Parameterized Maze Generation Algorithm for Specific Difficulty Maze Generation

Liam Peachey
ljpeach18@earlham.edu
Department of Computer Science
Earlham College
Richmond, Indiana, United States

ABSTRACT

Mazes as a problem domain have wide applications, from art and games to testing decision algorithms. While there has been focus on the topology of mazes, recent work on generating specific levels of difficulty of maze is more scant. Additionally, existing models for analyzing a maze's difficulty may not accurately reflect difficulty for both humans and computers, with some mazes classified as more difficult appearing easier to a human solving the maze. This paper explores a three piece method for generating mazes of specific difficulty using a neural network, parameterized maze generation algorithm, and difficulty model.

KEYWORDS

neural networks, maze generation, graph analysis
.

1 INTRODUCTION

Mazes are a well known and easily understood problem domain in AI research. They have been produced throughout history for purposes of entertainment, defense, and decoration. The structure of a maze impacts the ease of which a human or algorithmic agent may navigate the maze. Either a human or an AI could easily solve a maze that is just a straight line to the goal, but it might be easier for a human to spot overarching patterns within a maze than for an algorithm. How would such a maze be classified? Models exist by which the difficulty of a maze may be estimated, but these have primarily been aimed at algorithmic agents [8]. Bellot et al.'s approach integrates McClendon's difficulty analysis, and produces a measure of how fun a maze is, integrating some of the ways humans scan mazes [1].
Additionally, existing research on maze generation is limited to finding algorithms that generate more complex mazes, or mazes with specific topology, such as image mazes [3, 9, 13]. There currently are no unified systems for generating mazes of specified difficulty using only one algorithm. This research produced an algorithm that generates mazes of a desired resultant difficulty to guide generation. The process uses a neural network to transform a difficulty rating and maze dimensions into a series of parameters that will affect how the maze is generated. The neural network will be trained using varying difficulty models, with the resulting mazes being tested using Gabrovšek's method for measuring the difficulty of a maze through the use of agents to determine which difficulty models produce the most accurate ratings [3]. The difficulty models used are McClendon's difficulty model, and Bellot et al.'s model[1, 8]. This research aims to introduce new factors to include in difficulty analysis as well.
This paper provides a detailed background review in section 2, design and implementation details in section 3, experimental results in section 4, conclusions in section 5, and a review of future work in section 6.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CS488 Senior Capsone, , Earlham College
© 2022 Association for Computing Machinery.

2 BACKGROUND

This background section will cover the primary sections and foundations for this project. The first section covers approaches for how mazes are typically generated. The second section covers existing models and techniques for defining the difficulty of a maze. The final section covers ways that machine learning has been used to assist in maze generation.

2.1 Maze Generation

Mazes at their cores are cell graphs. The edges of the graph can be traversable, or not. Those that are not are considered walls. In a typical maze, cells consist of four edges, forming a grid. However, this representation can be bent to create other structures as well [7]. For the purposes of this research, I will be focusing on the standard four neighbor cell graph. Perfect mazes are defined as mazes that have a singular path connecting any two cells [1]. While not all mazes are required to have this attribute, it does allow for easy maze generation through the use of spanning tree generating algorithms [5]. As a result, maze generation typically makes use of existing algorithms for generating these spanning trees.
As mazes are well known, there are already a number of algorithms to generate mazes. The Recursive Backtracking method chooses a cell on the graph, marks it visited, adds it to a stack, is marked as visited, and then chooses a random neighbor that has not been visited to join with. That neighbor becomes the next active cell, and the process repeats until there are no unvisited neighbors around the active cell. At this point, the current cell is abandoned, and a new cell is popped from the stack. Once the stack is empty, the algorithm halts [11]. This method is essentially a depth first traversal.
Kruskal's algorithm works by separating the graph into sets and then joining them back together one cell at a time. When two cells are joined, they become part of the same set. Cells are only joined

with cells in different sets. Once the whole graph is part of the same set, the algorithm completes [11].
Prim's algorithm is similar to the recursive backtracking algorithm. It also makes use of a frontier to connect new cells. Instead of popping the next cell off the stack however, it randomly chooses a cell from within the frontier, and it then adds that cell to the "visited" or "in" group, and adds unvisited neighbors to the frontier [11]. This is the approach that this research will be building off of.

2.2 Difficulty

Bellot et al. outlines existing methods for assessing the difficulty of mazes, and produces their own model for calculating how fun a maze is [1]. McClendon's difficulty ratings, which Bellot et al. used and refined in their model, is based on the complexity of the solution path multiplied by the complexity of all branches in the maze [8]. Bellot et al.'s approach takes into account the number of what they refer to as "non-significant" walls [1]. According to them, mazes with a low number of non significant walls and the highest difficulty as rated by the McClendon model are the most fun.
Finally, Gabrovšek uses a number of agents to explore a maze, using their performance to determine which algorithms produce the most difficulty mazes [3]. Bellot et al. also produced two hybrid algorithms that they used for generating mazes that they determined to be more fun [1]. However, this and many other works failed to create systems of generating mazes of a specified difficulty.

