**Project 3: Shortest Path**

***COP3530: Data Structures and Algorithms***

**Team Name:** Trifecta

**Team Members:** Benjamin Hsu, Matt Hansen, Richard Liu

**GitHub URL:** <https://github.com/matthew3hansen/DSA_Project_3/tree/master>

**Link to Video: “In Progress”**

**Problem:** This project will seek to find both the: shortest path of travel from one point in a city to another and the path that is most well-lit (has the highest percentage of intersections with lights). A point, for the purpose of this project, is defined as a street intersection in a square-grid-style street-blocks city.

**Motivation:** Urban environments can prove to be dangerous to travel, especially at night, and during times of civic turmoil, such as currently being witnessed in the United States. Even before all of the current economic turmoil, there has been a substantial increase of urban citizens getting robbed or either attacked. Finding the shortest path of travel will help reduce the time spent traveling in such potentially dangerous environments and rather traveling in safe healthy environments reducing the risk of getting caught in a hazardous situation. Alternatively, the traveler’s desired path might instead be the path that is most well-lit and spots where crime rates are low, so finding this path as an alternative gives more options to the traveler.

**Features:** When a “starting point” street intersection and an “ending point” street intersection is entered; the program will output the sequence(s) of street intersection names to travel that is/are the shortest path(s) to travel.

The program will calculate this shortest path using Dijkstra's Shortest Path First algorithm. A second path that features the most well-lit path, defined as the path with the highest percentage of intersections with lighting, will be found as well.

Using a csv file which is used as a make-shift map from the user to the destination, the program will map out the safest path to take using body relative directions (forward, backward, left, right) for the user to understand easily which direction to move in.

**Data:** Street intersections will be created for an imaginary city that has a “square grid”-style city block layout. Street names are generated using a random data set generator (https://www.mockaroo.com/). The data set will first be filtered to remove duplicate street names before use in the program.  
  
A to-be-determined amount of the intersections will be selected to have street lighting. The distribution method of this lightning is still to be determined - whether randomly distributed among the intersections, or specifically along certain roads, or a mix of these two methods.  
  
In order to fulfill the “at least 100,000 data points” requirement of the project, a minimum of 633 unique street names (e.g. 317 streets on 1 axis, and 316 streets on the other axis) are needed to populate the grid with their intersections. For the purpose of this project, a 500x500 grid will be used.  
  
A fictitious city will be randomly generated with self-made code. This random generation code will take as inputs the size of the grid to be made, and the percentage of grid that will be occupied by buildings. Each point on the generated grid will then be randomly made to be either an open road intersection, or a point blocked by a building. In order to ensure that no roads are fully enclosed by buildings, the code will then check for such fully enclosed roads and “open” them up to the rest of the map.  
  
This city will use the street naming convention in which the street name is defined by its latitude or longitude, and not by continuity of the actual street. For example, if a street is blocked by a building, the street of the same latitude (or longitude) on the other side of the building will be the same street name.

**Data Structures / Algorithms Implemented:** Dijkstra’s Algorithm and Adjacency List

**Data Structures Used: Adjacency List, 2D array, dictionary**

**Responsibilities:**

A highly tentative team distribution of responsibilities is as follows. The actual distribution of these tasks will almost certainly not be as exactly listed here, but this section is included in order to fulfill the project proposal requirements.

Generating data to build fictitious city: Benjamin Hsu

Creating the graph data structure: Richard Liu / Benjamin Hsu

Implementing Dijkstra’s algorithm: Richard Liu / Matt Hansen

Web/game visual interface: Matt Hansen / Richard Liu

Documentation: Benjamin Hsu / Matt Hansen

**Analysis(Purpose of each function):**

Any changes the group made after the proposal?

findDimensionsOfMap(): This function reads in a file, and iterates through the number of rows to count the number of rows, and finds the length of the line to find out how many columns there are.

readFile(): This function takes in the rows and columns, and then reads in the horizontal and vertical street names.

insertNodes(): This function adds all nodes in a 2D array.

createAdjacencyList() : This function takes in the number of rows, and the number of columns and then iterates through to add to the adjacency list.

findShortestPathAdjacencyList(): This function is our algorithm to navigate from the start point to the end point of the map and prints out the directions from point A to point B.

**Complexity Analysis per function:**

findDimensionsOfMap(): Time complexity O(n), Space complexity O(1). This function reads in a csv file going through each line and returning the dimensions of the map. (rows and columns)

readFile(): Time complexity is O(n+m) where n is equal to the rows and m is equal to columns. Overall space complexity is O(N+M) since we are storing N rows, and M columns in two separate list.

insertNodes(): Time complexity O(n\*c) where n is the number of rows and c is the amount of chararacters in the variable readInLine. All of the lines inside the double for loop are constant operations inserting into the 2D array. Overall space complexity O(n2).

createAdjacencyList() : Overall time complexity is O(n\*m) where n is equal to the number of rows and m is the number of columns. This function also uses the function readFile() and insertNodes() but since there is a double for loop inserting into the adjacencyList, O(n\*m) dominates. (**Work in progress**)

findShortestPathAdjacencyList(): (**Work in progress**)

**Overall Complexity Analysis:** (**Work in progress**)

**Reflection:**

**References:**

<https://www.mockaroo.com/> ( Random Set Generator)

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm> (Dijkstra’s Algorithm)