Comparative Analysis of A* and RRT* Pathfinding Algorithms for Autonomous Drone Navigation in Different Environments

Colin Brennan
colinpbrennan07@gmail.com

Nicholas Ciordas
nicholas.ciordas@gmail.com

Samhita Pokkunuri
samhita.p@yahoo.com

Nickolas Regas
nicka.regas@gmail.com

Adam Wahid
adamw@tutamail.com

Will Wands
wowands@gmail.com

Shreya Srikanth*
ss3645@scarletmail.rutgers.edu

Governor's School of New Jersey Program in Engineering & Technology
July 26, 2024

*Corresponding Author

***Abstract*—Unmanned Aerial Vehicles (UAVs) have developed significantly in recent years, allowing for the facilitation of improvements in many different aspects. The importance of navigating various environments autonomously cannot be understated and may allow for expanded opportunities when considering future utilization ranging from military to personal use. This research compares different pathfinding algorithms and their benefits and drawbacks in distinct environments. The performances of each algorithm in the mazes could be used to determine when to use each algorithm when applied to real-world scenarios. The environments were broken up into four kinds of mazes: ladder mazes, single solution mazes, high-density obstacle mazes, and low-density obstacle mazes, with each representing a unique real-world environment. The pathfinding algorithms A*, a grid-based algorithm, and Rapidly-exploring Random Tree* (RRT*), a sampling-based algorithm, were visualized using Pygame and compared along four metrics through real-world testing: flight time, compute time, flight distance, and margin of error (displacement from the goal node). These metrics showed that A* performed three to five times faster than RRT* in compute time for every maze type. Furthermore, RRT* performed better than A* in flight time and flight distance for low-density, high-density, and ladder mazes; however, it performed similarly to A* in single solution mazes. These results provide valuable insight into selecting the most appropriate algorithms based on specific situational needs.**

I. Introduction

Autonomous navigation has recently become at the forefront of various developmental technologies. Numerous algorithms, including deep learning and Ground Control Systems (GCS) [1], enable drones to traverse environments filled with obstacles and varying terrain without human error. These algorithms allow drones to simulate real-life applications, ranging from precision agriculture and ecological sampling to search and rescue and military-based operations. As of 2024, the autonomous drone navigation market has expanded to $9.73 billion (USD) and is expected to grow at a Compound Annual Growth Rate (CAGR) of 15.97% by 2030 [2]. This predicted growth underscores the importance of designing autonomous UAV systems that are deployable across various professions, offering cost-efficiency and safety advantages through simulations before field testing.

The emergence of pathfinding algorithms to navigate these complex environments has become crucial for enabling robust outdoor experimentation. Algorithms such as A* and Rapidly-exploring Random Tree (RRT*) are extensively researched in this domain. The A* algorithm is renowned for its accuracy and efficiency in finding the shortest path in grid-based environments, while RRT* excels in precision and handling dynamic obstacles. Thus, there is a critical need to develop drone navigation systems that can employ these algorithms to accommodate all types of UAVs and situations.

II. Background

A. DJI Tello EDU

The DJI Tello EDU quadcopter was used to test each pathfinding algorithm in different mazes. The DJI Tello EDU has the advantage of easy programmability, as it can function with Scratch, Python, and Swift [3]. Furthermore, the EDU can pair with the SDK 2.0 development toolkit to convert the search algorithm's results into drone movement [4].

B. Pygame Library

To visualize both algorithms, the program utilized the Python library Pygame, which includes variable modules for displaying the maze layout, the solution path, and key points such as the start and end locations. In this case, the code updated the maze layout and the corresponding solution while illustrating the starting and ending points. This made it easier to understand the functionality of each algorithm and map out how the drone would move in a similar environment [5].

C. Types of Mazes

Single solution mazes are useful due to their long winding paths and reliability in producing extreme test environments for the drone. They represent crowded indoor situations where there is only one path from the start to the exit position. An ideal single solution maze has no loops (i.e., places where the path runs back into itself) but has dead ends. This complexity provides a rigorous test of the drone's navigation capabilities.

Random obstacle mazes feature obstacles placed at random coordinates with a predefined density to simulate more arbitrary areas such as forests. Adjusting the randomness and density of obstacles for diverse test scenarios helps closely mimic aerial navigation conditions. For example, low-density obstacle mazes feature open areas that enable the drone to build up speed and allow it to make shallow turns to avoid obstacles. Conversely, high-density obstacle mazes force the drone to take steep turns in rapid succession.

Ladder mazes generate large obstacles that must be completely circumnavigated for the drone to progress to the goal node while leaving certain areas wide open, allowing the algorithm to choose from multiple paths. This maze is representative of a city arrangement where a drone can take multiple bypass routes around large buildings. This maze tests the algorithm's ability to select the most efficient path even when multiple options are presented.

