Understanding the Shortest Path Problem

Ruksana Khan and Melissa Viator

BU MET College Computer Science Department

CS566 – Dr. Alexander Belyaev

Spring 2022

Table of Contents

Abstract

In computer science, the shortest path problem is a NP-complete problem where the solution finds a graph's optimal path. Shortest path planning is essential in numerous applications such as telephone networks, social networking applications, flighting agenda, designated file server, autonomous cars, robots, mapping platforms, etc. One of the most popular graph algorithms to solve the shortest path problem is Dijkstra's algorithm. While Dijkstra's algorithm successfully finds the optimal path among a graph's network of vertices, the time complexity, O(E * logV), is extremely inefficient as the number of vertices and edges of the graph increases. To juxtapose the runtime of Dijkstra's algorithm the project introduces a path planning algorithm, Rapidly-Exploring Random Trees (RRT). RRT's time complexity, O(n * logn), is asymptotically optimal in comparison to Dijkstra's algorithm as RRT does not visit each vertex or edge in the graph. However, RRT does not necessarily find the most optimal or shortest path. To overcome the limitations of these algorithms, this paper proposes the combination of RRT and Dijkstra's algorithm. In doing so, the RRT + Dijkstra algorithm solves the shortest path problem in a practical runtime. To implement the RRT + Dijkstra algorithm, a series of experiments were performed in a 2D environment where the results were compared to the original Dijkstra's algorithm and RRT algorithm. We discovered that the combination of RRT + Dijkstra was an innovative perspective to successfully find the shortest path with an improved runtime.

Introduction

The shortest path problem is an infamous question in graph theory that aims to find the optimal paths among a network of vertices in a graph. An algorithm to solve the shortest path problem will find a path between two vertices in a graph such that the total sum of the edge weights is minimum. There are several variations of the shortest path problem such as the single-pair shortest path, single-source shortest path, breadth-first search, all-pairs shortest path, and single-source widest path. For the purpose of this project, we will focus on the single-source shortest path which requires finding a collision free shortest path from a source vertex to all other vertices in a weighted graph. The single-source shortest path solution minimizes the sum of the weights of the edges in the path to find the most efficient solution.

To understand the importance of solving the shortest path problem, let's explore the real-world applications. Given the specific variation of the shortest path problem, there are a myriad of applications such as telephone networks, social networking applications, flighting agenda, designated file server, autonomous cars, robots, and, most notably, mapping platforms. When attempting to travel from position A to position B, for example, one encounters the shortest path problem due to the various routes available. Mapping platforms will utilize shortest path algorithms to find the minimum distance between the current position and the desired position. GPS capabilities permit the discovery of the shortest and fastest path between position A and position B where the edges connecting the two positions are weighted based on distance, traffic, delay-timing, etc. Thus, algorithms to solve the shortest path problem have fundamental and relevant real-world applications.

To begin our analysis of the shortest path problem, we will introduce and analyze one of the most popular algorithms in graph theory, Dijkstra's algorithm. First, we will seek to understand the applications and constraints of Dijkstra's algorithm. Next, we will introduce an opposing algorithm, Rapidly-Exploring Random Trees (RRT), and compare the strengths and weaknesses of each algorithm. After the initial analysis of each algorithm, we will implement and test the effectiveness of Dijkstra's algorithm versus RRT algorithm. With this understanding, we will explore a combination of Dijkstra's

algorithm and RRT in the hopes of improving the algorithm's effectiveness. The purpose of this project is to research and investigate the shortest path problem by exemplifying Dijkstra's algorithm alongside RRT and analyze the combination of the algorithms in hopes of finding an optimal solution to the shortest path problem.