2.3 Maze Generation and Machine Learning

Susanto et al. produced a genetic algorithm for generating a type of maze [12]. Their mazes are not the usual cell graph type, and are specific to the game they developed. They encoded patterns into genes that would then be used within the genetic algorithm [12]. Their method also seems to be suited to generating higher complexity levels. The goal of our algorithm is to produce mazes of a specific complexity without long training cycles at run time, so a genetic algorithm like this one would not be a good fit.

3 DESIGN AND IMPLEMENTATION

This project aims to gauge the effectiveness of the parameter set chosen for maze generation, and to determine whether it would be feasible to use machine learning techniques to convert a difficulty value into said parameters.
The overall framework of the project is outlined in Figure 1. Input difficulty is the desired difficulty of the maze. This is passed to the maze generator neural network (MGNN) which will return a set of parameters for the main algorithm in an attempt to produce a maze that matches. The generated maze is then passed to the difficulty model, which analyzes the maze difficulty. The MGNN then uses this value and the goal difficulty to grade, and proceeds to the next generation. This describes the learning loop. After this is completed, the algorithm may be used to generate a pool of mazes for agent functions to complete. The performance of these functions is then compared with the desired difficulty of the maze, as well as the difficulty model's assessment of the maze, and compiled as performance data.
The difficulty model will factor in the work of Bellot et al. to create measure of how fun a maze is, as well as re-examining McClendon's difficulty formula, which is already factored into Bellot et al.'s formula, to look for additional factors by which to determine a maze's theoretical difficulty [1, 8]. Distinct MGNN's have been trained on the difficulty models used to assess the usefulness of each model. Additionally, I will be using Gabrovšek's method of employing maze solving agents to determine maze difficulty and complexity [3]. I will be adding additional agents however, including heuristic searches, such as A* search, greedy, etc. for more accurate measures as to how a computer may solve a maze as well as how to push the model further.
If the data shows the approach is feasible, a neural network will eventually use difficulty models as feedback. Parameters include which cells will be selected for expansion, directional biases, and maximum ratio of certain types of intersection (outlined by Kim and Crawis) to the maze's size [6], and solution length. Other factors will likely be added or removed during the course of the project. The core of the algorithm will be based on recursive backtracking for maze generation [11]. Recursive backtracking makes use of a frontier built through cell discovery, which should allow for more control over determining how the maze will grow.
This work will be evaluated by comparing the desired difficulty rating of mazes generated, actual difficulty ratings as based on the models used, and the performance of the agent functions used later on. We will test multiple difficulty models for training the MGNN, and analyze all of them to see which works best with the recursive backtracking system we develop.

3.1 Neural Network

The idea for this section is to train a neural network to be able to generate the set of inputs for the maze generator given a desired difficulty and dimensions for the maze. This could likely be accomplished via a Generative Adversarial Neural Network given a large enough supply of pre-generated sets of mazes to use as a training set. The setup for this section is difficult as this problem demands what is essentially a reverse classification network.

3.2 Maze Generation

3.2.1 *Parameters.*

The goal for the maze generator was to find areas within generation to try to exert greater control over, while still maintaining the maze's randomness. The primary principle behind this maze generation algorithm is to repeatedly take nodes that are within the maze, and connect a random neighboring node until all nodes are within the maze. "within the maze" meaning there is a way to access the node from the starting point of the maze generation. This means that aside from the maze's dimensions, we can alter how we select the node to expand from (see section 3.2.4), and which neighbor is selected (see section 3.2.2). Additionally, the McClendon and Bellot difficulty models both stress that solution path length is an important feature of a maze, so the ranking of solution length can also be used as a parameter[1, 8]. Finally, we can use starting nodes within the maze to influence growth.
Starting nodes are given as collection of ordered coordinate pairs. They correspond to the positions of the nodes that will be initially

