Brian Pinson
U191813366
Project 1 Group 13
10/23/2017

Project 1 was written in C++, using only the standard library, and compiled in Visual Studios

Algorithm

The algorithm designed to solve the shortest distance problem uses a priority queue where the front of the queue is the shortest total distance to the final city. The total distance is defined as the current distance traveled + the estimated distance to the end. Only one version of the algorithm was needed solve for the straight line distance and the fewest links problems. The difference between the two solutions is setting the estimated distance to either one for fewest links or taking the distance between a city and the final city for the straight line distance.

The shortest distance problem is broken down into two main algorithms. The shortest distance algorithm is responsible for moving through a priority queue and keeping track of the total distance traveled. The addNeighbors algorithm adds all new neighbors into the queue. If a neighbor is already on the queue but this neighbor's total distance traveled is less, the city on the queue is deleted and the neighbor is added, else the neighbor is skipped. The addNeighbors algorithm is also responsible for keeping the queue organized by least total distance to most total distance. Together they create the list of all paths taken. From there a print function will decide which paths were dead ends and what was the optimal final path.

```
Shortest Path Algorithm
input: starting city
output: paths: list of all paths taken

1. push start into priority queue;
2. push start into paths;
3. double traveled = 0;
4. while(queue.front() != end)
5. – push queue.front() into excluded list;
6. – traveled = queue.front()
7. – addNeighbors(queue.front(), traveled)
8. – if(queue is empty)
9. –– print error message;
10.–– return;
11.–– end;
12.– end;
13. end;

addNeighbors Algorithm
input: city
output: all neighbors of the city

1. pop priority queue;
```

```
2. if(city is neighbors with the end city)
3. – push end city to front of queue;
4. – return;
5. – end;
6. for(i = 0; i < number of neighbors; i++)
7. – if(path is not on the excluded list)
8. -- if(straight line distance)
9. --- (if city.neighbor(i) already exist in the queue but has a smaller
distance traveled)
10. --- push city.neighbor(i) into queue;
11. --- end;
12. -- if(fewest links)
13. --- push city.neighbor(i) into queue;
14. -- end;
15. – end;
16. for(i = 0; i < size of queue; i++) // organizes queue from smallest to
largest
17. – for(j = i; j < size of queue; j++)
18. ---if(i > j)
19. ----- swap(i, j);
20. ----- end;
21. --- end;
22. end;
```

Heuristics:

The heuristic for this A* algorithm is estimating the total distance a path takes from beginning to the end. This is accomplished by storing the current total distance taking from the starting city to the current city and adding in the estimated distance of the current city to the end. For fewest links the estimated distance is one step. For straight line distance the estimated distance is the length between the current point and end point. The estimate is an underestimate as it presumes the smallest distance from the end is either one step or a straight line towards the end. A priority queue always explores the point estimated to be the closest to the end.

Program Design

The class PathFinder is responsible for taking in the user's input, formulating the shortest path using either the straight line or fewest link heuristic, and printing out the output. First in the constructor is the function setup(); which is responsible for handling the user input. After asking the user to input the location and filename of the textfiles it will double check that the files exist and can be successfully opened. All the information in the textfiles are added into a vector of structs called cities. The cities struct holds the cities name, coordinates, neighbors' name, and estimated distance from the end. After setup() has taken in all of the user inputs which include: connections file, locations file, starting city, ending city, excluded cities, heuristic, and printing method the constructor begins calculating the paths taken.

The two main algorithms responsible for calculating the paths taken are ShortestPath(); and addNeighbors(); which are explained further in the Algorithm section. From the starting city all neighbors are added into a priority queue organized from least total distance to most total distance. The neighbors from the front of the queue are added into the queue, the front is popped, and the queue is reorganized. If the queue is empty the ending path is impossible to reach and a warning error is sent. Otherwise the end is eventually found.

After the end, or last possible path, is found the printing function printPath() prints the output as requested by the user. The optimal path prints only the path that leads to the end. It is calculated by taking the worst paths taken during the PathFinder algorithm. This works because PathFinder() always takes the best possible path, even though it might lead to a dead end. To avoid the dead ends during printing, the optimal path would be the longest estimated distances taken before the end city was found. If the user requests step-by-step then the printPath() will intentionally print all paths taken, including dead ends. This is accomplished by backtracking anytime a dead end is reached during printing.