Dijkstra's Algorithm

One of the most popular graph algorithms to solve the shortest path problem is Dijkstra's algorithm. Given a particular graph, Dijkstra's algorithm aims to find the shortest path from the source vertex to any other vertex on the graph. Dijkstra's algorithm is greedy, meaning that the algorithm chooses locally optimal solutions that may lead to a global solution. In this case, the global solution is the shortest path from the source vertex to any other vertex on the graph. The basic concept of Dijkstra's algorithm starts with a directed/undirected, weighted, and non-negative graph and a source vertex. As the algorithm is in process, it maintains two sets: 1) a set that contains the shortest path tree and 2) a set that contains vertices not yet included in the shortest path tree. Until all vertices are included in the shortest path tree, the algorithm will iteratively add the vertex that is not yet included and has the minimum distance from the source vertex. The outcome of the algorithm is a tree of the shortest path that minimizes the total distance among the source vertex to all other vertices in the graph.

The outline below provides the steps of Dijkstra's algorithm:

Input: A directed/undirected, weighted, and non-negative graph ***G*** and a source vertex ***s***
Initialize the two sets:

1. Set of vertices included in the shortest path tree (**Shortest_path_tree**); initialize as empty
2. Set of vertices not yet included in the shortest path tree; initialize with all vertices included in graph ***G***

Initialize the keys (distance values) of the source vertex ***s*** as 0 and all other vertices as ∞
While the set of vertices not yet included in the shortest path tree is not empty:

- Add the vertex not yet included in the **Shortest_path_tree** that has the minimum key
- For each adjacent vertex: relaxation of edges and update the key if the new distance is less than the current key
    - The new distance is computed by the key of the parent vertex plus the weighted edge of the adjacent vertex

Output: The shortest path from ***s*** to ***v*** for every vertex ***v*** in ***G***

When evaluating the effectiveness of an algorithm, time complexity, the amount of time it takes an algorithm to run as a function of the length of the input, is a key metric in algorithm analysis. The time complexity of Dijkstra's algorithm is O(E * logV), where V is the number of vertices and E is the total number of edges in the graph. The evaluation of the time complexity for Dijkstra's algorithm is subjective. For example, when comparing the time complexity of Dijkstra's algorithm to Bellman-Ford's algorithm, another famous shortest path algorithm with a time complexity of O(E*V), Bellman-Ford's is considerably larger than Dijkstra's algorithm. However, when comparing the time complexity of Dijkstra's algorithm to A*, a path-finding algorithm that incorporates heuristics, Dijkstra's algorithm is far less efficient as A* does not visit every vertex in the graph but instead only incorporates paths that are

headed generally in the correct direction of the target. Thus, the runtime of Dijkstra's algorithm is satisfactory when the number of vertices and edges composed in the graph is small. However, the runtime becomes extremely inefficient as the number of vertices and edges increases. In addition to time complexity, it is important to consider space complexity, the amount of memory space required to solve the algorithm as a function of the length of the input, when evaluating the effectiveness of an algorithm. The space complexity of Dijkstra's algorithm is O(V), where V is the number of vertices in the graph. When compared to A*, the space complexity of Dijkstra's algorithm is superior as A* requires exponential storage with respect to the length of the solution. Thus, the time and space complexity of Dijkstra's algorithm could be considered an advantage or disadvantage depending on the algorithm of contrast.

In addition to time and space complexity, there are several components to consider when evaluating Dijkstra's algorithm. For example, the correctness of Dijkstra's algorithm is an advantage in that the algorithm guarantees the shortest path of a graph given at least one valid path. Additionally, once the algorithm is complete, there is a least weight path to all vertices in the graph, meaning you can find the shortest path to any point. While this fact is a benefit, visiting every vertex in the graph does require significant time and space. Dijkstra's algorithm is an uninformed search algorithm, meaning the algorithm does not require additional information about the state or space to determine the next step. While there are advantages to the uninformed nature of Dijkstra's algorithm, this can also be a disadvantage in that the algorithm can enter an infinite loop if there isn't a path to the desired vertex. Another disadvantage of Dijkstra's algorithm is that it is unable to handle negative weights, limiting the applicability of the algorithm. Understanding such pros and cons of Dijkstra's algorithm helps to understand its overall effectiveness and consider potential variations of the algorithm to solve the shortest path problem.

