Fibonacci Heaps as Priority Queues and Bench-marking

Aniket Kaulavkar, Gaurang Rao, Parth Shah[1]

***Abstract*— This paper highlights and compares the application level differences between binary heaps as priority queues and Fibonacci heaps as priority queues.**

I. Introduction

Fibonacci heaps are data structures capable of implementing priority queue operations. They consist of a collection of heap-ordered trees. The amortized run time of certain operations(such as insert), have been found to be better for Fibonacci heaps when compared to other data structures that implement priority queues, such as binary heaps. [1] Some optimizations such as restricting the size of the circular linked list which holds the root node of the heaps to $log(n)$ where n are the total number of nodes in the heap, optimize performance of certain operations such as deletion of a node. [2]

II. Bench-marking Fibonacci Heaps Against Binary Heaps Using Dijkstra's Algorithm

A. Introduction

Dijkstra's algorithm is an iterative algorithm that calculates the shortest path from one particular starting node to all other nodes in the graph. The algorithm iterates once for every vertex in the graph; however, the order that we iterate over the vertices is controlled by a priority queue. The value that is used to determine the order of the objects in the priority queue is the distance from our starting vertex. The priority queues contains only the fringe nodes. By using a priority queue, we ensure that as we explore one vertex after another, we are always exploring the one with the smallest distance. We will discuss the different data structures that can be used to implement a priority queue and how Fibonacci Heaps perform in the following sub-section.

B. Data Structures and Implementation

Priority Queues can be implemented using heaps, linked lists and arrays. Arrays perform the worst among these. Linked lists only perform a little better when it comes to deletion, as the elements don't have to be shifted. Heaps are generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list. In a Binary Heap, *getHighestPriority()* can be implemented in *O(1)* time, *insert()* can be implemented in *O(Logn)* time and *deleteHighestPriority()* can also be implemented in *O(Logn)* time. With Fibonacci heap, *insert()* and *getHighestPriority()* can be implemented in O(1) amortized time and *deleteHighestPriority()* can be implemented in *O(Logn)* amortized time. In the following sub-section, we have compared STL's priority queue[3], to our own implementation of Fibonacci heaps. The priority queue in the Standard Template Library internally uses a binary heap. Hence, the only performance improvement for Fibonacci Heaps can be seen in the insert operation.

[1]In alphabetical order as there was equal contribution. For Advanced Algorithms (UE17CS311) with Prof. Channa Bankapur, PES University

TABLE I
TIME COMPLEXITY COMPARISON

	Insert	Get-min	Delete-min
Linked List	O(1)	O(n)	O(n)
Binary Heap	O(log n)	O(1)	O(log n)
Fibonacci heap	**O(1)**	O(1)	O(log n)

C. Results and Conclusion

We used a random graph generator to generate 5 test cases, and compared the performance of the two data structures. The results were obtained on a system with a quad-core i5-8259U processor and 8GB RAM. For brevity, we have attached the comparison graphs only for one out of the five test cases. In Figure 1, we can see that Fibonacci heaps perform marginally better than STL's priority queue implementation. Figure 1 represents a reasonably large graph with 418 vertices, and 724 edges.

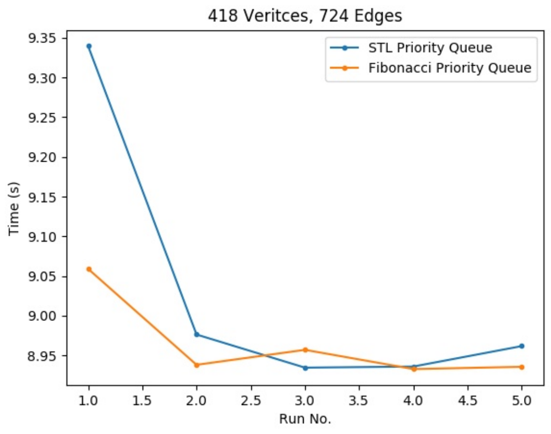

Fig. 1. Times for 418 Vertices, 724 Edges

In Figure 2, we compare the two data structures' average performance in each of the 5 test cases. In most of the runs, the average time shows that Fibonacci heaps perform better. This performance improvement can be credited to it's faster inserts.

