Comparative Analysis of Priority Queues in Graph Algorithms

Luke Bird
Riley Fee
Noah Bradley
Computer Science / The University of Alabama
Tuscaloosa, Alabama, United States of America

***Abstract*—Priority queues are fundamental data structures that play a critical role in many graph algorithms, including Dijkstra's shortest path algorithm and Prim's minimum spanning tree algorithm. Although theoretical analysis suggests that advanced priority queues such as Fibonacci heaps provide superior performance as inputs approach infinity, their practical benefits remain debated due to implementation complexity and hidden constant factors.**

In this study, we implement and experimentally evaluate pairing heaps and Fibonacci heaps within the context of Dijkstra's and Prim's algorithms. We analyze their performance across varying graph sizes and structures, focusing on random, grid, and synthetic worst-case graphs. Our results highlight the trade-offs between theoretical and real-world performance, offering insights into when advanced heap structures provide empirical benefits.

***Index Terms*—Dijkstra, Prim, pairing heap, Fibonacci heap, priority queue, experimental analysis**

I. Introduction

Efficient data structures are integral in improving the performance of graph algorithms. Dijkstra's shortest path algorithm and Prim's minimum spanning tree algorithm are two of the most widely used algorithms in computer science. Both algorithms utilize a priority queue to repeatedly select the minimum-priority element and update priorities through decrease-key operations. As a result, their efficiency depends strongly on the speed of these operations.

Most basic implementations of these algorithms use binary heaps due to their simplicity and ease of implementation. However, research has shown that more advanced data structures, such as Fibonacci heaps, can improve performance significantly. Fibonacci heaps allow for decrease-key operations to be completed in constant amortized time, which leads to improved efficiency for Dijkstra's and Prim's algorithms. Despite this advantage, Fibonacci heaps are rarely used in real-world applications because of their comparatively higher overhead and implementation difficulty.

Pairing heaps have emerged as a simpler alternative that often performs well in practice. Although their worst-case time complexity is not fully understood, pairing heaps frequently achieve competitive or superior performance compared to Fibonacci heaps. This difference is a key theme in algorithm engineering: theoretical optimality does not always translate into real-world efficiency.

The goal of this project is to compare Fibonacci and pairing heaps when used in Dijkstra's and Prim's algorithms. We constructed modular implementations of Dijkstra's and Prim's algorithms so they can be interchanged. We then evaluated their performance across a range of graph sizes and structures, including sparse, dense, and worst-case scenarios. We focused on total runtime, the relative costs of extract-min and decrease-key operations, and when advanced priority queues yield meaningful benefits.

This work helps us understand when advanced data structures provide real benefits and when simpler alternatives are more practical. The results highlight the importance of experimental testing in algorithm design.

II. Background

A. Dijkstra's Algorithm

First conceptualized in 1956 by Edsger Dijkstra, Dijkstra's shortest path algorithm is characterized by its simplicity. In fact, it is so simple that he came up with the solution in twenty minutes, with no writing materials, while sitting in a cafe with his fiancé. It was published three years later, shocking him when the easy to understand algorithm became the cornerstone of his fame.

Dijkstra's algorithm begins by assigning each node a distance, with the starting node getting zero and all others getting infinity. Next, select the node with the shortest distance and compare the distance of each directly connected node with the selected node's distance value plus the distance to the compared node. Then, set the compared node's distance to the smaller of the two. After considering all of a node's neighbor's, it can never be rechecked. This process repeats until it selects the destination node or no node can be selected.

B. Prim's Algorithm

Also known as Jarník's algorithm or Prim-Dijkstra's algorithm, Prim's algorithm was actually first developed in 1930 by Vojtěch Jarník. It was then independently rediscovered and republished in the 1950's by both Robert Prim and Edsger Dijkstra, although Prim got there first.

Prim's algorithm is simple in execution. first, initialize a tree starting at any arbitrary point in the graph. Next, repeatedly follow the lowest weight edge and add the corresponding vertex to the tree, repeating until all vertices have been added. the algorithm returns a map of the shortest path

C. Priority Queue Operations

A priority queue is a data type very similar to a classic queue or stack, in that it roughly holds to a first in, first out strategy. The key difference is that each element is also triaged according to its Priority value, with higher or lower values being given preference according to the implementation of the structure.

