**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.

**Tools/Languages/APIs/Libraries used: Python, C++ and the Pygame Library**

**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

Any changes the group made after the proposal?

**Complexity Analysis per function in the Worst Case Scenario:**

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

*readFile():* Time complexity is O((r+c)\*(m)) where n is equal to the rows and m is equal to columns. In this function we are inserting into two separate arrays by iterating through the rows and columns and assigning a string to each of them. Since each of the for loops uses the function strip() which has a worst case complexity of O(m) where m is the length of the string being stripped.

*createArray():*

* This function uses the readFile() function, so this function will also contain its time complexity along with the rest of the function.
* Initializing a two-dimensional array using its rows and columns has a worst case time complexity of O(r\*c) where r is the rows, and c is the columns.
* Using a double for loop, this function goes through the number of rows, and uses the function strip(O(m)) and split(O(m) before going through the inner for loop. The split method has a n O(m) complexity because we are only striping a string with size 1 where the length of string is denoted as m. The inner for loop iterates through each character in the line and inside the for loop, it then assigns a string value depending on which if statement it passes through. The time complexity of this specific part is O(r\*(m3)) where r is the rows, m is length of string.

Overall runtime for *createArray()* is equivalent to O((r+c)\*m + (r\*c) + (r\*m3)). Simplifying we get O((r+c)\*m + r\*m3) worst case time complexity.

*createAdjacencyList():* Time complexity of this function is O(r\*c) where r is the rows, and c is the columns. This function uses a nested for loop to iterate through and append each value to the adjacency list.

*findShortestPathSingle():*

* This function finds the shortest and safest path to take when the user does not want to see alternate routes.
* Overal complexity for this function is O(n\*(n+m + o)))
* Starting at the first while loop, it goes through all the nodes that haven’t been looked and increments until it reaches the length of adjacency list. Time complexity is O(n) where n is the length of adjacency list.
* Inside the while loop the first for loop iterates through the length of adjacency list and then find the node with the lowest distance. Hence this section O(n) runtime. The second loop inside the while loop iterates through the length of adjacent nodes of the node with the lowest distance. We can call this length m hence this part would be O(m) runtime.
* Adding on, the next inner while loop appends each pointer to previous Node to the array pathStack. And the next inner while loop pops it off the stack after it prints the direction.

Hence this section’s complexity would be O(o+o)

*findShortestPathMultiple()*

* This function finds the multiple safe paths incase the user wants to find multiple alternate routes.
* Overall time complexity for this function O(n\*(n+m + o\*(s+t) + o))) which simplifies to

O(n\*(n+m + o\*(s+t))). Where n is the length of adjacency list, m is length of adjacent nodes, o is length of original array, and s and t are equal to length of previousPath-1 and previousPath respectively.

* Starting at the first while loop, it goes through all the nodes that haven’t been looked and increments until it reaches the length of adjacency list. Time complexity is O(n) where n is the length of adjacency list.
* Inside the while loop the first for loop iterates through the length of adjacency list and then find the node with the lowest distance. Hence this section O(n) runtime. The second loop inside the while loop iterates through the length of adjacent nodes of the node with the lowest distance. We can call this length m hence this part would be O(m) runtime.
* The next loop in the sequence is a while loop that checks if sourceReached is equal to false, which could act as an infinite loop if it is never reached. Inside this loop, we have a loop iterating through the length of the array. Inside that loop contains two nested for loops that iterates through the length of the previousPath-1 and the previousPath. Hence the time complexity of this section would be O(o\*(s+t)) where o is equal to originalArrayLength, s is equal to length of previousPath-1 and t is equal to length of previousPath.
* The last inner for loop iterates through the originalArrayLength which we can denote as o.

*shortest\_path\_visual():*

* Overall time complexity for this function is O(a + a\*(w+n)). Where a is length of aList, w equals length of weight map and n is equal to length of adjacent nodes of the min index.
* Starting with the first for loop, this iterates through the aList hence O(a) time complexity where a is length of aList.
* The next while loop compares the length of computed with length of aList. Since computed starts off at length of 0 and aList starts off with a length greater than 0, this means we are appending values until it reaches the length of aList. The first for inner loop inside this while loop iterates through the weight map and the second loop iterates through the length of adjacentNodes hence this section would have a O(a\*(w+n)) run time where w is equal to the length of the weight map and n is equal to the length of adjacent nodes of the minimum index.

**Reflection:**

**References:**

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

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

<https://www.quora.com/What-is-the-runtime-for-Python-split-built-in-method> (Split() Method time complexity)

<https://stackoverflow.com/questions/55113713/time-space-complexity-of-in-built-python-functions/55114114> (Strip() Method time complexity)