Read Ballew
Thien-An Vu
4 - 30 - 18
Professor Helsing
CSCE 2100.001

Design Document

Design Process:

Read:

The design process for Project 3 was actually really easy, coding the different searches is where most of the challenge laid for myself. We decided immediately to break the project into two main pieces, the depth first search, and the breadth first search. After doing so, we needed to create a rough idea of what the program would entail. This was done a paper and whiteboard the vast majority of the time, otherwise it was quick implementation in the code during the actual coding session. Pseudocode needed to be created initially, and we sought advice a few times during that stage.

We created our pseudocode by thinking out how we would 'attack' this project from both of our points of view. We ended up having to go to the peer mentor for advice, and she helped us develop many different ideas to try. Our design was decided together to avoid having too dissimilar code, as that would make it hard to adapt to one another's style. The path the data took was short and simple. First, it would be read in from a user defined file; then it would be sorted and output in the form of an adjacency matrix. After the adjacency matrix was created, it was time to move on, to the different searches. A menu was created to sort through the different options presented in the file, and then the user is given a selection option. Each search was implemented by careful thought and planning for the data, and included two functions, one that called a function that would call itself recursively, to complete the search properly. Once the data was returned, all of the information is sorted appropriately and printed for the user's viewing pleasure.

Thien-An:

For this project, my partner and I decided to have this project broken up to 2 major parts, DFS and BFS. The basic algorithm was written on paper and white-board. We discussed on how to specifically write each search algorithm. I drew out the graph for both examples in the clearest way possible and it turned out to be very helpful since we needed the graph to write our DFS and BFS. The first part of the project is similar to the previous one we've done, taking in the data and store them. Although we have 2 separate search algorithm but they're fairly similar, the difference is how we push in the nodes for stack and queue for the respective algorithm.

We went to the peer mentor to make sure our thought process is correct and on the right track since we had a little bit of difficulty understanding DFS and BFS. The peer mentor explained and wrote pseudo code on the white to help us with the thoughts process. We started to work on the code after that and fixed any errors as we came through them. The most challenging part was to find the correct path of the for each search method. Which is implementing both DFS and BFS correctly to display the right path. On paper, we would draw out the stack and queue for the respective search methods and see how it works first and then actually implemented it in the code.

Data Structures:

Read:

The data for this project was mostly put into and sorted into vectors. My partner and I chose vectors, due to familiarity, and ease of utility. Ease of utility, because a vector is a great stand in for both stacks and queues. The only difference between the two was actually the way you pick a node off the stack, or pick a node from the beginning of the queue. We read in the data from the file, and put the values into a vector of structures. With each structure holding both nodes that were connected, and the propagation time between the two nodes, verifying and utilizing the data was extremely easy. After the data is stored using vectors, it is simply referred to, to create other vectors, each used for a different portion of the project.

The main problem with choosing vectors, was actually the sheer amount of options to move, create, destroy, and view data in a vector. Having the vector start at 0, and the actual list of nodes start at 1 was the only real challenge in using a vector (which was solved very quickly and easily). Otherwise, using vectors for BFS for example, worked flawlessly. I simply chose nodes to view children of from the beginning of the queue, instead of the back of the vector, like you would for a stack. To justify using vectors instead of an actual stack or queue, I think it was better to work with a data structure that was so versatile, and readily understood by both members of the team.

Thien-An:

In this program, we agreed on using vectors since we're both very familiar with it and it also helps with the designing process. Vector is very flexible since we can add and remove each element depending on the current situation. Using vector as a container rather than the stack and queue themselves is a little bit different like instead of dequeuing we just removing the first element in our "queue". For our DFS, we just have a horizontally "stack" and popping off is removing the last element in the vector for our case.

We compiled our makefile with C++ since we use #include chrono to calculate the time for each search algorithm. Organizing the data for this project was important since everything node we read in from the file must be recorded and be used later in the project. After that we also have to sort the connected to nodes and push them in order. More vectors are created to store explored, unexplored temporary and pathing. Overall, organizing the data for the project needed to be done correctly otherwise we would run into things like deleting nodes that don't exist in the stack or queue.

Functionality:

Read:

Data in terms of functionality for my section of the project, runs through the program as such: programs starts -> menu ->main function -> BFS function -> BFS pathing function -> menu quit choice -> program ends. This means that the smaller functions that I designed, current node ends, and calculation of propagation time are called variably throughout the whole program, but each small section that was created by either of us, was created with the potential to be used throughout the entire program, for any use either of us might have.

So once data exited the main section, in the form of a vector of connections and integers of the permanent beginning and ending selections made by the user... DFS is called, and then BFS. BFS pathing is called recursively, this way the function does not stop until the end condition is met. Either the queue becomes completely empty, or the final node is found. It cycles through the queue repetitively, searching for the smallest child that is unvisited, unexplored, and is directly connected to the parent node we are currently searching with. Once it is found, we move on to the next node, and erase the node we just looked at from the queue, adding it to the path.

Once the data is compiled completely, we print out the completed path, and calculate the time it took for the BFS to run, if it really ran through the predetermined network. This value is printed, along with how long it took the actual breadth first search to run, as well as the actual path it took from the starting node to the finished node. If no path was found, that is printed in lieu of printing times and the path. The program finishes running its searches and returns to the main menu, where you can choose to run the search for any of the choices outlined in the file.

Thien-An:

First step of this program is to pull in all the data and stored property. Then we use the connected nodes along with their weights to create the adjacency matrix. For DFS we first find the connected nodes of the beginning node, store and sort them in order from the smallest to the largest in our unexplored vector and since we use vector as our stack, we have to push in the top of the stack which is the largest number in the vector and check for the children of that node. The beginning node will always be pushed into the stack. We delete the nodes that have no undiscovered children. We also have a function if a node is moved to explored, it will be removed from unexplored list. The function is recursively called until the stack is empty or the beginning node is deleted which result in no path found.

After finding a path or no path found, the function will print out the path that it went through to find the beginning and ending node. We also calculate the propagation time by adding the weight of each node in the path to find the total weight. Along with that, the propagation time will also be printed for each option. After that the user can select another path that they want to check for the program. We also found the time it took to find the path for each selection.

When the program is ran, it'll print out the beginning node, end node and connections of those nodes along with the weights. After that we print out the adjacency matrix and the path choice for the user to select. After selection, the program will find a path for DFS and BFS with the weight for the path with the time to find a path for that particular choice.

Organization of Responsibilities:

Organization of the different responsibilities we had was split up primarily by function. Writing the code was something we had to do in the presence of one another, otherwise the variables would be a little hard to understand from each other's perspective.

Both Members:

int main();
adjacencyMatrix();
doesConnectionExist();

Read:

BFS();
BFSpathing();
currentNodeEnds();
calcPropTime();

Thien-An:

DFS();
DFSpathing();
checkOrder();
checkExplored();