A Comparative Study of Maze Solving and Maze Creation Algorithms

Marie Milanowski and Evan Pochtar

Abstract

One of the most crucial problems within computing is navigating and solving difficult systems with search algorithms. These problems are presented in many different ways, including mazes, images, data searching, and numerous others. Good pathing relies on speed, efficiency, and ability to navigate complex situations and problems. In this case, procedural two-dimensional (2D) maze generation and solution is one way to test these algorithms. By analyzing different metrics, the strengths and weaknesses of each search algorithm can be derived and the best one of certain sizes and types of mazes can be determined.

1 Project Description

With the growth of Artificial Intelligence (AI), the necessity for efficient use of search algorithms has grown significantly. With more search methods available than ever, the performance and use cases of the various search algorithms is extremely important. For this project, the team decided to explore various popular search algorithms and their ability to solve maze problems. Each algorithm was explored significantly and then used to find solutions to mazes of varying sizes. The solutions and solution times of each algorithm were then compared and the team determined which algorithm(s) performed the best.

In order to test this information, the team created an application that creates a maze of infinite rows and columns (to a reasonable extent). Then, the application attempts to solve the maze with all of the algorithms being compared and analyzes their solutions and performance times. The search algorithms explored were A*, Depth First Search (DFS), Breadth First Search (BFS), Simulated Annealing, Genetic Algorithms, Best-First Search, and Uniform Search. In addition to finding an overall "best" search algorithm, the team also looked at algorithm performance differences between different maze sizes in order to determine a "best" search algorithm for different maze sizes.

The exploration of maze problems has proven to be beneficial in many instances, especially in artificial intelligence. One of the more common introductory AI problems is the robot problem with a robot in a room full of obstacles trying to find a path from point A to point B. This simple problem highlights the basic goals of AI and how developers can approach problems of this type, leading to the solution of AI problems on a much larger

scale. This is just one type of path-finding problem that benefits significantly from the analysis of maze problem solutions. The other significant application of maze problems is the invention of GPS and its ballooned popularity. With this increase of smartphone use and general public reliance on mapping services such as Garmin and more recently Google Maps, Apple Maps and Waze, the necessity of efficient maze problem solutions is more important than ever. These types of problems have been integrated into the everyday lives of a vast majority of people and their efficiency and accuracy are crucial to the success of many. Map search problems are simply more complex maze problems and knowing more about which algorithms perform the quickest and most accurately can immensely benefit the performance and continued success of these applications.

The team chose to focus on this problem for reasons very similar to the reasons that maze problem optimization is important in general. They found the versatility of maze problems interesting and benefit from those solutions in their everyday lives. The knowledge of search algorithm preference and performance is critical to many applications and provides an easy way to test them and come to a conclusion on what should be used.

2 Problem Background

2.1 Maze Generation

"Analysis of Maze Generating Algorithms"[Gab19] provided six possible algorithms to create a 2D maze with "wall" nodes and "path" nodes, as well as provided four search algorithms that solve these mazes. The authors went in depth testing each algorithm for the complexity, speed of creation, and the number of nodes created. The mazes complexity was defined by the dead ends, visited intersections, and the average steps taken to solve by four different search algorithms. The team decided to use the "Recursive Backtracking" method for the creation of the maze in this project for its relative simplicity and speed, while still being able to create complex structures.

The main takeaway from "The quest for the perfect perfect-maze"[KC15], on the other hand, was an algorithm to calculate the complexity of the maze, taking into account the max length, speed of solving, the minimum length, and the total number of nodes. Junctions, crossroads, and number of wall nodes are all important to creating the "perfect" maze. In this case, the goal was to increase the average amount of time taken, and the number of nodes searched. The team used this to evaluate the properties of the randomly generated maze that Recursive Backtracking created and took that into account when recording the results of the searches. This highlighted the differences between search algorithms in different possible scenarios, providing a more holistic view.

2.2 Greedy DFS

"A Greedy Approach in Path Selection for DFS Based Maze-map Discovery Algorithm for an autonomous robot" [MSIS12] explored the behavior of an autonomous robot in an unknown

maze problem, specifically the behavior when using Depth First Search (DFS). Multiple variants of the DFS algorithm were explored including a greedy approach and more traditional DFS approaches. The performance of each algorithm was recorded and compared in order to understand the effects that each of the modifications to the simple algorithm had on the solution to the maze. This study allowed the team to see how the different aspects of the DFS algorithm affected the solution time and results. This knowledge also helped the team understand why the performance of the DFS algorithm fell where it did in the final performance results of the study as well as what situations the algorithm performed better or worse depending on the characteristics of the problems.