Rapidly-Exploring Random Trees

Robotic path planning is a concept of finding a trajectory through an environment that connects a robot's start point and target point. The concept achieves a similar goal as the shortest path problem in that the algorithms aim to find a path between two points. However, the solution to the robotic path planning problem does not necessarily find the most optimal or shortest path. Often, the environment of a path planning problem is visualized in a binary occupancy grid which includes obstacles the robot must avoid. As a result, robotic path planning "determines how an object can navigate through a space with known or unknown obstacles while minimizing collisions" (Robotic Path Planning). Rapidly-Exploring Random Trees (RRT) is a sampling based algorithm to solve the robotic path planning problem. The basic concept of the RRT algorithm starts with an environment that includes the start vertex, target vertex, and possibly some obstacles. First, RRT randomly selects a new vertex anywhere in the state space and connects the vertex to the nearest vertex in the tree. A maximum distance can be set to limit the distance the new vertex is from the nearest vertex; this lessens the likelihood that the randomly selected vertex will cross an obstacle or travel too far in the opposite direction of the target vertex. If an obstacle interferes with the new vertex, then no new vertex is added to the tree and the algorithm continues. The RRT algorithm is effective because when there are large, unexplored areas of the state space, there is a fifty percent chance a random vertex is placed in this open area. Thus, new vertices are placed in unexplored areas causing the tree to rapidly expand. Once the path is in some threshold of the target vertex, a viable path is formed from the start vertex to the target vertex, solving the robotic path planning problem.

The outline below provides the steps of the RRT algorithm:

Input: A planning domain, start vertex v_{start}, target vertex v_{target}, and threshold distance d
Initialize the path tree, T, as empty that will include the set of vertices included in the path (the RRT)

- Add v_{start} to T
- While v_{new} is not within the threshold distance, d, of v_{target}
  - Select random vertex, v_{new}
  - Find the nearest vertex, $v_{nearest}$, to v_{new}
  - If the line connecting $v_{nearest}$ and v_{new} is greater than the threshold distance → v_{new} equals the vertex along the same line at the threshold distance
  - If the line connecting $v_{nearest}$ and v_{new} is not obstructed → add v_{new} to T
- Add v_{target} to T

Output: An RRT, T, that is a path from the start vertex, v_{start}, to the target vertex, v_{target}

Next, consider the effectiveness of the RRT algorithm in comparison to other path planning algorithms as well as Dijkstra's algorithm. The time and space complexity of RRT is O(n * logn) and O(n), respectively, where n denotes the total number of iterations. Notice that RRT's asymptotic analyses for time and space complexity is similar to Dijkstra's Algorithm. However, the fundamental difference is that RRT's time complexity is based on the number of samples while Dijkstra's time complexity is based on the number of vertices and edges included in the graph. Thus, the runtime of the RRT algorithm is asymptotically optimal in comparison to Dijkstra's algorithm. Such that, as the number of vertices and edges in the graph increases, RRT's time complexity is favorable as the algorithm does not visit each vertex or edge in the graph. Additionally, compared to the A* algorithm, RRT is likely to use fewer vertices since the vertices can be spaced further apart based on the maximum connection distance. Overall, the time complexity of RRT is a key strength when comparing the runtime of various graph and planning space algorithms.

While the analysis of time and space complexity is important in understanding the efficiency of an algorithm, understanding the tangible pros and cons of RRT is also critical for the purpose of this project. Such that, the RRT algorithm is guaranteed to find a path as the number of samples approaches infinity – but most likely a tremendously smaller number of samples. For comparison, the algorithm, Random Trees, does not accomplish this goal. While RRT finds a viable path, it is unlikely that it is the most optimal path. Given the stochastic nature of the RRT algorithm, the resulting path tends to zig-zag as new vertices are added to the tree. Thus, RRT works well in situations where the problem is only searching for a valid path and not necessarily the shortest path. The variation of RRT, RRT*, was introduced to find a more optimal path. While additional steps are added to the RRT* algorithm, the path becomes shorter and more optimal as the number of iterations increases. Understanding the strengths and weaknesses of the RRT algorithms begs the question if the concept of the path planning algorithm can be used in conjunction with shortest path algorithms such as Dijkstra's Algorithm.