D. Dijkstra and A* Algorithms

Dijkstra is a grid-based pathfinding algorithm that can output the shortest path between two predefined nodes by utilizing a cost function. A cost function is a mathematical equation determining the "price" (in this case, distance) of certain actions, which is then used to maximize the program's efficiency. In this case, the cost function is distance-based,

$$F(N) = G(N)$$

with its parameter being the distance between the node the algorithm is currently exploring and the start node. This value is repeatedly updated with each program iteration to reflect the smallest known value for each node. The algorithm begins by assigning a value of infinity to each node, the only exception being the source node's value which is set to zero. The starting node branches out to its neighboring nodes, defining the cost of each neighboring node based on the cost of their respective connections. The node with the smallest value is then selected, and its neighbor's values are calculated [Fig. 1]. As this process continues, the cost of certain nodes (distance from current to starting node) becomes optimized as surrounding weights are redefined [6]. Each node can undergo relaxation, or a cost reduction once, in which a new, more optimal path is discovered from the starting node to the current node. One

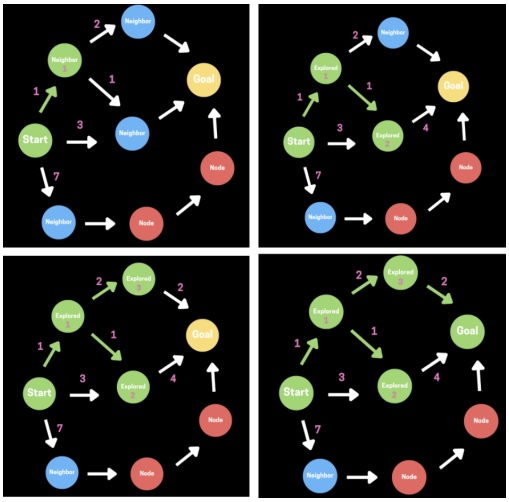

Fig. 1: Dijkstra Pathfinding Algorithm

advantage of Dijkstra is its utilization of a parent dictionary, which stores each node's immediate predecessor. Since the algorithm can simply retrace its steps through the parent dictionary once it computes, reconstructing the shortest path from the start node to the goal node is very straightforward. While Dijkstra is effective, it struggles with solving large, complex mazes, as it has no heuristic implementation (the algorithm's ability to estimate its distance to the goal node). For example, it would reward traveling away from the goal node if the path leading away from the goal is shorter than the path leading toward it.

A*, another grid-based search algorithm, leverages a heuristic to enhance performance. Like Dijkstra's approach, A* operates on a grid of nodes with predefined obstacles, a designated start point, and a fixed goal point. However, A* distinguishes itself by incorporating a heuristic that estimates the distance from the evaluated current node to the goal node. This heuristic guides the algorithm to prioritize exploring paths that are closer to the goal, thereby potentially reducing unnecessary exploration and optimizing the search process. [7]

The cost function equation A* utilizes is the sum of two functions:

$$F(N) = G(N) + H(N)$$

G(N) represents the distance between the starting and current node, and H(N) represents the heuristic, the straight line distance between the current and goal node. During the A* algorithm, the F(N) of the surrounding nodes is calculated, the node with the smallest value is selected, and the process repeats until the goal is reached. A* calculates the F(N) for the current node's neighbors, appends each to a priority queue, and chooses to explore the node with the lowest F(N) while recording its parent node. As more nodes are explored, the

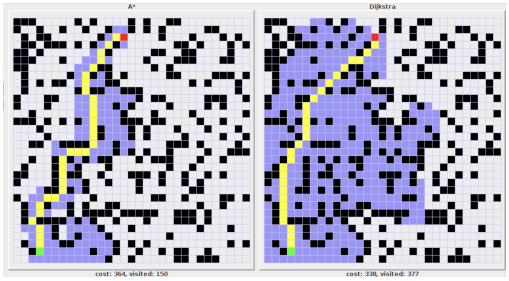

Fig. 2: A* vs. Dijkstra
Source: Adapted from [9]

F(N) of neighboring nodes that may have been previously evaluated are recalculated to ensure that the shortest possible paths are being considered. If the new F(N) is lower when coming from a different parent, that node's parent is updated to reflect the new path [7]. Eventually, the algorithm will return the quickest path to the goal, where its added heuristic effectively reduces the number of nodes the algorithm has to explore.

A* often performs an order of magnitude better than its predecessor. When the size of the maze is at 300 nodes, A* can comfortably complete the maze in under 0.05s, while Dijkstra's algorithm takes 0.5s on the same machine. This discrepancy is due to Dijkstra's algorithm having to search a much higher proportion of those nodes when compared to A* [8].

As can be seen, the inclusion of this heuristic fundamentally alters A*'s approach and differentiates it from Dijkstra's approach. While Dijkstra's algorithm guarantees finding the shortest path by exploring all possible nodes uniformly, A* narrows its search based on the heuristic guidance. This heuristic-informed decision-making enables A* to often reach the goal more efficiently, especially in scenarios where the grid is large and complex.

However, the success of A* heavily depends on the quality of the heuristic chosen. A well-designed heuristic can significantly improve the algorithm's performance by accurately estimating the remaining distance to the goal, guiding A* more swiftly toward the optimal solution. However, an overly optimistic or inaccurate heuristic might lead A* astray, potentially missing the optimal path or increasing computational overhead [10].

