Prim's Algorithm Analysis

Tyler Laudenslager
CSC 402 - Data Structures 2

April 9, 2021

Abstract

I will test the efficiency between two data structures by using them as a means to implement Prim's algorithm to find the minimum cost spanning tree of a connected graph. One implementation used a C++ Standard Library priority queue. The other used a custom priority queue called HeapADT. I first cover program structure and design of Prim's algorithm implemented in the C++ programming language. I then cover the details of how both priority queues are implemented. Furthermore, I discuss the results of the data analysis I performed and give the conclusion that the custom HeapADT priority queue is slightly faster than the Standard Library priority queue.

1 Introduction

I am going to conduct a performance analysis on Prim's algorithm that is implemented in two different ways using the C++ programming language to find the minimum cost spanning tree of a connected graph. One implementation used a standard library priority queue that simulates a binary heap. The other used a non standard library priority queue that simulates a binary heap. Both implementations are the same except for the differences already mentioned above.

1.1 Program Design

The two implementations of Prim's algorithm written in C++ were designed to be clear, concise and readable. The program reads in a file that contains a connected graph in the format described below. The first line of the file has the number of vertices of the connected graph. The subsequent lines in the file contain the unique edges in the connected graph. The lines that are meant to represent edges have the following format "vertex one","vertex two", and "edge cost" with spaces in between each value in the line. The vertices in the line represent the nodes in the graph that are incident to a particular edge. The "edge cost" is the cost of travel to get from one vertex to the other along the edge. The vertices in the edge are labeled using the lowercase English alphabet starting with the character "a". The program works with any connected graph being a graph of degree 2 up to and including any graph of degree $n - 1$ with n being the amount of vertices in the graph. The vertex that the program starts at is arbitrary, meaning I will get the same results no matter which starting vertex is used.

1.1.1 Structure & Control Flow

Both implementations of Prim's algorithm begin by reading in a file that represents a connected graph as described above. The programs then store all the edges of the graph in a adjacency list data structure. The adjacency list is a mapping of vertices, to an unordered set of neighboring edges, in a undirected graph. Then the program creates a priority queue, that holds all the edges in the graph, that need to be visited with the minimum cost edge always at the beginning of the queue. The program maintains a vector that marks which vertices have been visited so far in which the value is either true or false corresponding to the index the vertex represents. If the algorithm has not visited this vertex yet the cost to travel along the minimum cost edge is added to the total cost of the final tree. I also save the edge in a mapping where the cost of the edge is the key and the edge is the value. I then add all other edges, neighboring the vertex we are currently at to the priority queue (only if adding the edges does not create a cycle in the graph). This pattern continues until the priority queue is empty and all edge options have been considered with priority going to the least cost option. After the algorithm runs, we now have: the minimum cost spanning tree in the mapping of edge costs to edges; and the total minimum cost of the spanning tree.

1.1.2 Standard Library Priority Queue

One of the implementations of Prim's algorithm uses a C++ Standard Library priority queue from the #include queue header library. The priority queue constructor takes three formal arguments as follows: priority_queue(type of element, underlying container, compare function). The first argument specifies the type of element I am storing in the priority queue. The second argument specifies the underlying container I want to use to contain the elements that need to be prioritize. The third argument specifies a comparision function for ordering. I simply insert an element, into the priority queue, and the priority queue handles all the complexity of ordering. We cannot choose which element gets removed. All we can do is get the next top element from the priority queue. In this particular case, the top element in the priority queue is always the edge with the least cost of travel.

1.1.3 HeapADT Data Structure Priority Queue

The second implementation of Prim's algorithm uses a custom generic priority queue, implemented as a binary heap called HeapADT, that can store any type of element we choose. This data structure uses a vector as the underlying container to store values. After every insertion into the vector we make sure the vector maintains the binary heap invariant in which, the parent node is less than or equal to the children nodes. Notice that when we remove an element from the priority queue we also have to maintain the binary heap invariant as well. Using a vector, we can access each node in the binary heap as follows. The parent node of any node x, where k represents the index of the element, is located at index

$$\lfloor\frac{k-1}{2}\rfloor \ni k \in \mathbb{W} \qquad \mathbb{W} = \{0, 1, 2, ...\} \tag{1}$$

The left child of a parent node can be located in the vector at the index of

$$2k + 1 \ni k \in \mathbb{W} \qquad \mathbb{W} = \{0, 1, 2, ...\} \tag{2}$$

where k is the index of the parent node the root being index 0. The right child of a parent node can be located in the vector at the index of

$$2k + 2 \ni k \in \mathbb{W}. \qquad \mathbb{W} = \{0, 1, 2, ...\} \tag{3}$$

The notation above can be translated as follows. The index of the right child, of a parent node, is 2k+2 such that k is in the set of whole numbers. This is because the index of the first element in the vector begins at zero instead of one. Being able to find which element I want directly, using the above equations, I can simply swap value locations inside the vector. This swapping of value locations in the vector is needed to maintain proper ordering of the priority queue. The same property holds with the HeapADT priority queue as with the Standard Library priority queue. We can only remove the element that is next in the queue, which is the edge with the least cost.