Implementation

In this section, we will conduct a series of experiments to analyze the real-time performance of Dijkstra's algorithm and the RRT algorithm. The experiments utilize and modify source code to

implement and test the algorithms in MATLAB (Dijkstra Source Code, RRT Source Code). A series of five experiments were performed for each algorithm. For each experiment, a start vertex and an end vertex were randomly selected where the algorithm must find a valid path connecting the two vertices. The planning space – an x-y coordinate grid with a minimum of 0 and maximum of 100 for each axis – was replicated for each algorithm and experiment. To simplify the experiments, no obstacles were included in the planning space. The vertices available were all integer (x,y) coordinates in the planning space, totaling to 10,201 vertices included in the graph. Note that the implementation of Dijkstra's algorithm was theoretical as the completion of running the algorithm on a complete graph with 10,201 vertices and 52,025,100 edges is impractical, taking approximately 208,550,040 iterations based on an O(E * logV) runtime. Figure 1 below summarizes the experimental results from the implementation of the algorithms.

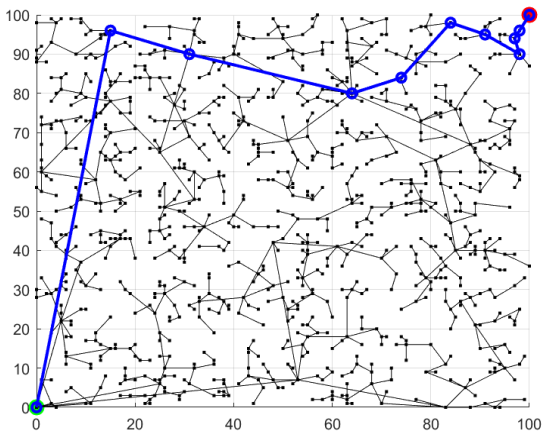
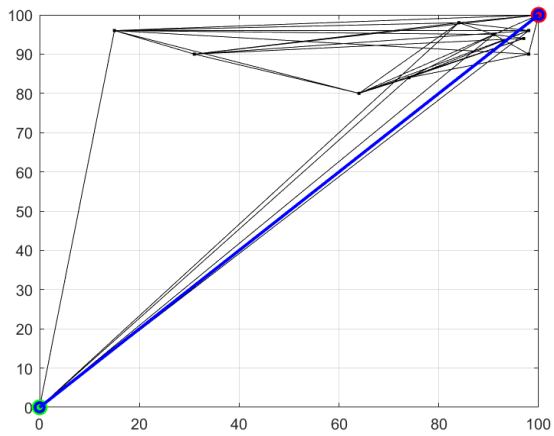
To analyze and compare the effectiveness and efficiency of the algorithms, the following factors were measured during each experiment: path length (in units); total path vertices; and runtime (in number of iterations). As expected, Dijkstra's algorithm correctly finds the shortest path; however, the runtime of Dijkstra's algorithm is astronomically high. This experiment revealed the severe limitations of Dijkstra's algorithm as the number of vertices and edges of the graph increases. When the number of vertices of a fully-connected graph is high, it is extremely computationally expensive to visit each vertex in a graph in order to find the shortest path to the target vertex. Comparatively, the RRT algorithm does find a valid path from the start vertex to the target vertex, but the algorithm does not return the shortest path. The experiments confirm that RRT does not solve the shortest path problem. However, the runtime of RRT is significantly smaller than Dijkstra's algorithm because RRT does not visit each vertex in the graph. The results demonstrate that the average number of iterations for RRT is approximately 1.3M times smaller than Dijkstra' algorithm. By performing real-time experiments of Dijkstra and RRT, we confirmed and evaluated the effectiveness and efficiency of the algorithms in finding a solution to the shortest path problem.