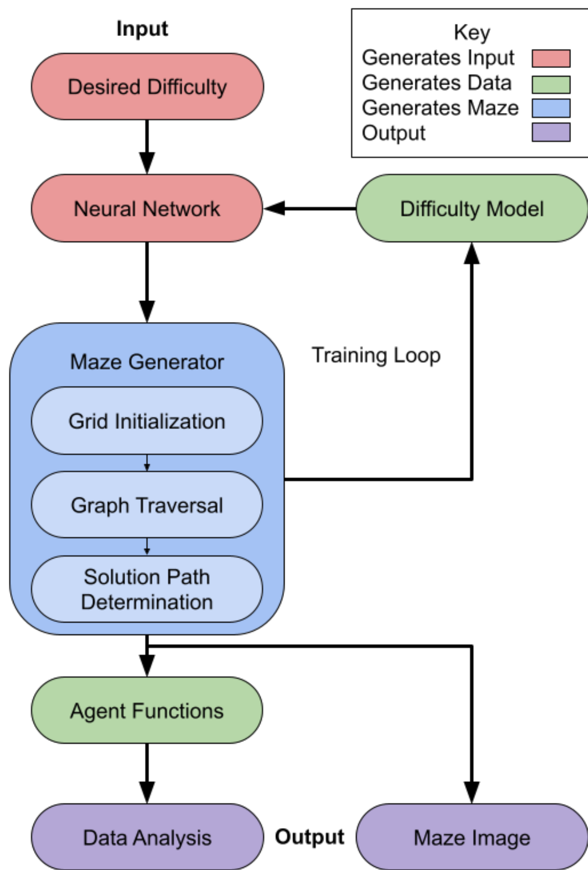

Figure 1: Architecture Diagram

placed within the frontier list as described in section 3.2.4.
Existing maze algorithms include Depth First, Breadth First, and Random traversals [11]. Depth first selects the newest nodes added to the maze first, breadth first selects the oldest, and random traversals choose randomly. This algorithm takes this idea, and is implemented using a list rather than a stack/heap so that any "expandable" node may be selected. Nodes that have been expanded from may be re-expanded from, so they must be placed back into the pool along with new nodes. This creates an opportunity for three parameters: old node insertion point, new node insertion point, and the node selection point. To allow for random selection, each point is represented as a ratio from zero to one, where zero is the start of the list and one is the end of the list. A range of 0-1 for all parameters results in a random traversal behavior. 0-0 for node insertion and 1-1 for node selection results in BFS behavior. 0-0 for all results in DFS behavior.
To affect what neighbor is selected, I used directional bias. McClendon and Bellot both make use of path complexity, defined as the number of turns on a given path, so including a way to affect how curvy a path is should allow for greater control over complexity, and therefore difficulty[1, 8]. This bias is defined using three weights, 0-1 each, representing a selection weight for left, right, and straight. These directions are relative to the initial entry into the node. If a node is added to the maze through a path from the bottom, straight maps to the top of the node, left to the left of the node, etc. If it is entered from the right, then straight maps to the left of the node, and left maps to the bottom, etc.
Solution path length is dictated by how long a given path is compared to all others. The parameter is given as a number 0-1, corresponding to the general ranking of the desired solution path to others. 0 is the longest solution path, 1 is the shortest.

3.2.2 *Nodes.*

Nodes describe all possible positions within the maze. Pointers to neighbors can be considered hallways.Essentially, if a node is accessible from a given node, there is a hallway connecting them. If not, there is a wall between them, or they're just not adjacent. For the purposes of this research, all hallways are limited to adjacent nodes. Nodes store a set, which describes whether they're inside or outside of the maze (value is 0), as well as which starting node they belong to. Nodes also store their coordinate to allow for easy identification, access, as well as to assist heuristic algorithms in the evaluation phase (see section 3.6).

3.2.3 *Graph.*

The maze as a whole is constructed through the graph formed from the above interconnected nodes. The first stage in the maze generation process is constructing a full grid of nodes. There are a few ways to do this: Maintaining a 2D array of nodes, where nodes only point to their neighbor if there is a hallway connecting them, and maintaining only a pointer to a starting node, where all nodes within the graph maintain pointers to their neighbors, and have additional variables which determine whether the connection is a wall or a hall. Both methods have their advantages and disadvantages. The first method has quick access for any given node coordinate, but requires maintaining a separate array in addition to the maze nodes. The second method has O(length+width) time complexity for accessing a random node, but only requires space to store the nodes. During the actual traversal of the maze, fast access is not particularly valuable as traversal may be handled through the nodes pointers to their neighbors. This means that on large mazes, space savings become more valuable than access time savings. This choice does not make an impact on the results of the research, but choosing the node only approach also allows for non-grid mazes.

3.2.4 *Pre-Traversal.*

Before traversing the maze graph, starting points for the maze are assigned distinct set values. As the maze is constructed by expanding into spaces that are not part of the maze, starting nodes form islands of nodes that are inaccessible to each other. By assigning each starting node a set, and having each new node added to that set's island, once the entire graph has been traversed, we can go through and join each set, removing islands and constructing a single perfect maze (see section 3.2.6).

3.2.5 *Frontier and Traversal.*

Traversal is accomplished by popping nodes with open edges that are part of the maze from the frontier, expanding them, and pushing both the new addition and old node if it has remaining open edges back into the frontier. Once the frontier is empty, the entire maze

has been traversed. For a node selected for expansion, an available neighbor to create a hall to by considering the directional bias parameter. The bias parameters are used to create a weighted random number, which corresponds to the desired neighbor to expand into. The frontier refers to the collection of nodes from which the maze can be expanded. These nodes are all within the maze, although they may not be within the same starting node "set." There is also the possibility that a node can become a dead end while it is still in the frontier. This is handled by simply passing by dead ends and popping a new value. As the algorithm has been parameterized to include a random range for pushing old and new nodes, and pop position, the frontier needs to have fast insertion and removal operations, while maintaining insertion order. I accomplished this by creating a linked list with a pointer to the node at the middle of each range of values (old, new, pop). This reduces access time by a small amount while maintaining a fast insertion and removal speeds.

3.2.6 *Set Joining.*

Once the graph has been traversed and all nodes are within a set corresponding to a starting node, the sets must be joined to form a complete maze. The algorithm chooses a starting set and finds all nodes that border a different set. One of those nodes is randomly selected, and random neighboring cell of a different set is chosen. The set that node belongs to is added to a set of sets that are considered "within the final maze" so that it can be ignored for all further perimeter checks. Nodes that only border nodes of a sets within the final maze are removed from the list of perimeter nodes. The new set of nodes is traversed again, adding all additional perimeter nodes to the list. This process continues until there are no perimeter nodes, meaning that all sets are within the final maze.