2.3 The A* Search Algorithm

"A comparative study of A-star algorithms for search and rescue in perfect maze" [LG11] explored the effects of the A* algorithm by utilizing various versions of the A* algorithm to solve the same maze. The results of the various algorithms were then compared with each other as well as the DFS algorithm as a benchmark algorithm. The different A* algorithms used in the study help readers understand the effects of the algorithm and highlighted the unique characteristics that make it useful in specific situations. These comparisons helped establish the characteristics of the A* algorithm and its usefulness when solving maze problems. Additionally, the comparison to the DFS algorithm helped put into perspective the impact of A* in comparison to other algorithms.

While the team has chosen to not observe the D* Lite [KL02] algorithm in this study, D* Lite was derived from the A* algorithm in an attempt to optimize it and reduce the running of repetitive code. Because of this, the analysis of D* Lite also takes a deep dive into the properties and tendencies of A*. Exploring A* by means of another algorithm helped the team understand the impacts that different aspects of the algorithm have. This allowed them to see the impacts of keeping the repetitive code that D* Lite aims to eliminate have on the results of the algorithm. This knowledge helped explain why the A* algorithm ended up behaving the way it did, especially in comparison to the other algorithms studied.

"Incremental A*" [KL01] explored another modification of the A* algorithm, in this instance, incremental modification via the Lifelong Planning A* algorithm. Similar to the D* Lite algorithm, the modification of the A* algorithm requires significant knowledge of the traditional A* algorithm. Comparing the properties of A* to the modified Incremental A* algorithm allowed the team to better understand A* and how it behaved in the maze problem and in comparison to the other algorithms being compared. While the team didn't explicitly study the Incremental A* algorithm, the modification and reasons that the researchers chose to modify the algorithm in the way that they did allowed the team to decide in the conclusion of the project whether or not A* was the best solution algorithm and what aspects are prevented it from gaining that distinction.

2.4 Genetic Algorithms for Solving Shortest Path

In the article "Genetic Algorithms for Solving Shortest Path Problem in Maze-Type Network with Precedence Constraints"[KK19] it went into detail about how a Genetic Algorithm with many different variable settings would search a maze and try to find the shortest path. Mazes are unique in comparison to other search problems as they have rare feasible paths. The search algorithm was mostly tested on mazes in this study, which gave it the potential to perform differently than how it would on other search problems. This limited the amount of inputs into the Genetic Algorithm itself, which hadn't been extensively studied in the past. Complexity was added onto the algorithm by adding a performance function, so it was easy to tell which maze sizes and layouts worked best or worst for the Genetic Algorithm. This was crucial to the team as this function had to be implemented keeping in mind a large maze size and having very few feasible paths to the end.

2.5 Simulated Annealing

In the journal article "Simulated Annealing"[BT93] it went into detail describing the method and implementation in pseudo code of the Simulated Annealing algorithm. Although this section of the total journal was short, it described both how fast the algorithm converged, and how it dealt with larger areas. The initial temperature state, and the slow decrease of it, was outlined in many different scenarios, and a most efficient number was eventually found where the algorithm works at it's best. This information was crucial to the team as the mazes created were of an indeterminate size. The journal also mentioned how the algorithm should and can be implemented in realistic situations, and how to measure the success. The algorithm itself was modified to search through a maze and function correctly under the correct conditions.

2.6 A Comparative Study of Maze Solving

The paper "A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory"[SDF+10] described the differences between popular searching algorithms, and how they interacted with the limitations of a maze in particular. Uniquely, they mentioned some search algorithms such as hugging left wall or hugging right wall, that did not appear in many other maze search specific papers. They also represented these algorithms in flow charts of how the functions interacted with their surroundings, helping readers understand the algorithms without having to understand pseudo code. Every test was done clearly, with different types of mazes and different complexities, making it clear where different algorithms shined compared to others. They created these comparisons using statistics like total nodes searched and time to solve maze, which are critical to choosing a correct search.

