Maze Generation Algorithms Using Graphs

Advisor: Prof. Dr. Tınaz Ekim

Group Members:

Berat Kubilay Güngez - 2021402087

Didem Uslan - 2021402027

Ebrar Baştan - 2020402078

Selahattin Eyüp Gülçimen - 2020402066

Abstract

This study examines the structural features of different mazes and their solving difficulties. It models the mazes as graphs and implements various generation algorithms. It utilizes both existing algorithms from the literature alongside the ones that are modified based on some strategic approaches. The fixed-size mazes are evaluated based on a set of topological metrics. Maze solving performances are analyzed through various techniques. These techniques consist of the existing maze-solving algorithms, a newly introduced hybrid search method and also a reinforcement learning based approach. Experimental results indicate that the maze complexity does not solely depend on one factor. In this study, it is observed that algorithms like Recursive Backtracker and Randomized DFS generate mazes with longer paths and fewer intersections, whereas Prim algorithm variants generate shorter and densely branched mazes. By conducting a principal component analysis, it is revealed that the most distinguishing property across different mazes is the average dead-end length from the solution path. In addition, solver performance across different maze types is compared for various metrics regarding the solution process and the most effective agents are identified for each maze type. Furthermore, mazes with loops are examined and solver performance analysis is conducted separately for these cases.

Keywords: Mazes, Maze Generation Algorithms, Maze Solving Algorithms, Graph Based Models, Looped Mazes, Maze Complexity, Difficulty of a Maze, Maze Metrics, Principal Component Analysis, Reinforcement Learning, Solver Performance

Table of Contents

1. Introduction

Mazes have been an interest for human beings throughout history. They have appeared in different forms, built with different materials [1]. They serve various roles. For instance, it is said that they had served as spiritual guides for pilgrims in Ancient Egypt [2]. However, these early designs, known as “labyrinths”, were not as complex as we come across today, having a single path. Labyrinths also became a part of architecture and art. Then, they have evolved into more complex structures, to mazes, as a source of entertainment in some cultures [3]. Today, we see maze-like structures in topology, architecture, video games, and many more.

The complexity of a maze is a controversial and somewhat subjective matter. This makes mazes an interesting topic for mathematical and computational analysis. Maze generation and solving methods also take an important place in literature, and various algorithms have been developed for both generation and solution.

This study begins by asking the question: “What defines the difficulty of a maze?”, then progresses toward the goal of generating mazes with different levels of difficulty. The key point is to model the mazes as graphs and to benefit from the characteristics of graphs for the discussions and analysis performed.

1.1. Objective of the Project

The primary objective of this project is to generate mazes with varying difficulty levels using various maze generation algorithms and to examine how different structural properties influence their complexity.

The difficulty of a maze is dependent on the solving approach, therefore, this study also aims to evaluate the generated mazes using various solving agents. The study tries to identify the maze characteristics that contribute most significantly to the difficulty measure from the perspective of each solver by observing the performance for each maze type.

1.2. Scope

This study includes the generation of mazes using selected maze generation algorithms from the literature, and also developing new ones. For the solving techniques, both existing

algorithms and new approaches will be introduced. The study also covers the implementation of these algorithms. The graph background of algorithms and the graph representation of the mazes help the study be built in a structured way.

The scope includes defining measurable indicators of difficulty, such as the number of visited nodes during the solving process. Emphasis is placed on computational analysis and objective evaluation of difficulty using algorithmic agents in a controlled environment.

The study covers only 2D and planar mazes generated using grids. Eight existing generation algorithms and five modified algorithms will be implemented. Some of the algorithms generate mazes with loops. Five maze-solving algorithms from the literature and two more solving agents, where one of them is a modified algorithm and the other is a learning based approach, will be performed.

To provide a clear understanding of the workflow, a context diagram with the project's objectives and related tasks has been constructed. This diagram can be found in Appendix 1. Context diagram illustrates the whole process. Starting with the question, “what makes a maze difficult?” since our main objective is to generate mazes with varying levels of difficulties. Different generation algorithms, some already existing and some modified during the project, are used for maze generation purposes. Pre-defined and calculated metrics of such mazes are then kept track of as maze analysis. On the other hand different solver algorithms are implemented on these mazes. Then solution results are analyzed accordingly. Final analysis of maze metric results and solver results gave conclusions about maze difficulty which helped us generate mazes with varying difficulty according to different generation and solving algorithm combinations.

1.3. Limitations

Despite the comprehensive scope of the study, some limitations should be acknowledged. In this study, only two-dimensional grid-based mazes are analyzed. Differently shaped or three-dimensional mazes are not considered. For each algorithm 100 samples of mazes are generated. The study is restricted to fixed-size mazes (25x25 for classical algorithms and 10x10 for Deep Q-Learning experiments). The mazes include a single starting and a single exit node, both located in the same place for all mazes.

This study is purely computational and theoretical and does not include human participant studies on their performance in solving mazes. Therefore, the study does not analyze human problem-solving behavior. It does not reflect cognitive limitations.

Additionally, due to computational constraints, the learning-based approach (DQN solver) could not be applied to a large sample with large-sized mazes.

It should also be noted that not all algorithms are suitable for all solving agents. For instance, the Hand On Wall algorithm is excluded from the set of solving agents performed for mazes with loops, since the algorithm may get stuck in a loop and fail to find the solution.

Lastly, in this study, only the Randomized Prim algorithm is modified to have several variants. Other than Randomized Prim and Hunt and Kill algorithms, no algorithm is extended to generate loops.

1.4. Literature Review

Fauzan Hilmi discusses how to find a solution path of a maze and also focuses on the generation of mazes in his study named "Implementation of Prim's and Kruskal's Algorithms on Maze Generation". He implements spanning-tree-based methods. The maze-graph mapping is encountered here. He emphasizes that there are some differences between Prim's and Kruskal's algorithms even though they generate mazes with similar backgrounds. He keeps some measures related to these mazes. One of them is the approximate percentage of dead ends cells upon the total cells. He observes that Prim's algorithm generates more dead-ends. He also examines the generation time and the solution path length. He observes that Prim's algorithm is more complex, and Kruskal's algorithm is faster in generation and concludes with selecting Kruskal's algorithm as a better option [4].

In Analysis of Maze Generation Algorithms, Gabrovsek aims to rank different maze generation algorithms based on the difficulty levels of mazes. He conducts this study using 6 generation algorithms and 4 solving approaches. He considers 3 maze properties to analyze maze difficulty: maze size, number of dead-ends, and number of intersections. He then introduces measures related to solver performance: number of steps, visited intersections, and visited dead-ends. He experimentally analyze and obtain ranking among mazes for each solver performance metric. He averages the rankings to come up with the overall rankings

among mazes. The algorithms he chose for this study are Recursive Backtracking, Hunt and Kill, Kruskal, Prim, Aldous-Broder; Random Walk, DFS, BFS, and Heuristic DFS. His study excludes mazes with cyclic structures. He observes that the Aldous-Broder and Wilson Algorithms are the ones on top in the final rankings. This means the agents solve them not so easily. Their tendency to generate UST's is considered to be the main reason. On the other hand, Hunt and Kill and Recursive Backtracking takes places at the bottom. They are considered to be solved in an easier way because of the similar behavior of the solving agents [5].

Kurokawa utilizes the maze-graph mapping too. He generates mazes using Eulerian graph characteristics. In the method he introduces, the solution path goes along with the line of a closed one stroke drawn curve which has two Eulerian circuits. These two circuits have identical vertices and edges but they construct different circuits. Two distinct edge cycle sets can be bi-partitioned according to adjacency between the regions surrounded by cycles. Also, each bi-partitioned cycle set can be converted to a cycle being a solution path of the maze [6].

McClendon aimed to quantify the maze difficulty and presents a method to achieve it. He uses various complexity measures and brings them together to come up with overall complexity. He discusses the maze complexity providing various formulas and calculate a reasonable upper bound for the measures [7].

In his thesis, Foltin touches upon human problem-solving, emphasizing mazes and maze-solving. He explores human interaction with mazes, and utilizes graph theory-related generation techniques. He mentions the differences in algorithmic generation processes and delves into relevant topics in graph theory. He concludes his work by analyzing human behavior and problem-solving performance [8].

1.5. Structure of the Report

This report contains seven main sections. Following the introduction, Section 2 provides essential background information. It also includes maze definitions, graph-based representations, and relevant metrics used to evaluate maze complexity and solver performance.

Section 3 introduces maze generation algorithms, both existing and modified ones while Section 4 covers the solving approaches. A new solving algorithm is introduced alongside existing ones and a reinforcement learning-based approach is conveyed in detail.

Section 5 focuses on the technical implementation. This section includes explanations on programming tools, data structures, and visualization methods. Section 6 reports experimental setups, results, and comparative analysis. It delves into the relation between maze characteristics and solver performance. Finally, the last section concludes the study with a summary of findings. It suggests the potential extensions of the work. In the appendices, additional code snippets, outputs, and figures are provided.

2. Background

2.1. Maze Concepts and Definitions

To approach the maze generation and evaluation process more systematically, it is essential to make clear definitions of key maze components. This section provides the terminology and conceptual framework which later analyses are going to be built upon.

A maze is a complex network of paths, generally designed as a puzzle to be solved by finding a path from the start point to the end. It differs from a labyrinth, which usually has a single path with no branches to the center. In this project, a maze is represented on a 2D grid structure, where each cell represents a possible location and each wall represents a boundary between adjacent cells. A cell can be open (accessible) or blocked by a wall (inaccessible). The structure of the maze is determined by the arrangement of these walls and open paths.

A maze can be represented as a graph where each accessible cell is a node and each path between neighboring cells is an edge. This representation which will be covered in the next section enables the use of graph algorithms for generation, analysis, and solving of mazes.

Key concepts in maze design that will be useful later on include:

- **Perfect Maze**: A maze with no loops and exactly one unique path between any two cells. A maximally acyclic spanning tree in graph terms.
- **Imperfect Maze**: A maze that includes loops, meaning multiple paths may exist between points. Not a tree in graph terms.
- **Dead End**: A path that terminates without reaching the solution.
- **Intersection** : A cell connected to more than two paths, creating decision points among different paths during traversal.
- **Corridor**: A sequence of cells with no branching.
- **Loop**: A cycle in the maze, allowing alternative routes between points.
- **Solution Path**: The path from the start point to the end point of the maze. In perfect mazes, this path is unique. In imperfect mazes, there might be multiple of such paths.
- **Node (Vertex):** In the graph representation of a maze, each cell in grid structure is a node.

- **Edge**: In the graph representation of a maze, a direct path between two adjacent nodes is an edge. If two cells are connected, there is no wall between them, an edge exists between their corresponding nodes.

Each maze has a start point and an end point. For this project, the start is fixed at the top-left corner, and the end is at the bottom-right corner. Maze complexity is often tied to the discussed maze concepts, such as dead ends, intersections, path lengths, and whether or not loops are present.

2.2. Graph Representation of Mazes

First of all, before representing mazes as graphs, recalling some assumptions made in Section 1.3, Limitations, might be helpful for understanding. Firstly, mazes can be in any shape, such as a square, circle, rabbit, house, etc. In this study, square mazes will be used for analysis because they are easier to implement. Secondly, in our square mazes, there will be one starting point on the upper left corner and one finishing point on the bottom right corner. Lastly, in the main analysis, mazes will contain no loops and multiple solutions. There will be one unique solution. As a special case, mazes with loops will be analyzed specifically.

The question of this section is how we can represent a maze as a graph. A square maze contains paths and walls. On the graph representation side, the square structure can be implemented with a grid having vertices in every intersection. If there is a connection between vertices, there will be a path in the maze. If there is no connection, then there will be a wall. In a maze with white and black areas in which white areas represent the ones in which one can move, in the graph side, white areas correspond to vertices that are connected with their edges, while black areas correspond to possible edges that are not used while creating the maze. The starting point can be seen as the root, while every end is a leaf. Since there is no loop in the main structure and it is a fully connected structure, these mazes are spanning trees in the graph representation. In addition, since there is a unique solution, these are called perfect mazes in the literature. [9]

To gain a deeper understanding of the described structure, one can refer to the visualized example provided. In Figure 1, a 6×6 maze is represented in graph format, and to its right, in Figure 2, the corresponding maze layout is shown.