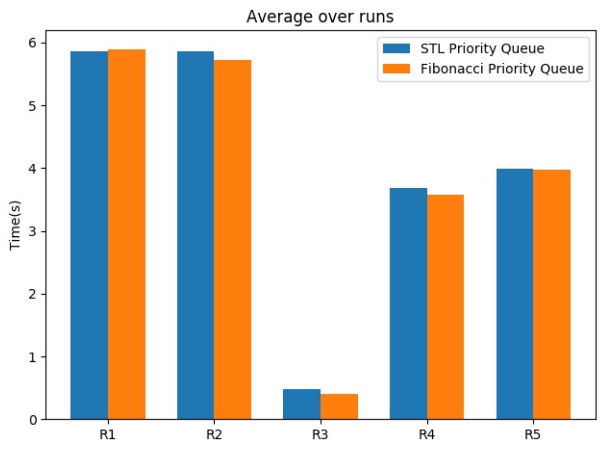

Fig. 2. Average Times over Multiple Runs with Varying Vertices, Edges

III. Process Scheduling Using Fibonacci Heaps

A. Introduction

With an increase in demand for efficiency across computing, and the requirement of prioritising certain processes in a system to meet these demands, we try to implement a scheduler based on a Fibonacci heap, and compare it against a priority scheduler implemented using a binary heap, in the Xv6 operating system.

B. Approach

The Xv6 operating system comes built with a round-robin scheduler. It stores a new process which is created and in the runnable state in the process table. A process is then arbitrarily chosen from this table and changed to the running state. To first benchmark the processes running on a priority scheduler, we modify the process structure in Xv6 to include a priority, represented by an unsigned integer.
We then use a random function to assign a random priority to every process that is generated within the system. This allows us to create a basic priority scheduler, that goes through the process table to find the process with the highest priority before running it.
We use a process spawning program to generate a fixed number of processes by using the fork method to create children processes. Note that each of these processes will now be assigned a random priority as they are created. Each child process goes over the same finite loop doing some arbitrary calculation to ensure that the processes don't terminate too quickly.
We run a general Fibonacci heap user program to test the working of a Fibonacci heap in the operating system.
We then try to modify the scheduler to create a Fibonacci heap based scheduler. At this point, due to the complexity involved in modifying the scheduler to use Fibonacci heaps, we decide to simulate processes, by running various programs using a Fibonacci heap as a user program and bench-marking them against a binary heap (Table II). The simulation of process scheduling was implemented by running a user program that simulated a maximum of 100 runnable processes, with about 75,000 context switches between them.

C. Results and Conclusion

As seen in the results (Table II), we notice that a Binary heap marginally outperforms a Fibonacci heap in our implementation. This could be reasoned out by considering the the relatively small (at-most 100) processes that our program simulates. A Fibonacci heap is expected to significantly outperform other priority queue data structures only at larger input values, as the difference between O(1) operations and O($log(n)$) operations is not visibly significant for relatively small values of n.

TABLE II
Xv6 Process Simulation Times (s)

Run	Fibonacci Heap	Binary Heap
1	0.2466	0.2025
2	0.2487	0.1969
3	0.2000	0.1888
4	0.2031	0.1941
5	0.2419	0.1950

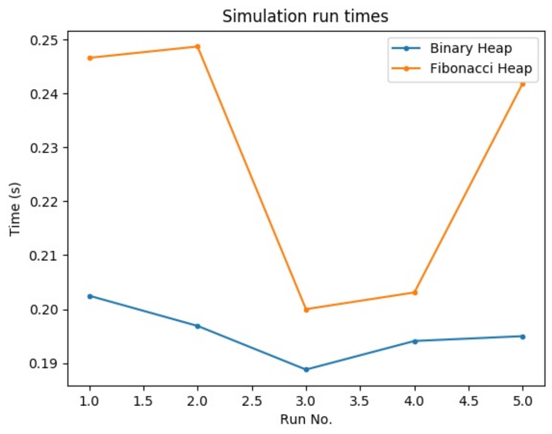

Fig. 3. Simulation run times with 100 processes, 75,000 context switches

References

[1] Fibonacci Heap (Wiki): https://en.wikipedia.org/wiki/Fibonacci_heap
[2] Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, Micheal L Fredman et al. Journal of the Association for Computing Machinery, Vol 34, No. 3, July 1987.
[3] C++ Standard Template Library http://www.cplusplus.com/reference/queue/priority_queue/