E. RRT and RRT Algorithms*

The Rapidly-exploring Random Tree (RRT) algorithm is a sampling-based path planning algorithm designed to efficiently search spaces consisting of multiple dimensions [11]. It is easily scalable from two to three dimensions and beyond, whereas more traditional search methods become computationally impractical at this size. While A* is grid-based and must commit all nodes to memory regardless of exploration, RRT (sampling-based) only needs to keep track of explored nodes. As the number of dimensions increases, the state space A* must commit to memory grows drastically, but RRT remains unaffected.

The RRT algorithm operates by generating random points within a specified region and connecting these points to the nearest node in the existing tree. A new node is added along the line segment connecting the random point to the nearest node at a distance capped by a predefined step size. If this new node is unreachable (i.e. the path to reach it passes through an obstacle), it is discarded, and a new point is generated. This process repeats iteratively, with the tree expanding until it reaches the goal. To improve efficiency, RRT biases the generation of points towards large unexplored areas, reducing redundant exploration in densely populated regions. While RRT effectively finds a feasible path in a gridless environment, it does not optimize it. The resulting route can be jagged and inefficient since RRT does not recalibrate or refine it after it is initially found. This can be problematic for applications such as autonomous drone navigation, where a smoother path is essential [12].

RRT*, the successor to RRT, addresses these limitations by introducing iterative path optimization. Unlike RRT, RRT* not only finds an initial feasible path but also refines it over time. After generating an initial path, RRT* continues to add new nodes and re-evaluate existing connections. It does so by searching within a specified radius around each new node to identify potential nodes for re-connection, which helps in minimizing the path length and improving overall path quality.

RRT* dynamically adjusts the connections between nodes to ensure the path is continuously optimized. While improving path quality, this refinement process increases the algorithm's computational complexity and execution time. The trade-off between path optimality and computational efficiency is a key consideration in real-world applications.

In many circumstances, RRT* does require four times more computation time than its predecessor (sometimes more depending on termination criteria), but this drawback is countered by the algorithm being asymptotically optimal [13]. While RRT could run forever and maintain a non-optimal solution, RRT* will approach the optimal solution with each iteration of the algorithm. For this reason, implementing a termination criterion is crucial for enhancing the practical performance of RRT*. Due to the possibility of infinite iterations, developers must terminate the code after it iterates a specific number of times. Importantly, developers must balance the number of iterations or node additions to achieve a feasible trade-off between computational expense and path quality.

III. Methods

This research compared the performance of A* and RRT* in various types of mazes to simulate different real-world environments. The Python library 'mmaze' was used to create single solution mazes, storing maze nodes in an array where obstacles were denoted by one and open nodes by zero. The maze was drawn using the Pygame library, with obstacles in black and nodes in white [5]. The Kruskal algorithm was

applied to create a minimum spanning tree, eliminating closed loops [14].

Two additional types of mazes, low and high-density obstacle mazes, could be created by adjusting the number of randomly generated obstacles. Both mazes ensured blocks could not be drawn close to the top-left and bottom-right corners to prevent from enclosing the maze's start and end.

All mazes were randomly generated to facilitate rapid testing and the collection of large data samples, minimizing outlier effects on the findings' accuracy.

After Pygame simulations, the code was adapted for drone movement using the Tello SDK 2.0 and its commands [4]. A* employed a grid-based coordinate system based on the size of each obstacle in the maze. RRT* is not restricted to the size of the obstacles and can place a node at any point. Both algorithms executed drone movement similarly, controlling the Tello EDU with commands relative to its current position.

A. A* Program

The algorithm constantly maintained a priority queue `open_set` to keep track of nodes that were being currently evaluated. The initial node, defined by `start`, began the process. The algorithm checked for the `g_score`, representing the distance from that node to the starting point, the `h_score`, representing the heuristic value, and the `f_score` of the node, which is the total cost value of reaching the current node from the starting node, including its estimated distance from the goal. All nodes surrounding the starting node became part of the `open_set`, and the algorithm selected the node with the lowest `f_score` to explore first, which became the new source node. This process repeated continuously until the goal node was detected. When this occurred, the `came_from` dictionary reconstructed the shortest path by tracing backward from the goal to the starting position. A* code can be referenced in section VII-A.

B. RRT* Program

RRT* generated a path using the branching node technique. The algorithm then optimized this path over multiple iterations. Each iteration generated more nodes, reducing the number of jagged movements and turns, thus slowly making the path shorter.

The algorithm initialized a start and goal node and used a sampling-based pathfinding system to create a tree rooted at the start node that branched out and eventually reached the goal node. This gradual expansion occurred by adding randomly sampled nodes (within the bounds of the maze and max guess radius defined at the start of the algorithm). During this process, the `step_size` parameter constantly dictated the max distance between each node and the random tree configurations branching out. This parameter was essential, as it affected the granularity of the search: if the `step_size` became too low, the search to approach the goal node often became lost or uncanny. If the `step_size` was too high, it led to divergent, reckless behavior. After the goal node was detected, the algorithm linked the goal to the starting node based on the connections developed in the tree and its overall growth. To optimize the performance, the space bar was clicked, after which the program increased the number of nodes utilized in the algorithm; thus, connections became reevaluated, and nodes were adjusted to minimize the path costs. This enhanced the algorithm's capability to find near-optimal paths while adapting to changes in the environment or new information.