2.7 Background Information Summary

Comparing the various methods of both maze creation and maze solving made evident to the team that, above all, performance capabilities vary widely between maze sizes, complexity, and the goal of the maze itself. Although some of the methods compared such as A* have a better performance on average, certain algorithms can start to become more efficient, especially as the mazes grow larger. Through various performance functions and complexity scores of mazes and algorithms alike, a final conclusion was then reached on which search algorithm performs best depending on the given mazes.

3 Proposed Solution and Approach

In order to analyze the performance of different search algorithms, the team built an application that generates a two dimensional maze of "infinite" rows and columns (to a reasonable extent) and then attempts to solve the generated maze(s) with each of the search algorithms being studied. The team then measured the solutions generated in terms of number of cells searched, number of cells in the final path, and the runtime of each algorithm and compared each metric. In addition to finding a "best" algorithm ranking for each metric, the metrics were also combined to create an overall best algorithm ranking based on all three metrics. Using multiple rankings allowed for an algorithm to be selected off of specific factors for specific problems, allowing for greater efficiency in niche situations.

The team also chose to generate mazes of different dimensions and compared the performance of the various algorithms on different size mazes to determine if some algorithms performed better on smaller mazes as opposed to larger mazes and vice versa. The application was built to take in the maze dimensions before calculating the runtime of each algorithm, so it was overall easier to include this portion of the study as the maze size was modifiable in the application built. This aspect of the project also allowed for a more specific algorithm selection for specific situations.

The team chose to build their own maze generation application in order to accommodate the modification of the maze size and to better fit their needs specific to this study. This allowed the team to run the study multiple times with various sizes of mazes for simple comparison. The application built by the team takes in the desired maze dimensions and from there, generates a maze of the desired dimensions and then solves the maze with each of the algorithms being observed. The solution path of each algorithm is then printed and a graph of the runtime of each algorithm is displayed in a separate window. Additionally, there are many versions and modifications of the algorithms studied, so building the application from scratch allowed the team to specify which version of each algorithm to use and to ensured continuity when running the application multiple times with the same size maze. In this study, the most basic version of each algorithm was used in order to understand the base behaviors of each algorithm and to better identify the reasoning for each runtime without a significant number of other factors affecting the runtimes that would have to be accounted for when comparing the different algorithms.

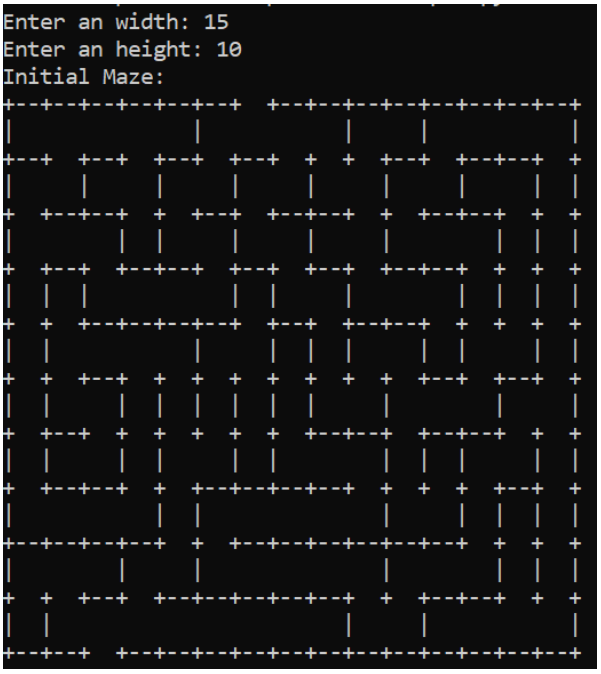
4 Experiment Design

In order to explore and solve this problem, the team utilized Python. This language was chosen because it allows for the use of packages such as numpy and matplotlib for easy integration. Python is also the most commonly used AI language, so it allowed the team to replicate the conditions where these algorithms would most likely be used. Matplotlib was used to graph the results of the search algorithms, providing a final ranking for each. Numpy, on the other hand, was used to track the amount of cells searched, the length of the final paths, and the total time the algorithm took to solve the maze. Other imports and modules used were, random, timeit, math, heapq, and copy.

