PREVENTION OF STREET HARASSMENT THROUGH CONSTRAINED SHORTEST PATH ALGORITHMS

Isabel Mora
Universidad Eafit
Colombia
imorar@eafit.edu.co

Andrea Serna
Universidad Eafit
Colombia
asernac1@eafit.edu.co

Mauricio Toro
Universidad Eafit
Colombia
mtorobe@eafit.edu.co

ABSTRACT

Street sexual harassment is an issue impacting the safety and well-being of citizens. Such a problem is prominent in Medellin, where women repeatedly say they feel unsafe while walking. Despite the negative side effects of street harassment, the lack of algorithms that warm citizens of areas where harassment is likely to occur perpetuates people's exposure to these situations. In this paper we present an algorithm that calculates routes based on both distance and safety in order to improve security perception, encourage walking and prevent cases of harassment. Specifically, a modified version of Dijkstra's algorithm was used to calculate a path between two given locations, with either harassment risk or distance as an additional constraint. The algorithm was implemented with a time complexity of O((V + E)logV), and a memory complexity of O(V + E), where E represents the streets and V the intersections of Medellin's map. The algorithm has an average execution time of 9.4 seconds. Overall, the results obtained revealed that there's always a compromise to be made between distance or harassment risk when choosing a particular route. Specifically, the variable that is constrained in the algorithm is the one that will be prioritized in the resulting path.

Keywords

Constrained shortest path, street sexual harassment, secure-path identification, crime prevention.

1. INTRODUCTION

Gender-based street harassment occurs when a person is subject to unwanted sexual comments, gestures and actions by a stranger in a public space. These types of actions are identified because they are directed at the victim due to their gender or sex. Actions which classify as street harassment include catcalling, sexist slurs and names, public sexual demands, public flashing, masturbation and rape [5].

Gender-based street harassment is an issue worldwide, with a disproportionately greater effect on women. Although it's a problem described as transnational, developing countries, which often have weak infrastructure, poor lighting conditions in public places and unsafe transport systems, often have worse cases of street sexual harassment. For instance, a study conducted by YouGov found that Bogota, Mexico City and Lima were among the top five worst cities in terms of verbal harassment [5].

Evidently, street harassment is an issue affecting safety perception in cities, on top of contributing to negative feelings in victims, particularly women. According to BMJ [6], street harassment makes victims feel humiliated (53.7%) and disturbed about their body (29.3%), and causes sleep and appetite problems. In Medellín specifically, 34.4% of teenage girls say they are victims of street harassment several times a day, while 60% of women feel Medellin is not a safe city due to its persistent patriarchal culture [7]. Thus, the constant threat of harassment when walking around a city, including Medellin, leads to fear, and can even stop people from walking or using the city's transportation network. Most importantly, the lack of data and algorithms that warm citizens of areas where harassment is likely to occur perpetuates people's exposure to these types of situations.

1.1. Problem

As seen by the figures presented in the introduction, street sexual harassment is an issue deeply affecting safety in cities, including Medellin. However, people do not have access to data that can help them avoid areas of the city where sexual harassment incidents are likely to occur. Even worse, citizens and tourists usually rely on navigation platforms, such as Waze or Google Maps to obtain routes for traveling around the city. These types of applications calculate routes based solely on distance, which not only fails to inform users of unsafe areas, but can also potentially lead people to take risker routes just because they are shorter. Thus, there's a great need to develop an algorithm that calculates routes within the city based on both distance and safety. In the case of sexual harassment, this means developing a constrained shortest path algorithm that provides users with routes that are short and have a low risk of sexual harassment. Through the implementation of this algorithm, security perception in the city would improve greatly, more people, especially women, would be confident about walking and taking public transportation, and cases of street harassment would decrease significantly.

1.2 Solution

In this work, we propose a modified version of Dijkstra's algorithm as a solution to the problem of street sexual harassment in Medellin. This particular algorithm was chosen because of its capacity to handle graph inputs with a large number of nodes, such as Medellin's map. Additionally, the algorithm was ideal to modify in order to fit the constraints of the problem at hand. The algorithm design is thoroughly explained in section 4.2.

Two main algorithms were designed: one that calculates the shortest path without exceeding a weighted-average risk of harassment *r*, and another that calculates the path with the lowest weighted-average risk of harassment without exceeding a distance *d*. However, in addition to these two complex algorithms, we also decided to implement two unconstrained version of the algorithm. That is, we created a version of the algorithm that outputs the path with the shortest distance, and another that outputs the path with the lowest risk. These two simple algorithms were created so that its unconstrained outputs could be compared with the results of the constrained algorithms.

1.3 Article structure

In what follows, in Section 2, we present related work to the problem. Later, in Section 3, we present the data sets and methods used in this research. In Section 4, we present the algorithm design. After, in Section 5, we present the results. Finally, in Section 6, we discuss the results and we propose some future work directions.

2. RELATED WORK

In what follows, we explain four related works to path finding to prevent street sexual harassment and crime in general.

2.1 Urban Navigation Using SafePaths

Galbrun, Pelechrinis and Terzi [1] were the first to define safe routes with the use of criminal data. Specifically, concerned with growing insecurity in cities, and aiming to take advantage of publicly available datasets, the investigators sought to improve the quality of life of those traveling around cities by developing an algorithm that uses criminal data to yield navigation options based on both distance and safety. The preliminary experiment was done using data from Chicago and Philadelphia, which they used to create a city risk model that included the probability of crime on any given road segment. Overall, their central objective was to find a short and low-risk route between two given locations. However, since these two variables cannot be computed together as a single problem, the investigators developed a solution based on a bi-objective shortest path problem that outputs a set of paths that have varying degrees of safety and distance [1]. Despite their success, this SafePaths problem is based on crime in general, and fails to address specific types of crime, such as street harassment, along a certain path.

The investigators working on the SafePaths problem used a deterministic algorithm to obtain the best route, which is done by computing all possible routes between two given locations. Overall, given a fixed pair of origin and destination locations, the algorithm computes two important paths: the safest path and the shortest path. When these two initial paths differ, as it is often the case, a recursive algorithm is then used to consider, in each iteration, an intermediate non-dominated path between the original intervals. This process is repeated until the difference between the shortest and the safest path is minimal [1].

2.2 Safe Routing with Crowdsourcing

A group of investigators in Mexico City, much like the authors of the work described in section 2.1, were concerned with security issues that arise in big cities and urban centers, especially in developing countries where crime is more widespread. Popular mobile systems compute paths based entirely on distance, and don't consider safety hazards that may arise along a recommended route. Thus, the researchers, developed an approach that integrates crowd-reported crime data with official government data in order to obtain safer routes in Mexico City for locals and tourists alike. The academic paper states that a preliminary experiment was implemented with 75% of effectiveness [2].