After completing the simulations, the code was altered to allow for drone movement, using the Tello SDK 2.0 and its relevant commands. In both the A* and RRT* codes, a coordinate system was set up to help direct the drone to the correct point. RRT* code can be referenced in section VII-B.

C. Bridging A* to DJI Tello EDU

Since A* employs rigid turning, as it is a grid-based algorithm, the nodes were marked at exact distances away from each other, so the entire path of nodes was determined based on each node's relationship to its precursor. As the drone traversed through the defined path, it utilized the relationship between each node (Right, Left, Forward, and Backward) to determine its next movement. Using the coordinate grid was essential because it helped determine which direction the drone could move in a sequential manner. Based on these values, the DJI Tello EDU was commanded to move in those respective directions with a step size of 50 cm.

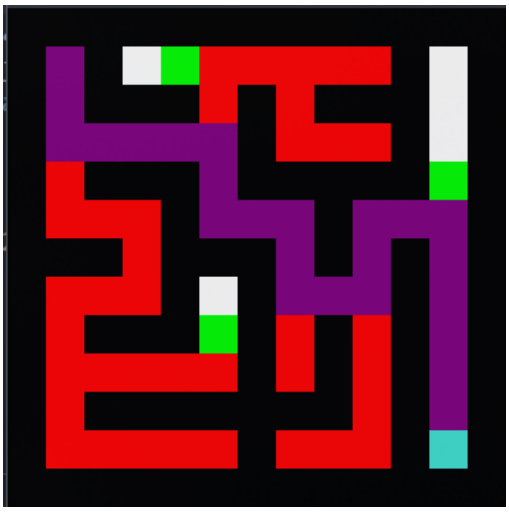

Fig. 3: A* Pathfinding Algorithm Implemented in Pygame

D. Bridging RRT* to DJI Tello EDU

Since RRT* is a sampling-based algorithm, meaning it employs jagged movements rather than rigid turns, a system of coordinates describing the nodes taken in the final path was used to track the drone's movement and, thus, the path it must take. The `go_xyz_speed` allowed the drone to traverse from node to node instead of in a specific direction. Notably, the DJI Tello EDU had a hardware cap that prohibited it from traveling less than 20 cm in two dimensions. For this reason, the predetermined step size had to be enlarged so that the space

between nodes exceeded 20 cm. The algorithm, furthermore, was commanded to undergo 30 iterations of optimization before running the code to ensure that the path was near optimal.

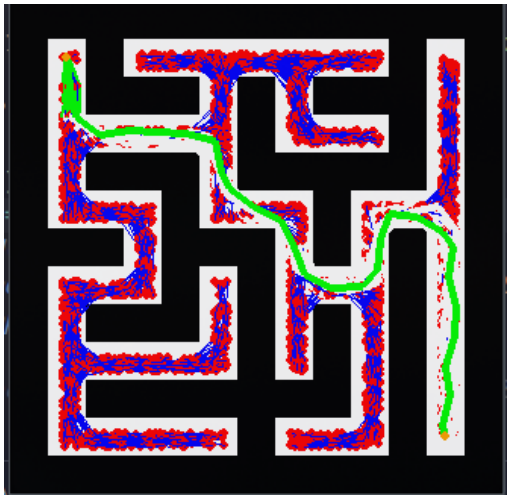

Fig. 4: RRT* Pathfinding Algorithm Implemented in Pygame

E. Constructing Randomized Mazes

Overall, 48 tests were conducted, 24 for each algorithm. For each test, four metrics were analyzed: flight time, compute time, flight distance, and margin of error (distance between the drone's final position and the goal node). The 24 tests for each algorithm consisted of four distinct maze types: single solution, ladder, high-density random obstacle, and low-density random obstacle, all randomly generated three times, with each maze being tested twice. However, generating randomized mazes for both tests could provide an unfair advantage to a specific algorithm's performance if fewer turns, for example, were required. Thus, once a maze was generated for one of the algorithms, it was reused as the maze for the second algorithm, forcing the two algorithms to run on the exact same randomly generated maze.

By quantifying the performance of each algorithm across various scenarios, advancements could be made in navigation strategies, ultimately aiming to enhance the reliability and efficiency of drone operations in real-world applications.

F. Setting Up Testing Environment for A and RRT**

This paper aimed to bridge the gap between virtual and real-world applications by transforming maze simulations into physical replicas suitable for testing autonomous navigation algorithms like A* and RRT*.

A conversion factor was employed to translate the dimensions of the Pygame simulation in pixels to the dimensions of the real-world testing environment in centimeters, ensuring consistency across all maze types. Furthermore, using a conversion factor helped pinpoint the exact positions of the drone during experimentation to provide a general marker of the path without having to do excessive mathematical calculations. Given that each maze shared identical dimensions, a single physical model sufficed and could be used across both algorithms. In the code, each grid space represented a value of 50 cm in real life; thus, creating an 11 x 11 block maze would result in a 5.5 by 5.5 meter maze, the boundaries of which were carefully measured and marked with tape.