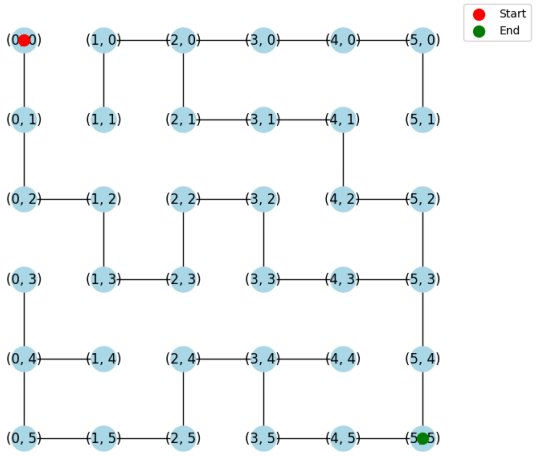


Figure 1: Graph Representation of a 6×6 Maze

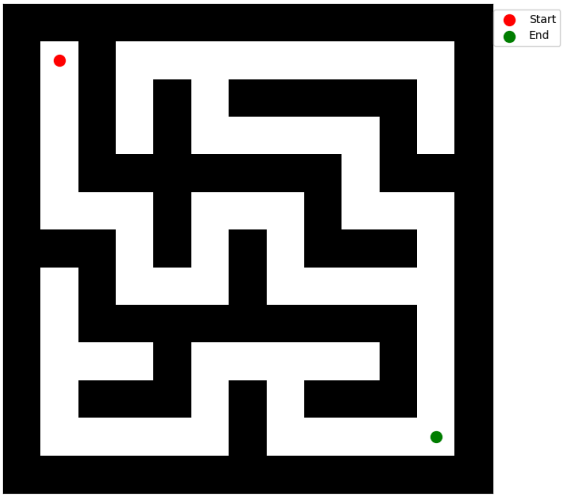


Figure 2: A 6x6 Maze

2.3. Maze Metrics

To evaluate and compare the topology and complexity of mazes generated by different algorithms, a set of quantitative metrics was defined and applied. These metrics are grounded both in previous literature and in intuitive insights into how the structural properties of a maze affect its solving difficulty. Each metric captures a different aspect of the maze's design, providing a broad perspective for analysis.

Below is an overview of the explored metrics:

- **Number of Intersections**: An intersection is defined as any cell connected to three or more neighboring paths. These represent decision points among different paths.
- **Number of Dead Ends**: A dead end is a cell with only one accessible neighbor, offering no further forward movement. These often increase the potential for missteps and backtracking.
- **Solution Length**: Refers to the number of steps in the shortest valid path from the start to the end point of the maze.
- **Dead-End Crossroads**: Counts the number of dead ends that branch directly off the correct solution path. These are roads that branch off from the solution path and ends up at dead ends eventually.

- **Tempting Count**: Denotes the number of intersections located directly on the solution path. These points can mislead solvers into exploring incorrect paths.
- **Solution Path Tortuosity**: Measures how deviated the solution is by taking the ratio of the actual solution path length to the Manhattan distance between the start and end points which is the minimum possible solution. A tortuosity value greater than 1 indicates deviation from a straight-line path.
- **Average Distance to Dead Ends (from Solution Path)**: Calculates the average distance from the solution path to the dead ends that diverge from it.
- **Average Distance to Dead Ends (from Intersection Point)**: Measures the average distance from each dead end back to its nearest intersection.
- **Average Distance to Dead Ends (from Start Point)**: Determines the average distance from the maze's starting point to each dead end.
- **Dead-End Tortuosity**: Applies the concept of tortuosity to dead ends, measuring how indirect each dead-end path is relative to its minimum possible length as Manhattan distance. This captures the extent to which these paths can mislead solvers spatially.
- **The Number of Turns in Solution:** Calculates how many times the direction changes along the shortest solution path of a maze from the start point to the end point.

These metrics were applied consistently across both perfect (loop-free) and imperfect (with loops) mazes. For mazes with loops, shortest paths were used in all solution path -related calculations to maintain comparability.

Additionally, it is important to note that some metrics particularly those measuring average dead-end lengths from different reference points were found to be highly correlated. To avoid redundancy and improve clarity in interpretation, such overlapping metrics were excluded from the final analysis.

Altogether, this set of metrics offers a structured and comprehensive framework for analyzing the complexity and topology of mazes generated by various algorithms.

2.4. Solver Statistics

Every maze-solving agent has different working principles, which makes their performance vary across distinct maze types. To analyze the performances and make a comparison in both ways, one is comparing solver performances for each maze generation algorithm, while the

other is comparing the “difficulty” of different maze generation algorithms for each solving agent, some metrics and statistics will be used.

Below is an overview of the explored metrics regarding the performance and behavior of solving agents:

- **Number of Visited Nodes:** Refers to the total number of vertices visited by the agent during maze solving. This metric also corresponds to the number of iterations, as the agent moves one vertex per iteration. Time is intentionally not recorded, as it may introduce inconsistencies due to hardware or computational differences.
- **Solution Path Length:** Denotes the length of the shortest valid path from the start to the end of the maze. There is one unique solution in perfect mazes.
- **Excess Steps:** Calculated by subtracting the solution path length from the number of visited nodes. This metric reflects the extra exploration performed by the agent and serves as a more reliable indicator of maze difficulty when comparing mazes of the same size.
- **Exploration Ratio:** Defined as the ratio of excess steps to the solution path length. Although this metric provides insight into the relationship between exploration and path efficiency, it may show mazes with shorter solution paths are more difficult. Therefore, it is not used directly in comparative evaluations.
- **Intersection Coverage:** Represents the proportion of intersections encountered by the agent during the solving process, calculated as the number of intersections visited divided by the total number of intersections in the maze. This metric helps assess how much of the maze’s decision space was explored.
- **Dead-End Coverage:** Measures the proportion of dead ends encountered during maze solving. It is calculated as the number of dead ends visited divided by the total number of dead ends. Like intersection coverage, this metric aims to evaluate the extent to which the agent explores misleading paths.

In addition to traditional solving agents, Q-learning is employed as a reinforcement learning-based approach. Due to its unique learning dynamics, additional metrics are defined:

- **Converged Epoch Number:** Indicates the number of episodes required for the Q-learning agent to learn the optimal path through the maze.

- **Epoch per Dead End:** Calculated by dividing the converged epoch number by the total number of dead ends. This provides insight into how dead ends impact the learning process.
- **Epoch per Average Dead-End Distance:** Represents the ratio of the converged epoch number to the average Manhattan distance from dead ends to the solution path. This metric evaluates the effect of dead ends on learning performance.
- **Epoch per Average Dead-End Reward:** Measures how the average penalty associated with dead ends affects convergence, by dividing the converged epoch number by the average dead-end reward.
- **Epoch per Solution Reward:** Assesses the influence of the final goal (solution reward) on the convergence behavior of the agent.

In summary, the statistics that will be kept for the maze solving side are visited nodes, solution path length, excess step, exploration ratio, intersection coverage, and lastly, dead end coverage. Also, Q-learning has its own defined statistics, which are primarily based on the converged epoch number and reward & penalty system.

3. Maze Generation Algorithms

Maze generation algorithms with different backgrounds are selected through a literature review. Randomized Prim, Randomized Kruskal, Wilson, Sidewinder, Aldous-Broder, Hunt and Kill, Recursive Backtracker, and Randomized Depth-First-Search algorithms are selected, and then their code implementations are made.

In the continuation, some variations are applied to the Randomized Prim Algorithm. They are named as Initialized Prim, Stochastic Prim, and Depth-First Prim algorithms.

Randomized Prim and Hunt and Kill algorithms are selected to be modified for cyclic mazes. Loop Hunt and Kill and Loop Prim algorithms are implemented for the purpose of having mazes with loops. The selection is made for these two algorithms because they have fundamentally different principles. While the Prim algorithm is based on generating a minimum spanning tree (MST), Hunt and Kill performs a random walk and a "hunt" phase in its strategy [10].

3.1 Prim-based Variants

3.1.1 Randomized Prim's Algorithm

Randomized Prim's Algorithm is a maze generation technique adapted from Prim's Minimum Spanning Tree algorithm. It is a common algorithm in the literature. The fundamental idea is to grow the maze from an initial starting point by iteratively adding one of the unvisited neighboring cells, gradually expanding the maze until it is fully generated. It tracks each unvisited neighboring cell of visited cells and selects one randomly [10].

The algorithm works as follows:

1. Start with a single cell and mark it as visited
2. Add all neighboring walls of the cell to a wall list
3. While there are walls in the list:
    a. Pick a random wall from the list
    b. If only one of the cells that the wall divides is visited:

1. Break down the wall (connect the cells)

2. Mark the unvisited cell as visited

3. Add the neighboring walls of the cell to the wall list

c. Remove the wall from the list

4. Continue until all walls have been processed [11]

In Table 1, characteristics of the Randomized Prim's maze are given by averaging the metrics over 100 mazes. For a better understanding of these values' meaning, rank is provided. The rank shows the position of each algorithm relative to others (among 10 maze generation algorithms) for only that specific metric. Rank 1 represents having the minimum average measured value, while rank 10 represents having the maximum average measured value.

Metrics	Avg. Measured Value	Rank
# of Intersections	168.94	8
# of Deadends	198.21	9
Avg. Length of Deadends	35.22	2
Solution Length	51.32	3

Table 1: Characteristics of Randomized Prim's Algorithm

3.1.2 Initialized Prim Algorithm

This algorithm is the first modified version of the Randomized Prim Algorithm. It begins constructing the maze by introducing an arbitrary path. The aim is to generate an initial path from one node to another, allowing a portion of the solution path to be predefined and the overall structure to be partially known in advance. The idea is to manipulate the maze generation process and reduce randomness to some extent. After establishing this initial path, the algorithm proceeds to generate the remaining parts of the maze by applying the logic of the Randomized Prim Algorithm.

<u>The algorithm works as follows:</u>

1. Start with the pre-generated path(or paths)

2. Add all walls adjacent to the path to a wall list

3. While there are walls:

    a. Pop a random wall from the list

    b. If the cell on the other side hasn't been visited:

    - Connect current cell to neighbor through the wall

    - Mark neighbor as visited

    - Add neighbor's walls to the wall list

4. Continue until no walls remain

Note that this algorithm is not included in the experimental step. However, the idea of having a predefined structure in a maze - which one can control the characteristics in that part - can be used in further studies to generate mazes with controlled levels of difficulty.

3.1.3 Loop Prim's Algorithm

Loop Prim's Algorithm is an extension of Randomized Prim's Algorithm. It is modified by introducing new edges to the maze that is generated by the Randomized Prim's Algorithm. Each new edge creates a new loop in the maze. Since the mazes without loops are spanning trees in this context, all nodes are connected. In spanning trees, if the number of edges is one less, not all nodes would be connected, and if it is one more, then there exists a cyclic structure.

Mazes with different loop densities can be generated with this algorithm since it also takes an argument called "loop density". This parameter determines the number of loops to add with the equation below. The result is rounded down.

$$\text{loop amount} = \lfloor \text{loop density} \times (\text{maze size} / 2) \rfloor$$

Equation 1. Loop Amount Calculation

For a maze with size 25x25, and loop density = 0.01;

$$0.01 * (25 * 25 / 2) = 3.125$$

The result is rounded down and found to be 3.

Sampling has been made with three different loop density values: 0.01, 0.10, and 0.25.

The algorithm works as follows:

1. Start with a single cell and mark it as visited
2. Add all neighboring walls of the cell to a wall list
3. While there are walls in the list:
    a. Pick a random wall from the list
    b. If only one of the cells that the wall divides is visited:
        1. Break down the wall (connect the cells)
        2. Mark the unvisited cell as visited
        3. Add the neighboring walls of the cell to the wall list
    c. Remove the wall from the list
4. Continue until all walls have been processed
5. For the given parameter (loop density), add edges randomly as many as the loop amount

In Table 2, characteristics of the Loop Prim's maze are given by averaging the metrics over 100 mazes. The values are for the mazes with a loop density of 0.1.

Metrics	Avg. Measured Value
# of Intersections	195.81
# of Deadends	173.23
Avg. Length of Deadends	28.65
Solution Length	50.22

Table 2: Characteristics of Loop Prim's Algorithm

Mazes with loops are discussed separately. The comparative analysis is made between the mazes with loops and their original versions with no loop. Therefore, they are not included in the rankings of metric values.

3.1.4 Depth First Prim Algorithm

Depth First Prim is another modified version of the Randomized Prim Algorithm. The base of the code and the logic come from the Randomized Prim Algorithm. While the Ranodmized Prim's algorithm tends to distribute nodes in a short area of cells, in this variation, a strategy is attempted where the neighbor selection is made so that the paths extend to less explored areas.

The algorithm works as follows:

1. Start at the initial cell and add its walls to a wall list
2. While there are walls:
    a. Check for less explored areas (cells with ≤1 visited neighbors)
    b. If in a dead-end sequence and less explored walls exist:
        - Choose randomly from less explored walls to extend dead-end
        - Increment depth counter
    c. Otherwise:
        - Choose wall randomly
        - Reset depth counter and generate new max depth
    d. Connect cells through chosen wall if neighbor unvisited
    e. Add new neighbor's walls to wall list

In Table 3, characteristics of the Depth First Prim's maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	198.27	10
# of Deadends	248.31	10
Avg. Length of Deadends	34.43	1
Solution Length	51.10	2
Table 3: Characteristics of Depth First Prim's Algorithm		

3.1.5. Stochastic Prim Algorithm

In the last variation of Randomized Prim's Algorithm, the aim is to reduce randomness. Neighboring node selection is made probabilistically. Weights are assigned to neighboring nodes based on their degrees.

