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: https://www.youtube.com/watch?v=2Z9Ud5UKPZs&feature=youtu.be

Problem: Walking on a safe path from point A to B has always been a big problem for urban cities. Due to the recent up rises due to racial inequality and the increasing number of Covid-19 cases, the world is getting even more dangerous. 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.

This program features a simple user interface that allows the user to pick two points on the map and our algorithm will map out the safest path/paths for your depending on your choice selection. This program will also allow you to choose an option between choosing a visual map or if you just want simple directions to get to wear you are going. On top of that, we also allow you to configure your street names or blocks for the intersections.

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 Dijkstra algorithm functions & text-based menu: Benjamin Hsu
- Pygame GUI implementation of graphical Dijkstra algorithm: Matt Hansen
- GitHub organization of project files: Richard Liu
- Documentation creation & project 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 range of weights would be more interesting of an application of Dijkstra’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). We also had to use a smaller subset of the large map for the visual since when the large map was used, it made the graph really small and very hard to see. To fix this we made a small subset to allow for better visualization.

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 $O(r*(m^3))$ where r is the rows, m is length of string.
- Overall runtime for *createArray()* is equivalent to $O((r+c)*m + (r*c) + (r*m^3))$. Simplifying we get $O((r+c)*m + r*m^3)$ 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 a single shortest and safest path to take.
- Overall 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 multiple safe routes of the same shortest distance.
- 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 label 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.

draw_path():

- Overall time complexity is O(e (r * c + a * l * q)), where: e is the number of edges between the source node and the destination node, r and c represents the rows and columns of the grid respectively, a is the number of nodes in the adjacentNodes list, l is the length of the intersection's name as it checks if two strings are equal, and q is the string length of the single letter N,S,E, or W as it compares the second element in the pair which is a string letter.

Reflection:

The actual run-times for our various algorithms were compared with a test of navigating from grid-position (0, 0) to (9, 9). Running on the same computer of one teammate for 5 test runs total, the text-based single path algorithm took an average of 5.4 seconds when factoring in map weights and 6.9 seconds when not factoring in map weights. The text-based multiple path algorithm took an average of 5.7 seconds when factoring in map weights and 20.5 seconds (892 paths found) when not factoring in map weights.

While Dijkstra'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 Dijkstra'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 $O(\log(n))$ complexity, an improvement over the $O(n)$ 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 Dijkstra's.

During our project, we were trying to find better ways to implement this idea of finding the safest path for the vulnerable in a real-world scenario. Such has implementing a crime map to simulate the dangers of going through neighborhoods showing how much crime is in the area the user is walking through. However, since of our time constraint, we weren't able to implement a full-scale version such as an IOS/Android app that would allow the user to use a real time map and show all the dangers of walking in each area. Although this desktop application is just a prototype, we are very proud of what we accomplished and hopefully this idea will be improved in the future.

As a group, our overall experience was great. Although we had some trouble with combining different experience levels together since we all have levels of expertise, the were able to push through and create our project. Such as one of our group members did not know any python, but he was able to learn python at a breakneck pace and achieved almost perfect mastery of the language. Most of the group did not know about the Pygame library before-hand so there was definitely a learning curve of all of us and also we were all pretty new to algorithms such as Dijkstra's shortest path algorithm hence we had to actually wait to learn it in class before we start on the project. On top of all of that, we had some trouble with coming up with project ideas and how to implement them. Most of us is fairly new to creating applications or user interfaces so we had to learn a library called Pygame to create the user interface.

What's next for Safest Path Finder?

If we were to do the project again, I think we would make a mobile app using react and Google's Map API to enhance the user's experience. Since our current application is constraint to the desktop and also the maps aren't based on actual real time maps, the user cannot really use this in a real-world scenario. We could probably make a version that captures recent data of car-accidents, burglaries, riots and the amount of people to further simulate the user experience. Also, instead of using python we could probably use a language like JavaScript for the backend part and front end we can use mostly html/CSS.

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