Vertical and horizontal tapes divided the outline into a grid system, aligning with the virtual model's grid nodes. This approach facilitated rapid maze construction, crucial for conducting many trials. At designated start and goal nodes, marked with an 'X', the midpoint was calculated. Placing the drone precisely at the center of this grid ensured consistent starting conditions across experiments [Fig. 5]. The accuracy of these placements was essential for assessing each algorithm's performance, particularly in gauging the margin of error (distance from the goal node to the drone's final position) inherent in autonomous navigation algorithms.

Fig. 5: Image of DJI Tello at its Starting Position

G. Calculation of Metrics

Quantifying the performance of A* and RRT* algorithms involved measuring several critical factors to facilitate a comprehensive comparison. To minimize the effects of external variables on test measurements, all tests were conducted on the same laptop

Four key metrics were measured to accurately assess each algorithm's performance. Firstly, a stopwatch was used to calculate the flight time, providing insights into how efficiently each algorithm navigated through diverse maze configurations. This measurement highlighted variations in traversal efficiency based on the complexity of the paths generated by A* and RRT*.

Secondly, the margin of error was evaluated by determining the distance between the drone's landing position and the midpoint of the goal node. Both the x-coordinate and y-coordinate displacement were measured to generate a direct indicator of each algorithm's precision in reaching its intended destination within the maze. These outcomes revealed which

algorithm consistently achieved more accurate results across different test scenarios.

Compute time represented the third critical metric, where the time taken to set up the maze and the path the drone would follow runs before the drone takes off. Comparing the compute times between both algorithms provides insight into whether one algorithm can perform faster and thus allocate fewer resources. This way, modifications to the code can be made quicker, which increases the developmental process.

Lastly, the total distance traveled by the drone was tracked during each test run. This metric not only highlighted the overall efficiency of each algorithm in terms of path optimization but also provided a practical measure of how well they managed the drone's movements within the maze environment.

Systematic data collection through these measurements aimed to objectively compare the effectiveness of A* and RRT* algorithms across diverse scenarios. The insights gained from these quantitative analyses contribute to refining autonomous navigation strategies, enhancing their applicability in various scenarios that potentially mimic real-world applications.

IV. Results

Performance metrics across all tests showed a large variation of measurements, but certain patterns were consistently observed throughout the different terrains.

A. Flight Times

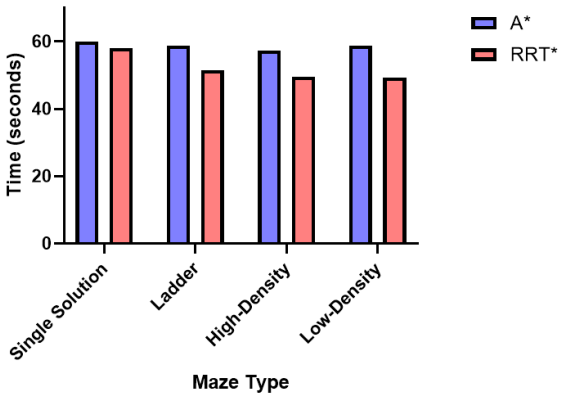

Fig. 6: Average Flight Time of A* and RRT* in Different Environments

The flight time measured the time the drone took to traverse through the entire maze, from takeoff to landing.

The results in Figure 6 show that RRT*, on average, took less time than A* to traverse through the paths in all environments except single solution mazes. In single solution mazes, the difference in flight times was negligible. This difference in traversal time can be attributed to the algorithmic differences in both pathfinding methods. Since A* is a grid-based pathfinding algorithm, it is constricted to 90° movements. This is not optimal in environments where many turns are needed, such as high-density obstacle mazes. It is also not optimal in open environments such as ladder mazes, or low-density obstacle mazes because it will not be able to utilize the open space to produce diagonal movement. In single-solution mazes, much of the maze is made of obstacles and ninety-degree turns, so the benefits of RRT* become more negligible.

B. Flight Distances

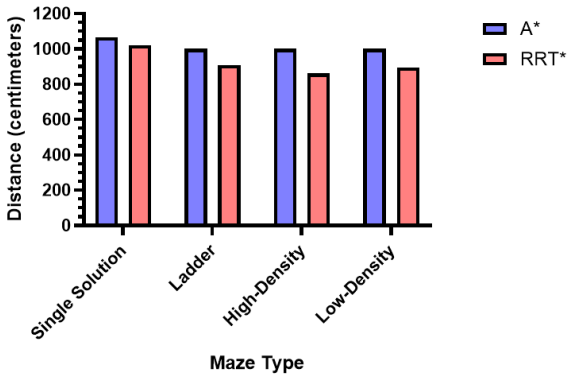

Fig. 7: Average Flight Distance of A* and RRT* in Different Environments

The most important metric for this research was flight distance, as each algorithm was optimized to find the shortest route to the goal node. In all experiments, the drone's theoretical flight distance from start to finish was measured. For A*, each grid box was predetermined to be 50 centimeters apart, allowing the distance to be counted post-testing. On the other hand, the algorithm for RRT* would use its conversion factor to translate the pixel distance between each node into real-world distances.