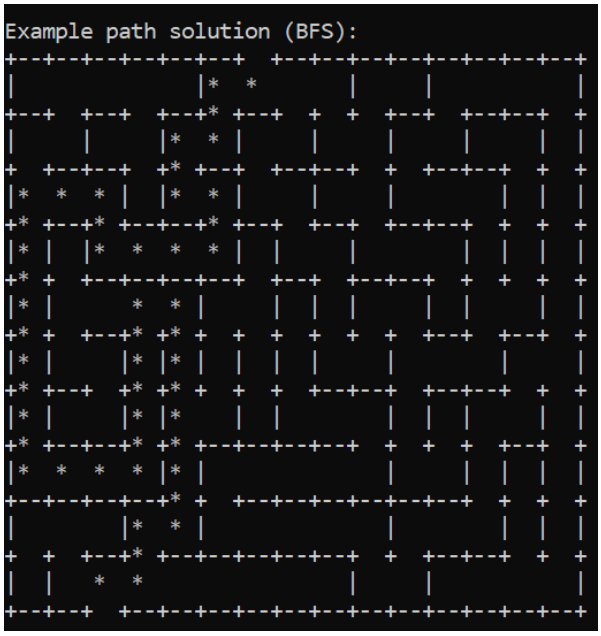
The maze creation algorithm was created using a random walk (RW) agent[Gab19]. Using a mix of horizontal walls defined as "+–" and vertical walls defined as "|", the maze is outputted to a computer terminal to a user specified height and width. The random walk algorithm is then given a random range of the height, and a random range of the width. Stepping through each node one by one in a random direction each time, it creates a wall between where the algorithm currently is, and its surrounding nodes. The algorithm guarantees any node on the maze to be able to reach any other point on the maze, with no isolated spaces or areas due to how the algorithm travels through the puzzle. This allowed for the placement of a random start node at the top of the maze and a random end node at the bottom of the maze without any risk of the maze being unsolvable. The start and end nodes are represented with openings in an otherwise closed off maze, which can be seen in Figure 1a. When the search algorithms are run, the maze is then displayed with the asterisk symbol, "*", along the final path returned by the search as shown in Figure 1b. This allowed the team to inspect the final outputs, and determine the differences between them.

BFS, DFS, A*, Best-First Search and Uniform Search have been some of the most used path finding algorithms. These were implemented in a similar fashion to the usual execution of these algorithms, adapted to work with the specific random walk maze generation and metric tracking being used in this study. The A* search algorithm and the Best-First Search algorithm both used the Manhattan heuristic to efficiently score nodes. The Simulated Annealing[BT93] and Genetic Algorithms[KK19], on the other hand, are not normally implemented to solve 2D mazes. This is because, unlike the popular algorithms, they are not guaranteed to return the shortest path, as seen in Figure 2a. The way these algorithms were adapted to this specific problem was by using the randomness to travel in decreasingly random paths, until finally narrowing on a path to the end. The temperature and cooling rate for Simulated Annealing was manually set per maze height and width. To verify that the temperature and cooling rate did not cause any issues before the final ranking, the algorithm was run 50 times, and after some adjustment to make it more efficient, was run an additional 50 times. If the algorithm failed to reach the end destination, the temperature and cooling rate were considered invalid. Generally, in this problem, the temperature had to be very high, as sometimes the algorithm would get stuck on sides of the maze the shortest path would never visit, and therefore quickly drop temperature without making progress. For the Genetic Algorithm, the fitness function was defined by the Manhattan heuristic. Similar to Simulated Annealing, the generations and initial population of the algorithm were adjusted

Figure 1: Maze Examples



(a) An example of a inital 15x10 maze.



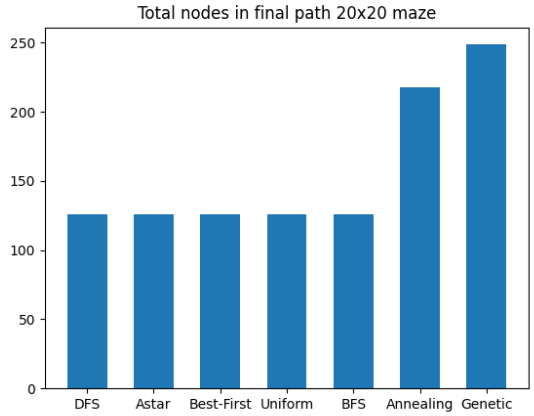
(b) The same maze with the BFS path algorithm overlayed onto it.

manually, using retesting to validate its efficiency and function. Other variables within the algorithm were static such as the mutation rate which was placed at a five percent with 80 percent crossover rate and a 20 percent leveling threshold. Additionally, the top ten percent of chromosomes with the highest fitness were carried over to the next generation using elitism. This allowed the team to derive the best results from both regularly used path finding algorithms, and some algorithms that are not regularly used.