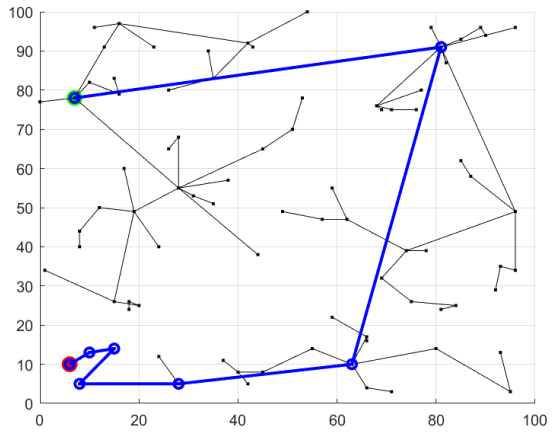
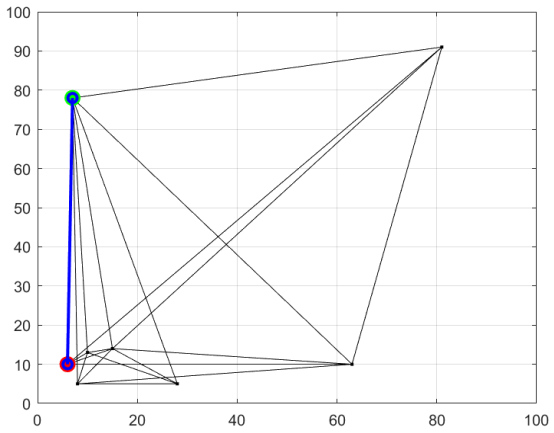
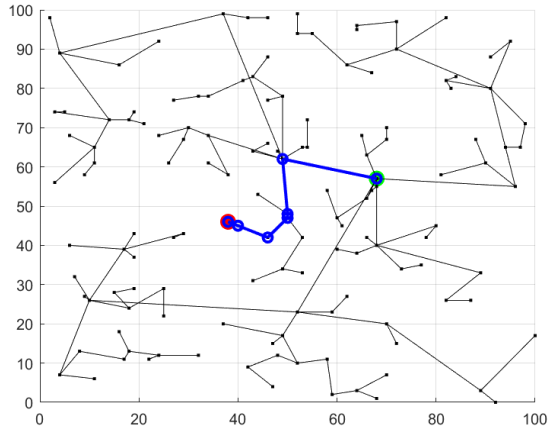
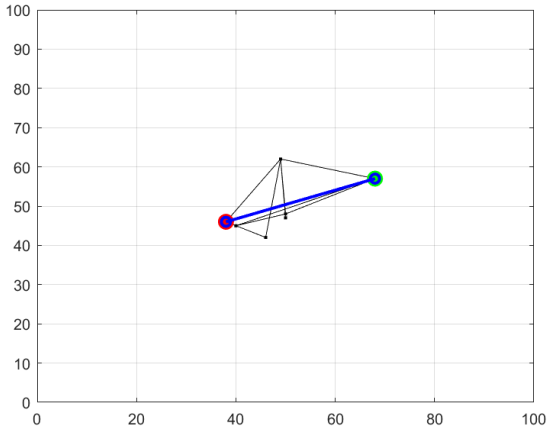
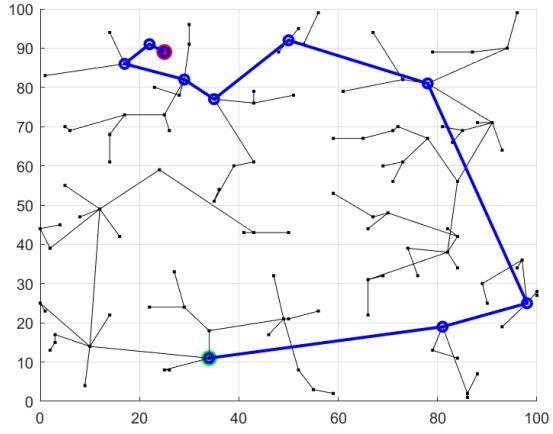
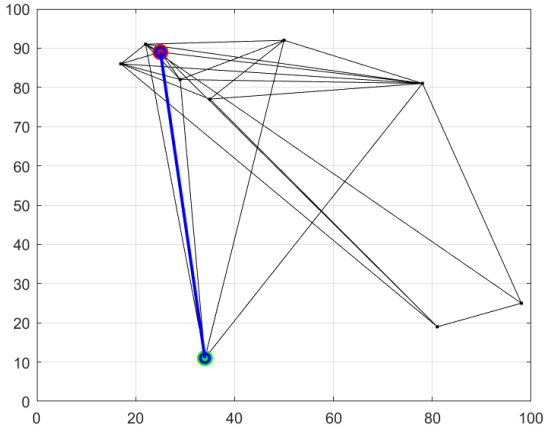
From the experiments, we identified the strengths and weaknesses of Dijkstra's algorithm and the RRT algorithm. We determined that while Dijkstra's strength is in correctness (i.e., finding the shortest path), RRT's strength is in efficiency (i.e., minimizing the number of iterations). With this in mind, we are seeking a combination of RRT + Dijkstra's algorithm to solve the shortest path problem in a practical runtime. The combination simply works by 1) implementing the RRT algorithm in the original planning space and 2) implementing Dijkstra's algorithm in the path created by RRT. The experiments highlight the success of this combination as the shortest path was found in a runtime only slightly higher than RRT. Figure 1 and Figure 2 below include the results and graph of the RRT + Dijkstra algorithm. The visualization of the planning space first illustrates the valid path created by RRT and then demonstrates the shortest path created by Dijkstra's algorithm on the RRT path. From the algorithms we analyzed, the RRT + Dijkstra algorithm is the most optimal in solving the shortest path problem.

