**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, physical distance-wise, from one point in a city to another and the path that is the least dangerous. 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 has the least crime rate 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 least-dangerous defined as the path with the lowest cumulative crime-rate will be found as well. The program also has the option to calculate and output multiple paths of the same minimum distance. An additional feature allows for a GUI display of the map and the calculated route.

Using a csv file which is a map of a randomly generated city, the program will find the specified shortest path (either by physical distance or by crime rate), and will output the ideal route(s) to take, either by text or by a visual display of the map and route, depending on the option chosen.

**Data:** The csv file that represents the city was generated using a self-made map generator program. The generated city is of the "square grid" format. A certain percentage of physically blocked intersections (e.g. buildings) is specified, and was 20% for this case, and the generated map is ensured to not have any "enclosed" streets. Street names are generated using a random data set generator (https://www.mockaroo.com/).

In order to fulfill the “at least 100,000 data points”, a 500x500 grid was generated, which has 250,000 elements. With a 20% case of "blocked" intersections, approximately 200,000 of these grid elements are "open" intersections that can be considered data points.

**Tools/Languages/APIs/Libraries used:** Python (main code), C++ (random map generator), Pygame Library (GUI display of the map & route)

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

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

**Responsibilities:**

* Generating data to build fictitious city: Benjamin Hsu
* Text-based Diijkstra algorithm functions & text-based menu: Benjamin Hsu
* Pygames GUI creation & implementation of graphical Diijkstra algorithm: Matt Hansen
* GitHub organization of project files: Matt Hansen + Richard Liu
* Documentation creation & management: Richard Liu

**Any changes the group made after the proposal?:**

The original proposal specified that the map of the city would be marked with intersection that have "lighting" and "non-lighting" intersections, and, along with finding the shortest physical distance path, would have the ability to find the most-lit path. However, with this plan, the graph's weights would be limited to only two values: a weight for lit intersections and a weight for unlit intersections. It was decided by the group that a graph with a wider ranger of weights would be more interesting of an application of Diijkstra's algorithm for weighted graphs, so, instead of intersections being marked as "lit" or "unlit", the map was instead generated with numerical indications of an intersections crime rate (where a value of 1 is the least crime, and a value of 9 is the highest).

**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:**

While Diijkstra's algorithm is accurate, it isn't the "best" algorithm for finding a shortest path between two nodes of a graph, as there are many optimizations that can be implemented to its base description. For example, the bulk of the run-time for our current implementation of Diijkstra's algorithm, appears to be the portion of code that searches through the list of nodes for the next minimum "dist" value to process next. This portion could be optimized by storing the nodes in a minimum heap, so that the algorithm can, instead of iterating through the large list of nodes each "cycle" of the algorithm, simply pop off the top-element, which has complexity, an improvement over the complexity of linearly iterating through the nodes. A lesson from this, is that there are many more improvements that can be made for such shortest-path-finding problems, including even using other algorithms than Diijkstra's.

[INSERT MORE REFLECTION HERE  
As a group, how was the overall experience for the project?

Did you had any challenges? If so, describe.

If you were to start once again as a group, any changes you would make to the project and/or workflow?

Comment on what each of the members learned through this process.]

**References:**

<https://www.pygame.org> (GUI creation)

<https://www.mockaroo.com/> (Random Street Name 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)