1.2 Expectations

I expect that the Standard Library priority queue will be noticeably faster than the custom HeapADT data structure. This is because I believe that the Standard Library priority queue should be more optimized considering it is a part of the core language that has been tested before being distributed. I hope to find something interesting in how long the implementations take to find the minimum spanning tree, of all applicable degrees of graphs with vertices of 8 and 9 in particular. I am also hoping I can discover a particular graph of a specific amount of vertices and in-degree, that takes substantially longer time to find the minimum spanning tree than other graphs.

2 Experiments

I realized very early on, thinking about how I wanted to design the way I analyzed the data, that I needed to automate the process. This would provide a streamlined way to test both implementations of Prim's Algorithm with any number of graphs as many times as I needed. This automation of the boring stuff, so to speak, allowed me a lot of time to think about the data I collected and substantially reduce the amount of time needed to collect the data to be analyzed.

2.1 Data Collection

One of the first thoughts I had was to limit the amount of code I needed to change in both implementations of Prim's algorithm. I decided I could use input redirection and output redirection, which are built-in features of the Bash shell in Linux for any input/output I needed to supply to the programs. In order to avoid errors, very few modifications were made to the C++ implementations. I decided to make three bash scripts. The first one would make all the connected graph files I wanted to run through Prim's algorithm. The second bash script would handle testing each graph that was produced using both implementations of Prim's Algorithm. Lastly, the third bash script gets the range of vertices each graph will have and runs the second bash script ten times. This way I can run one bash script and get a new timing file almost instantly instead of manually entering the timings, into an excel spreadsheet, which is tedious and impractical.

2.1.1 Bash Scripting

gen_graphs.bash The first bash script I wrote was designed to generate a graph file using a Python3 program I built. This program generates the properly formatted connected graph files that are used as input to C++ implementations of Prim's Algorithm. The graphs can be of any in-degree within the limits of graph theory. An odd numbered vertex graph can only have even in-degrees. Example, if I have a graph with six vertices than the degrees can be 2, 4, or 6. In contrast a even numbered vertex graph can have any number of in-degree starting at 2 up to and including $n - 1$ in-degree such that n is the total number of vertices in the graph.

analyze.bash The second bash script I wrote collects all the graphs that were generated using the **gen_graphs.bash** script above. The script then runs each graph through both implementations of Prim's algorithm. I then append each timing as well as the implementation used with the vertex and degree of the graph tested separated by commas (commonly referred to as a CSV (Comma Separated Values) file). I carefully extracted the necessary data, from the name of the graph file, using regular expressions. This allowed me to keep track of which timings belong to which graph and implementation I used.

gen_data.bash The third and final bash script I wrote takes two arguments that represent a range of vertices to collect data one. I understood that the more data I had to analyze the more things I might be able to uncover. I preferred to have more data, than I would need, instead of not enough data. For each number in that range specified **gen_graph.bash** will produce the corresponding graph. After all graphs have been generated, the graphs will exist in a folder so I can run **analyze.bash** as many times as needed. In this particular case I ran **analyze.bash** 10 times.

2.2 Python3 & Pandas

Naturally, since I created a CSV (Comma Separated Values) file, I decided to import and use the Pandas library which is a third party library designed to be used with the Python3 programming language. I could not have described the Pandas library better than how the official website explained the library. "The Pandas library is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language". Pandas allows for sorting, filtering and plotting of data into an understandable format that can be analyzed. This allows for much more expressive ways to show the data you have found to others without much work. Using the Pandas library, along with a Juptyer Notebook, is a very common way to show others an analysis of the data without going into the details of explaining syntax of a programming language.

2.3 Jupyter Notebooks

I wanted the data analysis to be automated as well so I could get a fresh data file and instantly know how the data analysis had changed with new data instead of making new graphs each time I have new data. To do this, I employed the help of a Jupyter Notebook. Jupyter Notebooks are used to have an executable document, that anyone can modify and execute existing snippets of code. In this specific instance I can generate graphs such as bar charts and histograms from the data I have gathered and then save the notebook. I can provide the notebook, along with the code I created, to others that can continue where I have left off and do more data analysis. A Jupyter Notebook can be restarted and ran with fresh data anytime giving much more insight into data than an excel spreadsheet.