In graph terminology, the degree of a vertex refers to the number of edges connected to that vertex.

Probabilities decrease as the degree of nodes decrease. For instance a node with degree of 3 has a higher probability than a node with a degree of 1, therefore the selection tends to favor neighbors with more connections which leads the resulting maze to have more junctions (3-way connection) and more crossroads (4-way connection).

<u>The algorithm works as follows:</u>

1. Start at the start cell and add its walls to the wall list
2. While there are walls:
    a. Calculate weights based on neighbor degrees
    b. Choose a wall probabilistically based on degree weights
    (higher degree = higher probability of being chosen)
    c. If the cell beyond the wall hasn't been visited:
    - Add it to the visited set
    - Connect it to the current cell

- Add its unvisited neighbors to the wall list

3. Continue until no walls remain

In Table 4, characteristics of the Stochastic Prim's maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	174.94	9
# of Deadends	196.78	8
Avg. Length of Deadends	42.10	4
Solution Length	48.02	1

Table 4: Characteristics of Stochastic Prim's Algorithm

3.2. Randomized Kruskal's Algorithm

Randomized Kruskal's Algorithm is a very common maze generation algorithm in the literature. Its fundamental approach is similar to Randomized Prim's. It tries to find the minimum spanning tree. However, the generation approach differs. In this method the edges are selected randomly rather than nodes. Unlike node selection, not all edges need to be selected. Edge selection is made in a way that prevents loop formation. The algorithm stops when all nodes are connected [10].

The algorithm works as follows:

1. Start with a list of all possible edges in the grid

2. Shuffle the edges randomly

3. For each edge:

a. Check if the cells it connects are in different sets

b. If they are:

- Add the edge to the maze (connect the cells)

- Merge the sets of the connected cells

c. Continue until all cells are connected [11]

In Table 5, characteristics of the Initialized Prim's maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	163.39	7
# of Deadends	187.85	7
Avg. Length of Deadends	41.30	3
Solution Length	75.16	5
Table 5: Characteristics of Randomized Kruskal's Algorithm		

3.3 Hunt and Kill Variants

Hunt and Kill Algorithm is an algorithm from the literature. A variation of this algorithm named Loop Hunt and Kill is implemented in this study.

3.3.1 Hunt and Kill

This algorithm constructs the maze by iteratively expanding from the current cell to a randomly selected unvisited neighbor. Unlike Randomized Prim's Algorithm, it only looks for the neighbors of the current cell and not to the neighbors of all cells in the generated paths. When the current cell has no unvisited neighbors, the algorithm performs a systematic scan of the grid to find a visited cell that has at least one unvisited neighbor and continues the process from there. In short, the algorithm performs a random walk. When stuck, it hunts for a new node where it can continue its random walk [10].

The algorithm works as follows:

1. Start at a cell and add it to visited cells

2. While possible:

    a. If current cell has unvisited neighbors:

        - Choose random unvisited neighbor

- Connect current cell to chosen neighbor

- Move to chosen neighbor

b. If no unvisited neighbors (dead end):

- Hunt for an unvisited cell that's adjacent to a visited cell

- If found, make it current cell and connect to a visited neighbor

- If no unvisited cells remain, maze is complete [11]

In Table 6, characteristics of the Hunt and Kill maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	58.39	1
# of Deadends	60.21	1
Avg. Length of Deadends	73.05	8
Solution Length	107.10	8

Table 6: Characteristics of the Hunt and Kill Algorithm

3.3.2 Loop Hunt And Kill

This algorithm starts by generating a maze with the Hunt and Kill Algorithm. Similar to the Loop Prim Algorithm, new edges are added to the existing maze based on the specified number of loops. Loop amount calculation is done in the same way as the Loop Prim Algorithm.

Sampling has been made with three different loop density values: 0.01, 0.10, and 0.25.

The algorithm works as follows:

1. Start at a cell and add it to visited cells

2. While possible:

a. If current cell has unvisited neighbors:

- Choose random unvisited neighbor

- Connect current cell to chosen neighbor

- Move to chosen neighbor

b. If no unvisited neighbors (dead end):

- Hunt for an unvisited cell that's adjacent to a visited cell

- If found, make it current cell and connect to a visited neighbor

- If no unvisited cells remain, maze is complete

3. For the given parameter (loop density), add edges randomly as many as the loop amount

In Table 7, characteristics of the Loop Hunt and Kill maze are given by averaging the metrics over 100 mazes. The values are for the mazes with a loop density of 0.1.

Metrics	Avg. Measured Value
# of Intersections	107.19
# of Deadends	52.21
Avg. Length of Deadends	43.12
Solution Length	70.70

Table 7: Characteristics of Loop Hunt and Kill Algorithm

3.4. Sidewinder Algorithm

The Sidewinder Algorithm is an existing algorithm. It builds the maze row by row. It creates horizontal runs and occasionally connects them upward to the previous row. Since upward connections in the top row are not possible, it forms an unbroken corridor there. The result is a maze with strong horizontal bias and relatively few vertical links [10].

The algorithm works as follows:

1. Process the maze row by row from top to bottom

2. For each row:

a. Keep track of cells in current run

b. For each cell in row:

- Randomly decide to either:

  * Carve east and continue run

  * End run by carving north from random cell in run

c. Reset run after carving north

3. Continue until all rows are processed [11]

In Table 8, characteristics of the Sidewinder maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	153.60	4
# of Deadends	171.00	4
Avg. Length of Deadends	42.29	5
Solution Length	59.36	4

Table 8: Characteristics of the Sidewinder Algorithm

3.5. Wilson's Algorithm

The Wilson algorithm is one of the random-walk based algorithms. It uses a loop-erased random walk. It starts at a random cell, then picks another random cell and traces a path between them. If the path loops back on itself, it erases the loop, and the path continues from the base of the loop.

Once the random walk reaches the initial starting cell, the path becomes a part of the maze. The process repeats by selecting new random cells and walking through the maze until all cells are connected. This algorithm generates uniform spanning trees [10].

The algorithm works as follows:

1. Start with one visited cell

2. While there are unvisited cells:

   a. Pick a random unvisited cell

b. Perform a random walk until hitting a visited cell

c. If the walk creates a loop:

- Erase the loop portion

d. Add the final walk path to the maze

e. Mark cells in path as visited

3. Continue until all cells are visited [11]

In Table 9, characteristics of the Wilson's maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	160.10	6
# of Deadends	182.68	6
Avg. Length of Deadends	45.53	6
Solution Length	77.88	6

Table 9: Characteristics of Wilson's Algorithm

3.6. Aldous-Broder Algorithm

The Aldous-Broder algorithm is another algorithm that generates uniform spanning trees. Its fundamental idea is performing random walks throughout all of the maze. The random walk may revisit already visited cells, and the path continues without any removal of loops. As it moves, it marks unvisited cells as visited and keeps walking until the maze is fully constructed [10].

The algorithm works as follows:

1. Start at a random cell

2. Choose a random neighbor

3. If the neighbor hasn't been visited:

- Connect current cell to neighbor

- Mark neighbor as visited

4. Move to the neighbor cell

5. Repeat steps 2-4 until all cells are visited [11]

In Table 10, characteristics of the Aldous-Broder maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	157.97	5
# of Deadends	179.53	5
Avg. Length of Deadends	46.36	7
Solution Length	81.10	7

Table 10: Characteristics of Aldous - Broder Algorithm

3.7. Recursive Backtracker

This algorithm is a common one and relies on backtracking as the name implies. It begins with the starting point in the context of this study. It can begin with a random cell in other implementations.

The algorithm branches out in a random direction. It basically wanders through the maze in a depth oriented way. When it branches itself into a corner, unlike Hunt and Kill Algorithm, it backtracks to previously visited vertex and continue [10].

The algorithm works as follows:

1. Start at initial cell (0,0)

2. Mark current cell as visited and add to stack

3. While stack is not empty:

    a. If current cell has unvisited neighbors:

        - Choose random unvisited neighbor

        - Add passage between current and chosen cell

- Mark chosen cell as visited

- Push chosen cell to stack

b. Else:

- Pop cell from stack (backtrack) [11]

In Table 11, characteristics of the Recursive Backtracker maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	62.03	3
# of Deadends	63.56	3
Avg. Length of Deadends	122.41	9
Solution Length	216.70	9

Table 11: Characteristics of Recursive Backtracker Algorithm

3.8. Randomized Depth-First Search

This algorithm follows a very similar strategy to Recursive Backtracker. It starts at an initial cell and checks for unvisited neighbors. It proceeds by selecting one randomly. If no unvisited neighbors remain, it marks the previous cell as the current one, and continues the process from there [12].

The algorithm works as follows:

1. Start at initial cell

2. While there are unvisited cells:

a. Get unvisited neighbors of current cell

b. If there are unvisited neighbors:

- Choose random unvisited neighbor

- Add wall between current and chosen cell

- Make chosen cell current and mark as visited

- Add chosen cell to stack

c. If no unvisited neighbors:

- Pop cell from stack and make it current

3. Continue until stack is empty [12]

In Table 12, characteristics of the Randomized Depth-First Search maze are given by averaging the metrics over 100 mazes.

Metrics	Avg. Measured Value	Rank
# of Intersections	61.86	2
# of Deadends	63.47	2
Avg. Length of Deadends	122.65	10
Solution Length	230.80	10

Table 12: Characteristics of Randomized Depth-First Search Algorithm

4. Maze-Solving Algorithms

Generating mazes with different algorithms and creating samples helps to analyze the properties and features of mazes with different generation algorithms. For example, it can be said that mazes generated by algorithm X have the longest solution paths on average. However, making comparisons based on features only is not very meaningful since it only gives an idea about the features, not the hardness of these features. What makes a maze more difficult is more about how it is solved by agents. That is why different solving agents will be used for comparison purposes, and in this part, some information about these solving agents will be provided. [13, 14]

4.1. Breadth-First Search (BFS)

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that systematically explores a graph level by level. Beginning from a designated starting node (often referred to as the root), BFS first visits all nodes directly connected to the start—i.e., those at distance one. After all nodes at the current level have been explored, the algorithm proceeds to the next level, visiting all unvisited neighbors of the previously explored nodes. This process continues iteratively until the target node is reached or all reachable nodes have been explored. [15]

The algorithm works as follows:

1. Initialize the queue with the start node and its path
2. While there are nodes to explore:
    a. Get the next node from the queue
    b. If the current node is the end, reconstruct the path
    c. For each unvisited neighbor:
        - Add to the queue with the updated path
3. Return solution path

4.2. Depth-First Search (DFS)

Depth-First Search (DFS) is a classical graph traversal algorithm that explores as far as possible along each branch before backtracking. Unlike Breadth-First Search (BFS), which traverses the graph level by level, DFS prioritizes depth by recursively visiting neighboring vertices along a single path. Starting from a designated root node, the algorithm selects one of its neighbors and continues visiting the neighbor of the most recently visited node.

This process continues until it reaches a terminal node or a dead-end—i.e., a vertex with no unvisited neighbors. At that point, the algorithm backtracks to the most recent vertex with unvisited neighbors and resumes traversal. This recursive exploration continues until all reachable vertices in the graph have been visited. [16]

The algorithm works as follows:

1. Initialize the stack with the start node and its path
2. While there are nodes to explore:
    a. Get next node from stack (LIFO)
    b. If the current node is the end, reconstruct the path
    c. For each unvisited neighbor:
        - Add to the stack with the updated path
3. Return solution path

4.3. Hybrid Search

This method represents one of the algorithmic contributions of our work, alongside our modifications to various maze generation algorithms such as Depth-First Prim, Stochastic Prim, and others. In the Hybrid Search approach, Breadth-First Search (BFS) and Depth-First Search (DFS) are strategically combined to balance exploration and depth.

The algorithm begins by exploring the maze using Breadth-First Search (BFS) until a predefined threshold—approximately one-third of all nodes—has been visited. After reaching

this threshold, it identifies the longest path discovered during the BFS phase and switches to a Depth-First Search (DFS) strategy, following this path toward the exit.

Experimental results indicate that this method functions as a more general-purpose solver, performing consistently well across different maze types. In contrast to traditional BFS and DFS, which exhibit performance fluctuations depending on maze structure, Hybrid Search maintains a relatively stable and effective behavior regardless of the underlying maze topology.

The algorithm works as follows:

1. Start at the entrance node.
2. Perform Breadth-First Search (BFS), exploring nodes level by level.
3. Continue BFS until approximately one-third of all nodes in the maze have been visited.
4. Identify the longest path discovered during the BFS phase.
5. Switch directly to Depth-First Search (DFS) from the end of this path.
6. Continue exploration using DFS, prioritizing unvisited neighbors.
7. Terminate once the exit node is reached.