3.2.7 *Solution Path Determination.*

Solution paths are limited to nodes along the edge of the graph. This is not necessary, but it's helpful for human readability. The process for deciding the start and end node pair requires finding the length of the path between all pairs of perimeter nodes. As mazes generated by this algorithm are considered perfect mazes, there is only one path between any two given nodes, so we do not have to consider shortest or longest paths. [1]. We quickly determine the solution path by converting the maze into a tree, where each node of the tree stores its depth, and a reference to its parent. We maintain a list of tree node positions for all perimeter maze nodes. The algorithm for determining the path length between perimeter pairs is as follows:

```
while the tree nodes are not the same:
    if either tree node has a depth of zero:
        Add both depths to the path length
        break
    if the tree nodes have different depths:
        move to the tree node's parent
        add 1 to the path length
    if the tree nodes have the same depth:
        move both nodes to their parent
        add 2 to the path length
return the path length.
```

These path lengths are used as keys while the node pair is stored as an item. The keys are sorted, and then node pair with a key ranked at the spot indicated by the solution rank parameter is returned as the start and end points for the maze.

3.3 Difficulty Models

3.3.1 *McClendon Difficulty.*

This difficulty model is used as the foundation for difficulty analysis during the project. The model is represented by the equation:

$$\delta(M) = log\left[\gamma(T)\prod_{i=1}^{n}\gamma(B_i) + 1\right]$$

Where δ represents the difficulty of a given maze, γ represents the complexity of a given path, M represents the maze, T represents the solution path, and B represents a branching path[8]. B is contained within the set of n paths in the maze that are not part of the solution path. The complexity of a path is defined as:

$$\gamma(B) = D(B)\sum_{i=1}^{n}\frac{\theta(w_i)}{d(w_i) * \pi}$$

Where θ is the change in direction for a given position in the maze measured in radians, w_i is a position within the maze, D is the length of the given path and d is the length of a given position in the maze.[8] Since our maze generator only generates rectangular mazes, there are parts of this equation that can be simplified [1]. All turns within the maze are 90°, so if there is ever a turn, it can be measured as $\frac{\pi}{2}$, simplifying our function.

$$\gamma(B) = D(B)\sum_{i=1}^{n}\frac{1}{2d(w_i)}$$

To calculate the difficulty, we find the solution path, tracking its length, and marking all intersections it contains as the start of branching paths, including the start and end points of the solution if either of them are not dead ends. We traverse the solution path, counting how far between each turn, using that value as our $d(w_i)$ value. Once the traversal has completed, we multiply by the length of the solution. As all complexity results for the solution path, and branching paths, are multiplied together within a logarithm, we can immediately calculate $log\,(\gamma(T))$, and set aside. Next, we perform a similar set of operations for each branching path. However, as these paths branch, we consider the start and end points of any given path to be marked with an intersection and an intersection, or an intersection and a dead end. If we hit another dead end, we mark that intersection just as we did for the solution path, and move on. Once we finish exploring a branch, we take the logarithm of its complexity value, and add it to a sum. We take the logarithm early in order to avoid massive numbers that can overflow quickly.
While this method works well for categorizing mazes, it has a short coming for the set up of the maze generator within this project. Primarily, for mazes with extremely short solution paths, such as when the start and end points in a maze are adjacent, due to considering all branching paths of the maze, the maze's difficulty remains large, despite not being difficult to solve for a human, or machine.

3.3.2 Bellot et al. Model.

This model uses the McClendon difficulty model as a base, and considers walls to be an additional important factor in solving a maze. Specifically, the Bellot et al. Funness model is interested in how the number of insignificant walls interacts with difficulty. Non significant walls are walls that are removed in their "non-significant wall removal" process [1]. Non significant walls are walls that, when viewed within a 2x2 grid of maze nodes, are not connected to any other wall. The process for removing non significant walls is as follows: mark where, for all 2x2 sets of adjacent nodes, there are more than 2 walls. Next, find all non-significant walls, and delete them. Find all non-significant walls, and delete them again so long as they are not marked. Continue this process until there are no more non-significant walls. Count the number of removed walls. This value is then divided by the McClendon difficulty for the maze, and this becomes the funness rating for the maze.

3.4 Maze Testing Via Agents

While human testing is difficult to organize and harder to get ethical clearance for, testing via AI is quick and easy. As such, we chose to use a series of agent functions to solve the mazes our algorithms produced, analyzing performance to see how well each agent does for each maze iteration. This approach is also based on Gabrovšek's previous work [3]. Each technique used for this evaluation traverses through the maze using node expansion (a given position being considered a node, which can be expanded to generate the different resultant moves from a given position) so we can use the metrics "nodes expanded" and "nodes generated" to see how much work each agent had to do. Nodes expanded refers to how many individual nodes we directly examine and generate children from, while nodes generated refers to how many nodes we generate total. We can also use time as a metric to measure efficiency, but our primary concern is the performance of the maze, not the efficiency of the algorithm, so this metric will receive less focus. One metric that I believe would require a deeper look would be the number of dead ends the agent hits. This would offer insights to how easy it was for the agent function to find the correct path in addition to how many nodes it took to fill the correct path out completely. Additionally, as a property of our mazes being that they are "perfect" mazes, any path from start to end that one of the agent functions finds will be the optimal solution. This means we can focus solely on the efficiency of each of our agents rather than the fit of the solution they find.