The approach proposed by the researchers consists on collecting Tweets that are related to crime and integrating them with crime data from official governmental institutions through an automatic system that considers descriptions and attributes in the data. After this, the Bayes algorithm is used to classify data that couldn't be integrated automatically. Most importantly, this algorithm is used to assign probabilistic crime rates to specific parts of the city. After collecting, sorting and classifying the data, a mobile application was developed to display the safest route between two given locations. The safe route is obtained though Dijkstra's algorithm, in which the weight of the nodes is derived from the average number of crimes in the given location. Thus, the algorithm outputs a route that avoids places with higher crime rates [2].

2.3 Personalized Safest Route

Analysis of criminal activity reveals that a significant number of offences towards civilians occur when people are in transit throughout a city. This, coupled with increasing rates of criminal reports throughout the word, motivated researchers Tarlekar, Bhat, Pandhe, and Halarnkar [3] to determine the safest route between two given locations in order to improve people's safety while traveling. The researchers developed a mobile application that outputs a personalized safest route based on the user's gender and age. To develop their application and algorithm the researchers used official data based on geographical locations that includes reports of criminal incidents that have occurred over the past 12 years in San Francisco.

The algorithm used by the investigators to obtain their desired result was the Iterative Dichotomiser 3 or ID3 algorithm. Initially, the application receives an input with a starting location and a destination. With these two locations a route is calculated, and all the possible streets along that route are analyzed based on the dataset relevant to the specific user. In other words, risk is calculated based on data that matches the user's gender and age. The safety route between the two locations can be calculated by the safety of

the streets that make up the route. Thus, the ID3 classifier, which generates a decision tree, checks the safety of all the streets and outputs a value of "Yes" or "No". The streets marked with a "No" are analyzed further and are the ones that determine the overall risk of the route. At the end of the process, the route with minimum risk value is selected as the safest and is returned to the user. The implementation of the algorithm detailed in the academic article was successful, with a reported accuracy of 86% [3].

2.4 Preventing Harassment with Anonymous Reports

As the final related work, we present the SafeStreet application created by researchers Ali, Rishta, Ansari, Hashem and Khan [4]. Troubled by the lack of attention given to widespread forms of public harassment, such as catcalling and groping, along with the negative psychological effects of these experiences on women, the researches aimed to create an application to help victims avoid public harassment incidents. Specifically, through the application, victims can anonymously self-report occurrences of street harassment with the ultimate goal of creating a large database of assault-related statistics. With this large and dynamic data base created through crowdsourcing, other application users can get live data of areas of the city prone to harassment and, subsequently, opt to take routes that avoid these areas [4]. Compared to the related works presented in sections 2.1, 2.2, and 2.3, the SafeStreet application is the most specific, as it was designed distinctively to reduce sexual harassment, instead of crime in general.

The application's main feature is a map where users can see combined records of harassment which were previously reported in the servers. The harassment statistics are shown through hierarchical clustering, which allows reports to be shown for both large areas, like counties and cities, and small regions, such as distinct roads. If a user wants to travel to a certain location, SafeStreet yields both a safe path and an unsafe path. In addition, the application can provide users with the safest time of the day to travel to the desired destination [4]. The exact algorithm used by the application to output the safest path is not detailed in the article, but it's clear that SafeStreet work only with the harassment statistics and does not consider distance, which is a great deficiency.

3. MATERIALS AND METHODS

In this section, we explain how data was collected and processed and, after, different constrained shortest-path algorithm alternatives to tackle street sexual-harassment.

3.1 Data Collection and Processing

The map of Medellín was obtained from Open Street Maps (OSM)[1] and downloaded using Python OSMnx API[2]. The (i) length of each segment, in meters; (2) indication whether the segment is one way or not, and (3) well-known binary representation of geometries were obtained from metadata provided by OSM.

For this project, we calculated the linear combination that captures the maximum variance between (i) the fraction of households that feel insecure and (ii) the fraction of households with income below one minimum wage. This data was obtained from the quality of life survey, Medellín, 2017. The linear combination was normalized, using the maximum and minimum, to obtain values between 0 to 1. The linear combination was obtained using principal components analysis. The risk of harassment is defined as one minus the normalized linear combination. Figure 1 presents the risk of harassment calculated. Map is available on Github[3].

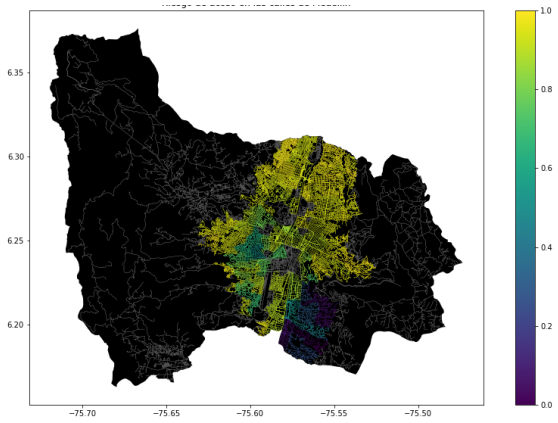


Figure 1. Risk of sexual harassment calculated as a linear combination of the fraction of households that feel insecure and the fraction of households with income below one minimum wage, obtained from Medellin's 2017 Life Quality Survey.

3.2 Constrained Shortest-Path Alternatives

In what follows, we present different algorithms used for constrained shortest path.

3.2.1 A* Algorithm

A* is an algorithm based on graph theory that is widely used to find the shortest path between two given locations. The algorithm is particularly useful for finding paths in graphs that include obstacles or unwalkable segments. This is ideal for constrained shortest path problems, as areas where a certain condition is exceeded can be taken as obstacles through which the path cannot pass. Overall, A*

[1] https://www.openstreetmap.org/

[2] https://osmnx.readthedocs.io/

[3] https://github.com/mauriciotoro/ST0245Eafit/tree/master/proyecto/Datasets/

works by maintaining a tree of paths originating at the starting node and extending those paths one edge a time until the desired condition is obtained. Most importantly, the A* algorithm is based on heuristics, as it uses a heuristic function to determine a (probabilistic) cost of reaching the destination from each node [10].

