Nick Zewiski

CS 3050

**Final Project Report**

**Problem Statement:**

For this project we were tasked with designing a maze pathing algorithm. This was to test our understanding and knowledge of graph searching algorithms. The premise of the project is that two robots have been planted in a maze and need to find paths to their respective exits. The program takes a text file that acts as the floorplan of the maze with markings to indicate each robots starting position and each respective exit. The maze files must adhere to a particular format otherwise the program will report an error and terminate. Walls of the maze are represented as ‘#’ characters, the starting position for robot 1 is labeled with an ‘s’ or ‘S’, it’s exit labeled as ‘e’ or ‘E’, the starting position of robot 2 is labeled as ‘f’ or ‘F’, and it’s exit is labeled as ‘l’ or ‘L’. Spaces in the maze represent possible positions where the robots could move to as long as the robots don’t have to “teleport” to reach them. Robots can only move one space at a time, in the cardinal directions (i.e. up, down, left, or right), and they can’t move through walls. Another constraint that had to be accounted for was the size of the maze which could not be assumed. The program was to be written in C or C++ using either Netbeans IDE or a Makefile, using only standard libraries. We were not allowed to use any graph theory libraries but were given access to array, vector, stack, heap, list, and queue libraries. For a project this large, a fair amount of error checking, formatting, and documentation is required to make sense of the source code.

**Process:**

The first thing that had to be decided was which language to use. I decided on C++ because of the usefulness of objects when implementing things like linked lists and graphs. From there, I set up the code that would prompt the user for an input file name and read from the file a two dimensional Maze array character by character. The array size is set with a maximum value to make looping through it easier. Since we can’t assume the actual size of the input array, the maximum value is fairly large but could be made larger if necessary. The program only populates the elements of the array that are necessary, leaving the rest of them set to NULL. The reason I chose this approach is because the program is designed to take a maze of any shape and size. This means that the user could input a maze file where every row is a different length making looping through the array somewhat tricky. Once the Maze is stored in a local array, the file is closed. If there is any kind of error opening, closing, or reading the file, the program throws an error value and a message is displayed letting the user know what went wrong. The program also increments the vertex counter variable as it reads characters from the input file. After successfully closing the file, the program begins looping through the local maze adding new unique nodes to the node array for every start, end, and <space> character encountered. The program does not allow for more than one start or end position for either respective robot. If such a case is encountered, the program reports the error and terminates. Nodes are not created for newline characters or end of line characters since those are to be treated as walls. The node array consists only of potential robot positions. This block of the program also error checks for any rogue characters. If a character is encountered in the maze that doesn’t meet the constraints of the maze file, an error is reported and the program is terminated. Within the same block, the program checks to see if there should be any edges between the current node and the surrounding nodes and increments a counter based on the outcome of that check. The program then uses the known vertex and edge count to create a graph of edges that is necessary for the Bellman-Ford algorithm I implemented. The edges themselves require a source, destination, and weight with an optional type. Earlier in the program, during the node array population process, each node is given an ID number and those ID numbers are utilized at this point in the program to populate the source and destination fields of each edge in the graph. Since all edges connect two adjacent vertices, each edge weight is set to one. The program loops through the node array reading vertex ID’s and checking for edges, adding them to the graph when appropriate. Once all the edges have been added to the graph, it is then passed to the Bellman-Ford function I implemented. The Bellman-Ford function used isn’t a true Bellman-Ford function however. Since it is known that all the edge weights in the graph are positive, I was able to drop the last third of the algorithm that checks for negative weight loops. The algorithm used can be seen below in Figure 1.

**Figure 1 – Bellman-Ford Algorithm**

void Bell\_Ford( graph\* graph, int src, int dist[] ) {

int V = graph->V;

int E = graph->E;

int i, j, u, v, weight;

for( i = 0 ; i <= V ; ++i )

dist[i]= \_\_INT\_MAX\_\_;

dist[src] = 0;

for(i = 1 ; i <=V ; ++i){

for(j = 1 ; j <= E ; ++j){

u = graph->edgeArr[j].source;

v = graph->edgeArr[j].destination;

weight = graph->edgeArr[j].weight;

if(dist[u] != \_\_INT\_MAX\_\_ && dist[u]+weight < dist[v])

dist[v] = dist[u] + weight;

}

}

} //End Bellman-Ford

The Bellman-Ford algorithm takes an integer array and a source vertex ID as its arguments and alters the array based on the output of the algorithm. The array stores the minimum distance of the index vertex from the source vertex. For example, if you have already run the algorithm with a source vertex ID equal to 5 and you want to know the minimum distance between vertex 5 and vertex 22, you would check the integer array from the algorithm at index 22 and what would be returned would be the minimum distance. From this distance array, if a viable path exists between the source and the target, it is possible to step backwards index by index and node by node from the target to the source, recording the steps along the way. This is what the getPath function does. The getPath function takes the two dimensional node array passed to it and turns it into a one dimensional array with each node’s ID equal to its index in the array. It also takes the distance array from the Bellman-Ford algorithm and an index and corresponding distance from which to start stepping backwards from. The getPath algorithm loops through the distance array looking for a distance that is one less than the current index. If such a distance is found, it checks the corresponding node in the node array to see if the current node is adjacent to the node that has been found with the correct distance. If the current node isn’t adjacent to the found node, the loop keeps searching for another distance that is one less than the current distance. If the found node is adjacent, the found node becomes the new current node and the corresponding direction is recorded in a string. The loop then restarts, again looking for a distance one less than the current node. This process repeats until the currant distance is zero, meaning we have reached the source vertex and the full path has been recorded. It should be noted that if a distance one less than the current distance doesn’t exist, that the program will enter an infinite loop, however the main function doesn’t call the getPath function if a viable path doesn’t exist. Main is able to check this by looking at the value from the distance array at the target index. If the Bellman-Ford algorithm computes that the distance between the source and target node is greater than the number of total edges in the maze (which is the maximum distance since all edge weights are one), then we know there is no path between the source and our target. In this case, it is reported to standard out that no path exists. The only other possible cases at this point is that either a start point, an end point, or both were not defined in the input file. In the case of a missing start point, the program will catch that a start node was never created, but since the index for the start defaults to 0 the Bellman-Ford algorithm will still run successfully. However, because of how I chose to tweak the algorithm, the distance associated with index 0 will always be “infinity” relatively speaking. This is how the program knows not to report a distance to a node that doesn’t exist. The program works similarly for a missing end point with the end index defaulting to 0 for the Bellman-Ford function.