3.4.1 A* Heuristic Search.

The A* Heuristic search is a commonly used search that takes into account both cost to reach a given position, as well as a heuristic value which estimates the cost to reach the goal [4]. A* is commonly used for solving path finding domain problems, and grid like mazes lend themselves well to the use of the Manhattan Distance heuristic model [2].

3.4.2 Greedy Best First Heuristic Search.

Greedy Best First Heuristic Search is another commonly used search agent that resembles A*. However, unlike A*, Greedy does not take solution depth into account, only the estimate of how far until the goal state [10]. This can lead to faster solutions that frequently do not produce optimal solutions. In this case, optimality is not a concern, so greedy provides a slightly different informed search to gauge the mazes off of.

3.4.3 Depth First Search.

Depth first search has been conceptually explored earlier in this paper, but the main idea is to explore as far down as one can before backtracking to an earlier position to explore further. This agent function essentially picks a path and sticks with it as far as it can before it's forced to choose a different one, exploring all the newest nodes first. It uses a stack as its foundation which can be helpful in visualizing how it functions.

3.4.4 Breadth First Search.

This search has also been conceptually explored, but the basic idea for this one is to explore all the oldest nodes first. It uses a queue as its foundation, which can be helpful in visualizing how it functions.

3.5 Evaluation

The maze generation process will be evaluated through the use of the difficulty models covered earlier as well as through the use of agent functions. This servers the purpose of providing more data that can be used to corroborate our observations, while also evaluating the fitness of the difficulty models used. It could be that a certain model of maze generation produces mazes that using a certain model may appear highly difficult where any agent function could solve it in two node expansions consistently. Each set of parameter inputs for a given maze will be tested 1000 times to gain a more accurate view of the typical maze of that setup. The data should tell us whether there's consistency and promise in the idea of using the parameters chosen for use in our algorithm to affect the difficulty of the maze. If the data supports the assertion that the parameters do hold a promising effect on the difficulty, then we can proceed to work with formulating a neural network or other machine learning technique for processing desired difficulty inputs.

4 RESULTS

The data shows that the parameters for maze generation can have a strong effect on the difficulty of a maze. For example, in Table 5, the Center starting point BFS Straight bias maze has a very low difficulty, while the Center starting point BFS right bias maze had a far higher difficulty (still below average for a 25x25 maze). Results show that DFS node selection tends to produce more difficult mazes by the McClendon model. DFS in these tests randomly selected a node from the first 10% of nodes within the active node frontier for expansion. Interestingly, the difficulty models seem to not reflect short solution paths in their final difficulty. This can most clearly be seen in Table **??** where the difficulties for the mazes rise the shorter the solution path becomes, while the amount of work any of the agent functions performs drastically decreases. As the models both use a multiplicative approach when combining the complexity of the solution path and the complexity of the branching paths, even if the solution path is extremely small, if there are many branching paths, the difficulty will be large.

In addition to performance testing, the mazes produced by different parameter sets have distinct visual topography. Using the two corners of the maze as the starting points produces a clear line down

the middle of the maze where the two "sets" (see section 3.2.6) of joined nodes meet. This effect can be seen in figures 2 and 3. This is more of an aesthetic distinction than a functional one however as there is still only one path from any two nodes. The agent function's performance also did not change significantly when paired with varying starting points (Table 1), although interestingly there was a jump in difficulty at 3 starting points. Directional bias also create interesting effects on the mazes. Figure 4 and 5 take on the appearance of mirrored windmills. Figure 6 forms swirling eddy-like passages. Figures 7 and 8 are similar, but the directional bias shows here as well. Understanding that longer curve-less passages lead to lower complexities, we can reasonably conclude that directional bias can play a role in difficulty determination. As seen in Table 5 and in Figure 2, certain configurations can create incredibly low difficulties as well.