A* starts its execution by calculation the value of the function f(n) = g(n) + h(n) for each of the neighboring nodes of the starting node. In this function, g(n) refers to the cost of the path from the origin, and h(n) refers to the cost from the destination. The heuristic part of the function is h(n), as this value cannot always be calculated deterministically due to obstacles [10]. In the traditional version of A*, where each node is a square on a graph, neighboring nodes vertically or horizontally have a cost of 10, and diagonal nodes have a cost of 14. Nodes that are part of obstacles are not assigned a value. After calculating f(n) for all the neighboring nodes, the algorithm decides to extend the path of the neighboring node with the lowest f(n) value. F(n) values are calculated for the neighboring nodes of the new node, and this process is repeated. The algorithm also keeps track of the node where a given f(n) value came from, as f(n) values for certain nodes are updated repeatedly when a shortest path is found. The algorithm finishes its execution when the path it chooses to extend is a path from start to destination or if there are no more paths to be extended [11].

The complexity of A* depends on the heuristic function that calculates the probabilistic weight to the destination. In the worst case, the algorithm can actually end up exploring all the connections in the graph, so the complexity turns out to be O(E), where E represents the edges of the graph. This occurs when the heuristic function does not output an accurate estimate for the distance to the destination. A good heuristic function decreases the branching factor of the algorithm, which reduces the nodes that have to be explored and therefore lowers the complexity [10]. In terms of memory, complexity is exponential to the number of nodes, since in the worst case it has to keep track of all the neighbors of the visited nodes, which are often repeated [19].

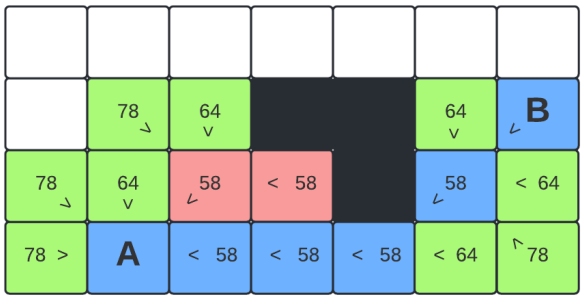


Figure 2. A* Algorithm example

3.2.2 Dijkstra's Algorithm

Dijkstra's algorithm, based on graph theory, is used to find the shortest path between nodes in a graph. The graph used in the algorithm is weighted, since each edge connecting nodes has certain value, to indicate how "costly" it is to travel through the edge. The most common variant of the algorithm sets a single node as the source and finds the shortest path from the source to every other node within the graph [8].

Dijkstra's algorithm starts by "labeling" each node with the known distance from the source. Before any calculations are made, the source node is assigned a value of 0, and all the other nodes are assigned a value of ∞. The values of the nodes distinct from the source are updated as the algorithm is executed, and the value will always represent the shortest distance found until that point. As the first step, the algorithm "travels" through all the edges connected to the source node, and updates the value of the distance taken to get to each of the "secondary" nodes. After this, the algorithm chooses the secondary node with the shortest distance, and calculates the new distances through the edges that depart from the choose secondary node. If a new distance is shorter than one found in the previous step, the distance assigned to that node is updated. In addition, the algorithm keeps track of the path followed to obtain the assigned shortest distance. Overall, Dijkstra's algorithm follows 3 main steps: updates node distances, keeps track of paths followed, and chooses the next vertex. Once all vertices on the graph have been chosen or "visited", the value assigned to each node, with its corresponding route, is the shortest distance and path from the source node [9].

If Dijkstra's algorithm is implemented with an adjacency matrix its time complexity is $O(V^2)$, where V represents the number of nodes in the graph. In an adjacency matrix, in the worst case, it takes O(V) to find the unvisited node with the shortest path, and also O(V) to update all the neighbors of the current node, since in the worst case a node can be connected to all the other nodes in the graph. Thus, time complexity comes out to be $O(V^2)$. Memory complexity for Dijkstra's with an adjacency matrix is also $O(V^2)$, because a V*V matrix is required to store all connections between nodes. If the algorithm is implemented with an adjacency list representation and a priority queue is included, the time complexity can be reduced to O((V+E) logV), where E represents the number of connections in the graph. With this implementation, the time to visit each vertex becomes O(V+E) and the time required to process all the neighbors of a vertex becomes O(logV), due to the optimization generated by the priority queue. With an adjacency list, memory complexity also drops to O(V+E), since only the neighbors of each node are stored [17].

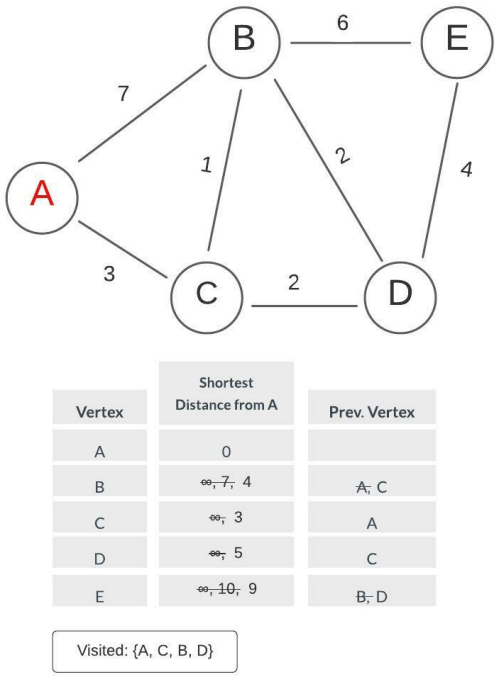


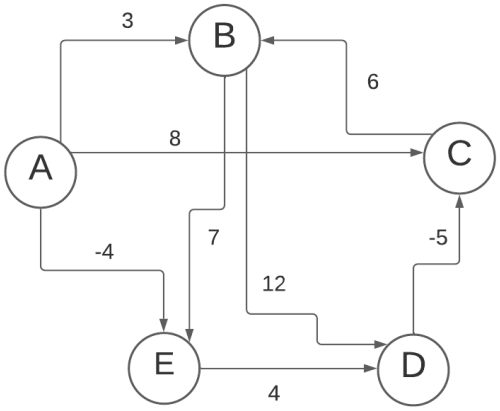
Figure 3. Dijkstra's algorithm diagram

3.2.3 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is commonly used for finding the shortest path between each pair of nodes in a graph. The graphs used with Floyd-Warshall are weighed and directed, meaning that each edge between vertices is assigned a value and a direction [12]. This algorithm is particularly useful for graphs that have edges with negative values, as these don't push the algorithm into infinite searches, as it is often the case with other algorithms like Dijkstra's.