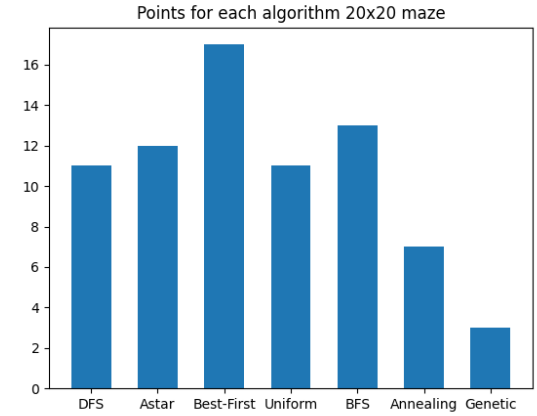
When the results were derived from searching the maze, it then was interpreted by a ranking system. For a specific height and width of a maze, ten different mazes were generated, and all seven search algorithms were run on each of those mazes. This was done so an algorithm did not get "lucky" and get given a randomly generated maze that it could search very quickly, providing a more fair experiment. For each algorithm, time taken to solve, nodes in final path, and nodes searched were tracked. For each iteration of the maze creation, the numbers were added together, and then divided by ten to get a final average number. Afterwards, using a simple numeric ranking system, they were placed from one to seven. The best algorithm for time and nodes searched was awarded seven points, and the points awarded incrementally decreased until the seventh place algorithm was awarded one point. For nodes in the final path, because most of the algorithms are guaranteed to find the shortest path, the computation was different. Those with the shortest path were awarded three points, the second shortest path was given two, and the third was given one. Then, the algorithms were finally sorted numerically to give a final ranking, and outputted to a

bar graph with both the algorithm name and the final point total, as shown in Figure 2b.