The three main types of priority queues are Binary heaps, Pairing heaps, and Fibonacci heaps. these all follow the same basic concepts, but with key differences.

D. Theoretical Complexity

Heap Type	Insert	Extract-Min	Decrease-Ke
Binary Heap	O(1)	O(log n)	O(log n)
Pairing Heap	O(1)	O(log n)	O(log n)
Fibonacci Heap	O(1)	O(log n)	O(1)

TABLE I
ASYMPTOTIC TIME COMPLEXITIES

III. IMPLEMENTATION

A. System Design

We decided to primarily use C++ for this project because it is what we were collectively most familiar with, and because it is inherently efficient on a low level.

Our program is organized efficiently, with each algorithm and heap being written in its own file. following object-oriented programming standards, these algorithms and heaps are written in the form of classes that can be called, saved, and used later.

B. Graph Representation

Our graphs are implemented as the Graph class stored in Graph.cpp. Because we are running trials using three different types of graph, there are three different functions that create the graph. Each one procedurally builds and organizes the graph according to the type of graph being created, using a randomly generated seed number as the foundation for the process.

C. Pairing Heap

Our Pairing heap follows the usual implementation of the data class, with the heap being split into its own file and organized into a class. The class is, in fact, for the nodes themselves, with each one carrying pointers to its first child, parent, and siblings. The file also includes all of the necessary functions to use the class as expected.

D. Fibonacci Heap

Similar to the Pairing heap, our Fibonacci heap uses a its own class structure saved in a separate file for modularity. The specifics of the class are very similar as well, with our implementation saving the individual nodes as objects within a larger heap object. The class functions are primarily within the FibonacciHeap class, which is one of the main differences in implementation between this heap and our pairing heap.

E. Instrumentation

Many parts of our data being collected can be tracked by simply outputting the right variables instead of getting rid of them. all of our values related to running time, however, are more difficult. For the times of each trial, there is a call to std::chrono just before and after they begin and end in main.cpp. There also several well times chrono calls to track the call times of the individual functions we are testing.

IV. Experimental Design

A. Environment

Our experiment runs a gauntlet of five trial groups. Each group runs four trials, which are made up of either Dijkstra's or Prim's algorithm and

Pairing heaps or Fibonacci Heaps. Once a combination is used in a group, it will not repeat.

B. Graph Types

We used three distinct types of graphs, each with the goal of modeling different cases based on the graphs which would often cause significant variations in runtime.. The graphs we used are the following:

- Random graphs (sparse vs dense)
- Grid graphs
- Synthetic worst-case graphs

C. Metrics Collected

We measured and collected a number of different types of data to better understand the results of the experiment. These measured variables include:

- Type of graph being measured
- Number of vertices in the graph
- Number of edges in the graph
- Algorithm in the current trial
- Heap in the current trial
- Trial number
- Trial runtime
- Number of insertions
- Number of Decreasemins
- Number of Decreasekeys
- Insertion time
- Decreasemin time
- decreasekeys time

We also have a large quantity of data that can be calculated from the variables that we collected.

V. Results

A. Total Runtime

See Figure 1 for an example of the total runtime.

B. Operation Times

Each operation count is the average time in milliseconds of each operation for the given combination of Algorithm and Heap, rounded down.

- Dijkstra/Fibonacci, Insert: 90,220
- Dijkstra/Pairing, Insert: 91636

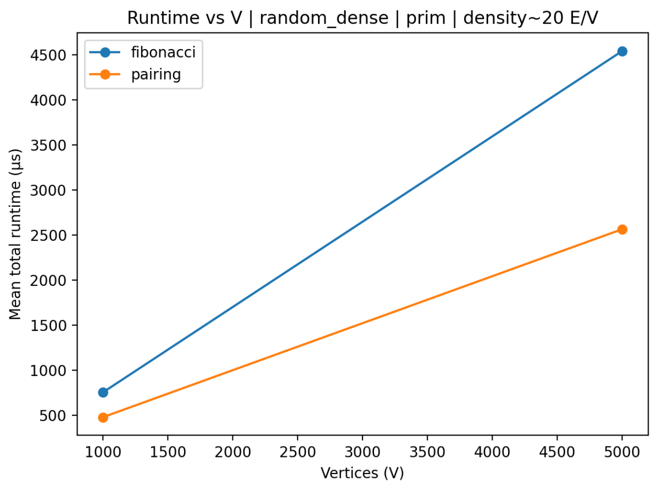

Fig. 1. Runtime comparison (placeholder).