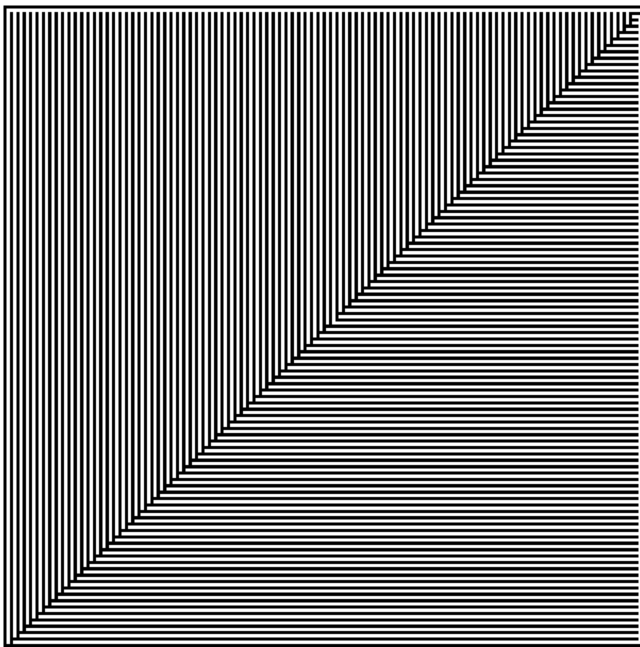

Figure 2: Corner Starting Points, BFS, Straight Directional Bias

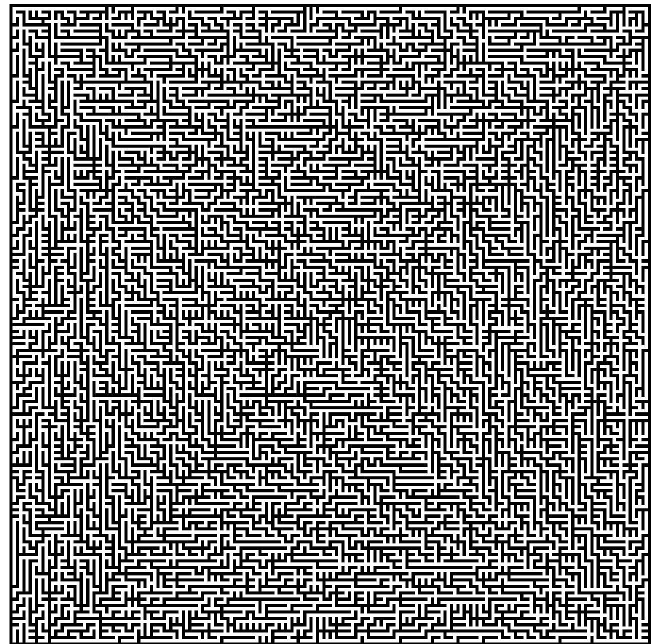

Figure 3: Corner Starting Points, Random Expansion, Right Directional Bias

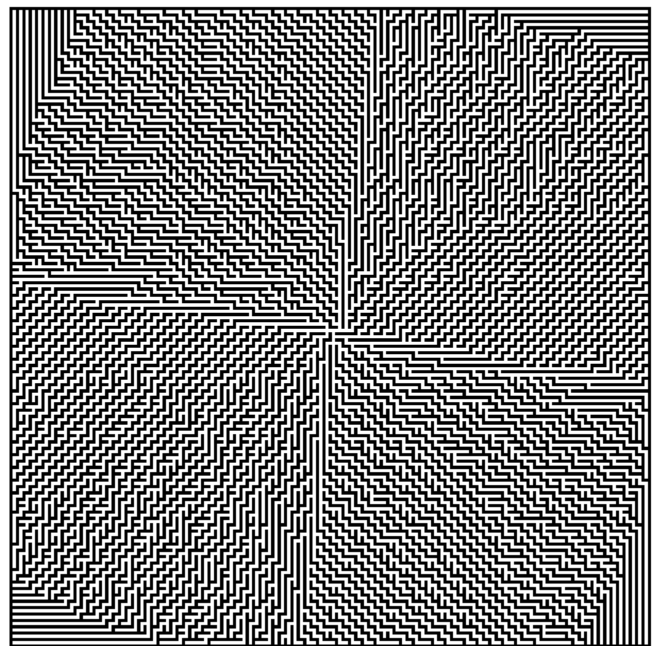

Figure 4: Center Starting Points, BFS, Right Directional Bias

4.1 Tables

5 CONCLUSION

Maze generation using the parameter set used within this research successfully influenced the resultant difficulty of the generated mazes. Certain input groupings can be used for more stark effects, such as the straight directional biases, and some inputs when used in combination can create drastically different results, such as BFS and straight directional biases. We conclude that given this success, a neural network or other machine learning technique should be able to formulate a model by which they are able to convert a difficulty into the set of inputs. Additionally, while the existing difficulty models are useful, there are holes in their techniques that lead to mazes with short solution paths being ranked at the same level or higher than mazes with longer solution paths, which the agent function testing shows does not accurately reflect the true difficulty of the maze.