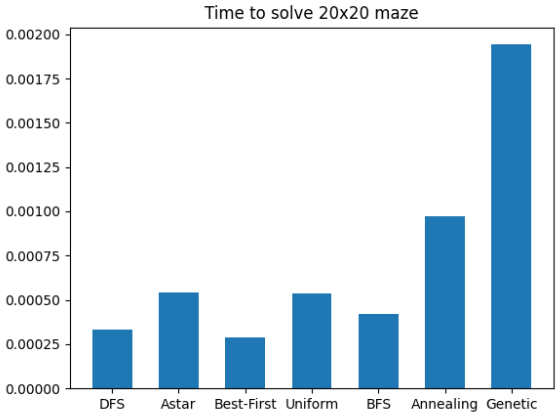
Figure 2: Graph Examples



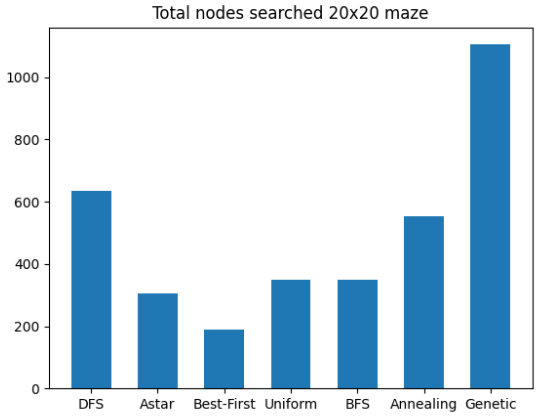
(a) Total moves to reach the end



(b) Final point total



(c) Total time to solve maze



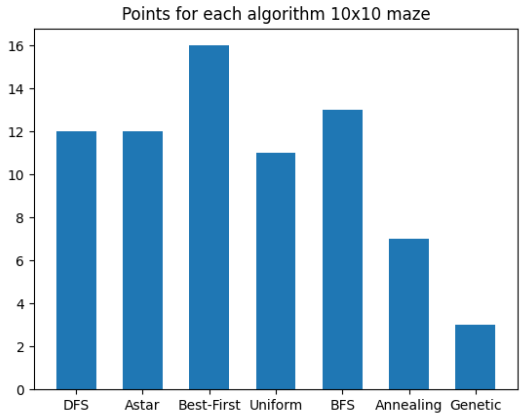
(d) Total nodes searched before end

5 Result Analysis

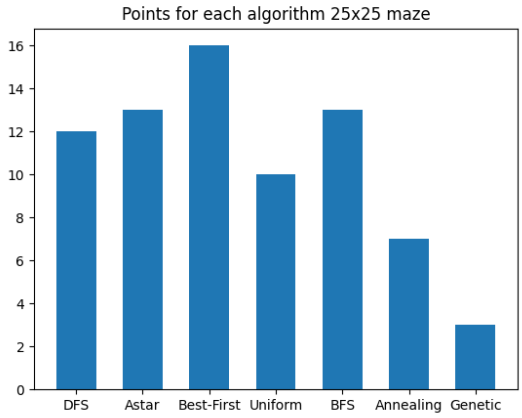
The application was run with mazes of size 10 x 10, 25 x 25, 50 x 50, and 100 x 100 and the point totals for each maze were then calculated and graphed for comparison. The resulting graphs can be found in Figure 3 for each maze size.

In all four maze sizes analyzed, the Best-First Search algorithm performed the best, regardless of maze complexity. In all instances, the different algorithms had a similar ranking in point totals. Best-First Search performed the best in all four cases with BFS being in second. A* had the same number of points as BFS in the 25 x 25 maze as seen in Figure

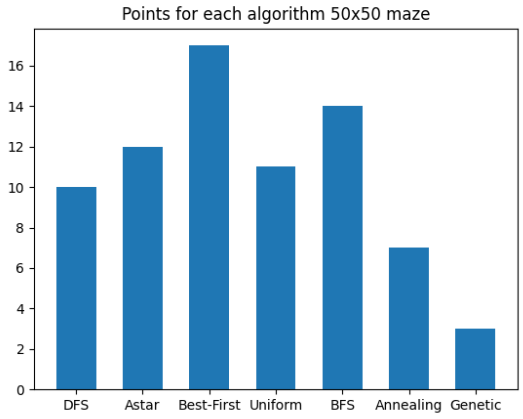
Figure 3: Maze Test Results



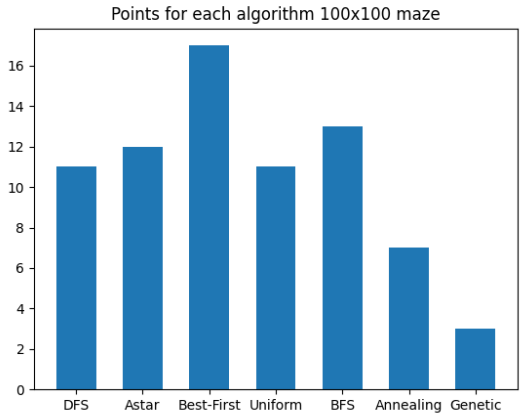
(a) Point results for a 10x10 maze



(b) Point results for a 25x25 maze



(c) Point results for a 50x50 maze



(d) Point results for a 100x100 maze

3b, but in the other three cases, A* was the next best *behind* BFS. DFS and the Uniform Search algorithm were the next two with Simulated Annealing and the Genetic Algorithm performing the worst. All four maze sizes had the same order of total point rankings, so the initial hypothesis that the various algorithms would perform differently depending on maze size was not correct. Instead, for mazes of any reasonable size, Best-First Search has the best performance regardless of dimensions.

The other major difference that was noted in the study results was the discrepancy between the A* and Best-First Search algorithms. This is because the two algorithms used the same heuristic in this study, so it was expected that the two would have closer point totals and be the top two performers. Instead, A* placed third consistently behind BFS and had point totals closer to the DFS algorithm than the Best-First Search algorithm. While no concrete explanation was found for this discrepancy, it could have been due to the idea

that BFS is better suited for this type of maze generation and maze problems in general. BFS and DFS also perform worse with large size problems, but since this study was only performed on reasonably sized mazes, BFS performed better than expected. DFS also had higher point totals than expected, and this was likely due to the same idea. Additionally, the graph in Figure 2d shows that DFS searched significantly more nodes than all other algorithms, other than the Genetic Algorithm, which was consistent across all maze sizes. This could also explain why DFS was consistently outperformed by A*.