3 Results

To begin, I can see that as the degree increases so does the time it takes to find the minimum spanning tree.

```
        difference
degree
2             7.64
3             7.77
4             6.40
5             6.82
6            12.66
7            14.20
8            14.43
9            15.30
10           22.15
11           26.20
```

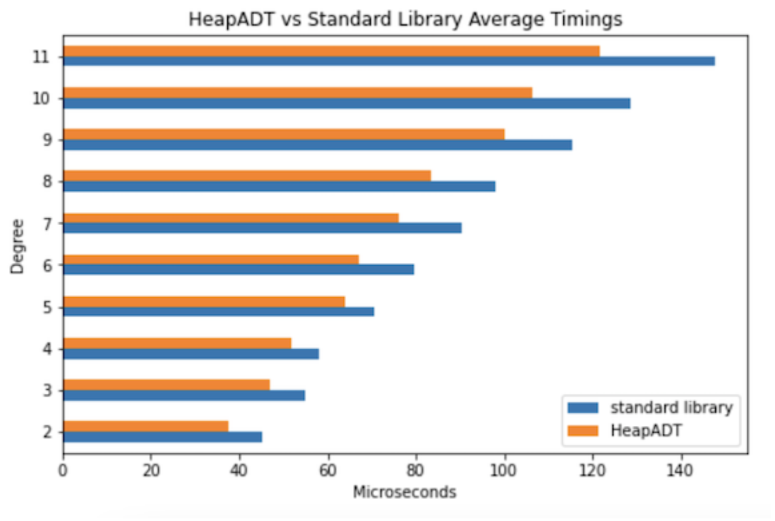

Figure 1: Average Timings for Each Degree

This is not surprising because there are more edges to look at as the degree increases. However, I can see a noticeable difference in the time it takes the HeapADT priority queue implementation and the Standard Library priority queue implementation. This is surprising because I thought that the Standard Library implementation would be faster than the HeapADT implementation. The data shows that as we increase the degree, the difference in time it takes to find the minimum spanning tree also increases between both implementations. This gives some insight into the data, however, there are more lower degree timings that were averaged out than the higher degrees. The horizontal bar chart above shows the average timings of graphs with vertices of 5 through 12 including all applicable degrees.

```
degree      two  three  four  five  six  seven
difference  8.8    3.4   7.3  11.2  9.6   15.7
```

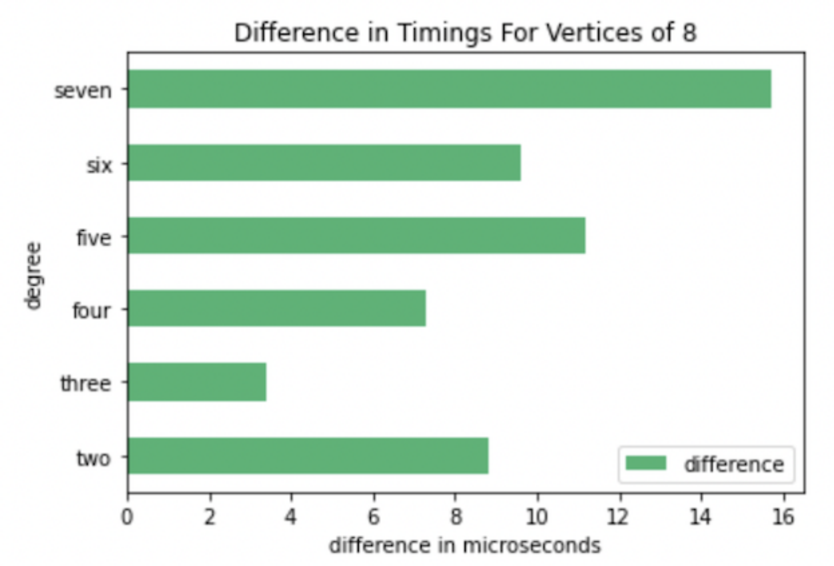

Figure 2: Difference in timings of graphs with 8 vertices

The previous graph shows that the HeapADT implementation is faster for all degrees, however, I wanted to know how much faster for specific vertices. Therefore, The graph above shows how much slower the standard library is depending on the in-degree all the vertices in the graph have. The difference shown is the timings for all applicable degrees of graph that has eight vertices. I notice that there is a slight difference in timings between the graphs that have an in-degree of three compared to graphs with an in-degree of two. This difference in timings as we see is double the time for the graphs with in-degree of two as compared with the graphs that have an in-degree of three. Also, I find it interesting that the difference in timings is longer for graphs that have an in-degree of five than for graphs that have an in-degree of six. I thought that the in-degree of six would have a greater difference than the in-degree of five timings. I was also surprised that graphs with a in-degree of 3 have a smaller difference in timings than the graphs with degree of two. All of the differences between both implementations are within twelve microseconds, which I would not say is a significant difference in performance depending on in-degree. I used graphs with eight vertices because it would provide more data to chart in order to obtain a better understanding.