4.4. A Search*

A* is a well-known graph traversal algorithm, similar in purpose to the previously discussed maze-solving methods such as BFS and DFS. However, unlike BFS and DFS, A* incorporates a heuristic function to guide the search process, making a decision in every iteration and enabling more efficient pathfinding. [13]

In this implementation, the Manhattan distance is used as the heuristic. For each unexplored node, A* evaluates a cost function defined as the sum of the distance from the start node (the actual cost so far) and an estimate of the remaining distance to the goal (the heuristic). The heuristic component—Manhattan distance—is chosen specifically because it aligns well with the grid-like structure of typical mazes, where movement is restricted to horizontal and vertical directions.

This heuristic enables A* to prioritize nodes that are more likely to lead toward the exit, balancing exploration efficiency with solution optimality. [17]

The algorithm works as follows:

1. Initialize the priority queue with the start node
2. While there are nodes to explore:
    a: Select the node with the lowest total cost, calculated as:
    total cost = distance from start + estimated distance to goal
    b: If the current node is the end, reconstruct the path and terminate
    c: For each neighbor:
        - Calculate the new cost
        - If a better path is found, update costs and add to the queue
3. Reconstruct and return the optimal path

4.5. Random Walk

This algorithm adopts an agent-based approach to maze solving. The agent explores the maze by randomly visiting cells until it discovers the exit. While the underlying exploration strategy is relatively simple, the implementation includes a key optimization: previously visited cells are recorded to avoid redundant exploration. This enhancement significantly improves the algorithm's efficiency by preventing the agent from revisiting already explored areas. [14]

The algorithm works as follows:

1. Start at the entrance
2. Randomly choose an unvisited neighbor
3. Move to that neighbor and mark it as visited
4. If no unvisited neighbors, backtrack
5. Continue until reaching the exit

4.6. Hand-on-Wall Strategy

The Hand-on-Wall algorithm is a well-established and intuitive method for solving mazes. The agent follows a simple rule: it places one hand on a wall and continuously follows it until the exit is found. Despite its simplicity, this strategy has proven effective and has historically been popular in maze-solving competitions, particularly in Japan. Its widespread adoption persisted until competition organizers began incorporating loops into maze structures to increase difficulty. It is important to note that the Hand-on-Wall method is guaranteed to succeed only in simply connected (perfect) mazes—those without loops. In mazes containing cycles, the agent may become trapped in an infinite loop, rendering the strategy ineffective. [18]

The algorithm works as follows:

1. Start at the entrance of the maze
2. Choose a wall side (left or right) to follow
3. Move forward while keeping the chosen hand in contact with the wall
4. At each junction or decision point, turn in a way that maintains contact with the wall
5. Continue this process until the exit is found

4.7. Deep Q-Learning

This algorithm is based on the Markov Decision Process framework, where an agent learns to make decisions by interacting with an environment in discrete time steps. In an MDP, the outcome of an action depends only on the current state and action, not on the sequence of previous events — a property known as the Markov property. [24]

Instead of following a predefined path or rule set, DQN enables an agent to learn a policy that maximizes the expected cumulative reward over time by approximating the optimal action-value function, also known as the Q-function.

Starting from an initial state, the agent chooses actions according to an ε-greedy strategy — sometimes selecting the best-known action (exploitation), and sometimes a random action

(exploration). Each action yields a reward and leads to a new state. These current state–action–reward–next state transitions form the foundation of learning. [21]

The Q-function is updated using the Bellman Equation, which expresses the relationship between the value of a state–action pair and the value of the subsequent state:

$$Q(s, a) = r + \gamma \cdot \max_{a'} Q(s', a')$$

Equation 2. Bellman Equation

Here, s represents the current state, a is the action taken in that state, and r is the immediate reward received after taking that action. The resulting state is denoted by s′, and a′ refers to all possible actions that can be taken from s′. The term γ, called the discount factor, determines the importance of future rewards relative to immediate ones. [23, 25]

In practice, DQN uses a deep neural network to approximate this Q-function and updates it by minimizing the error between the predicted and target Q-values.

4.7.1. Q-Network

The neural network used in this implementation serves as a function approximator for the Q-values, estimating the expected cumulative reward of each possible action in a given state. It is a fully connected feedforward network with two hidden layers. Each layer applies a linear transformation followed by a non-linear activation function (SiLU), enabling the network to capture complex relationships between state features and action values. The input layer receives the current state representation, and the output layer produces one Q-value per possible action, allowing the agent to evaluate and compare its options. By training on past experiences, the network gradually approximates the optimal action-value function that guides the agent's decision-making process. [22]

4.7.2. Maze Environment

The environment represents the maze as a two-dimensional grid derived from its graph structure, where 0 denotes open paths and 1 denotes walls. The agent starts at position (1, 1)

and aims to reach the goal at (size-2, size-2), navigating through allowed (non-wall) cells. At each time step, the agent receives a default step penalty of -0.05, which encourages faster convergence to the goal by penalizing longer paths. [20]

Invalid actions—attempts to move into walls or out of bounds—are penalized with a stronger reward of -1, discouraging ineffective exploration. Revisiting previously visited states imposes an additional penalty of -0.2, designed to promote the discovery of new paths and reduce cyclic behavior.

A key feature of this environment is its reward shaping around dead ends. Dead-end cells (with degree 1 in the underlying graph) that are not the goal or start nodes are precomputed, and each is assigned a negative reward scaled between -0.4 and -1.0 based on its L1 (Manhattan) distance to the goal. The farther a dead-end is from the goal, the more it is penalized. This discourages the agent from entering paths that do not contribute to task completion.

Reaching the goal terminates the episode and yields a cumulative penalty equal to the total magnitude of all dead-end penalties, serving as a strong negative signal to prevent trivial loops or artificially prolonged trajectories. Together, these reward mechanisms encourage efficient, goal-directed behavior while dynamically penalizing inefficient exploration patterns. [19]

In Figure 3, an example of dead-end penalties and finding the solution reward can be observed visually.

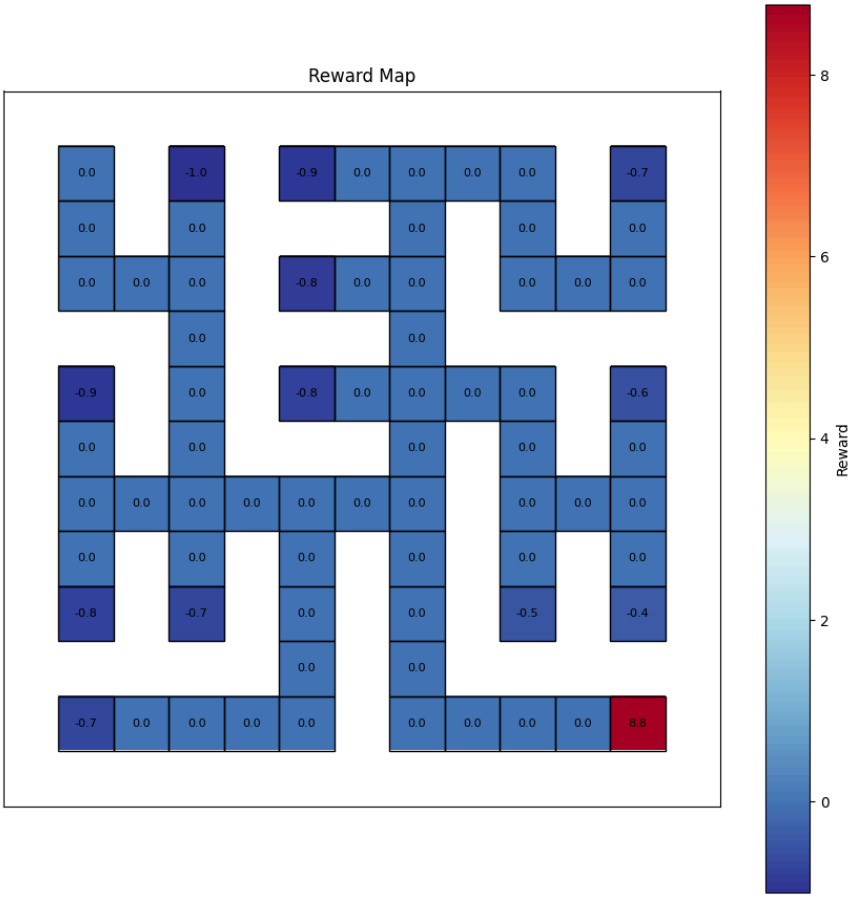


Figure 3: Reward Map Example

4.7.3. Agent

The learning agent operates within the maze environment by interacting at each time step through perception, action selection, and learning from consequences. At every step, it observes the current state of the maze and selects an action based on its Q-network's predictions, guided by a stochastic policy that can follow either ε-greedy or softmax-based exploration, depending on the configuration.

The agent uses a buffer to store transitions of the form (state, action, next state, reward, done), which are later used for training. The total reward collected during an episode is monitored, and if it drops below a threshold (set as one-third of the maze size negatively), the episode is forcefully terminated. This introduces a performance-based early stopping condition.

When choosing an action, the agent not only considers Q-values but also applies label smoothing through a small α (e.g., 1e-6) to avoid overconfidence in a single action. This helps stabilize learning, especially in sparse reward environments like mazes.

After each step, the agent updates its internal state and accumulates reward, applying stronger penalties for invalid moves, revisits, or reaching dead ends, as defined by the environment's shaping. In addition to decision-making, the agent is equipped to visualize the learned policy across the maze, highlighting the direction it would take at each state and the corresponding max Q-values, offering an interpretable view of the agent's learned strategy.

Every 20 iterations, the agent outputs its policy visually, as shown in Figure 4. Note that in Figure 4, the policy appears to be fully optimized, as the agent can reach the exit from any position in the minimum number of steps.

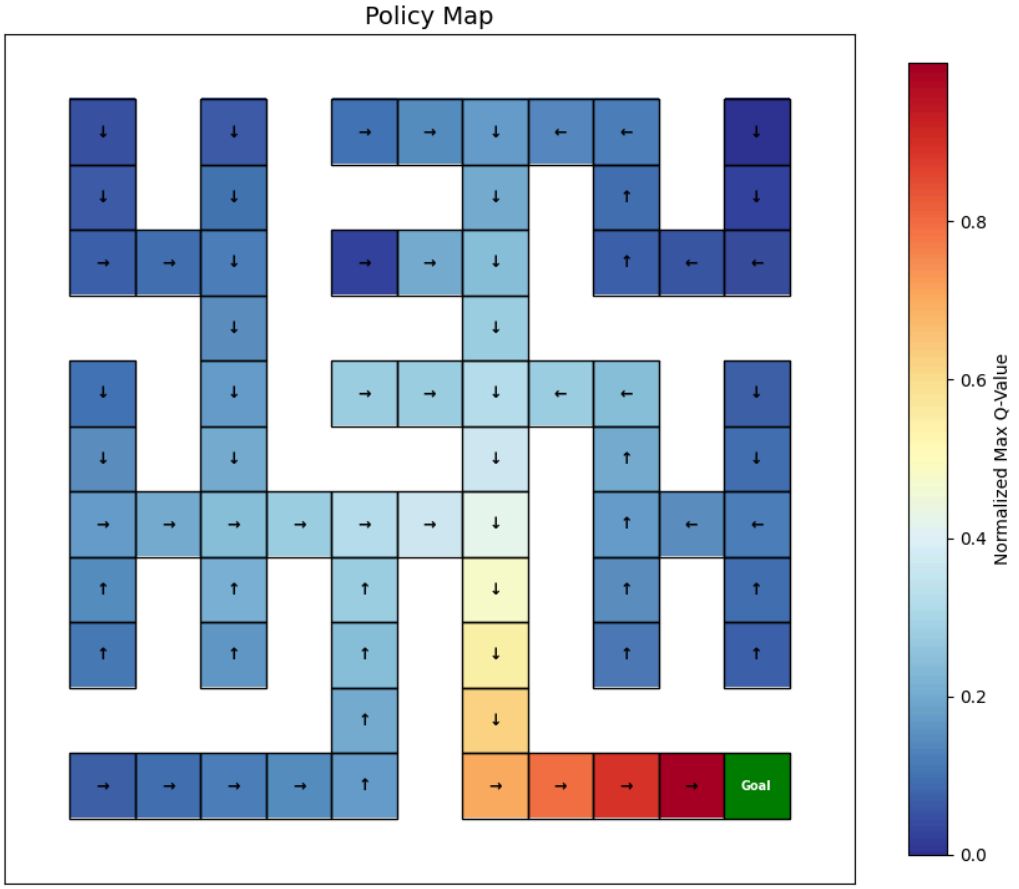


Figure 4: Policy Map Example

5. Implementation Details

5.1. Programming Language and Environment

The project was implemented using Python 3.12, selected for its simplicity, flexibility, and extensive library support for numerical computation, graph processing, and visualization. All development and experiments were conducted on a MacBook Air equipped with an Apple M3 processor and 16 GB of RAM, running macOS Sequoia. Several key Python libraries were utilized throughout the implementation. NumPy was used for efficient grid-based maze representation and array operations. NetworkX enabled graph-based representations of mazes and supported structural analysis as well as solver algorithms. Matplotlib was employed for static visualization of mazes and solution paths, and also supported the generation of dynamic animations. For the Deep Q-Learning component, PyTorch was used to implement neural networks and training routines.