Simulated Annealing and the Genetic Algorithm being in the last two places in all four maze types is the outcome that the team expected. Both search algorithms work best with complex data, and are not usually suited for path finding when the search space is smaller. Due to the randomness involved in both, the final search paths ended up being longer than the guaranteed shortest path returned by the five other algorithms. This also caused them to take longer to solve and the search space to be larger. This resulted in the two algorithms consistently placing last in every category, leading to their bottom positions in Figure 3. Some important things to note with these algorithms is that they are very variable depending on the static inputs. For Simulated Annealing this could be the temperature and cooling rate and for Genetic Algorithms this could be mutation rate or any of the other randomness factors included. Simulated Annealing consistently placed higher than the Genetic Algorithm due to Genetic having to run every generation completely, forcing the algorithm to consistently take much longer then Annealing. Their overall performance could have been improved with continuous testing and adjusting, and possibly moved up in the point totals.

6 Future Work

There are many ways to expand this idea and improve its functionality and purpose of use. One way would be to simply add more search algorithms. Using niche algorithms, such as Trémaux's algorithm or the pledge algorithm, might be better at certain maze layouts then others. Another possible route of expansion could be to add more metrics the search algorithms are tested on, such as amount of turns. Amount of turns could be important to the real world, because if these search algorithms were used in a physical scenario with a robot, reducing the amount of turns could be crucial. Testing at different maze sizes, creation of a 3D version of this experiment or a real life recreation using computer vision could be invaluable to deciding an efficient and "perfect" maze searching algorithm.

Additionally, as mentioned in the Result Analysis section, the order of the top four performers was consistent, but not what was expected by the team. Therefore, further modification could be done to the algorithms in order to understand why the top four algorithms performed as they did and whether that was an anomaly or the intended outcome.

7 Conclusion

Maze problems have numerous uses both within and outside of artificial intelligence including many applications in day to day life. The efficiency of the search algorithms used to solve these problems is vital to the performance of anything that depends on path-finding problems. In order to further analyze some of the most popular search algorithms and their performance on maze problems, seven popular search algorithms were tested and compared on generated mazes of different dimensions in order to determine the best search algorithm. After measuring and comparing the performance of the algorithms being studied, it was determined the regardless of problem size, Best-First Search is the best overall algorithm for solving mazes of this generation method. Best-First Search consistently outperformed the other six methods and had the best metrics in almost all of the individual metrics as well as overall. In order to maximize the efficiency of maze problems and get the most optimal solutions, Best-First Search should be utilized in any maze problems of reasonable size.

References

[BT93] Dimitris Bertsimas and John Tsitsiklis. Simulated Annealing. *Statistical Science*, 8(1):10 – 15, 1993.

[Gab19] Peter Gabrovšek. Analysis of maze generating algorithms. *IPSI Transactions on Internet Research*, 15(1):23–30, 2019.

[KC15] Paul Hyunjin Kim and Roger Crawfis. The quest for the perfect perfect-maze. In *2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES)*, pages 65–72, 2015.

[KK19] JunWoo Kim and Soo Kyun Kim. Genetic algorithms for solving shortest path problem in maze-type network with precedence constraints. *Wireless Personal Communications*, 105:427–442, 2019.

[KL01] Sven Koenig and Maxim Likhachev. Incremental A*. Technical report, In Proceedings of the Neural Information Processing Systems, 2001.

[KL02] Sven Koenig and Maxim Likhachev. D* lite. In *Eighteenth National Conference on Artificial Intelligence*, 2002.

[LG11] Xiang Liu and Daoxiong Gong. A comparative study of a-star algorithms for search and rescue in perfect maze. In *2011 International Conference on Electric Information and Control Engineering*, pages 24–27, 2011.

[MSIS12] Md. Sazzad Mahmud, Ujjal Sarker, Md. Monirul Islam, and Hasan Sarwar. A greedy approach in path selection for dfs based maze-map discovery algorithm for an autonomous robot. In *2012 15th International Conference on Computer and Information Technology (ICCIT)*, pages 546–550, 2012.

[SDF+10] Adil M.J. Sadik, Maruf A. Dhali, Hasib M.A.B. Farid, Tafhim U. Rashid, and A. Syeed. A comprehensive and comparative study of maze-solving techniques by implementing graph theory. In *2010 International Conference on Artificial Intelligence and Computational Intelligence*, volume 1, pages 52–56, 2010.