Report for LAB 4

Leticia Dupleich Smith

Deadline: December 22, 2024 23:59

Task description

The purpose of this lab assignment was to implement Dijkstra's algorithm when navigating the Dutch train system. The task involved finding the most time efficient path between two cities (inputted by the user) accounting for any number of disruptions (also inputted by the user). The main goal of this lab was to, using a given map of the train system, develop a functioning code with Dijkstra's algorithm and efficiently find the best alternative possible route given various options. This assignment also encouraged us to explore different time efficiencies of multiple data structures and when one is more applicable than the other. In this case, heaps were very important to make this code efficient so part of the task of this project was to learn how to code Dijkstra's algorithm with the implementation of heaps.

Structures and Algorithms

Dijkstra's algorithm works through a weighted graph, so the first step was to create a graph data structure. Here, the graph was represented with an adjacency list since there are relatively few edges (sparse graph) and thus the time complexity would be better ($V^2 > V+E$). Each node contains the integers *index* and *weight* and a pointer to the next node *next*. In this way, every node of the graph is connected to all of its neighbors via a linked list. The first helper function used for graphs is *init_graph* which initializes the graph, setting each neighbour to NULL. Moreover, the neighbours (edges) have to be inserted manually and the function *addEdge* allows this to be less repetitive since the function is called every time an edge is added. On the other hand, *removeEdge* is useful when the user inputs a certain number of disruptions since it removes the corresponding edges. Finally, to avoid any memory leaks and successfully pass the valgrind test, the memory used to create the graph also has to be freed with *freeGraph*.

The code also uses a heap to implement a priority queue. Each node of the heap contains its corresponding station name and its distance to the starting node - to avoid the repetitive use of *strcmp*, each station name was mapped onto an index using the function *mapStationIndex.* The heap is created using an array of nodes, a *front* and a *size*. One of the helper functions for the heap is *removeMin* which is responsible for removing the element with the minimal distance from the heap, in this case the root. Another function *enqueue* is responsible for adding elements onto the heap. For both *removeMin* and *enqueue*, it is imminent to make sure that the heap's structural property is not changed and thus the functions *downheap* and *upheap* have to be used. Finally, *decreaseDistance* updates the distances to their minimal capacities in the same way that Dijkstra's algorithm updates pseudo distances every time a smaller one is found.

That brings us to the most important function: *Dijkstra*. This function creates a min heap and sets all of the distances to infinity initially except the root. The function has a main while loop that goes through all of the nodes in the graph, starting with the one with the smallest distance. For each node, it looks at its neighbors, calculating the potential new distances from the current node. If a shorter path to a neighbor is found, the neighbor's distance and previous node are updated, and the heap is changed to show the new shorter distance. This process continues until all nodes are visited or the heap is empty.

Evaluation of the program

To test the project, I used the *make test* and CodeGrade. Once the code passed *make test*, it was tested on CodeGrade and this is where issues began to show. The main issue was that the code was not reading the station "Den Haag" correctly since it contains a space which made the test take an extremely long time to run (~8 minutes). In order to solve this, *scanf(" %[^\n]", x)* was used rather than *scanf(" %s", x)* which would effectively take consideration of the space and pass most of the tests on CodeGrade. Another issue that was presented by CodeGrade was a memory leak. This issue was not as complicated to fix since I knew what was probably missing in the code to create this memory leak. It turns out that the code was freeing all of the data structures and memory except that used for the graph. This is when I realized that I forgot to implement the function *freeGraph*. As soon as this function was added, the test was also passed.

Aside from CodeGrade, another issue happened when adding and removing edges from the graph. Since the assignment stated that the weight from city a to b was the same as from b to a, I just assumed that adding one edge between the two was enough. I did not, however, take into consideration that there was still a reverse edge that had to be labeled, it simply had to be labeled the same as the forward edge.

Conclusion and Self-Reflection

This assignment helped me to understand more about how to implement data structures together. It also enhanced my understanding of graphs since I had to decide which implementation to use and how to keep adding and removing edges. It also showed me how small details like a space in "Den Haag" can have very important changes to the code and sometimes prevent it from functioning properly.

Implementing the heap was rather easy since a lot of the functions were recycled from the previous weekly exercise *heap*. The challenging part was using a heap in Dijkstra's algorithm. It was a bit hard to visualize exactly what role the heap would be playing in the algorithm and how it would benefit its function. To help to understand this, I once again drew it on paper and brainstormed exactly what I wanted each function to do in the algorithm. This really helped me to write the *Dijkstra* function since I knew what I wanted from it.

Like most labs, I am still curious as to what other ways there are of implementing the same algorithm and how they would affect its complexity since I did not really look at other implementations of Dijkstra's algorithm on C. Furthermore, in the future I would like to solve the bonus exercises which I was not able to do this time.