Based on the data analysis, the flight distance for A* tended to be 11% more than that of RRT*. This is because A* can only travel along straight paths, no matter how optimized it is. However, RRT* allows the drone to move diagonally and travel closer to the walls, allowing for shortened distances when traveling around corners and along stretches of open space.

C. Compute Times

In every maze type, RRT* took three to five times longer to compute than A*, as shown in Figure 8. This occurred because RRT* continuously adds new nodes to the path to further optimize it and shorten the distance between the starting and goal nodes. Each experimentation for RRT* was optimized 30 times, reforming the algorithm's path length without significantly damaging the computation time.

Out of all of the environments tested, RRT* had the largest compute time in high-density mazes. This can be attributed to it making more drastic path changes each iteration due to the high density of obstacles.

There was no large variation across the same algorithm type for the other mazes. Regardless of the type of maze utilized, as long as the algorithm was the same, the compute time remained relatively constant.

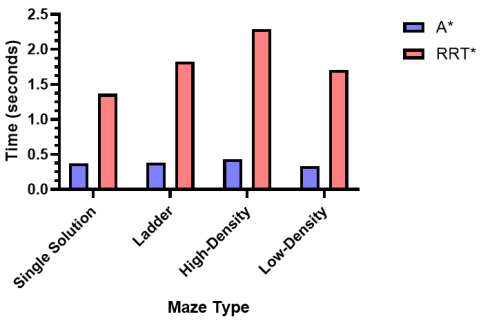

Fig. 8: Average Compute Time of A* and RRT* in Different Environments

D. Margin of Error

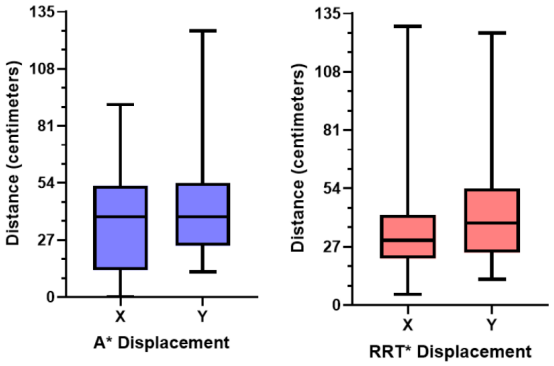

Fig. 9: Box and Whisker Plots of Error

While testing the drone's flight, it was found that there was a median cumulative error of 48.7 cm as measured from the center of the expected goal point to the landing point. Broken down, it was found that A* had a median error of 58.6cm and RRT* had a median error of 41.6cm. This is because while traversing the grid through the coordinates dictated by both A* and RRT*, each movement caused a small but noticeable drift. This drift accumulated and led to a significant error when the drone eventually touched down away from the expected goal point.

Distance Traveled	Average Error
50 cm	16 cm
200 cm	21 cm
350 cm	37 cm
500 cm	40.25 cm

TABLE I: Average Error Per Distance Traveled

According to Table 1, there was a positive correlation between the distance traveled in a straight line and the error from the expected endpoint. Since A* has a larger average flight distance, as shown in Figure 7, this is the most likely explanation for why A* has a higher margin of error than RRT*.

V. Conclusion

A. Discussion

Pathfinding algorithms are becoming increasingly important for autonomous drone navigation, as drones are becoming capable of traveling through random paths without needing human-based systems or a GPS. This study analyzed two prominent pathfinding algorithms, A* and RRT*, in the context of autonomous drone navigation through differing maze environments.

The A* algorithm emerged as a robust solution to scenarios where computation speed is the driving factor. It demonstrated rapid convergence to the shortest path in all maze environments but lacked efficiency in flight distance. Due to its lack of computational resource usage, it is well-suited for surveillance or inspection missions that necessitate a quick response. Operations that necessitate speed in response and have limited computing power rather than pinpoint accuracy would benefit from employing this algorithm.

Conversely, the RRT* algorithm showcased a distinctive approach rooted in sampling-based methods designed to explore environments with complex and unpredictable obstacles. While RRT* exhibited longer computational times due to its iterative path refinement process, it excelled in generating paths that were notably smoother and more adaptable to unique environmental conditions.

In practical applications, the choice between A* and RRT* depends on the specific requirements of the mission and the characteristics of the environment. A* remains the algorithm of choice for scenarios where compute speed is paramount. Although A* reflected a similar flight time to that of RRT* in single solution mazes, its lack of need for computational resources gives it a slight edge when navigating environments similar to that type of maze. Some real-world examples that A* would be optimal for include warehouse navigation to conduct inventory, indoor three-dimensional mapping for renovations, agricultural monitoring, and package delivery. Its ability to compute optimal paths ensures rapid response times and efficient resource utilization, making it valuable for small and large-scale business use while keeping costs low as it doesn't require expensive hardware.

On the other hand, RRT* excels in environments where path accuracy and adaptability are crucial, such as dynamic urban settings or complex indoor environments. Despite its longer computational times, RRT* offers superior performance in navigating through varying environments, as seen through its faster flight time and smaller error upon landing in three out of four maze environments. RRT* is especially more advantageous in situations where precision is the most important factor, such as dense forest, surveillance operations that require radar avoidance, and search and rescue in tight spaces. The algorithm's ability to maneuver through small areas precisely makes it a valuable tool when working with areas where it would be dangerous to send humans.