5.2. File and Module Structure

Due to the structural nature of the project, it is organized in a modular fashion. Each functional component serves a distinct purpose: maze generation, maze solving, Deep Q-Learning, visualization, and evaluation. This structure is designed to enhance clarity, promote extensibility, and ensure the reproducibility of experiments.

- *MazeGenerationAlgorithms/* Contains the full set of implemented maze generation algorithms, including both classical and randomized approaches as previously described.
- *MazeSolutionAlgorithms/* Includes all implemented maze solving algorithms mentioned earlier.
- *DeepQ-Learning/* Includes the implementation of the fully connected Deep Q-Network. Previously trained models are stored in the files/ subdirectory.
- *Outputs/* Includes notebooks and output files for generating and storing maze animations. Generated visuals and animations are saved in the Outputs/ subdirectory.
- *Statistics/* Contains statistical analysis notebooks and data files related to maze generation, solving, and Q-learning

5.3. Data Structures Used

The implementation employs a combination of key data structures to effectively represent, manipulate, and visualize the maze environment.

- **Graph Structure**: The logical structure of the maze is represented using an undirected graph, where each cell corresponds to a node and paths between adjacent cells are modeled as edges. This enables efficient neighbor lookups, connectivity checks, and support for traversal algorithms used in the solving process.
- **Grid Structure**: A two-dimensional NumPy array is used to store the visual layout of the maze. Cells are encoded as either walls (1) or paths (0). A higher-resolution version of this array is also maintained for smooth visualization, allowing for detailed coloring and animation during exploration and solution rendering.

Additional data structures play supportive roles in the implementation. These include coordinate tuples to identify cell positions, sets for efficiently tracking visited cells, and deques to manage the order of exploration. These structures ensure quick access, clear indexing, and dynamic control over algorithmic flow. Lists and dictionaries are also used where necessary to store auxiliary data such as animation frames and position mappings.

5.4. Visualization / Animation Approach

To support analysis and interpretation of maze generation and solution processes, a dedicated visualization module was implemented using Matplotlib and Networkx. Static visualization functions (draw, draw_solution, visualize_maze_as_graph) are provided to display the structure of the generated maze, highlight the start and end points, and illustrate the solution path using graph-based shortest path algorithms.

For dynamic visualization, an animation approach is used during the maze generation phase. Each state of the maze grid is captured after each update and stored as a frame. These frames are then animated using Matplotlib.animation.FuncAnimation, effectively showing the progression of the maze being carved in real-time. The animations can be displayed inline in Jupyter notebooks using IPython.display.HTML, or exported as .mp4 files into the Outputs/ subdirectory.

This animation technique not only enhances the interpretability of how various maze generation algorithms work but also provides an engaging way to compare structural characteristics and complexity across algorithms.

6. Experiments and Results

6.1. Experimental Setup

In order to evaluate the characteristics of mazes and solver performance on different types of mazes, a series of experiments are designed. For each maze generation algorithm, 100 maze instances were generated at a fixed size of 25x25. This uniform sizing ensured consistency in the comparison of structural and computational properties across algorithms.

Each generated maze was represented in both a grid-based format and a graph-based structure. The grid format, implemented using NumPy arrays, was used for visualization and evaluation purposes, while the graph representation, constructed using NetworkX, was utilized for extracting maze characteristics and enabling solver applications.

To analyze the characteristics of the mazes, several features were computed for each instance, including the number of dead ends, the number of intersections, the solution path length, and various other structural metrics.

To analyze the impact of each maze structure in terms of solving difficulty, all generated instances were processed using the aforementioned solvers. For each run, performance metrics such as the number of visited nodes, excess steps, dead-end coverage, and various other indicators were recorded. These metrics were then averaged over the 100 maze instances corresponding to each algorithm in order to ensure statistical reliability and to enable meaningful comparisons of solver behavior across different maze types.

It is worth noting that, due to computational constraints, Deep Q-Learning was only applied to 10x10 mazes and limited to 15 instances per algorithm. In addition to standard performance metrics, specific indicators related to learning behavior were also tracked. These included the number of epochs required for convergence, the ratio of convergence epoch to the average dead-end reward, and other indicators highlighting the relationship between the applied penalty for dead ends and the learning performance of the agent.

6.2. Maze Metrics Results

To facilitate a fairer comparison and comprehension of the structures of mazes generated by different generation algorithms, a set of already developed measures was gathered and used on more than 100 maze instances for each algorithm. Those measures, introduced in Section 2.3, test a variety of maze attributes. The analysis of those results on metrics provides information on the structure characteristics of the mazes generated by each algorithm.

A general analysis discovered that maze algorithms create distinctive structural patterns which can be generally grouped based on the character of their different features. Recursive Backtracker, Randomized DFS, and Hunt and Kill algorithms are most likely to create mazes with long, coiling solution paths and relatively small numbers of branch points. They are characterized by low intersection density but high measures in both tortuosity and turn number. Their dead ends are deep and well-separated.

On the other hand, algorithms like Prim's, Kruskal's, and Wilson's yield narrower and branched mazes. These mazes have shorter paths to the solution in a dense but shallow network of dead ends and shallow intersection points. Although they yield shorter paths, these mazes are more deceptive since they have numerous deceptive intersection points and branches, most of which run directly along the solution path. This kind of construction is misled frequently but with shorter dead ends. The results point out that difficulty in mazes is not a function of any single measure but one that arises from the interaction between several features.

A more thorough analysis of intersection and dead-end density continues to uphold the construction. Depth First Prim's, Wilson's, and Stochastic Prim's algorithms all gave mazes with a high rate of intersections across runs consistently, frequently more than 180 for a single instance alone. These intersections are followed by high dead-end numbers, in some cases higher than 170, leading to very branched, dense but shallow structure. In contrast, Recursive Backtracker, Randomized DFS, and Hunt and Kill created mazes with significantly fewer intersections, sometimes less than 60 and lower dead-end densities. These are more open, structures with deeper dead ends which provide fewer distractions.

Discussed analysis results above can be validated through the figure S below. It shows the maze structure of Randomized Kruskal, Hunt and Kill and Recursive Backtracker algorithms

according to 3 metrics: # of intersections, solution length and average dead end length (from solution path).

As can be observed from the figure, the Recursive Backtracker algorithm generates mazes with longer solution paths and deeper dead ends with lesser intersection points. While, Randomized Kruskal with a higher number of intersection points creates shorter solution paths and shallow dead ends. Hunt and Kill on the other hand is moderate on both ends.

When measuring solution length and tortuosity, a similar difference appears. Recursive Backtracker and Randomized DFS generated some of the most tortuous solution paths, which were more than 150 steps and had tortuosity from 2.5 to 3.0. These mazes will route solvers in tortuous paths that initially appear simple but are challenging because they have deeper dead ends. In contrast, Prim-based algorithms and Sidewinder mazes boasted much shorter solution paths of 50 to 80 steps on average and much reduced tortuosity factors between 1.2 to 1.6. Such structures favor a more spread structure but balance their simplicity with encapsulating the solution path with false intersections and dead ends that increase the risk of error. Generally speaking, longer solutions are highly associated with higher tortuosity and higher turn counts, and shorter solutions to more interconnected mazes mean less tortoisty but higher potential for intersection based decision-making.

Turn count, a measure of how often the solution path turns, is another measure of navigational complexity. Mazes by Recursive Backtracker and Randomized DFS had the highest number of turns and averaged about 24, meaning that they had very winding and long path lengths. In contrast, mazes by Stochastic Prim's and Randomized Prim's had significantly fewer turns closer to 11 or 13 exhibiting a trend towards linearity. These mazes are more dense at intersections, in which the solver is frequently tasked with making choices at closely spaced intersections, and less disorienting by turns.

Further insight was obtained from the study of dead-end distance measures, which establish how far from various reference points such as the start, the solution path, and key intersections dead ends exist. Mazes produced by Recursive Backtracker, Hunt and Kill, and Randomized DFS both had the highest mean and maximum distance to dead ends from any point of reference, with means typically in excess of 35 and up to greater than 90 at most. By comparison, Prim-based methods generated mazes with dead ends more closely bunched

together in clusters around the central course or major junctions. These parameters reduce the penalty of exploration while increasing the frequency of distractions and favor a more in-your-face type of misdirection distraction over more profound dead-end distraction. The rate of dead-end crossroads and the level of seductive intersections of those who fall along the right solution path are two particularly insightful statistics.

Depth First Prim's and Loop Prim gave the highest numbers in both classes, often giving more than 100 dead-end crossroads and more than 50 inviting intersections. In contrast, Recursive Backtracker and Randomized DFS had much smaller such distractions, with similarly low numbers often under 30, which reflects their linear and deterministic nature. These two measures are strong indicators of solver confusion potential and have a special impact for agents based on local, myopic exploration heuristics. Beyond direct measures of individual metrics, an analysis of correlations across metrics revealed underlying structure dynamics that support these findings.

There is a very strong positive correlation between the number of intersections, the number of dead ends, and the intersection count. This would imply that denser maze structures are more deceptive in nature as they pack intersections tightly along and around the solution path. Further, mean and maximum dead-end distances regardless of whether they are derived from the beginning, the solution path, or tightly packed intersections agree. Finally, dead-end intersections were found to have a high correlation with the overall dead ends as well as the number of seductive intersections to justify concluding that certain types of mazes actually put misleading features in and around the main solution path. From the metric scores, each maze generation algorithm can be categorized based on the structural preferences it facilitates. To explain the correlation between metrics, Plots below are provided. They show the results of generation algorithms according to maze metrics.

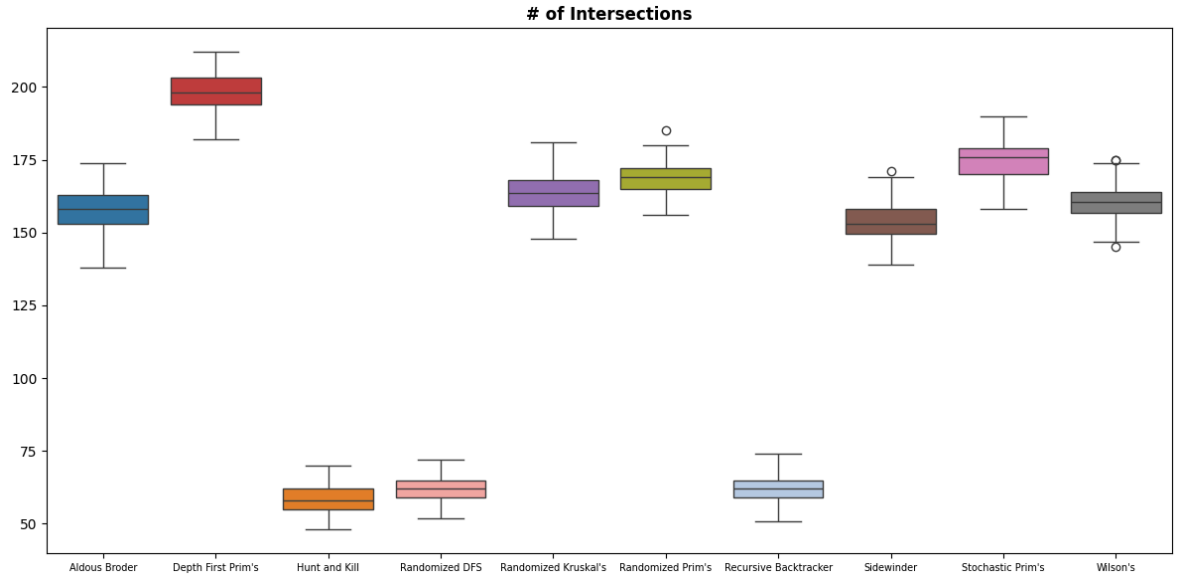


Figure 5: Average # of Intersections of Generation Algorithms

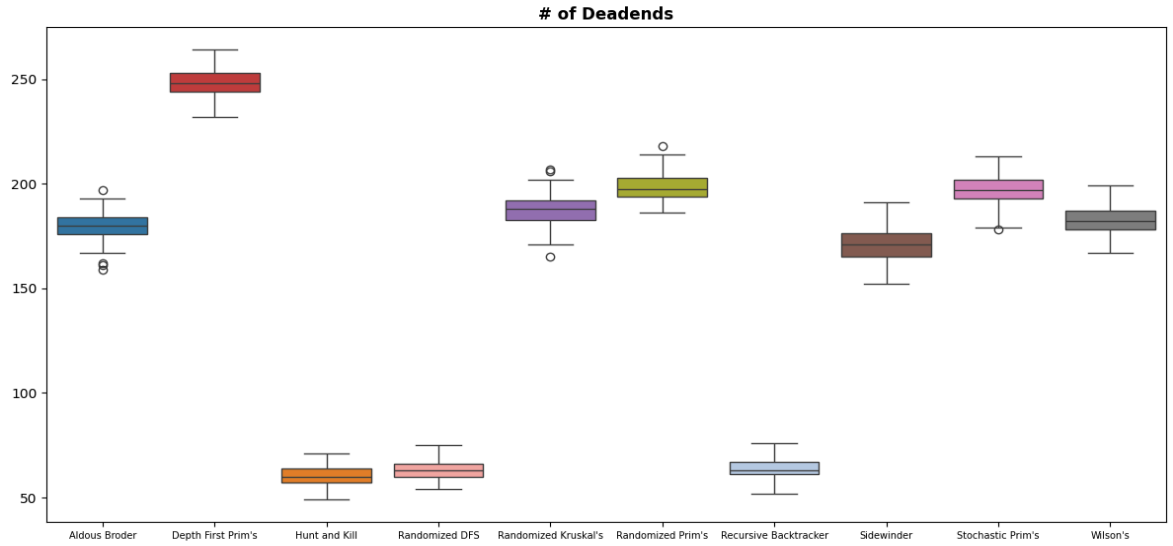


Figure 6: Average # of Dead Ends of Generation Algorithms

These two tables clearly shows the positive correlation between # of dead ends and # of intersections as their average results according to each maze type demonstrates a resembling distribution. Similar comparative discussion can be built upon all metrics, by evaluating such metrics' distributions among all maze generation algorithms as in this example. For instance, according to our discussion we would also expect solution path length and dead end length

(from solution path) metrics to have similar box plots among algorithms too since they are positively correlated.

