Time Complexity of Johnson's Algorithm : $O(EV + V^2\log(V))$

Binomial Heaps

Decrease Key : While implementing a decrease_key operation in a binomial heap, the node carrying its new, decreased value keeps on getting swapped with its parent node till its parent node carries a value smaller than it, or till it reaches the root list. In case the node with its value decreased reaches the root list, the head/min pointer pointing to the minimum value node in the heap is updated/not updated accordingly based on simple comparison.

Extract Min : In the extract_min operation, the minimum node in the heap is identified with the help of the head/min pointer. After that, that single node is deleted, and all the children of that node in that tree are recursively integrated by consecutive union operations. The condition that is to be satisfied is that there should be only one tree in the heap corresponding to a single degree. Incase while calling this operation, two trees of the same degree are found, they are merged into a single tree of degree being one more than the previous degree, and the same operation is recursively applied to this new tree. The head/min pointer is then updated.

Fibonacci Heaps

Decrease Key : In the decrease_key operation for fibonacci heaps, in case the node whose value is being decreased is at the root list/still carries a value more than its parent, then nothing is done. Otherwise, the node is cut off from its parent and added to the root list. If that node's old parent was not marked, it becomes marked, and otherwise is also similarly cut off from it's parent and that procedure is recursively repeated till an unmarked node/node of the root list is encountered. The head/min pointer is subsequently updated as usual.

Extract Min : Extract min operation in fibonacci heaps consists of two basic parts, that is firstly identifying the minimum node in the heap with the help of the head/min pointer and subsequently adding the children of that node to the root list while deleting that node. After that, the consolidate function is applied to the heap, which is similar to the merging of trees in binomial heaps except for the fact that the criteria for merging two trees here is based on whether the trees have the same rank and not degree. Rank basically just refers to the number of children of that node. After deleting the minimum node, the head/min pointer is further updated as usual.

Time Complexity Analysis (Experimental)

Results shown below are the times it takes the code running Johnson's Algorithm using different heap data structures to run. To generalize, we take the graph size to be very large. The test cases used to run the analysis are given here.

Before analysing using values of time taken for execution, the code is analysed as to which aspects of it are working, and what are not.

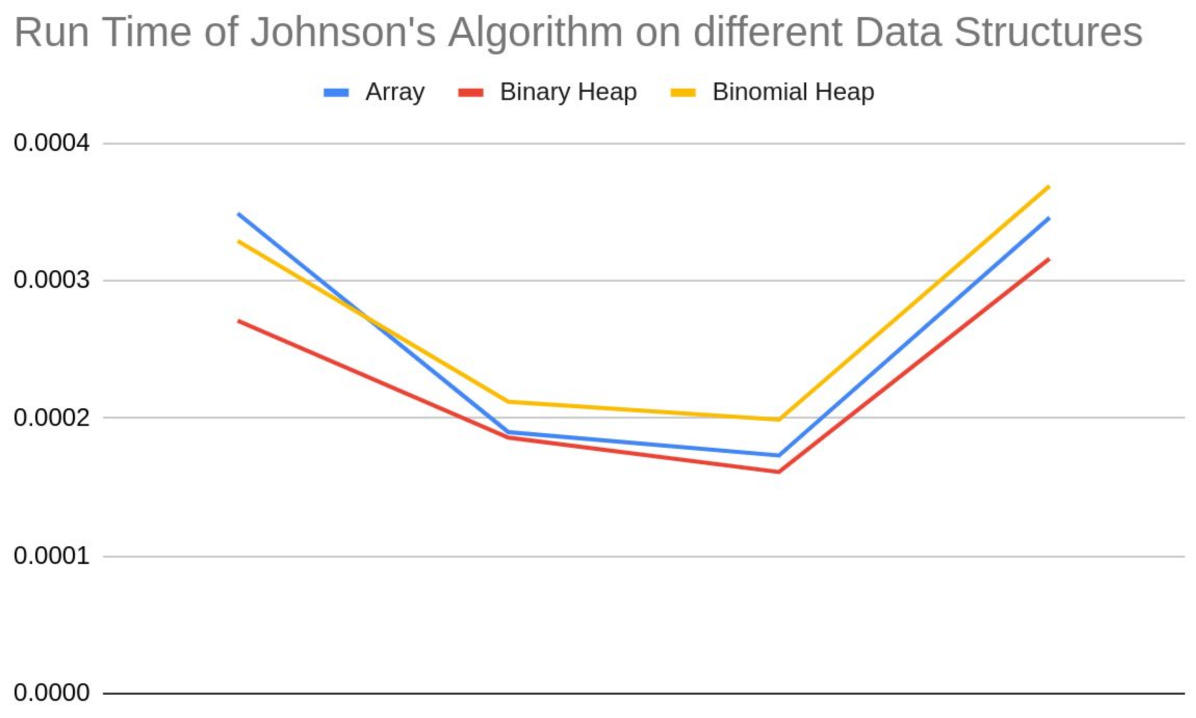
Status of the Code
Array based Implementation : Working well on all values of V
Binary Heap based Implementation : Working well on all values of V
Binomial Heap based Implementation : Working well on all values of V
Fibonacci Heap based Implementation : Currently stuck on a segmentation fault

Evaluating different test cases (under large V in range [30,50]), the time taken for the execution of Johnson's Algorithm Follows :

Test Case	Array	Binary Heap	Binomial Heap	Fibonacci Heap
1	0.000349	0.000271	0.000329	Segfault
2	0.00019	0.000186	0.000212	Segfault
3	0.000173	0.000161	0.000199	Segfault
4	0.000346	0.000316	0.000369	Segfault

Each data entry corresponds to the run time using that data structure in seconds.

This can further be plotted graphically as :



In this chart, the Fibonacci Heap has been omitted as it was not able to give results since the code could not be debugged completely from the segmentation fault stage due to insufficient time.

Conclusions :

- It is evident that a binary heap implementation works better than array based implementation, just by analysing the run times for each of the four tested test cases.
- When it comes to analysing the binomial heap, though run times can be approximated to be similar to the array based implementation, it would however mimic the binary heap implementation more closely when the value of V is significantly increased.
- Another factor to go over the cases when different heap data structures show similar characteristics to arrays to each other is the case when $E \sim O(V^2)$. In this case, the complexity expression in Johnson's Algorithm turns similar for all the three data structures of heaps, that is $O(V^3)$.
- As for fibonacci heaps, a similar statement can be made (under the consideration that the code would have worked). It would have also mimicked the binomial heap type of distribution, but under significant increase in the value of V (not very easy to implement randomly, without having a negative edge weight cycle), the O(1) amortized complexity of the decrease_min operation would change the overall complexity to $O(EV + V^2\log(V))$, which could make a big difference in larger graphs, where $E \sim O(V)$. If $E \sim O(V^2)$, then the complexity once again resembles a binary/binomial heap.