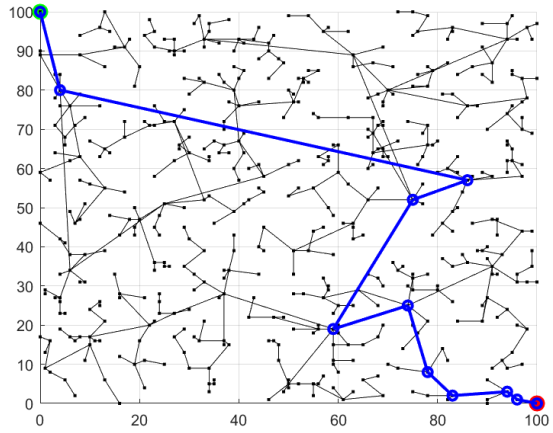
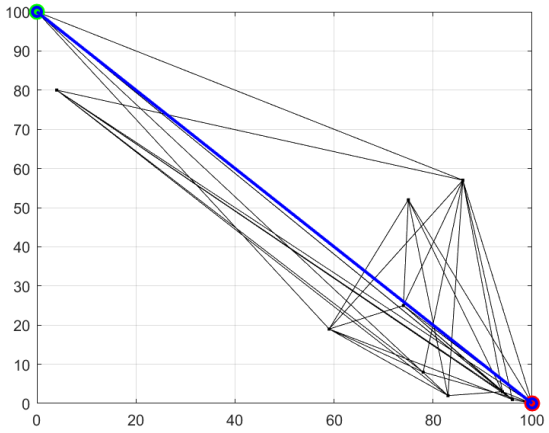
Figure 1: Table of the experimental results from Dijkstra, RRT, and RRT+ Dijkstra algorithms