Discussed analysis results above can also be validated comparatively for maze types. For instance, the figure 7 below, shows the maze structure of 3 distinct generation algorithms, Randomized Kruskal, Hunt and Kill and Recursive Backtracker according to 3 metrics: # of intersections, solution length and average dead end length (from solution path):

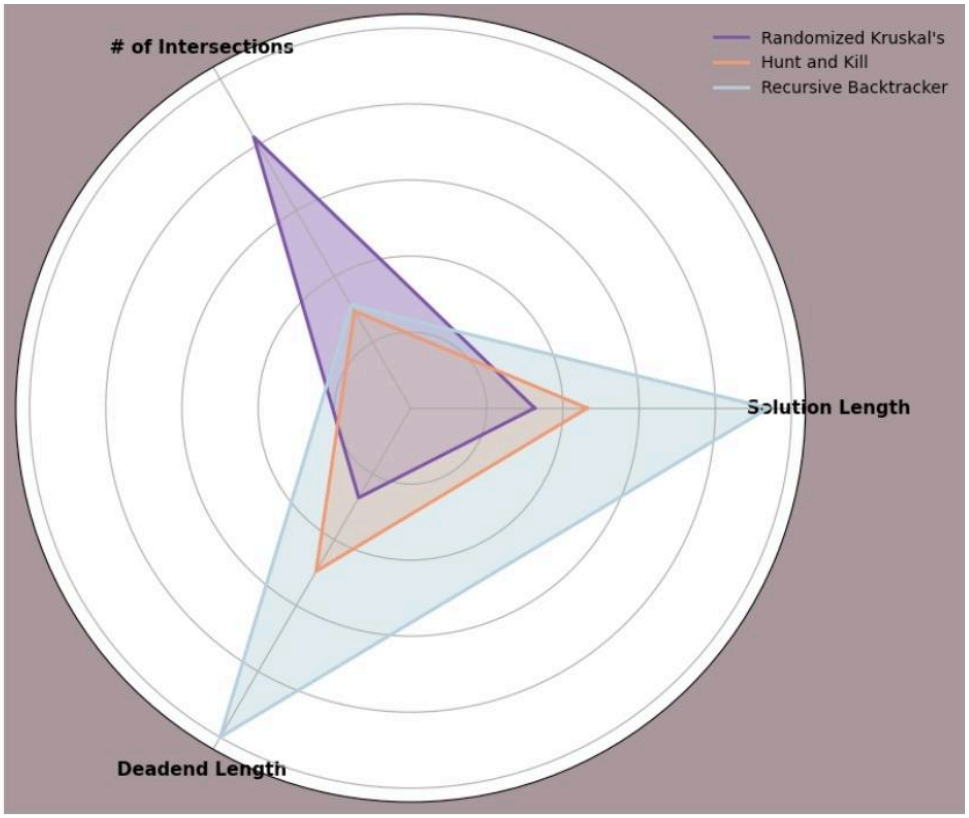


Figure 7 : Spider Plot of Generation Algorithms

As can be observed from the figure, the Recursive Backtracker algorithm generates mazes with longer solution paths and deeper dead ends with lesser intersection points. While, Randomized Kruskal with a higher number of intersection points creates shorter solution paths and shallow dead ends. Hunt and Kill on the other hand is moderate on both ends. Similar comparative discussion can be built upon other maze generation types with different pairs of metrics to examine their behaviour closely.

All the maze metric analysis that is done in this section according to different maze types can also be observed at Maze Metric Results Table at Appendix 3 comprehensively. Which

provides the quantified results of maze metrics for each generation algorithm with 100 samples.

Before ending the metric analysis it's important to note that the patterns observed in looped mazes align with those seen in non-looped mazes, but there are some important differences too. Looped mazes introduce an extra parameter called loop density, which measures how many loops are present within the maze. As loop density increases, the number of alternative solution paths might grow too, often leading to shorter solution paths since there's more than one way to reach the end. At the same time, a higher loop density usually means more intersections, as the added loops naturally create additional branching points. This reinforces the patterns discussed earlier, showing how loops contribute to a more interconnected and complex maze structure. Table 13 is the maze metric results of looped mazes for 100 samples of each type with different loop densities as can be seen below:

Maze Type	Loop Density	# of Intersections	Solution Length	Shortest Possible Solution	Solution Path Tortuosity	# of Deadends	Deadend Crossroads	Avg Distance to Deadend(From Solution Path)	Max Distance to Deadend(From Solution Path)	Avg Distance to Deadend(From Intersection)	Tempting Count
Loop Hunt and Kill	0.00	58.39	107.10	48.0	2.23	60.21	18.15	69.39	183.65	73.05	16.38
	0.01	61.74	99.46	48.0	2.07	58.16	18.01	64.95	169.40	68.21	16.47
	0.10	107.19	70.70	48.0	1.47	52.21	19.51	41.92	103.13	43.12	22.36
	0.25	176.68	57.64	48.0	1.20	40.74	18.87	29.53	67.33	30.08	29.83
Loop Prim	0.00	168.94	51.32	48.0	1.07	198.21	113.49	32.61	86.36	35.22	38.05
	0.01	172.60	50.84	48.0	1.06	196.95	114.38	32.31	84.24	34.92	37.81
	0.10	195.81	50.22	48.0	1.04	173.23	106.19	27.29	69.02	28.65	40.18
	0.25	236.40	49.24	48.0	1.02	140.63	94.92	23.54	57.09	24.23	45.24

Table 13: Maze Metric Results of Looped Mazes

6.2.1. Dimensionality Reduction

After a comprehensive maze metric analysis, To deepen our understanding on the underlying patterns across generation algorithms, A principal component analysis (PCA) was conducted. This reduction technique has been applied to the whole set of metrics aiming to capture the most significant sources of variance within a simplified representation. An initial 2 dimensional PCA gave the results shown in the Figure 8 below:

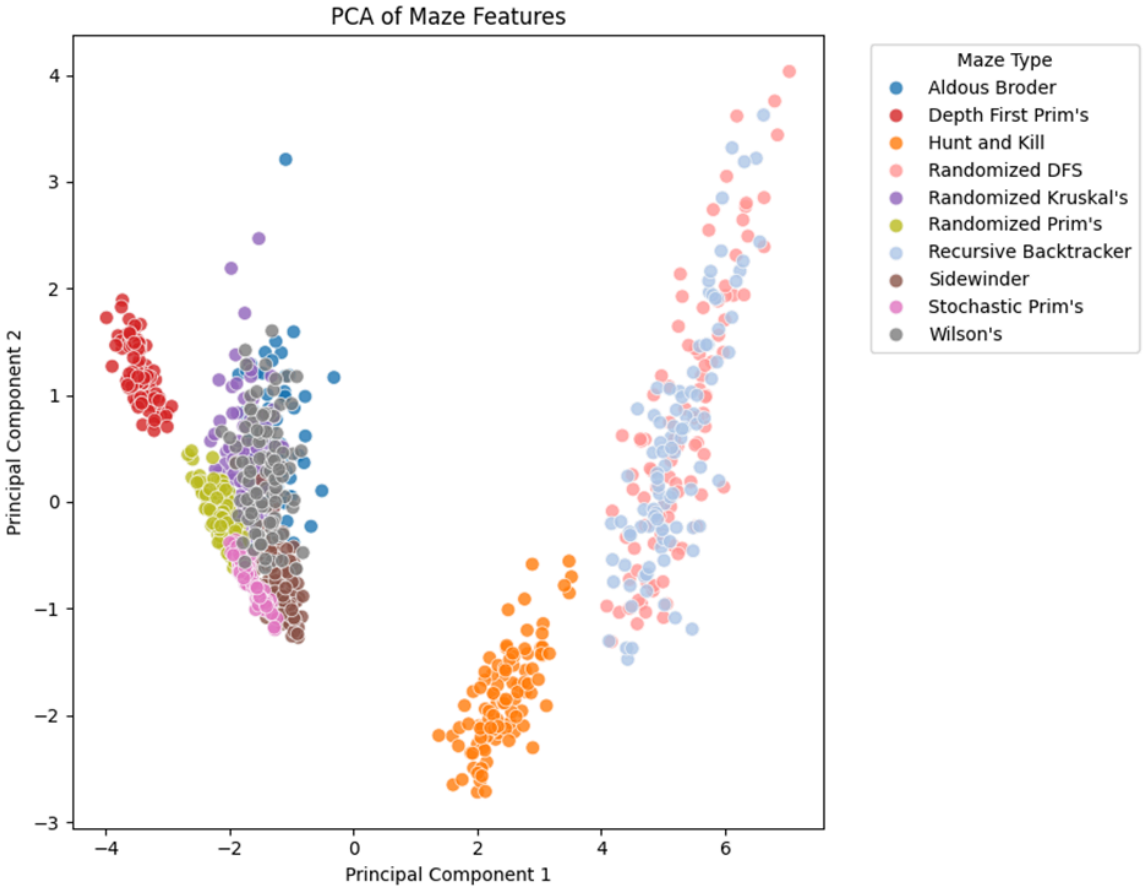


Figure 8: Initial 2 Dimensional PCA Results

However, The results revealed that a single principal component accounted for the majority of the total variance (over 80% to precise) in the dataset, indicating that much of the structural variation among maze types could be effectively summarized along one dominant dimension. An one dimensional PCA gave the results shown in the Figure 9 below:

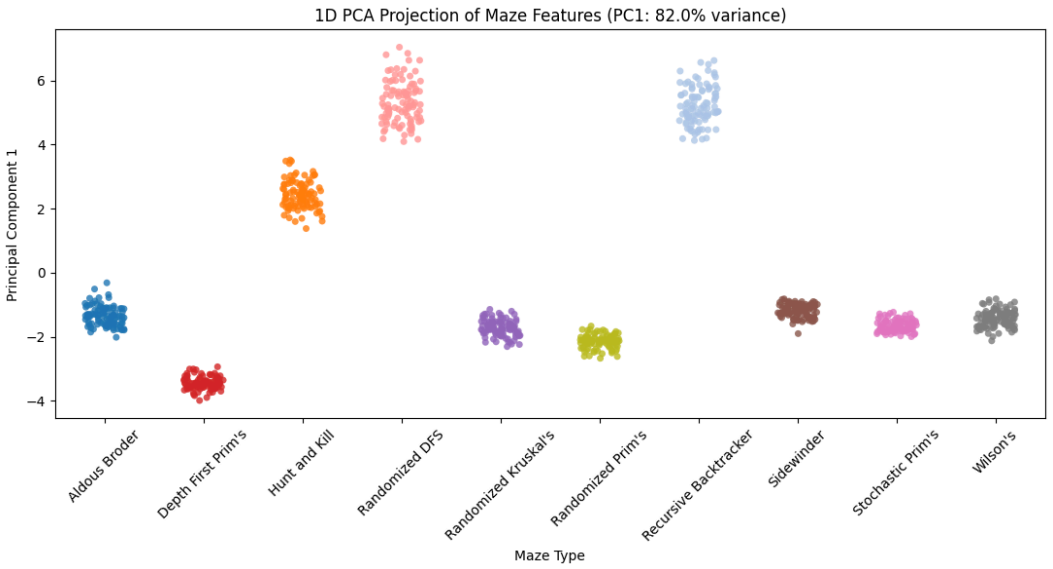


Figure 9: Improved 1 Dimensional PCA Results

When examining the distribution of the principal component across different maze types, a clear similarity was observed between this reduced dimension and the box plot of the 'Average Distance to Dead Ends (from Solution Path) metric. Figure 10 below presents the box plot for average dead-end distance across all algorithms:

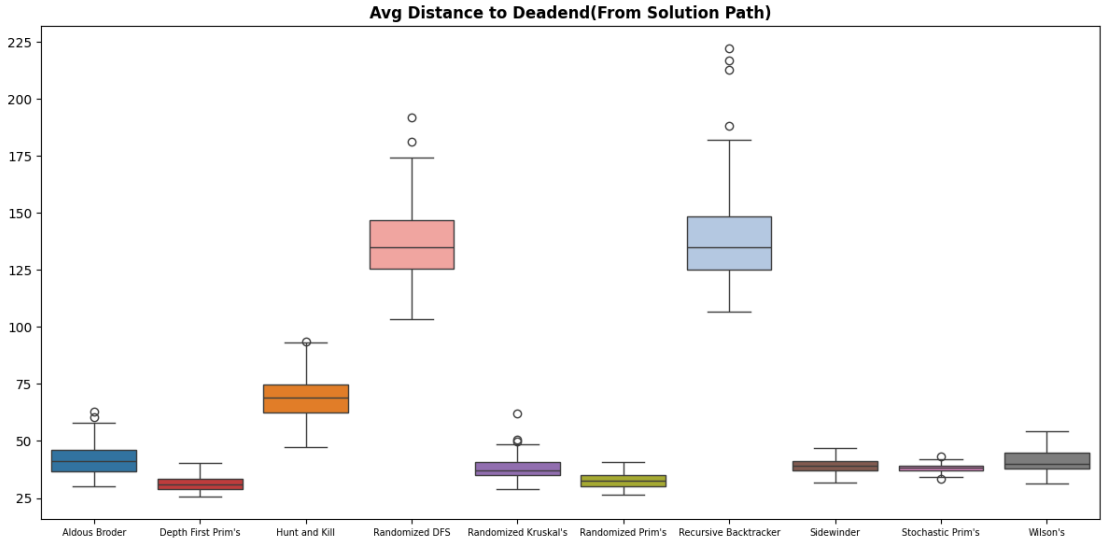


Figure 10: Average Distance to Dead Ends (From Solution Path) Metric Box Plot Results

This visual and statistical resemblance suggests that this metric plays a central role in capturing structural differences between mazes. In other words, the average depth of dead ends from the solution path appears to be a key characteristic that distinguishes one maze generation algorithm from another.

6.3. Solver Statistics Results

As described in the previous sections, the main part comprises 10 different generation algorithms, which are solved by 6 different solving algorithms, as well as the Deep Q-Network. In addition, there is a side discussion related to whether having a loop makes a maze easier or harder. In this part, the results of the solver statistics will be provided. Firstly, the results will be provided from the agents' perspective, then for every maze generation algorithm, another comparison will be made.

Before providing the details, a small discussion related to the solver statistics can be useful in light of the results. Recall that visited nodes, solution path length, excess step, exploration ratio, dead end, and intersection coverage are the related statistics for the solving agents part. And as it was discussed, making comparisons based on only visited nodes or solution path length is not very meaningful since mazes with longer solution paths by definition will have more visited nodes, which does not make them necessarily more difficult. Thus, the excess step is a more insightful statistic. Also, when the algorithm is not on the solution path, it spends its time hitting dead ends or visiting intersections more than required. That is why the excess step gives very similar results to intersection and dead-end coverage, in comparison. Also, since the exploration ratio may give a bias towards solution path length, it will not be directly used for analysis. However, it also gives similar results with the excess step metric. Thus, for the following results and comparisons, the excess step will be used.

Starting with the results of Breadth First Search (BFS), its results are provided in Table 14. The generation algorithms are ranked with respect to excess steps from the lowest to the highest. As can be seen in Table 14, the algorithm solves Randomized DFS and Recursive Backtracker very easily compared to others. The hardest ones seem to be Depth First and Randomized Prims since their excess steps are the highest. In addition, the performance of the algorithm is close to Wilson's, Aldous Broader, Hunt and Kill, and Recursive Backtracker.

Maze Type	Visited Nodes	Solution Path Length	Excess Steps	Exploration Ratio	Dead End Coverage	Intersection Coverage
Randomized DFS	358.59	217.06	141.53	0.625133	0.4590	0.5019
Recursive Backtracker	376.91	227.62	149.29	0.631664	0.4904	0.5309
Wilson's	505.85	75.54	430.31	5.851359	0.7889	0.8289
Aldous Broder	514.22	77.84	436.38	5.784270	0.8080	0.8436
Hunt and Kill	555.12	102.64	452.48	4.675566	0.8579	0.9160
Randomized Kruskal's	532.35	74.30	458.05	6.315299	0.8338	0.8738
Stochastic Prim's	557.97	48.12	509.85	10.596175	0.8601	0.9248
Sidewinder	597.28	59.60	537.68	9.095445	0.9384	0.9673
Depth First Prim's	619.39	51.18	568.21	11.132606	0.9885	0.9976
Randomized Prim's	621.10	51.70	569.40	11.046560	0.9906	0.9982

Table 14 : The Results of BFS

The second agent is Depth First Search (DFS); its results are provided in Table 15. The easiest ones for this agent are the same as BFS, which are Recursive Backtracker and Randomized DFS. It also solves Hunt and Kill also in an easier manner, which may be a result of the longer path lengths. The worst performance is with Sidewinder and Stochastic Prim, which have short paths. Results align with the working principle of DFS. (It chooses a path and follows it until the end)

Maze Type	Visited Nodes	Solution Path Length	Excess Steps	Exploration Ratio	Dead End Coverage	Intersection Coverage
Recursive Backtracker	385.83	227.62	158.21	0.767573	0.4313	0.5675
Randomized DFS	394.21	217.06	177.15	0.951858	0.4590	0.5850
Hunt and Kill	318.09	102.64	215.45	2.178944	0.4721	0.5606
Randomized Kruskal's	363.98	74.30	289.68	4.024049	0.5284	0.6371
Randomized Prim's	348.95	51.70	297.25	5.746843	0.5324	0.5969
Depth First Prim's	349.19	51.18	298.01	5.825006	0.5429	0.5799
Aldous Broder	417.38	77.84	339.54	4.559968	0.6404	0.6965
Wilson's	429.17	75.54	353.63	4.851066	0.6786	0.6954
Sidewinder	466.85	59.60	407.25	6.852317	0.7096	0.7873
Stochastic Prim's	459.19	48.12	411.07	8.539042	0.7310	0.7400

Table 15 : The Results of DFS

The third agent is Hybrid Search, which is a combination of BFS and DFS. The results of this agent are given in Table 16. The easiest ones are the same as the other agents, which are Randomized DFS and Recursive Backtracker. On the hardest side, Sidewinder, Randomized, and Depth First Prims have close performances. The other generation algorithms in the middle have close performances to each other.

Maze Type	Visited Nodes	Solution Path Length	Excess Steps	Exploration Ratio	Dead End Coverage	Intersection Coverage
Randomized DFS	300.26	217.06	83.20	0.420196	0.2927	0.4018
Recursive Backtracker	321.06	227.62	93.44	0.463577	0.3154	0.4330
Wilson's	367.20	75.54	291.66	3.979505	0.5459	0.6267
Randomized Kruskal's	375.19	74.30	300.89	4.195078	0.5518	0.6482
Aldous Broder	384.58	77.84	306.74	4.135623	0.5675	0.6606
Hunt and Kill	416.80	102.64	314.16	3.305645	0.5975	0.7355
Stochastic Prim's	369.50	48.12	321.38	6.680108	0.5285	0.6472
Sidewinder	438.03	59.60	378.43	6.419790	0.6531	0.7489
Randomized Prim's	436.00	51.70	384.30	7.462266	0.6555	0.7458
Depth First Prim's	439.09	51.18	387.91	7.592020	0.6640	0.7496

Table 16 : The Results of Hybrid Search

The results of the Random Walk agent are provided in Table 17. The first two stay the same, the easiest ones, but the gap between performances of other generation algorithms is not that high in this setting. Randomized and Stochastic Prim and Randomized Kruskal are the last three, the ones having shorter paths.

Maze Type	Visited Nodes	Solution Path Length	Excess Steps	Exploration Ratio	Dead End Coverage	Intersection Coverage
Randomized DFS	397.18	217.06	180.12	0.923737	0.4521	0.5877
Recursive Backtracker	412.11	227.62	184.49	0.954368	0.4760	0.6158
Hunt and Kill	353.85	102.64	251.21	2.659744	0.4803	0.6150
Wilson's	344.92	75.54	269.38	3.722174	0.4918	0.6080
Depth First Prim's	320.61	51.18	269.43	5.283155	0.4742	0.5620
Sidewinder	335.46	59.60	275.86	4.718188	0.4877	0.5787
Aldous Broder	357.18	77.84	279.34	3.734480	0.5053	0.6271
Randomized Prim's	338.72	51.70	287.02	5.550316	0.5026	0.5934
Stochastic Prim's	337.01	48.12	288.89	6.000325	0.5021	0.5763
Randomized Kruskal's	370.38	74.30	296.08	4.149880	0.5344	0.6534

Table 17 : The Results of Random Walk

Hand on Wall has a unique working principle, and its results are provided in Table 18. The agent solves the Sidewinder very easily, which might be the result of the full corridor in the first row of the maze. The last three, the hardest ones, are Randomized Kruskal, Depth First Prim, and Randomized Prim, which have short paths and a large number of intersections. The agent also has a good performance on Hunt and Kill, Recursive Backtracker, and Randomized DFS. These mazes have longer paths and fewer intersections, which might show a pattern here.

Maze Type	Visited Nodes	Solution Path Length	Excess Steps	Exploration Ratio	Dead End Coverage	Intersection Coverage
Sidewinder	142.08	59.60	82.48	1.398448	0.1527	0.2791
Hunt and Kill	252.05	102.64	149.41	1.563902	0.3163	0.4727
Recursive Backtracker	398.92	227.62	171.30	0.846123	0.4547	0.5949
Randomized DFS	403.62	217.06	186.56	0.965582	0.4693	0.6040
Stochastic Prim's	300.02	48.12	251.90	5.224033	0.4383	0.5219
Aldous Broder	343.29	77.84	265.45	3.566837	0.4860	0.6062
Wilson's	349.30	75.54	273.76	3.731634	0.4994	0.6156
Randomized Kruskal's	352.78	74.30	278.48	3.851476	0.5055	0.6221
Depth First Prim's	331.63	51.18	280.45	5.512338	0.4903	0.5791
Randomized Prim's	342.81	51.70	291.11	5.648565	0.5059	0.6000

Table 18 : The Results of Hand on Wall

The last agent in the classic setting is the A* search, and its results are given in Table 19. The easiest two are Randomized DFS and Recursive Backtracker, while the hardest one is Hunt and Kill. This agent tries to minimize the distance travelled, and the reason for the worst performance of Hunt and Kill may be a result of the maze structure. There is no direct pattern for mazes in this agent.

Maze Type	Visited Nodes	Solution Path Length	Excess Steps	Exploration Ratio	Dead End Coverage	Intersection Coverage
Randomized DFS	328.75	217.06	111.69	0.479235	0.3958	0.4473
Recursive Backtracker	350.81	227.62	123.19	0.505338	0.4309	0.4859
Stochastic Prim's	225.07	48.12	176.95	3.673525	0.2126	0.4928
Depth First Prim's	341.44	51.18	290.26	5.580039	0.4475	0.6519
Wilson's	366.64	75.54	291.10	3.844968	0.5405	0.6250
Randomized Kruskal's	365.41	74.30	291.11	3.858243	0.5397	0.6264
Aldous Broder	376.94	77.84	299.10	3.779980	0.5620	0.6404
Randomized Prim's	395.60	51.70	343.90	6.565078	0.5595	0.7141
Sidewinder	415.05	59.60	355.45	5.890236	0.6189	0.7012
Hunt and Kill	486.89	102.64	384.25	3.874400	0.7226	0.8158

Table 19 : The Results of A*

After looking at the maze generation algorithms agent by agent, looking at the general pattern in a plot might be useful for the general conclusion. Figure 11 is plotted based on the excess steps per agent, and the analysis can be made by comparing the widths of every color. For example, for the agent Hand on Wall, the easiest mazes are generated by Sidewinder, while Depth First and Randomized Prims are the hardest ones. Making comparisons per agent and combining this information, Randomized and Depth First Prims have the largest widths, so they are the hardest ones, whereas Recursive Backtracker and Randomized DFS have the narrowest widths, so they are the easiest ones in general. The result is not in a random

manner, there is a pattern. Recall that Recursive Backtracker and Randomized DFS have the longest paths, as a result, the longest solution paths, and fewer intersections. On the other hand, Depth First and Randomized Prim algorithms have a large number of intersections and shortest paths. Let's keep this analysis and try to see if it applies to the Deep Q-Network side.