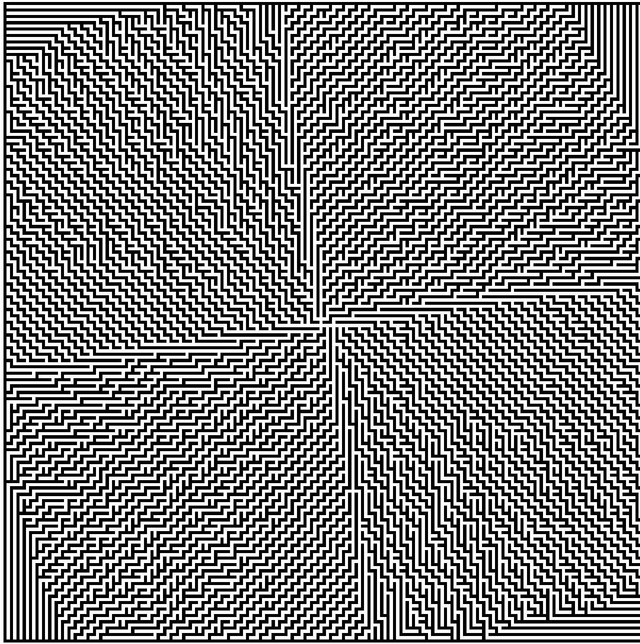

Figure 5: Center Starting Points, BFS, left Directional Bias

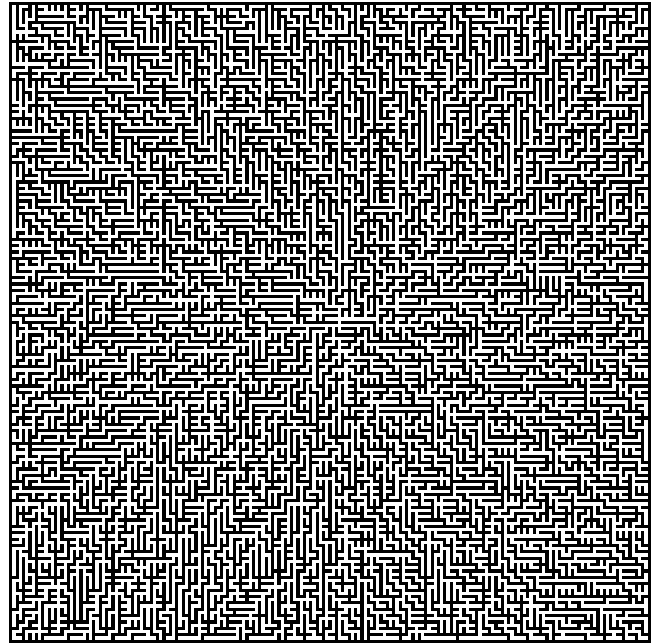

Figure 7: Center Starting Points, RND, left Directional Bias

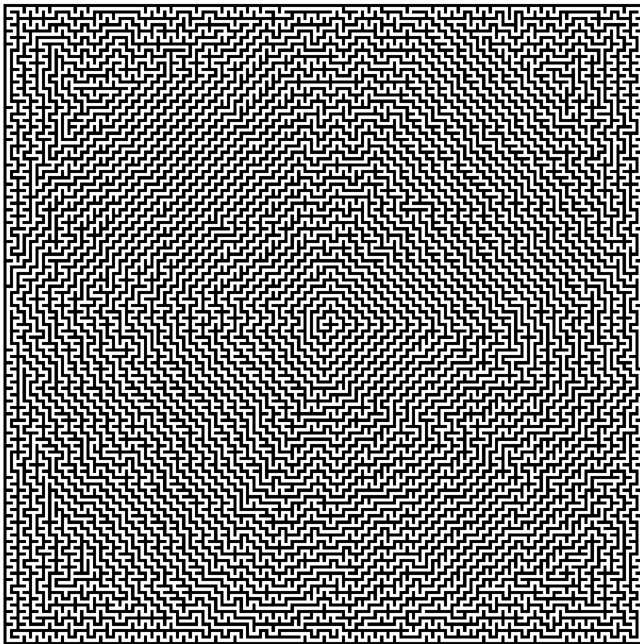

Figure 6: Center Starting Points, DFS, left Directional Bias

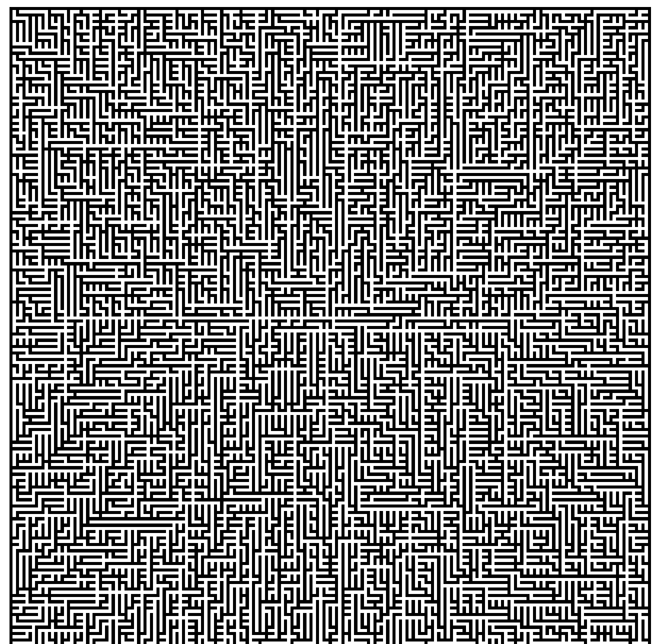

Figure 8: Center Starting Points, RND, strt Directional Bias

5.1 Future Work

The most pressing future work is to implement the neural network into the project and begin training. Other future work includes implementing intersection ratios as a parameter, expanding the algorithm to include imperfect mazes, attempting to add additional parameters to maze generation, changing the overall shape of the maze from a grid to any arbitrary bounds, adding additional neural network integration within the generation algorithm for more dynamic generation, adding additional dimensions to the maze (including three dimensional visualization), allowing nodes to connect to non-adjacent nodes, and implementing additional difficulty models and agent functions.

Table 1:
Starting Points Testing Results

Maze setup	1 strt pnt	2 strt pnts	3 strt pnts	4 strt pnts	5 strt pnts
mcclendon	132.2385	129.7770	138.6990	137.0579	136.6011
bellot	2.591534	2.643539	2.555488	2.587667	2.605675
BFS E	623	615	620	620	619
BFS G	624	618	623	623	622
DFS E	366	438	393	391	379
DFS G	391	460	414	414	403
A* E	490	529	471	507	501
A* G	506	541	492	523	514
Greedy E	429	415	327	369	366
Greedy G	449	437	354	396	392

Table 2:
Expansion Order Testing Results

Maze setup	Rnd Expansion	BFS Expansion	DFS Expansion
mcclendon	132.638474	83.659698	152.577087
bellot	2.580449	3.711855	2.188952
BFS E	623	624	621
BFS G	624	625	623
DFS E	363	374	361
DFS G	388	390	390
A* E	487	513	455
A* G	504	520	474
Greedy E	432	515	339
Greedy G	452	523	365