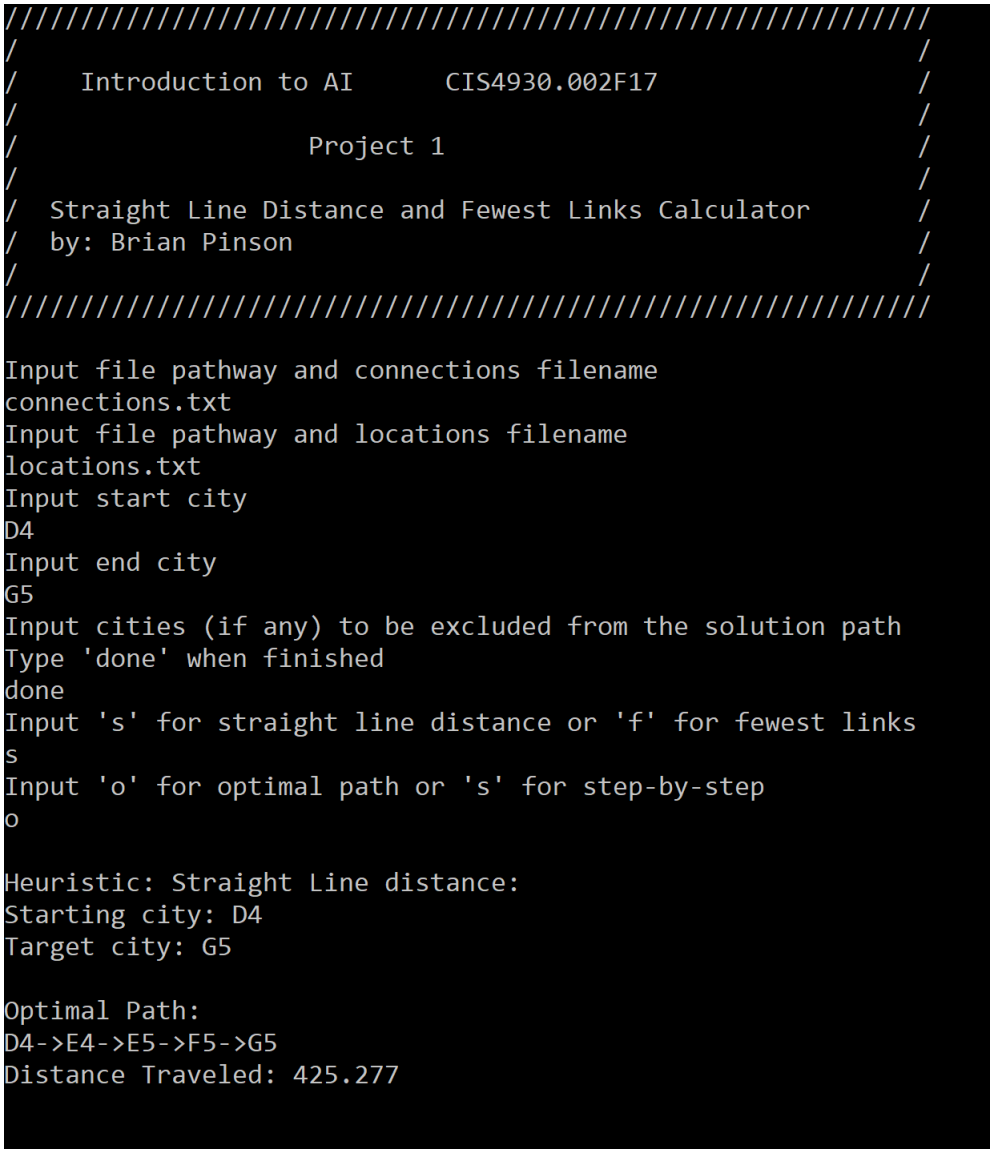
List of Functions:

```
void setup(); // responsible for getting receiving user inputs
void buildConnections(); // adds in cities and their neighbors
void buildLocations(); // adds in cities and their coordinates
void alphabetize(); // organizes neighbors alphabetically just like the sample output
void setStartCity(); // user inputs start city
void setEndCity(); // user inputs end city
void setExcluded(); // user inputs excluded cities
bool isExcluded(int); // checks if city is excluded
bool isEnding(int); // checks if city leads to end city
void setPathway(); // sets heuristic used
void setOutput(); // sets output method
void setEstDistance(); // estimates how far a city is from the end
double distance(int, int); // calculate distance of two cities
void setNeighbors(); // puts neighbors in a list
void ShortestPath(); // calculates shortest path
struct path { int loc = 0; double dist = 0; int step = 0; }; // struct holds location, distance
traveled, and steps taken in the path
void addNeighbors(path, double); // adds neighbors into the queue
bool isCloser(path); // checks if city was found at an earlier point
void printPath(); // prints path
```

Input/Output

The program uses command line for input and output (Fig. 1). The user is asked to input the connections textfile and location if the location is outside of the directory where the program is executed. Then the user inputs the locations textfile. Then the user must input a starting city, an ending city, and list any cities, if any, to be excluded from the calculations. Then the user inputs their desired heuristic, ‘s’ for straight line or ‘f’ for fewest links. Anything else sends an error message and the user must try again. Then they must input their preferred printing method: ‘o’ for optimal or ‘s’ for step-by-step.

The following is the interface created from command line



(Fig. 1)

Performance

Calculating performance has been difficult. When running the program using the sample input the program took 33% more time calculating fewest links. Results showed 0.04ms for straight line distance and 0.06ms for fewest links. While testing other longer paths results varied more by how many steps it took rather than the heuristic used. When calculating distance from A1 to G5 both heuristics needed 4 steps to reach the final city and both took 0.06ms to calculate the answer. Since both use the same algorithm to calculate the answer it is not surprising that they take roughly the same amount of time to calculate the path. The real variable that determines the speed of the calculation is not the heuristic but the number of paths taken, including dead ends, using the heuristic.

Team Work

The entire project was done by Brian Pinson.