- Prim/Fibonacci, Insert: 101,722
- Prim/Pairing, Insert: 78,050
- Dijkstra/Fibonacci, DeleteMin: 480,693
- Dijkstra/Pairing, DeleteMin: 125,329
- Prim/Fibonacci, DeleteMin: 481,402
- Prim/Pairing, DeleteMin: 99,695
- Dijkstra/Fibonacci, DecreaseKey: 98,365
- Dijkstra/Pairing, DecreaseKey: 90,262
- Prim/Fibonacci, DecreaseKey: 200,936
- Prim/Pairing, DecreaseKey: 172,233

VI. Discussion

A. *Do Fibonacci heaps provide practical benefits?*

According to our results, they do not. Fibonacci heaps were found to have higher runtimes than Pairing heaps in all cases we measured, leading the the conclusion that there is no reason to use the over pairing heaps.

They could be used in come cases, specifically when using Dijkstra's algorithm on a high density graph, but this case simply allows its runtime to stay a flat amount more than that of pairing heaps, instead of doubling it.

B. *How do pairing heaps compare in practice?*

Pairing heaps were faster in every case we tested. In most cases, algorithms using pairing heap data structures completed in half the time. In some of the cases using synthetic worst-case graphs, the pairing heap algorithm was the only one of the two to terminate without an error at all.

C. Which algorithm benefits more (Dijkstra vs Prim)?

The two algorithms receive aprovimately even benefits from the two heaps. in each of our tests, the algorithm used often had the least pronounced effect ut of all other factors on what the outcome would be.

D. How does graph structure affect performance?

Graph structure affects performance in several ways. One is that the density of node placement has a direct impact its runtime, leading to the only case in which fibonacci heaps are a semi-usable option.

Grid Graphs lead to drastically lower runtimes for both types of heap, regardless of whether they are using dijkstra or prim's algorithms. this is likely because of the even and uniform placements of the nodes doing away with any form of bottleneck.

Random graphs caused relatively even time patterns across the board, with the exception of one case. Dijkstra's algorithm with pairing heaps was slowed down do much in random, dense graphs that it is almost on-par with the speed of the fibonacci heap.

E. Why do theory and practice differ?

Theory and practice differ here for the same reason that they often do: theory is usually made under the assumption of ideal circumstances, and ideal circumstances are almost impossible to come about in real life.

In this case, the theory differs because it does not look at the full picture. The fibonacci heap function runs with a theoretical time complexity of O(1) in for most of its functions. However, one of the only longer functions, extract-min, takes up the vast majority of its runtime in our experiment, drastically increasing its runtime beyond what is predicted by the time complexity.

VII. Threats to Validity

As with any other experiment, there ar any number of factors that may have introduced error into these results and must be kept in mind when reading. these sources of error may include:

- Measurement Noise:

  Even in measurements taken by a computer, it is possible for there to be inaccuracies and slight deviatons from the perfect reading.

- Graph Generation Bias:

  Our graphs were generated using a randomly generated seed. While this works perfectly fine for our purposes, true random in computing is a problem that is still being solved today.

- Implementation:

  Human error can introduce mistakes anywhere in a process, even when we are being as careful as possible.

VIII. Conclusion

In conclusion, the Fibonacci Heap and Pairing Heap data structures offer massive benefits to both Dijkstra's algorithm and Prim's algorithm. They organize the paths in a way such that they can be searched faster and therefore return a solution in a fraction of the time.

After careful analysis of the two, however, it is clear that pairing heaps are indeed the superior option in most cases. When used they led to consistently faster runtimes, regardless of whether the algorithm they were being used with. This comes in contradiction to the common theory due to the heaps' respective member functions and their time complexities.