RRT* also has the advantage of handling multidimensional spaces, which can aid tremendously in applications such as 3D

mapping. While A* must map out the state space with a grid and therefore suffers from the curse of dimensionality as the number of nodes needed to be committed to memory increases drastically with the addition of new dimensions, RRT* excels when paths are not simply represented and include more non-convex areas.

B. Future Work

Although both search algorithms generated success in pathfinding, there are still limitations regarding predictive performance and UAV utilization. Utilizing different types of drones, including those suffering from over-actuated and under-actuated performance, could help with future in-depth experimentation. For example, testing with a quadrotor or VTOL tailsitter, such as the WingtraOne Gen II, which confers a more passive control design and lighter power expenditure, ensure the drone's performance would not be compromised by weaker structural integrity. Furthermore, over-actuated testing using designs with redundant fault tolerance and complex maneuvers (thus requiring more computational power) would ensure that a large power expenditure will not reduce the success of the algorithms. This testing can be done with designs similar to octocopters or hybrid VTOLs, such as the Autel Dragonfish [15]. The construction of the drone that is employed can provide another way of monitoring the success of both pathfinding algorithms and tailoring the needs of the drone to specific applications. For example, using materials such as Fiber Flax or fiberglass to construct a quadrotor chassis can increase durability and flexibility in more intense environments. Moreover, using Depron for aileron construction can increase the rotational movement of the drone, such that multiple sides must include attachments (i.e., agricultural precision).

Furthermore, the current model functions off a preplanned algorithm, where waypoints are defined so the robot can traverse through the maze. In other words, the robot's movement is controlled by itself and not the recognition of its surroundings. In the future, reducing the number of waypoints through artificial vision could be utilized in the algorithm to increase accessibility in various applications, especially in remote locations where GPS coordinates might not be properly defined. Integrating a LiDAR or IMU sensor gives the DJI Tello EDU the ability to send constant updates of linear acceleration, angular velocity, and ranges of light to the DJI Tello EDU SDK, which can increase its autonomous abilities. Using an IMU sensor, which usually consists of a 3-axis magnetometer, accelerometer, and gyroscope, the drone can analyze its three-dimensional position relative to its environment [16]. Similarly, a LiDAR could also be incorporated to improve on the drone's artificial vision. A LiDAR sensor uses remote sensing technology as pulsed lasers to measure the drone's orientation relative to the Earth. These light pulses produce a 3-dimensional image of the environment, improving the drone's ability to fly through an unfamiliar environment. Generally, the usage of drones alongside LiDAR produces the benefits of precise distance measurements, enabling the ability of precise data collection of surroundings [17]. This, once again, could enhance the autonomous ability of the drone and potentially reduce the amount of waypoints required in the algorithm.

The mazes used in this research represent a wide range of possible environments for the drone to find itself in, but it is not all-encompassing. The ladder, single solution, and random obstacle mazes can represent city, interior, and forest environments. These mazes allow for data on real-life parallels relating to everything from urban drone delivery systems to search and rescue in varying environments [18]. It is necessary to expand these situations to other types of natural habitats and various man-made structures. For example, maze generation algorithms that create paths that loop in on themselves would be useful for representing wider or more complex indoor structures. Three-dimensional mazes would also be useful in testing the storage demands of the two algorithms as the spaces get exponentially larger. Changing the maze generation and, thus, the real-world applications would also unlock possibilities for using new kinds of drones and algorithms. The ability to diversify important parameters allows autonomous navigation to be constantly improvable and applicable to all types of circumstances.

Implementing these techniques would give the drone complete autonomy, allowing it to navigate multiple environments. Furthermore, it would expand the algorithm to be utilized across various types of UAVs as the bridging commands are altered respectively.

VI. Acknowledgements

The success of this research would not have been possible without aid from various individuals. Foremost, this paper's authors would like to thank Rutgers University, Rutgers School of Engineering, and the Governor's School of New Jersey Program in Engineering and Technology for providing them with the opportunity to conduct this research, as well as Lockheed Martin for sponsoring the program. The authors of this paper would like to express gratitude to project instructor Shreya Srikanth for her guidance and mentorship throughout this research process, as well as Hugh Keenan, the group's Residential Teaching Assistant, for his monitoring and advising of group members. They would also like to thank the New Jersey Office of the Secretary of Higher Education and the Governor's School of New Jersey Program in Engineering and Technology's alumni for funding and supporting the following research. In addition, the authors appreciate the insights contributed to their research by project instructor Shreya's lab partner, Arvind Kruthiventy, and professor Dr. Laurent Burlion for his guidance and provision of essential resources from the D-141 Advanced Controls Laboratory at Rutgers University. The authors are also grateful to Vikram Setty for providing permission to use his A* and RRT* code as a foundation for the drone's algorithms. Furthermore, they acknowledge Jean Patrick Antoine, the Associate Director of NJ Governor's School of Engineering and Technology, and Dean Ilene Rosen, the Director of the NJ Governor School

of Engineering and Technology, for their tremendous support during the project. They would also like to thank DJI Tello and Visual Studio Code for providing them with the hardware and software necessary for this project.