Table 3:
Turn Bias Testing Results

Maze setup	Single Turn Only	Straight Only	Turn Only	No Bias
mcclendon	137.876236	119.555397	144.205704	132.426651
bellot	2.451557	2.984574	2.304719	2.587526
BFS E	623	623	623	623
BFS G	624	624	624	624
DFS E	362	368	364	360
DFS G	387	394	388	385
A* E	487	488	491	494
A* G	504	509	506	510
Greedy E	433	425	426	432
Greedy G	453	448	446	453

6 ACKNOWLEDGEMENTS

Thank you to David Barbella for his time, patience, feedback and willingness to help throughout the course of producing a project idea and proposal. Thank you to Charlie Peck for insight, motivation, and structure. Thank you to Sofia Lemons for all the advising, inspiration and support. Thank you to Igor Minevich for the data structure assistance.

Table 4:
Solution Ranking Testing Results

Maze setup	Sol Percentile 0	Sol Percentile 0.2	Sol Percentile 0.4
mcclendon	132.457642	138.961395	140.932816
bellot	2.58565	2.464546	2.432711
BFS E	623	519	443
BFS G	624	530	455
DFS E	363	417	398
DFS G	388	432	410
A* E	494	295	223
A* G	511	315	243
Greedy E	433	249	187
Greedy G	453	272	208

Maze setup	Sol Percentile 0.6	Sol Percentile 0.8	Sol Percentile 1
mcclendon	143.297333	146.051056	144.079926
bellot	2.387638	2.347678	2.941714
BFS E	355	162	1
BFS G	369	176	4
DFS E	394	361	294
DFS G	404	368	296
A* E	136	51	1
A* G	153	60	3
Greedy E	120	39	1
Greedy G	138	49	3

Table 5:
Misc Setups Testing Results

Maze setup	Corner BFS Straight bias	Corner DFS Left bias	Corner DFS Straight bias
mcclendon	21.521925	158.517014	138.543655
bellot	26.201122	2.110628	2.402767
BFS E	623	622	620
BFS G	624	624	622
DFS E	453	382	367
DFS G	464	408	395
A* E	501	488	459
A* G	508	503	480
Greedy E	345	336	339
Greedy G	361	364	367

Maze setup	Center BFS Straight bias	Center BFS Right bias	Center DFS Rigth Bias
mcclendon	38.979633	105.4664256	154.735107
bellot	12.642587	4.052686	2.155141
BFS E	624	624	615
BFS G	625	625	617
DFS E	213	322	359
DFS G	228	332	389
A* E	211	199	499
A* G	233	213	515
Greedy E	144	183	355
Greedy G	164	196	383

Maze setup	Center DFS Straight Bias
mcclendon	135.89505
bellot	2.463364
BFS E	616
BFS G	619
DFS E	356
DFS G	385
A* E	474
A* G	494
Greedy E	345
Greedy G	374

REFERENCES

[1] Victor Bellot, Maxime Cautrès, Jean-Marie Favreau, Milan Gonzalez-Thauvin, Pascal Lafourcade, Kergann Le Cornec, Bastien Mosnier, and Samuel Rivière-Wekstein. 2021. How to generate perfect mazes? *Information Sciences* 572 (2021), 444–459.
[2] Susan Craw. 2017. *Manhattan Distance.* Springer US, Boston, MA, 790–791. https://doi.org/10.1007/978-1-4899-7687-1_511
[3] Peter Gabrovšek. 2019. Analysis of maze generating algorithms. *IPSI Transactions on Internet Research* 15, 1 (2019), 23–30.
[4] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.
[5] Paul Hyunjin Kim. 2019. *Intelligent maze generation.* The Ohio State University.
[6] Paul Hyunjin Kim, Jacob Grove, Skylar Wurster, and Roger Crawfis. 2019. Design-centric maze generation. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*. 1–9.
[7] Xue Li and David Mount. 2016. Spherical Maze Generation. (2016).
[8] Michael Scott McClendon et al. 2001. The complexity and difficulty of a maze. In *Bridges: Mathematical Connections in Art, Music, and Science*. Citeseer, 213–222.

[9] Yuichi Nagata, Akinori Imamiya, and Norihiko Ono. 2020. A genetic algorithm for the picture maze generation problem. *Computers & Operations Research* 115 (2020), 104860.

[10] Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach* (3 ed.). Prentice Hall.

[11] Ms Shivani H Shah, Ms Jagruti M Mohite, AG Musale, and JL Borade. 2017. Survey Paper on Maze Generation Algorithms for Puzzle Solving Games. *International Journal of Scientific & Engineering Research* 8, 2 (2017), 1064–1067.

[12] Evan Kusuma Susanto, Rifqi Fachruddin, Muhammad Ihsan Diputra, Darlis Herumurti, and Andhik Ampuh Yunanto. 2020. Maze Generation Based on Difficulty using Genetic Algorithm with Gene Pool. In *2020 International Seminar on Application for Technology of Information and Communication (iSemantic)*. IEEE, 554–559.

[13] Jie Xu and Craig S Kaplan. 2007. Image-guided maze construction. In *ACM SIGGRAPH 2007 papers*. 29–es.