Primarily, the algorithm is executed with the use of a matrix, which is constructed from the given graph. Each entry [i][j] of the matrix represents the value of the edge between nodes i and j. If there's no edge connecting i and j, the matrix entry is given a value of ∞. Furthermore, the distance of a node to itself is 0, so the matrix's diagonal will always be equal to 0. After the matrix is constructed from the given graph, the algorithm calculates the distance between two nodes. In this process, it checks if there's any intermediate nodes between the two chosen originally. If this is the case, and if the distance when passing through the intermediate node is less that the direct distance between the two original nodes, the shortest distance value is updated in the matrix. This process is repeated for all the intermediate vertices, and then for every other pair of nodes in the graph. At the end of its execution, the algorithm outputs an updated matrix with the shortest paths between each pair of nodes in the graph. A modified version of the Floyd-Warshall algorithm includes an extra step that calculates the shortest-path tree for every node in the graph, which allows for the reconstruction of the path between two nodes [12].

The implementation of Floyd-Warshall requires 3 nested loops, one for the rows of the matrix, one for the columns of the matrix, and one for the intermediate nodes. That is, for each row [i] it checks the direct distance to column [j] by checking the entry [i][j]. Then it proceeds to compare this direct distance to the distance through all intermediate nodes. Thus, the time complexity of the algorithm comes out to be $O(V^3)$, where V represents the numbers of nodes in the graph. The memory complexity is $O(V^2)$, because a V*V matrix is required to store all connections between nodes [18].



	A	B	C	D	E
A	0	3	8	∞	-4
B	∞	0	∞	12	7
C	∞	6	0	∞	∞
D	∞	∞	-5	0	∞
E	∞	∞	∞	4	0

Figure 4. Floyd-Warshall algorithm input example

3.2.4 Breadth-First Search Algorithm

The breadth-first search algorithm is used for searching tree data structures until a node with a certain condition is found. The algorithm starts at the root node and explores each level of the tree completely before moving on to the next level. This differs from the depth-first search algorithm, which explores a tree branch until its last level before backtracking and exploring other branches. Among many other things, breadth-first search is used for finding the shortest path from a source vertex to all the other nodes on the graph. Unlike other algorithms, like Dijkstra's and Floyd-Warshall, breadth-first search is used on graphs that are unweighted. This means that edges between nodes are not assigned a specific length. Instead, the distance from a

certain node to the source node is determined by how many levels beneath the source node it is found, with each level being assigned a value of one [13].

BFS works by assigning two values to each node in the tree: a distance referring to the minimum number of edges needed to get to the source, and the name of the previous vertex in the shortest path. Before the algorithm is executed, the two values of every vertex in the graph are set to null and the source vertex is assigned a distance value of 0. Then, all the neighbors of the source node are visited first and assigned a distance value of 1, since they are one level below the source. Afterwards, the algorithm explores all the neighbors of the nodes whose distance is 1 and assigns them a distance of 2. This continues through all levels of the tree until all nodes reachable from the source are visited. As the algorithm get deeper into the tree, it must be careful not to visit nodes that have already been visited, so it can only explore nodes whose distance value is still null. Furthermore, to keep track of which neighbor node to explore from next, the algorithm uses a queue. Once a node is visited it is added to the queue, and the node that has been in the queue the longest is the one chosen to explore from next [14].

Most breadth-first search algorithms are implemented with an adjacency list representation of the graph. In this case, time complexity is O(V+E), where V is the number of nodes in the graph, and E in the number of edges. In the worst case, it takes O(V) time to access an unvisited vertex, and O(E) to visit all the edges of the current node. Thus, complexity comes out to be O(V+E). However, this really means that complexity if O(max(V + E)), due to the asymptotic nature of the O notation. Generally, BFS is implemented for graphs that have more edges than nodes, so complexity turns out to be O(E). Memory complexity is also O(V+E), since only the neighbors of each node are stored [14]. If BFS is implemented with an adjacency matrix representation, both time and memory complexity increase to $O(V^2)$ [13].

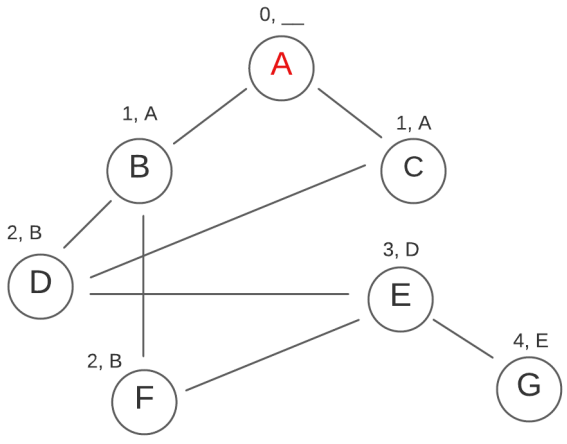


Figure 5. Breadth-first search

4. ALGORITHM DESIGN AND IMPLEMENTATION

In what follows, we explain the data structures and the algorithms used in this work. The implementations of the data structures and algorithms are available on Github[4].

4.1 Data Structures

In order to accurately implement the constrained shortest path algorithms, Medellin's map was represented with an adjacency list through the use of Python dictionaries. Such data structure serves to organize the connections between the different streets and is ideal as an input for the algorithm chosen to solve the problem at hand. The keys in the adjacency list correspond to all of the unique values of the "origin" column of the CSV file. The value for each origin is another dictionary, which has the destinations as the keys, and each value is a tuple with the corresponding distance and risk along that particular edge. Essentially, the structure of the adjacency list is: {origin: {destination1: (distance, risk), destination2: (distance, risk)}, origin2: {…}}. A visual representation is shown in figure 6. It must be noted that, for clarity, the nodes in figure 6 are labeled with letters. However, in the actual adjacency list nodes are labeled with their given latitude and longitude coordinates.

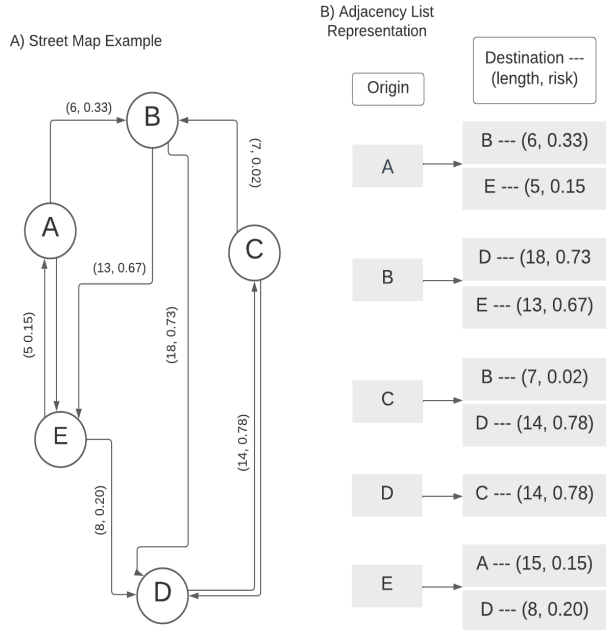


Figure 6. Street map example with its corresponding adjacency list representation.

