Programming Assignment 3: Graphs

Jacob Schuetter

16 November 2023

1. **Introduction**

Following the programming portion of these assignments, I often look back and wonder if I overcomplicated the process. Usually, it just comes down to a desire to implement a robust class that could hypothetically be used for further projects. This project is no exception. While there are still a number of member functions I would need to implement for a truly complete version of the class, I did end up going to quite a bit of trouble to implement and debug features that, for our single use case, probably weren’t necessary – like template classes, for example. As of the beginning of this project, we had not yet fully covered them in CS 216, so it took me a day or two to figure out how to get them to work, but eventually, everything fell into place. As usual, I implemented test functions in my main source file to test each class and demonstrate its functionality. To run, simply compile and run main() in Source.cpp.

1. **Minheap**

The most significant design choice I made with this program was to use the array-based tree implementation for Minheap rather than the node-based implementation. This decision was primarily to make it easier to insert nodes to the heap (allowing simply appending the node at the end of a vector instead of trying to keep a pointer to the next available node slot). This choice also led to a number of other effects down the line. Other than this, implementing Minheap went smoothly and about as expected. The only things I had to add after the fact were two functions weight(n) and count(n), in order to enable functionality I needed in the PQueue class.

1. **PQueue**

Implementing the priority queue was similarly straightforward. Again, there were only two additional member functions needed here, count(n) and getPriority(n), both of which I had to implement as I was trying to add Prim’s algorithm to my Graph class. I used Minheap to implement this class, since it seemed to be the most straightforward way to do it, rather than having to constantly sort another data structure. This worked beautifully for my implementation of Dijkstra’s algorithm later on, but I did discover that because of this choice, my current version of PQueue does not function as an ordinary queue. If I did this project a second time, I might make an effort to change that.

1. **Graph**

Graph is where things got a little hairy. The greatest changes I had to make while I was implementing it had to do with the class data members. Initially, I had used a hash table mapped to a linked list to store my vertices and edges, which worked for a while, and probably would have continued to work, but I realized it would be more efficient to use a second hash table instead. So, I made that change, and, as all such changes do, it broke things, but everything was back to functionality before too long. I also eventually decided to make hash tables for storing node predecessor and node distance into data members of the class. I initially was not doing this, since they are not technically properties of the graph itself, but between both search algorithms and their respective print algorithms, I eventually realized it would be much easier to simply store those maps as members of the graph object. This does mean that there could be erroneous or outdated values stored in these data members if called outside the member functions they are intended to be used with, but I did not run into any such issues in my implementation. If I wanted to fix this, I might make a new object containing these maps that would be the return value of each search algorithm and an argument of each print function. The printout of the adjacency map of the provided graph can be found below.

Vertices: | Adjacent vertices (weight):  
 Zerind | Oradea (71), Arad (75),  
 Hirsova | Eforie (86), Urziceni (98),  
 Arad | Sibiu (140), Zerind (75), Timisoara (118),  
 Bucharest | Urziceni (85), Pitesti (101), Giurgiu (90), Fagaras (211),  
 Craiova | RimnicuVilcea (146), Dobreta (120), Pitesti (138),  
 Dobreta | Craiova (120), Mehadia (75),  
 Eforie | Hirsova (86),  
 Fagaras | Bucharest (211), Sibiu (99),  
 Giurgiu | Bucharest (90),  
 Iasi | Neamt (87), Vaslui (92),  
 Lugoj | Mehadia (70), Timisoara (111),  
 Mehadia | Dobreta (75), Lugoj (70),  
 Neamt | Iasi (87),  
 Oradea | Sibiu (151), Zerind (71),  
 Pitesti | Bucharest (101), Craiova (138), RimnicuVilcea (97),  
 RimnicuVilcea | Craiova (146), Pitesti (97), Sibiu (80),  
 Sibiu | Oradea (151), Arad (140), Fagaras (99), RimnicuVilcea (80),  
 Timisoara | Arad (118), Lugoj (111),  
 Urziceni | Bucharest (85), Hirsova (98), Vaslui (142),  
 Vaslui | Iasi (92), Urziceni (142),