Experiment	Start Vertex	End Vertex	Shortest Path (units)	Algorithm	Path Length (units)	Total Path Vertices	Time (iterations)
1	(0, 0)	(100, 100)	141.42	Dijkstra	141.42		208,550,040
				RRT	203.76	11	998
				RRT + Dijkstra	141.42	2	1080
2	(7, 78)	(6, 10)	68.01	Dijkstra	68.01		208,550,040
				RRT	234.97	8	84
				RRT + Dijkstra	68.01	2	128
3	(68,57)	(38,46)	31.95	Dijkstra	31.95		208,550,040
				RRT	50.03	7	137
				RRT + Dijkstra	31.95	2	156
4	(34,11)	(25,89)	78.52	Dijkstra	78.52		208,550,040
				RRT	207.60	10	114
				RRT + Dijkstra	78.52	2	176
5	(0,100)	(100,0)	141.42	Dijkstra	141.42		208,550,040
				RRT	213.74	11	522
				RRT + Dijkstra	141.42	2	608

Figure 2: Visualization of the experimental results from RRT and RRT+ Dijkstra algorithms

Experiment	RRT	RRT + Dijkstra
1		

2		
3		
4		

5		

Conclusion

The purpose of this project is to understand and implement the shortest path problem. Dijkstra's algorithm was utilized to demonstrate a solution to the shortest path problem as the algorithm iterates through a graph to find the optimal path. We investigated the effectiveness of Dijkstra's algorithm by analyzing its strengths, weaknesses, and practically in comparison to other algorithms. While Dijkstra's strength lies in its correctness, the time complexity of Dijkstra's algorithm is impractical in most real-word scenarios. To juxtapose Dijkstra's algorithm, we introduced Rapidly-Exploring Random Trees (RRT), a sampling based algorithm to solve the robotic path planning problem. During the analysis of RRT's effectiveness, we found that the runtime of the RRT algorithm is asymptotically optimal in comparison to Dijkstra's algorithm. However, RRT does not find a solution to the shortest path problem. The analysis of the two algorithms exposed the complementary strengths and weaknesses of Dijkstra's algorithm and RRT. With this understanding, we explored the combination of Dijkstra's algorithm and RRT in the hopes of improving the runtime of the algorithm while maintaining a solution to the shortest path problem.

To further understand the effectiveness of Dijkstra's algorithm, RRT, and the combination of RRT + Dijkstra, a series of five experiments were performed. The experiments aimed to find a path from the start vertex to the target vertex of a graph and measured the resulting path length, total path vertices, and runtime for each algorithm. The implementation of the algorithms reiterated the finding from our initial analyses of Dijkstra's algorithm and RRT. Additionally, the implementation of the RRT + Dijkstra algorithm successfully found the shortest path in a practical runtime. We believe that the combination of RRT + Dijkstra was an innovative perspective to successfully find the shortest path from the start vertex to the target vertex. From this, we were able to achieve the goal of the project to understand the shortest path problem and implement an optimal solution.

Source Code

Dijkstra Source Code: Dbekaert. (2015, April 24). Stamps/dijkstra.m at master · dbekaert/stamps. GitHub. Retrieved March 7, 2022, from https://github.com/dbekaert/StaMPS/blob/master/matlab/dijkstra.m

RRT Source Code: Rapidly-exploring random trees algorithm. MathWorks. (n.d.). Retrieved March 7, 2022, from https://www.mathworks.com/matlabcentral/fileexchange/53772-rapidly-exploring-random-trees-algorithm?s_tid=FX_rc3_behav

Work Cited

Dijsktra's algorithm. GeeksforGeeks. (2021, September 10). Retrieved March 7, 2022, from https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Dijkstra's algorithm. Programiz. (n.d.). Retrieved March 7, 2022, from https://www.programiz.com/dsa/dijkstra-algorithm

Dijkstra's shortest path algorithm. Brilliant Math & Science Wiki. (n.d.). Retrieved March 7, 2022, from https://brilliant.org/wiki/dijkstras-short-path-finder/

Robotic Path Planning. Path Planning. (n.d.). Retrieved March 7, 2022, from https://fab.cba.mit.edu/classes/865.21/topics/path_planning/robotic.html

RRT- dijkstra: An improved path planning ... - dergipark. (n.d.). Retrieved March 7, 2022, from https://dergipark.org.tr/en/download/article-file/1254850