[4] https://github.com/isabelmorar/ST0245-001

4.2 Algorithms

In this work, we propose algorithms for the constrained shortest-path problem. The first algorithm calculates the shortest path without exceeding a weighted-average risk of harassment *r*. The second algorithm calculates the path with the lowest weighted-average risk of harassment without exceeding a distance *d*

4.2.1 First Algorithm: Modified Dijkstra's

A solution to the first problem was achieved through the implementation of a modified version of Dijkstra's algorithm. The algorithm designed differs from Dijkstra's traditional algorithm in three main aspects. First, this algorithm computes the shortest path between a source node and a destination node, while Dijkstra's traditional implementation is used to compute the shortest path from the source node to all the other nodes in the graph. Since we were only concerned with the shortest path without exceeding a risk of harassment between two given locations, it was a waste of time and memory to continue computing paths that did not concern the given destination, particularly considering the great number of nodes in Medellin's map. Similarly, the implemented algorithm, through an additional dictionary, keeps track of the current path with the shortest distance between the source and the destination, while the traditional algorithm simply outputs the distance without the corresponding path. This modification was essentially because the path between the two location is the most important part for the user. Finally, and most importantly, Dijkstra's algorithm was modified to include an additional condition. That is, a path between two nodes was only updated if the average risk through that path did not exceed the permitted maximum risk.

Essentially, the algorithm designed works through dictionaries and a central priority queue. Before any calculations are made, three dictionaries and initialized: one for distances, one for previous nodes, and one for average risks. The distance values for all nodes in the dictionary are assigned an initial value of "infinity" so they can be compared once the algorithm starts its execution. The other two dictionaries are initialized with "None", because they will be updated throughout the execution. The source node is assigned a distance value of 0, a risk value of 0 and a previous node value of -1. Additionally, the source node, along with its corresponding values, is added as the first item of the priority queue.

In the main loop of the function, the algorithm pops the first item in the queue and iterates over its neighbors, which are accessed through the main graph (adjacency list) passed into the function. For each neighbor, the algorithm checks if the current distance is shorter than the actual distance kept in the distance dictionary at that point [16]. If it is, the algorithm proceeds to compute the average risk along the path with the new shorter distance through the weighted arithmetic mean formula. If the calculated average risk is less than or equal to the permitted risk *r*, all of the values for the current neighbor are updated in the dictionaries and the neighbor is pushed into the priority queue. If the conditions are not met (distance is greater, or risk is above maximum), the algorithm simple continues and no values are updated. Execution stops once the item popped from the priority queue corresponds with the destination node. The function outputs the calculated shortest distance and the average weighted risk from the source to the destination, along with the dictionary of previous nodes so the path can be reconstructed with an additional recursive function. A graphic representation of the algorithm can be found in figure 7.

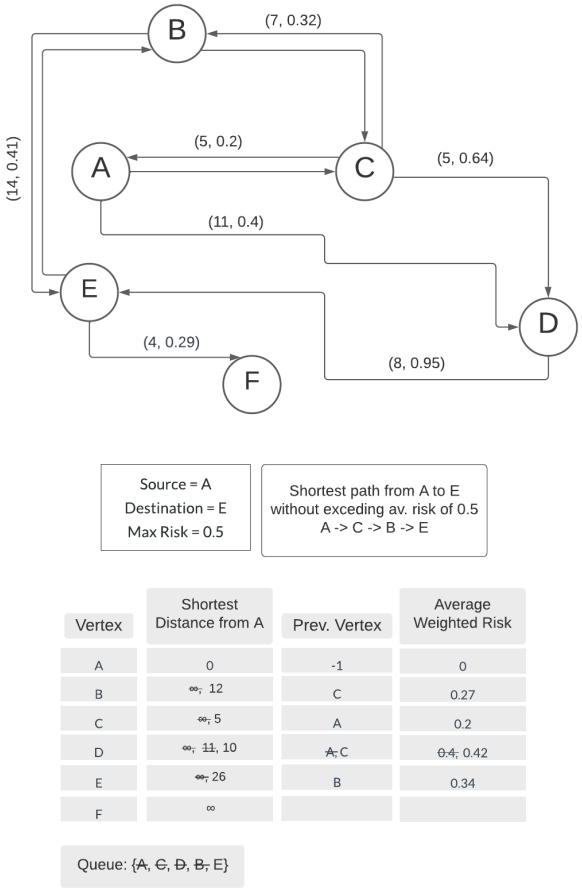


Figure 7: Example execution of the implemented algorithm (modified Dijkstra's) to solve the first constrained shortest path problem.

4.2.2 Second Algorithm: Modified Dijkstra's

The same modified version of Dijkstra's algorithm described in section 4.2.1 was used to obtain a solution to the second problem. In this case, we were concerned with the path with lowest risk without exceeding a maximum distance between two given locations. Thus, the additional constraint added to the algorithm for the second problem

was distance, while the central calculations were made with the average risk values. That is, a path between two nodes was only updated if the distance through that path did not exceed the permitted maximum distance.

The algorithm designed for the second problem works through dictionaries, a central priority queue, and an additional queue for visited nodes. For this algorithm it was necessary to add such additional queue, which was not needed for the first problem, because of the minimal differences that can arise when comparing average risks, since they are all values between 0 and 1. Without the queue for visited nodes, the algorithm constantly revisited nodes because of a slight difference between risk values. This, in turn, pushed the algorithm into infinite loops, since it often got stuck between two nodes. Such a problem was solved through the inclusion of the additional queue, which also greatly improved the algorithm's time efficiency.

Just like in the first algorithm, before any calculations are made, three dictionaries and initialized: one for distances, one for previous nodes, and one for average risks. For this algorithm, all the nodes are assigned a risk value of 1 at the start, so they can be compared once the algorithm starts its execution. The other two dictionaries are initialized with "None", because they will be updated throughout the execution. The source node is assigned a risk value of 0, a distance value of 0, and a previous node value of -1. Additionally, the queue for visited nodes and the priority queue are initialized with the source node.