The next graph shows the exact same difference in timings, however using graphs with 9 vertices instead. I uncovered something interesting in this graph that I hoped to find, but did not expect to find while analyzing the data. In the graph below, there is an almost multiple of three increase in time between an in-degree of four versus an in-degree of six. Also, I can notice that there is a noticeable difference in timings between degree six and degree eight. I found this interesting because there are more edges to analyze for a 9 vertex graph with each node having an in-degree of eight. I thought this would make the algorithm take a longer time to run than the same graph in which each node has a in-degree of six.

```
degree      two  four   six  eight
difference  5.5   6.0  17.2    8.1
```

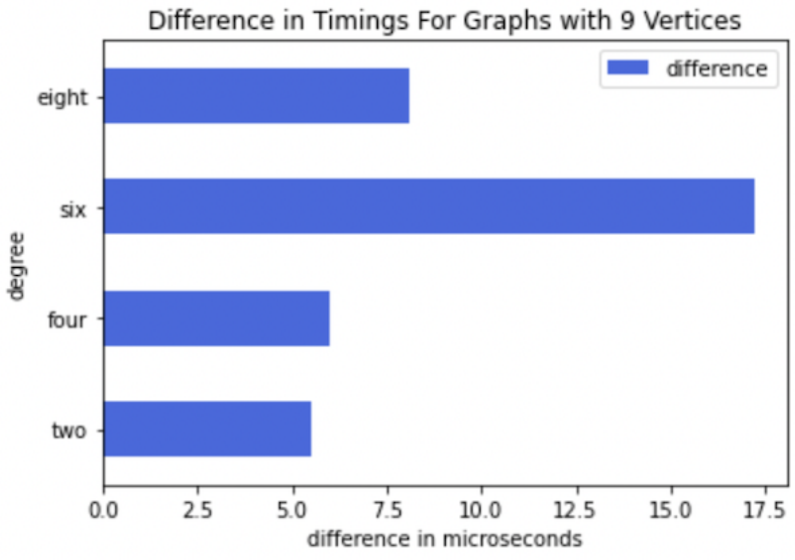

Figure 3: Difference in timings of graphs with 9 vertices

Notice how the in-degree differences of in-degree two and four are very close. However, when we want to find the minimum spanning tree of a graph with an in-degree of six the difference becomes quite apparent. While the differences in this graph do not exceed past 16 microseconds I find it interesting that a graph with an in-degree of six would have such a noticeable difference from an in-degree of eight. The reason that there are not any odd in-degree differences in this graph is because we are using a graph that has an odd number of vertices. I cannot have a connected graph with each node having an odd in-degree when the graph has an odd number of vertices, because adding an edge to the graph to increase one vertices' in-degree also will increase another vertices' in-degree. Thus, having a connected graph that has an odd amount of vertices as well as every vertex having an odd in-degree is impossible.

4 Analysis & Conclusions

From the data I have gathered and the analysis I have performed, I can conclude that the custom priority queue HeapADT is slightly faster than the C++ Standard Library implementation of the priority queue. I suspect because of Figure 1 that if I were to analyze graphs with a higher number of vertices I would start to see a larger difference in performance. It appears that there is large difference in the the time it takes to find the minimum cost spanning tree for graphs that have nine vertices, with an in-degree of six, than all other graphs that have nine vertices with any other in-degree. I suspect this is because the algorithm has to search for a vertex belonging to an edge that the algorithm as not visited yet. However, I understand there may be unknown factors that could be manipulating my data in unforeseen ways.

5 Bibliography

Dr. Spiegel - Author of HeapADT generic priority queue.

http://faculty.kutztown.edu/spiegel/CSc237/Examples/HeapADT/VectorImplementation/

Aakash Prabhu - Author of the starting code that was adapted to build a bigger program.

https://github.com/aakash1104/Graph-Algorithms/blob/master/PrimsAlgorithm.cpp

Project Documentation:
https://kuvapcsitrd01.kutztown.edu/ tlaud746/CSC402/Project2/

Various utilities I used to analzye the data and create the programs.

Pandas Website : https://pandas.pydata.org
Python Website : https://www.python.org
Jupyter Notebook: https://jupyter.org/index.html
C++ Programming Language