Comparison of the Fibonacci heap with its baselines

Design Laboratory Report

Course ID: CS59001

Name: Uppada Vishnu

Roll No: 16CS30037

Table of Contents

Table of Contents	2
Problem Statement	3
Introduction	3
Properties of Fibonacci Heaps	3
Memory Wise Implementation of Fibonacci heaps	4
Dijkstra algorithm	5
Experiments and Data	6
Results	6
Conclusion	7

Problem Statement

Implementation of the Dijkstra algorithm in an undirected graph using simple array as well as Fibonacci heap and measure the relative performance of the two implementations.

Introduction

Fibonacci heap is a modified form of a binomial heap with more efficient heap operations than that supported by the binomial and binary heaps. But unlike binary heap, a node in the Fibonacci heap can have more than two children.
The Fibonacci heap is called a **Fibonacci** heap because the trees are constructed in a way such that a tree of order n has at least F_{n+2} nodes in it, where F_{n+2} is the $(n + 2)^{nd}$ Fibonacci number.

Properties of Fibonacci Heaps

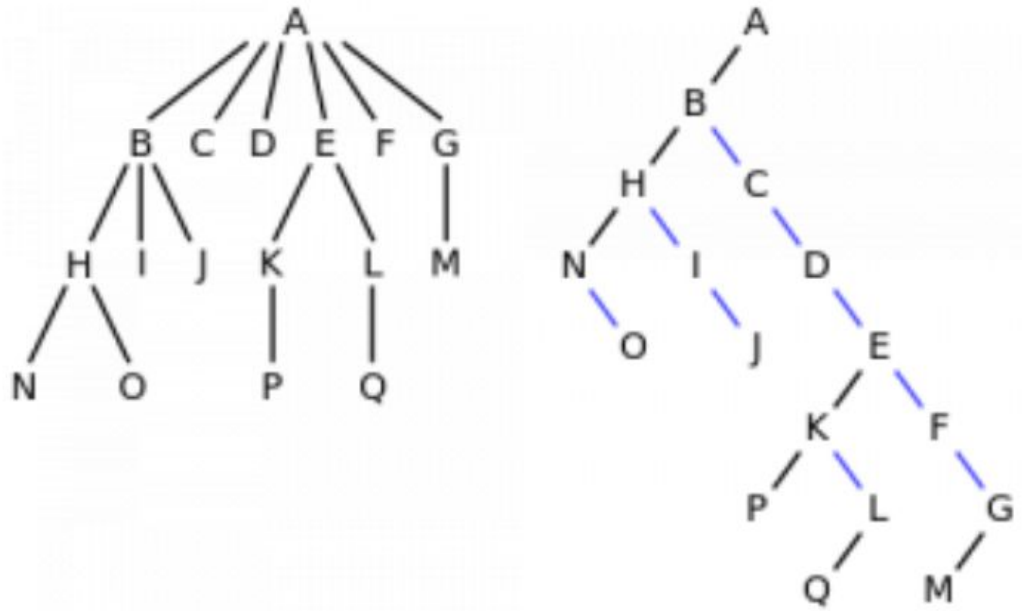
- The trees of the Fibonacci heap are unordered but rooted.
- Fibonacci Heap maintains a pointer to the minimum value (which is the root of a tree). All tree roots are connected using a circular doubly linked list, so all of them can be accessed using a single ‘min’ pointer.
- Nodes also contain a marked variable which makes sure that we don’t cut more than 2 children of a parent.

- Each Tree in it follows min-heap property.
- The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, the time complexity of these algorithms is `O(VLogV + ELogV)`. If Fibonacci Heap is used, then time complexity is improved to `O(VLogV + E)`.
- Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high.

The main idea is to execute operations in a “lazy” way. For example merge operation simply links two heaps, insert operation simply adds a new tree with a single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.

Memory Wise Implementation of Fibonacci heaps

- It is represented using *Left-Child Right Sibling Representation*.
- It is a different representation of an n-ary tree where instead of holding a reference to each and every child node, a node holds just two references, first a reference to its first child, and the other to its immediate next sibling.
- This new transformation not only removes the need for advanced knowledge of the number of children a node has but also limits the number of references to a maximum of two, thereby making it so much easier to code. Below is an example.



Dijkstra algorithm

- Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
- More specifically we are interested in the single-source shortest path.
- Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.
- The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Experiments and Data

- We generate a graph based on the user's given the number of nodes and density.
- Taking the same source node we run the Dijkstra algorithm using the Fibonacci heap, normal Array (O(n)), and Set data structure present in the Standard Template Library (STL) in c++.
- We compare the time taken by each approach in seconds.

Results

- The table below shows the time taken by each implementation of the Dijkstra algorithm in seconds.
- The order is :
  - Fibonacci heap
  - Normal Array
  - STL set

	Density			
Nodes	25	50	80	100
100	0.000457 0.000559 0.000605	0.000744 0.000914 0.000753	0.001123 0.002013 0.001231	0.001326 0.001495 0.001346
300	0.003467 0.004787 0.003511	0.007070 0.007848 0.008675	0.009522 0.011063 0.009851	0.012156 0.013857 0.013070
500	0.010042 0.012642 0.009240	0.019006 0.023681 0.018179	0.031184 0.037330 0.032587	0.038925 0.044536 0.038076

700	0.017062 0.025513 0.017703	0.039711 0.048241 0.038558	0.067359 0.074727 0.062446	0.085724 0.092894 0.078144
1000	0.040813 0.059776 0.039378	0.088742 0.105393 0.083218	0.152232 0.168830 0.138989	0.183861 0.203823 0.170036
3000	0.433164 0.764166 0.404979	0.909014 1.051410 0.827750	1.54217 1.66637 1.49320	1.91031 2.04064 1.75326
5000	1.32910 1.71624 1.17577	3.20152 3.59669 2.67692	4.99781 4.99393 4.08714	5.75430 6.63946 5.24999
8000	3.92193 5.05409 3.47851	8.14112 9.71516 7.35982	13.9564 14.9452 12.2636	20.3657 21.3926 16.7798

Conclusion

- We can observe that Normal array implementation is slower than Fibonacci and STL implementation. This can be accounted for O(n) time for getting the min priority node.
- For lower order of nodes and high density sometimes Fibonacci performed better than STL counterpart.
- For higher nodes and density we observe that STL outperforms the other two methods. This can be accounted for by the larger constant that is present in the time complexity of the Fibonacci heap.