In the main loop of the function, the algorithm pops the first item in the queue and iterates over its neighbors, which are accessed through the main graph (adjacency list) passed into the function. For each neighbor, if it's not in the queue for visited nodes, the algorithm computes the average risk along the path with the arithmetic mean formula. If this calculated average risk is lower than the current risk kept in the risks dictionary at that point, the algorithm checks if the distance along the new path is less than or equal to the maximum permitted distance *d*. If it is, all of the values for the current neighbor are updated in the dictionaries, the neighbor is pushed into the priority queue, and the neighbor's previous node is added to the queue for visited nodes. If the conditions are not met (average risk is greater, or distance is above maximum), the algorithm simple continues and no values are updated.

Execution stops once the item popped from the priority queue corresponds with the destination node. The function outputs the calculated shortest distance and the average weighted risk from the source to the destination, along with the dictionary of previous nodes so the path can be reconstructed with an additional recursive function. A graphic representation of the algorithm can be found in figure 8.

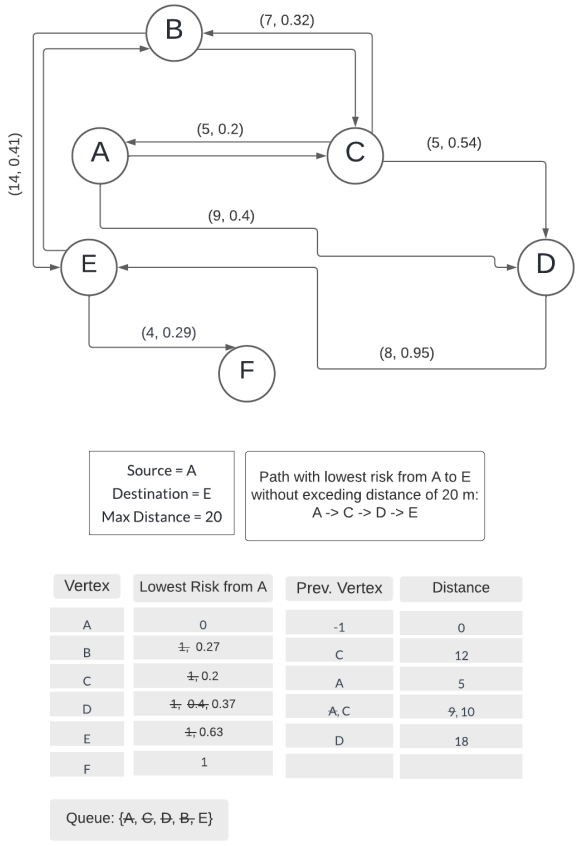


Figure 8: Example execution of the implemented algorithm (modified Dijkstra's) to solve the second constrained shortest path problem.

4.4 Complexity analysis of the algorithms

The algorithm implemented is a modified version of Dijkstra's. Since it was implemented with an adjacency list representation of the graph, and included a priority queue, it's time complexity is O((V + E)logV), where V represents the number of nodes in the graph and E represent the number of edges. Specifically, V represents the intersections and E represents the streets in Medellin's map. Essentially, the outer loop of the function is executed once for every time a node is added to the priority queue, which in the worst case turns out to be once for all the nodes in the graph. Inside the while loop, all of the neighbors of the current node are explored. Thus, if every node is added to the priority queue, the algorithm has to explore all the neighbors of all the nodes, so the complexity of this segment is actually O(V + E). Finally, each priority queue operation (adding or removing an entry) is O(logV). Hence, if all the nodes have to be added to the priority queue, the overall time complexity of the algorithm in the worst case is O((V + E)logV).

In addition to the functions with the constrained shortest path algorithms, a recursive function was implemented so that the calculated path could be reconstructed after the

execution of the algorithms. Such recursive function has complexity O(V), since only one recursive call is made, and, in the worst case, a computed path is made up of all the nodes in the graph.

In terms of memory complexity, an adjacency list was used for the representation of Medellin's map as a graph. Such data structure has memory complexity of O(V + E), since one stores all the nodes and the neighbors of each node. However, this means that complexity if O(max(V + E)), due to the asymptotic nature of the O notation. In Medellin's map, complexity turns out to be O(E), since there are more edges than nodes. Particularly, it's known that E = V*5 because a particular intersection has a maximum of 5 edges. Thus, this means that memory complexity is O(V*5) = O(V) because constants are ignored in O notation. In the worst case, if every node were connected to all the nodes, the memory complexity of an adjacency list is actually O(V^2). However, this scenario is unrealistic for this project, since it's impossible for every street to be connected to all the other streets on the map.

Algorithm	Time Complexity
Modified Dijkstra's	O((V + E)logV)

Table 1: Time complexity of the modified version of Dijkstra's algorithm, where V is the number of nodes and E is the number of edges in the graph. Specifically, V represents the intersections and E represents the streets in Medellin's map.

Data Structure	Memory Complexity
Adjacency List	O(V + E) = O(V*5) O(V)

Table 2: Memory complexity of the adjacency list where V is the number of nodes and E is the number of edges in the graph. Specifically, V represents the intersections and E represents the streets in Medellin's map.

4.5 Design criteria of the algorithm

Primarily, in order to make the constrained-shortest path algorithms more efficient, an adjacency list was chosen over an adjacency matrix for the representation of Medellin's map as a graph. The map has a great number of nodes (unique origin values), but each node is only connected to a few destinations. Thus, if this were represented in an adjacency matrix a great amount of memory (V*V) would be wasted, as the matrix would be mostly empty [15]. This is not the case with an adjacency list, where one only stores the neighbors of each vertex. Additionally, the algorithm implemented requires constant access to a node's neighbors, which can be done in O(1) in the adjacency list, since it was implemented through the use of Python dictionaries.

In regards to the algorithm itself, the main feature that was added to improve efficiency was a priority queue. Such data structure ensures that the next node to be explored is always the one with the lowest value, which guarantees that no unnecessary nodes are explored. In addition, the algorithm was modified to stop once the destination node was pushed from the priority queue.

In addition to the functions with the constrained shortest path algorithms, a recursive function was implemented so that the calculated path could be reconstructed after the execution of the algorithm. This function outputs a deque with all the street segments of the calculated path. A deque was chosen instead of a regular list because elements can be inserted and deleted in O(1). Throughout the program, recursion was only used in the function that reconstructs the paths. We decided not to use recursion in the actual algorithms because this could easily lead to exponential complexity if more than one recursive call was made. Such complexity would have made the algorithm impossible to work with due to the great number of nodes and connections in Medellin's map.