1. **Breadth-First Search**

Breadth-first search was fairly simple to implement. The only issues I ran into have, in large part, already been mentioned. First, I discovered that my PQueue class does not have the first-in-first-out functionality of an ordinary queue, so I had to change my implementation to use the standard library data structure. Other than that, the only major issues were in my print functions. Originally, I was going two functions: one to print BFS paths and another to print Dijkstra’s paths. In trying to implement these, however, I was running into a number of issues with output formatting that I still haven’t gotten to the bottom of. Instead, I decided to circumvent the process by changing the print path function to a find path function, changing the return type from void to a vector containing the nodes on the path. I then implemented a separate print function that would display the path contained in the vector. I also implemented a second version of the findPath function to find the path from the source to multiple destinations, simply for the ease of demonstration.

Shortest paths (unweighted):

Arad -> Sibiu: Arad, Sibiu

Arad -> Craiova: Arad, Sibiu, RimnicuVilcea, Craiova

Arad -> Bucharest: Arad, Sibiu, Fagaras, Bucharest

1. **Dijkstra’s Algorithm**

For Dijkstra’s Algorithm, like BFS, I originally implemented three separate auxiliary data structures, for storing each node’s predecessor, distance, and color. Then, as mentioned before, I moved the former two structs to be data members of the Graph object. The color map still remains in the implementation of the function, since it is not needed outside the function implementation. I did have to add one line to the provided pseudocode, to set the initial distance value of the source node to 0, but aside from that, everything went rather smoothly.

The biggest problem with my present implementation of Dijkstra’s is my decreaseKey function, in PQueue and (by association) Minheap. At present, the weights in the queue are separate from the node’s assignment in the distance map, which represents a rather significant memory inefficiency. Additionally, and perhaps more concerningly, my present implementation uses a linear search to find the node whose priority is being decreased. I’m not sure if there is a better way to do this or not, given that the Minheap structure is inherently unsorted, but it is a notable inefficiency in the code.

Below is the path Dijkstra’s found from Arad to Bucharest. It is different from the path found by the BFS algorithm, because, despite passing through one node more than is strictly necessary, the weighting pattern of the graph offers a “shorter” (i.e. of lesser total weight) route following the path below:   
Arad, Sibiu, RimnicuVilcea, Pitesti, Bucharest

1. Prim’s Algorithm

Being already somewhat familiar with Prim’s Algorithm from discrete math, it was not at all difficult to implement. It did involve implementing a count function for PQueue, similar to the standard function provided with the std::set class, as well as a getPriority function. Similar to decreaseKey, both of these functions make use of a linear search. Unlike the above, however, these functions could probably be improved or eliminated entirely.

As mentioned in class, adding an inQueue flag to each node could easily eliminate the need for the PQueue::count function, but given my choice to make the graph based in an array implementation, this would have required me to redefine the type around which my graph was structured and likely to rewrite the majority of my code as well. Rather than do that, I decided simply to go with what worked and was not overly time-consuming (or prone to overcomplicate my code). In a similar vein, PQueue::getPriority could also probably have been eliminated if I had instead used the distance data member of the Graph object to store the distance along the current shortest path to each node. As above, this could potentially be further streamlined if I had somehow integrated the distance data member into the PQueue as the weight of each node, but it seemed simpler and more robust to handle each separately.

The minimum spanning tree found by the algorithm may be found below, both as a printout and visual graph.

A map of the mountains

Description automatically generated with medium confidenceSource node: Arad  
 Vertex: | Predecesor:  
 Sibiu | Arad  
 Zerind | Arad  
 Hirsova | Urziceni  
 Timisoara | Lugoj  
 Oradea | Zerind  
 Fagaras | Sibiu  
 RimnicuVilcea | Sibiu  
 Craiova | Pitesti  
 Pitesti | RimnicuVilcea  
 Bucharest | Pitesti  
 Urziceni | Bucharest  
 Giurgiu | Bucharest  
 Vaslui | Urziceni  
 Dobreta | Craiova  
 Eforie | Hirsova  
 Mehadia | Dobreta  
 Lugoj | Mehadia  
 Iasi | Vaslui  
 Neamt | Iasi