COMP 150-Alg 2, Final Project:

Fibonacci Heaps and Dijkstra's Algorithm

Daniel Mahoney

August 11, 2020

A Fibonacci Heap is a heap data structure; being a heap means that it maintains easy access to the minimum element, and any element is less than or equal to its children.

Most importantly, the interesting operations are extracting the minimum and decreasing the key of a node, which is where the runtime is notably different from a regular heap.

For extraction, we consolidate the trees and combine ones if they have the same degree, so that at the end, there is at most one tree of each degree.

For decreasing a key, we can immediately move it to the root list layer, instead of heapifying as in a regular heap. This provides the decrease key function with an amortized runtime of $O(1)$, in comparison with the $O(\log(n))$ runtime with a regular heap.

This is particularly useful in Dijkstra's algorithm, which performs 1 extract minimum key for each vertex and 1 decrease key operation for each edge. Thus, the overall runtime for Dijkstra's with a regular heap is $O(E\log(V))$, while with a Fibonacci heap, it is $O(E + V\log(V))$.

Note that this is only better if $V << E$, or in other words, we have a pretty dense graph. A complete graph would have $E = O(V^2)$, which would have the runtime be

$$O(V^2 + V\log(V)) = O(V^2) \quad \text{vs.} \quad O(V^2\log(V))$$

0.1 Fibonacci Heap Implementation

In implementing the Fibonacci heaps, I ran into a few hurdles, particularly with the extract min function. The main difficulty was making sure I could loop through the root list (circular linked list) to hit each element only once, but also allowing for the possibility of moving some trees that we've already visited around by connecting with other trees in the consolidation step. I also had some difficulty with the use of the array to merge, which resulted in the Fibonacci Heap nodes to have even more space required per node. Each node has 4 pointers to other nodes (parent, children, left, right), 2 booleans uses for marking (one for decrease key marking, one for helping in the extract min function), and 3 integers for the value stored, the degree (i.e. number of children), and an ID for use in Dijkstra's later.

This can be used with the attached executable ./Fib, which takes user input including 'i <k>' to insert the value k, 'e' to extract the minimum, 'rm' to report the minimum, and 'dk <i><j>' to decrease the i-th key to a new value of j.

Note that the decrease key does not decrease the node with value i to value j, but takes the node that was inserted i-th (0 counting), and decreases its key to the value provided by user (j in the syntax given above). This was for ease of implementation for testing.

I did not implement merge because I have not yet implemented a copy constructor, and the merge function takes another FibHeap as argument. Though the merge function itself would be quite simple, just moving a few pointers to join the root lists.

0.2 Dijkstra's Implementation

To see the Fibonacci heap in action, I created a simple graph in order to run Dijkstra's algorithm. I wrote two versions of Dijkstra's, one that uses the Fibonacci heap I created, and one that uses a regular heap that I adapted from an old heap I made in COMP-15.

I then ran some speed tests on some large graphs to see the runtime in action. For extra verification of correctness, I also compared to see that the heap Dijkstra's and Fibonacci Heap Dijkstra's found the same shortest path lengths.

I predicted that for small graphs, the regular heap would be faster because of the extra overhead time spent in the Fibonacci heap with more pointers and more logic happening. Moreover, for sparse graphs, the regular heap would also likely be faster since if $E = O(V)$, then they have the same time complexity.

For this first set of tests, I had approximately $V\sqrt{V}$ edges. I randomly created edges, with random weights up to 10000. This does allow for repeat edges or self edges, but that doesn't affect the algorithm.

V	E	FibHeap (ms)	Heap (ms)
100	1000	.206	.098
1000	31000	3.750	1.831
10000	1000000	136.340	111.315
100000	31000000	5121.861	4821.051

Unfortunately, we see that in each of these cases, the regular heap is in fact faster. This was the highest power of 10 I could do for vertices before running out of memory for my program.

However, if we look at the trends, we can see how much slower the Fibonacci heap is. At 1000 vertices, it takes about twice as much time. At 10000, about 1.225 as much. And at 100000, about 1.062 times as much as the regular heap. This indicates that for large enough input, the Fibonacci heap may be faster, as we expect.

Another set of tests I ran, was to keep the number of vertices constant, but to increase the number of edges.

V	E	FibHeap (s)	Heap (s)
50000	100000	.168	.066
50000	1000000	.300	.193
50000	10000000	1.599	1.483
50000	100000000	16.160	15.765

These results again fall in line with above, where the regular heap is still faster at my memory limit, but the two sets of times seem to be approaching.

Finally, I ran a separate set of tests which instead of randomly creating edges, it instead created a complete graph. This means there are $\Theta(V^2)$ edges, although in practice I have as many edges as possible without self edges.

V	E	FibHeap (ms)	Heap (ms)
1000	100000	24.337	20.211
5000	2500000	493.779	477.215
10000	100000000	1963.657	1898.557

In conclusion, it appears that in the graphs that I am creating, a Fibonacci heap does not improve the actual runtime of Dijkstra's algorithm. It may be that for larger graphs, a Fibonacci Heap is indeed faster, as we have proved with the amortized runtime analysis. Or perhaps I have made a mistake in the implementation of the Fibonacci heap, or I have coded some portions of it inefficiently. At around the highest amount of memory I can store before being killed, the two heaps appear to be about the same.

0.3 Future Work

If I were to continue or extend this project, there are a few things I might try. Firstly, I would like to see larger graphs, as I have been somewhat constrained by memory limits. And for test data, I would be interested in seeing actual data, such as from a physical source, just to check if realistic graphs are roughly approximated by my random or complete graphs.

Other heap-like structures would be interesting to compare as well. For example, a binomial heap, which is quite similar to the Fibonacci heap but with more restrictions, should have the same amortized runtime as a regular heap. Thus, I would predict it to be in practice worse than the regular heap tested here, since there is more pointer overheap for a binomial heap. And it would likely also be slower than a Fibonacci heap, since the decrease key is still $O(\log(n))$ amortized for a binomial heap, compared to $O(1)$ amortized for Fibonacci, and both have a lot of extra pointer work and general overhead in implementation.

0.4 Links

- Box Folder for Code: `https://tufts.box.com/s/2s0o1buqkwb63kkstrhq6a3qkh2846cm`