Finally, for the actual display of the calculated path, an additional program was created that outputs the path graphically on Medellin's map through the use of GeoPandas. Although the maps are not the prettiest, we believe that outputting the paths graphically is essential for comparing results and for users to get a better understanding of the algorithms' results.

5. RESULTS

In this section, we present some quantitative results on the shortest path and the path with lowest risk.

5.1.1 Shortest-Path Results

In what follows, we present the results obtained for the shortest path without exceeding a weighted-average risk of harassment *r*.

Origin	Destination	Path	Maximum *r*
Universidad EAFIT	Universidad de Medellín	d = 6142.569 r = 0.758	0.77
Universidad de Antioquia	Universidad Nacional	d = 860.19 r = 0.845	0.85
Universidad Nacional	Universidad Luis Amigó	d = 1910.13 r = 0.842	0.845

Table 3. Shortest distances without exceeding a weighted-average risk of harassment *r*.

```
** Origin shown in red and destination shown in yellow **

Shortest path (red) -
Total Distance: 7323.31 meters
Average Risk: 0.741

Shortest path without exceeding risk of 0.65 (white) -
Total Distance: 7913.826 meters
Average Risk: 0.6
```

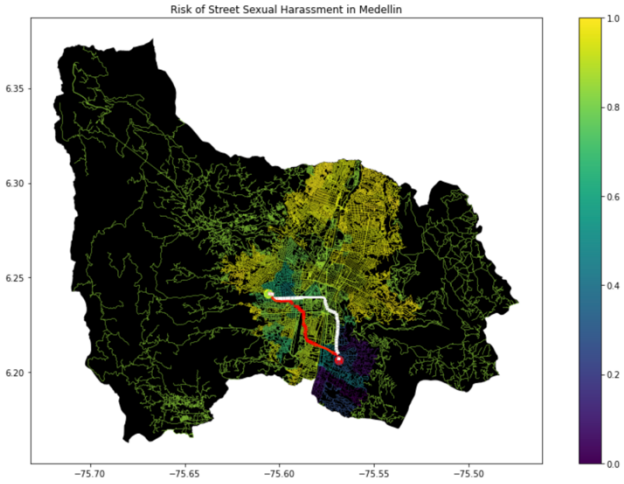


Figure 9. Example graphical output from Carrera 39 (red) to Carrera 83 (yellow). For comparison, the shortest path is shown in red, and the shortest path without exceeding a risk of 0.65 is shown in white.

5.1.2 Lowest Harassment-Risk Results

In what follows, we present the results obtained for the path with lowest weighted-average harassment risk without exceeding a distance *d*.

Origin	Destination	Path	Maximum *d*
Universidad EAFIT	Universidad de Medellín	r = 0.719 d = 6183.71	7000
Universidad de Antioquia	Universidad Nacional	r = 0.865 d = 815.44	820
Universidad Nacional	Universidad Luis Amigó	r = 0.849 d = 1472.52	1500

Table 3. Lowest weighted-average harassment risk without exceeding a distance *d* (in meters).

```
** Origin shown in red and destination shown in yellow **

Path with lowest risk (yellow) -
Total Distance: 4933.763 meters
Average Risk: 0.814

Path with lowest risk without excedding distance of 4700 meters (white) -
Total Distance: 4631.827 meters
Average Risk: 0.855
```

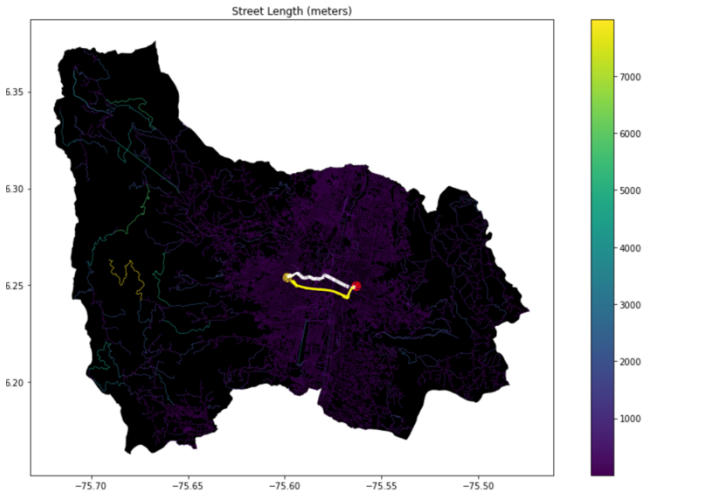


Figure 10. Example graphical output from Carrera 53 (red) to Avenida 80 (yellow). For comparison, the path with lowest risk is shown in yellow, and the path with lowest risk without exceeding a distance of 4,700 meters is shown in white.

5.2 Algorithm Execution-Time

In Table 4, we explain the relation of the average execution times for the queries presented in Table 2 and Table 3.

Calculation	Average execution times (s)
Universidad EAFIT to Universidad de Medellín	11.787 s
Universidad de Antioquia to Universidad Nacional	8.095 s
Universidad Nacional to Universidad Luis Amigó	8.292 s

Table 4: Execution times of the modified version of Dijkstra's algorithm for the queries presented in Table 2 and Table 3.

6. CONCLUSIONS

After testing both algorithms multiple times, with origins and destinations in all areas of the city, it can easily be concluded that there's always a compromise to be made

between distance or harassment risk when choosing a particular route. That is, the path with the lowest risk was almost never equal to the path with the shortest distance. In regards to this, the additional constrains added to the algorithms were essential for allowing a reasonable compromise between distance and risk depending on the needs and preferences of the user. For example, in the calculation of paths between Universidad de Antioquia and Universidad Nacional, we can see that the first algorithm outputted a path with a greater distance and a lower risk, due to the constraint of a maximum risk, while the second algorithm outputted a path with a shorter distance and a greater risk. This pattern was found in almost all of the test cases performed, leading to the conclusion that the variable that is constrained in the algorithm is the one that will be prioritized in the resulting path. This is essential for a future implementation in the city, as the two algorithms created can lead to very different results depending on what variable wants to be prioritized.

On another note, the multiple tests of the algorithms revealed that the difference between the path with the lowest risk and the one with shortest distance depends greatly on the area of the city that the user will travel through. For instance, if the origin and destination are close together, and in an area where the risk of harassment is homogenous, the path with the lowest risk will be similar (or identical) to the path the with shortest distance. If, on the other hand, one has to transverse the city to get from the origin to the destination, the difference between the two paths can be abysmal. Similarly, the area of the city also affects the constraints that can be imposed on the risk along a path. In areas like the north of the city, it's impossible, for instance, to find a path with a risk lower than 0.8.