VII. Appendix

The code utilized to create both pathfinding algorithms can be found under the following GitHub repository: https://github.com/nrgameace/GSETAutonomousDroneResearchProject.

A. A* Algorithm Code

```
def algorithm(draw, grid, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {block: float("inf") for row in grid for block in row}
    g_score[start] = 0

    f_score = {block: float("inf") for row in grid for block in row}
    f_score[start] = h(start.get_pos(), end.get_pos())

    open_set_hash = {start}

    while not open_set.empty():
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current == end:
            reconstruct_path(came_from, end, draw)
            end.make_end()
            return True

        for neighbor in current.neighbors:
            temp_g_score = g_score[current] + 1

            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + h(neighbor.get_pos(), end.get_pos())
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor], count, neighbor))
                    open_set_hash.add(neighbor)
                    neighbor.make_open()

        draw()

        if current != start:
            current.make_closed()

    return False
```

B. RRT* Algorithm Code

```
def rrt_algorithm(viz_window,start_node,goal_node,obstacle_list):
  distanceTheoretical = 0
  node_list = []
  node_list.append(start_node)
  erase_list = []
  run = True
  while run:
    new_node, node_list = add_new_node(viz_window,node_list,obstacle_list)
    if target_reached(new_node,goal_node):
      goal_node.parent = new_node
      pygame.draw.line(viz_window,BLUE,(new_node.x,new_node.y),(new_node.parent.x,new_node.parent.y))
      node_list.append(goal_node)
      display_final_path(viz_window,goal_node)
      run = False
```

REFERENCES

[1] Y. Hong, J. Fang, and Y. Tao, "Ground control station development for autonomous uav," in *Intelligent Robotics and Applications: First International Conference, ICIRA 2008 Wuhan, China, October 15-17, 2008 Proceedings, Part II 1*. Springer, 2008, pp. 36–44.

[2] Mordor Intelligence, "Uav navigation systems market - growth, share & analysis 2024," 2024. [Online]. Available: https://www.mordorintelligence.com/industry-reports/uav-navigation-systems-market

[3] DJI Store, "Official store for dji drones, gimbals and accessories (united states)," 2024. [Online]. Available: https://store.dji.com/product/tello-edu

[4] R. Tech, "Tello sdk 2.0," 2018. [Online]. Available: https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf

[5] Pygame, "pygame: A free and open source python programming language library for making multimedia applications like games built on top of the excellent sdl library," 2024. [Online]. Available: https://github.com/pygame/pygame

[6] A. Javaid, "Understanding dijkstra's algorithm," *Available at SSRN 2340905*, 2013.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Journals & Magazines*, vol. 4, no. 1, pp. 100–107, 1968. [Online]. Available: https://ieeexplore.ieee.org/document/4082128/

[8] B. M. Sathyaraj, L. C. Jain, A. Finn, and S. Drake, "Multiple uavs path planning algorithms: a comparative study," *Fuzzy Optimization and Decision Making*, vol. 7, pp. 257–267, 2008.

[9] YouTube, "Video title," Mar 2015, youTube video. [Online]. Available: https://www.youtube.com/watch?v=g024lzsknDo

[10] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic a," *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 224–243, 2016.

[11] J. Ding, Y. Zhou, X. Huang, K. Song, S. Lu, and L. Wang, "An improved rrt* algorithm for robot path planning based on path expansion heuristic sampling," *Journal of Computational Science*, vol. 67, p. 101937, 2023.

[12] I. Noreen, A. Khan, and Z. Habib, "A comparison of rrt, rrt* and rrt*-smart path planning algorithms," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 16, no. 10, p. 20, 2016.

[13] Z. H. Iram Noreen, Amna Khan, "Optimal path planning using rrt* based approaches: a survey and future directions," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 11, 2016.

[14] M. Zhou, "mmaze: A python maze generator and solver," 2023. [Online]. Available: https://github.com/MorvanZhou/mmaze

[15] M. Hassanalian and A. Abdelkefi, "Classifications, applications, and design challenges of drones: A review," *Progress in Aerospace Sciences*, vol. 91, pp. 99–131, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0376042116301348

[16] K. S. Hatamleh, O. Ma, A. Flores-Abad, and P. Xie, "Development of a special inertial measurement unit for uav applications," *Journal of Dynamic Systems, Measurement, and Control*, vol. 135, no. 1, p. 011003, 2013.

[17] K. Amer, M. Samy, M. Shaker, and M. ElHelw, "Deep convolutional neural network based autonomous drone navigation," in *Thirteenth International Conference on Machine Vision*, vol. 11605. SPIE, 2021, pp. 16–24.

[18] B. Shen, "Advances in pathfinding algorithms for games, route planning software, and automated warehouses," Ph.D. dissertation, Monash University, 2023.

[19] V. Setty, "A demonstration with visualisation/gui for robot path planning algorithms like a*, rrt, rrt*," gitHub. [Online]. Available: https://github.com/vikrams169/Autonomous-Robot-Path-Planning-Using-A-star-RRT-and-RRT-star