Andy Fleischer

Prof. Karla Fant

CS163

8 June 2021

Program 5 Efficiency Writeup - Graph

This program was required to store information on a trail system via a table abstract data type, using a graph data structure implemented with an adjacency list. In the graph, vertices represented trailheads and trail intersections, and each one had a name (dynamically allocated array of characters). Edges represented connections between trail vertices, and each had a length in miles (float) and a difficulty (integer from 1-5). Thus, this was a weighted graph. I chose to make my ADT a directed graph, since the user could easily make the directed graph undirected by adding twice, or keep it as directed.

This data structure worked very well for the assigned application. Since the application required complex vertices with connections to and from many other vertices, a graph data structure is the only data structure that makes sense. There are a few ways to implement a graph, but I would not switch away from an adjacency list. This is mainly because I know less about other implementations, but I do believe that the adjacency list is better than an adjacency matrix (2D array) or an edge list. An edge list would make traversal (like an organized display all) very difficult, and an adjacency matrix would have slightly less efficient traversal, since the program would have to search through every index, instead of just the next item in the list.

The most efficient aspect about my design and data structure is probably the display all function. As explained above, other graph implementations would not be as efficient for a display all function or really any traversal. The edge list is just very unorganized and would display the edges in the order they were added, not by vertex. The adjacency matrix would be very similar to the adjacency list used, but it will have to traverse through the unused indices of the arrays, which can become suboptimal with much larger data sets.

The biggest thing that would have not been efficient about my program would be sorting the data. A depth-first search could help slightly with sorting the vertices. However, unless the graph has all the proper connections, the data will not be absolutely sorted, the algorithm will only find the lexicographical path through all nodes. Luckily, there were no sorting algorithms in this project, so the least efficient was just adding vertices and edges. The only inefficiency in adding vertices was searching for the first open index, but that could be fixed by storing an int that keeps track of the current index. Adding edges was a bit slower since I had to find the index of each vertex given, then loop through the edge list to make sure the edge being added does not exist already. These inefficiencies would be negated in the adjacency matrix and the edge list, but they are so small (especially for the size of this given application) that I would not choose a different graph implementation.

If I had more time to solve this problem, there are a few changes I would make. I would not switch the abstract data type, data structure, or the graph implementation, but there are small changes I would make. First, I might add a way to input from an external data file so that I could see how the program handles much larger data sets both in memory usage and speed efficiency. I could also add some more functions, like a depth-first search algorithm, a find shortest path (either through all nodes or between two nodes), a total length calculator, and much more. As always, many functions could do with better error handling, especially my insert edge, which has many different ways that the given input could result in an error (the edge already exists, one of the edges is invalid, etc.). Finally, outside of the ADT, I could always make a better UI that has stronger input checking and looks nicer.