In terms of the performance of the algorithms themselves, although the execution time is relatively low, it can definitely be improved so users can get multiple possible paths in real time. Overall, when testing the execution of the algorithms it was found that the second algorithm, which took the risk as the weight of the edges, ran much slower than the first algorithm. This is probably due to the very small differences that can arise when comparing the average risk along two routes, considering that it's always a value between 0 and 1.

Overall, the algorithms were implemented successfully and the results obtained reveal the great problematics that can arise when the calculation of routes is based solemnly on distance, as the shortest path in certain areas of the city almost always resulted in a very high risk of sexual harassment. For a city as unsafe and patriarchal as Medellin the results of the project are of great use, since they reveal that sexual assault, security perception and the well-being of citizens, especially women, can be greatly improved by simply providing people with path-finding algorithms and applications that take into consideration their safety. In addition, the results also serve as an incentive to collect more accurate data on the risk of sexual harassment throughout the city, which is currently scarce, as it was shown that this information is crucial for the development of accurate and useful algorithms.

6.1 Future Work

In the future, we would like continue working on the project by making it more user-friendly and efficient. Primary, the graphical outputs of the paths can be improved through the implementation of dynamic maps, which are easier to understand for the user. Such improvement can be achieved through work in web development, as it would be ideal to have the program run on a website, where the dynamic maps can be outputted and easily accessed by the user. Similarly, the algorithm input can be improved, so that users can enter the name of their current location, instead of specific coordinates. Finally, the time execution of the algorithm can be reduced by working with established Python libraries that are more efficient.

On another note, for future work we would like to focus on the data used for the calculation of sexual harassment risk in Medellin. We believe that current data is simplified, as it does not include critical factors such as street lighting, time of the day, and user's demographics. Through work in statistics, we can improve the numerical representation of sexual harassment risk, which will improve the accuracy of the paths outputted by the algorithm.

Finally, with the great knowledge acquired throughout the project we would like to take the same problematic of street sexual harassment and solve it as a bi-objective optimization problem. That it, we would like to implement an algorithm that finds a path with the shortest distance and the lowest risk at the same time. Such solution can be achieved through methods like Pareto's frontier, which we will implement after work in optimization.

ACKNOWLEDGEMENTS

This work was possible thanks to the help of the teacher assistants of Data Structures and Algorithms I, who consistently helped with the implementation of the algorithms and were always willing to solve questions and concerns.

We thank the students of Data Structures and Algorithms I for constantly inspiring us to make the project better.

The first author is grateful to her parents for supporting her education at Universidad Eafit.

The authors are grateful to Prof. Juan Carlos Duque, from Universidad EAFIT, for providing data from Medellín Life Quality Survey, from 2017, processed into a Shapefile.

REFERENCES

1. Galbrun, E., Pelechrinis, K. and Terzi, E., 2015. Urban navigation beyond shortest route: The case of safe paths. [online] Science Direct. Available at: https://www.sciencedirect.com/science/article/pii/S0306437915001854.

2. Mata, F., Torres-Ruiz, M., Guzmán, G., Quintero, R., Zagal-Flores, R., Moreno-Ibarra, M. and Loza, E., 2016. A Mobile Information System Based on Crowd-Sensed and Official Crime Data for Finding Safe Routes: A Case Study of Mexico City. [online] Mobile Information Systems. Available at: https://www.hindawi.com/journals/misy/2016/8068209/

3. Tarlekar, S., Bhat, A., Pandhe, S. and Halarnkar, T., 2016. Algorithm to Determine the Safest Route. International Journal of Computer Science and Information Technologies, Vol. 7 (3), 1536-1540. Available at: http://ijcsit.com/docs/Volume%207/vol7issue3/ijcsit20160703106.pdf

4. Ali, M., Rishta, S., Ansari, L., Hashem, T. and Khan, A., 2015. SafeStreet: Empowering Women Against Street Harassment using a Privacy-Aware Location Based Application. [online] ResearchGate. Available at: https://www.researchgate.net/publication/300656203_SafeStreet

5. Stop Street Harassment, 2022. What Is Street Harassment? | Stop Street Harassment. [online] Available at: https://stopstreetharassment.org/about/what-is-street-harassment/

6. Neeti, R. and Pamela, O., 2012. Street harassment: an unaddressed form of gender-based violence. [online] BMJ Journals. Available at: https://injuryprevention.bmj.com/content/18/Suppl_1/A145.2.info

7. Ruta N Medellín, 2022. Reducir el acoso callejero: el reto de la Secretaría de Mujeres - Ruta N. [online] Available at: https://www.rutanmedellin.org/es/programas-vigentes/2-uncategorised/592-reto-de-mujeres

8. Wikipedia, 2022. Dijkstra's algorithm. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Algorithm

9. Spanning Tree, 2020. How Dijkstra's Algorithm Works. [video] Available at: https://www.youtube.com/watch?v=EFg3u_E6eHU

10. Abiy, T., Pang, H. and Tiliksew, B., 2022. A* Search. [online] Brilliant Math & Science Wiki. Available at: https://brilliant.org/wiki/a-star-search/#heuristics

11. Lague, S., 2014. A* Pathfinding (E01: algorithm explanation). [video] Available at: https://www.youtube.com/watch?v=-L-WgKMFuhE

12. Datta, S., 2020. Floyd-Warshall Algorithm: Shortest Path Finding. [online] Baeldung. Available at: https://www.baeldung.com/cs/floyd-warshall-shortest-path

13. Wikipedia, 2022. Breadth-first search. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Breadth-first_search

14. Cormen, T. and Balkcom, D., 2015. The breadth-first search algorithm (BFS). [online] Khan Academy. Available at: https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/the-breadth-first-search-algorithm

15. Geeks for Geeks, 2022. Comparison between Adjacency List and Adjacency Matrix representation of Graph. [online] Geeks for Geeks. Available at: https://www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/

16. Brad Field CS. 2022. Shortest Path with Dijkstra's Algorithm. [online] Available at: https://bradfieldcs.com/algos/graphs/dijkstras-algorithm/

17. Pandey, M., 2021. Dijkstra Algorithm. [online] Scaler Topics. Available at: https://www.scaler.com/topics/data-structures/dijkstra-algorithm/#code-482-4-0--time-complexity

18. Programiz. 2022. Floyd-Warshall Algorithm. [online] Available at: https://www.programiz.com/dsa/floyd-warshall-algorithm

19. Code Examples. 2022. Why is the complexity of A* exponential in memory?. [online] Available at: https://code-examples.net/en/q/1a2cc9