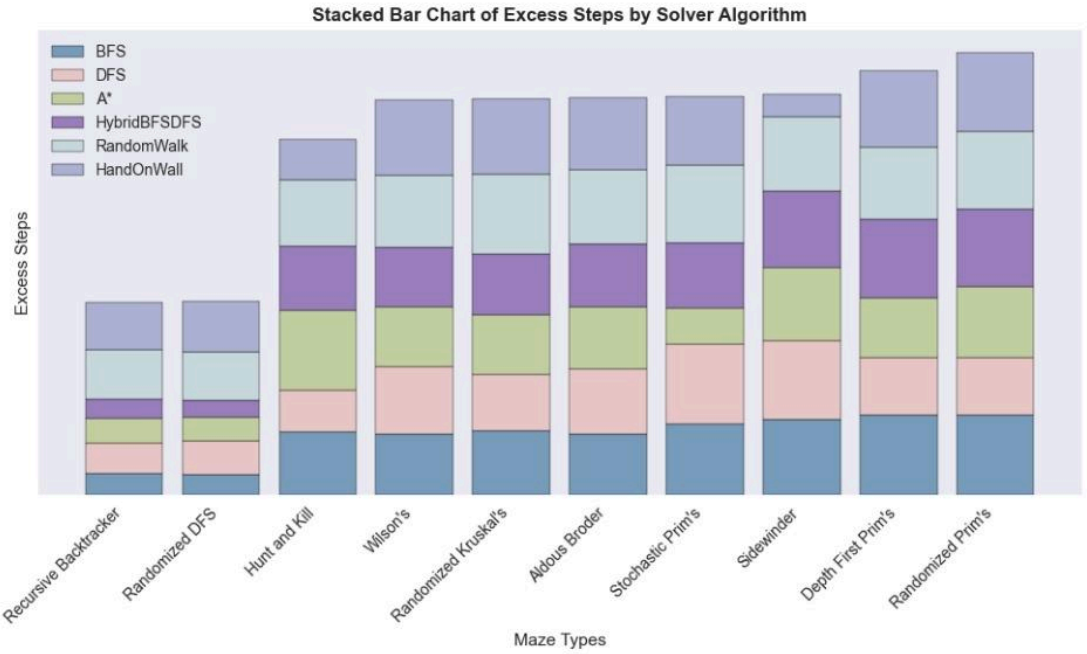


Figure 11: Stacked Bar Chart of Excess Steps by Solver Algorithm

On the DQN side, before analyzing the results, a short explanation about the statistics can be useful. Thinking of the difficulty of a maze, harder mazes should take more effort to learn, while the algorithm should learn the easiest mazes with less effort. Thus, the converged epoch number will be used as the main metric. The other statistics, converged epoch number divided by the number of dead ends, the converged epoch number divided by the average Manhattan distance from dead ends to solution, the converged epoch number divided by the average dead end reward, and the converged epoch number divided by solution reward will not be used directly for comparison. They are useful to see the pattern, but on the comparison side, they may create some bias due to the division operation.

A spider plot can be useful to show the DQN comparison. It is created based on the converged epoch number of every maze generation algorithm. As can be seen in Figure 12, the algorithm converged with the lowest number of epochs with Hunt and Kill, Recursive Backtracker, and Randomized DFS, whereas it requires more epochs to converge for

Randomized Kruskal, Depth First, and Randomized Prim. The results of Wilson and Aldous Broader are a bit surprising, but these two have similar backgrounds, so it might be due to the working principle. Other than this, the general result applies. In the end, mazes with longer paths and fewer intersections are easier, while mazes with shorter paths and higher intersections are harder for agents to solve.

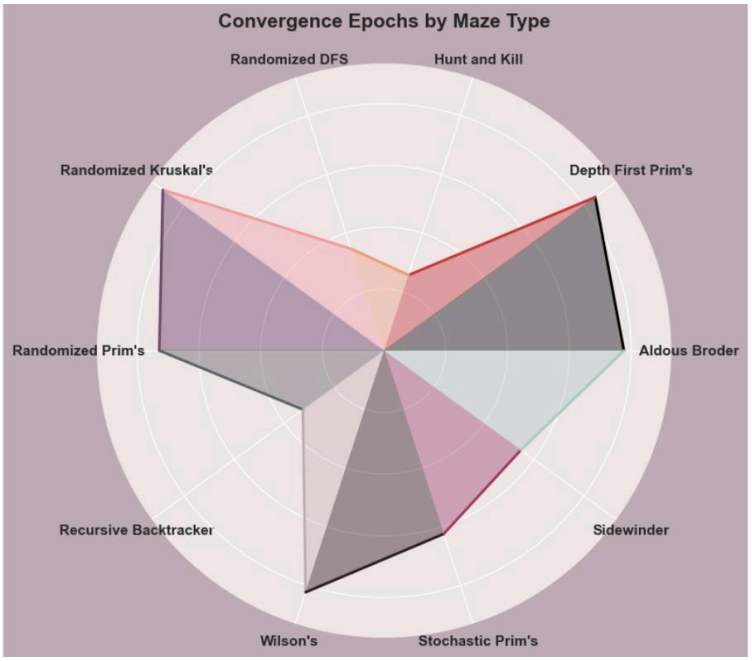


Figure 12: The Coverged Epoch Number by Maze Type

As it is mentioned in the previous sections, another research question is whether having a loop makes a maze easier or harder. For this purpose, mazes are generated with two different generation algorithms with distinct backgrounds, Hunt and Kill, and Prim. The amount of loops is designated by loop density, zero implying there is no loop, and as the density increases, the number of loops increases. The results of this run are provided in the Table 20. In this part, the comparisons are made based on the visited vertices statistic since the solution path length is not clear in mazes with loops. Hand on Wall is not used in this part as an agent because when it enters a loop, it may not go out due to its principle. As can be seen in the Table 20, BFS and Random Walk are not good at solving loops. They do not work in the most efficient manner, so the result can be expected as such. However, A* and Hybrid Search solves mazes with loops easily. Also, as the density increases, meaning there are more loops, mazes become easier. DFS does not have that clear pattern, but it solves Prim's mazes with loops in an easier way. The case with the Hunt and Kill might be the result of the structure of

the maze generation algorithm. Since A* tries to minimize the total distance travelled and Hybrid Search has the best sides of BFS and DFS, they can be used as the main results. These two agents give more intuition of hardness since they are trying to solve in a better way. Therefore, in this part, it can be said that, in general, mazes with loops are easier to solve for proper agents. Once the agent enters loops, there will be more options to go out and return to the solution path. Also, there might be more than one solution in this case. Thus, with a proper agent, they can be solved easily.

	Solver	A*	BFS	DFS	HybridBFSDFS	RandomWalk
Maze Type	**Loop Density**					
Loop Hunt and Kill	0.00	486.89	555.12	318.09	416.80	353.85
	0.01	479.85	557.48	339.93	406.57	361.12
	0.10	466.58	584.26	335.05	373.48	357.05
	0.25	429.62	609.10	326.75	312.67	379.04
Loop Prim	0.00	395.60	621.10	348.95	436.00	338.72
	0.01	344.36	621.68	365.17	411.22	334.93
	0.10	355.39	622.28	303.94	385.83	354.95
	0.25	324.95	622.87	310.30	340.67	351.67

Table 20 : The Results of Mazes with Loops

Until this part, the comparisons are made from the agents' perspective. The purpose was to see for agents in general which kinds of mazes are more difficult. This discussion can be expanded by asking if one has a maze generated by this algorithm, which agent should one use? To answer this question and reach a two-way comparison, for every maze generation algorithm, the performance of the agents will be compared. The numeric results based on the excess step metric are provided in Appendix 2.

Starting with Depth First, Randomized Prims, and Wilson's algorithms, Random Walk has the best performance among these three, and also, Hand on Wall has a performance close to this. Sidewinder has a special structure, and thanks to this, Hand On Wall performs extremely well. The agent Hand On Wall works well in some other types of mazes, too, which are Aldous Broder, Randomized Kruskal, and Hunt and Kill. As it is understood, it has a domination among other agents, and it was an expected result. This agent was used in racing in Japan until mazes with loops were introduced. It is good in practice since dead ends are

easy to deal with, and it does not require making a decision. It only follows a route directly. Continuing with maze types, as a special case, A* solves mazes generated by Stochastic Prim easily. Lastly, Hybrid Search can be used to solve mazes generated by Recursive Backtracker and Randomized DFS. These two have the longest paths with fewer intersections, so the combination of BFS and DFS working principle works well with them.

The last suggestion is about the special case, mazes with loops. Table 20 can be read horizontally for this purpose. Every row represents a maze type that is solved by different agents and results of statistic visited nodes is also provided. For the same maze, since the properties of the maze is the same, comparing the visited number of nodes gives the idea of performance. Nearly in kinds of mazes, Depth First Search (DFS) has the best performance. The result is not a coincidence since loops make paths longer by adding dead ends into the paths, and DFS solves mazes with longer paths better, the result is observed. Therefore, for mazes with loops DFS is a good agent to use in practice.

The summary of the suggestions can be seen in the Table 21:

Maze Generation Algorithm	***The Best Solving Agent***
Randomized Prim	Random Walk & Hand on Wall
Depth First Prim	Random Walk & Hand on Wall
Sidewinder	Hand on Wall
Randomized Kruskal	Hand on Wall
Aldous Broder	Hand on Wall
Wilson's	Hand on Wall
Hunt and Kill	Hand on Wall
Stochastic Prim	A*
Recursive Backtracker	Hybrid Search
Randomized DFS	Hybrid Search
Mazes with Loops	DFS

TABLE 21: The Best Solving Agent for Every Maze Generation Algorithm

7. Conclusion and Future Work

7.1 Summary of Findings

This project set out to systematically examine how different maze generation algorithms produce structurally unique mazes and how these structures affect the performance of various maze-solving agents to assess difficulty levels of mazes. Through the implementation of ten classical and modified maze generation algorithms such as Kruskal, Prim, Sidewinder, Randomized DFS, as well as multiple solving strategies including rule-based agents, graph search methods, and a Deep Q-Network (DQN). The study explored both the generation and solution process in depth.

A well defined set of maze metrics was developed and applied across 100 maze instances per algorithm, such as the number of intersections and dead ends, solution length, tortuosity, turn count, and several distance-based metrics (e.g., average dead-end distance from solution path, start point, and intersections). These metrics showed a multidimensional portrait of maze complexity.

Results revealed clear patterns of generation algorithms. Algorithms like Recursive Backtracker, Randomized DFS, and Hunt and Kill created mazes with long, solution paths with many turns, low intersection density, and deep dead ends resulting in structurally sparse but deeper misleading mazes. On the contrary, Prim-based, Wilson's, and Kruskal's algorithms produced shorter paths with more intersections and shallow dead ends, leading to more frequent but less deep dead ends as misleading components.

A Principal Component Analysis (PCA) was performed to reduce dimensionality and determine the contributors to structural variance. The analysis showed that over 80% of the variance could be captured in a single component, which closely resembled the distribution of the Average Distance to Dead Ends (from Solution Path). This suggests that this metric is especially powerful in characterizing maze complexity and distinguishing between generation strategies.

On the solver side, performance metrics to measure difficulty were defined. Such as excess steps, excess ratio, dead-end coverage, and convergence epochs for Deep Q Learning agent (DQN). Solvers like BFS, DFS, and Hybrid Search performed best on mazes with fewer

intersections and deeper dead ends, such as those produced by Recursive Backtracker and Randomized DFS. Prim-based mazes posed a higher difficulty due to their higher branching rates with shorter path lengths and high intersection counts.

The DQN agent was applied to mazes with the size of 10×10 due to computational limits. DQN solver converged more quickly on mazes with fewer intersections, and struggled on dense, deceptive layouts like those generated by Depth First Prim and Randomized Prim.

Lastly, a separate analysis on looped mazes showed that increasing loop density tends to decrease solution path length with multiple solution paths emerging and also decrease solver difficulty especially for agents like A Star and Hybrid Search while increasing the number of intersections.

In summary, this project clearly demonstrates how maze structure shaped by different generation algorithms directly influences solving difficulty and agent performance. By combining detailed structural maze analysis with solver benchmarking, we can make conclusions about the performance of solvers relative to the different maze topologies produced by different maze generation algorithms. These pair-wise combinatorial analysis gives us insights which not only deepen our understanding of algorithmic maze generation but also provides practical guidance for designing solvers or mazes tailored to specific use cases and difficulty levels.

7.2 Potential Extensions

While this project offers a wide-ranging study of solving and generating mazes, there are several directions remain open for future research and extension:

Human-centered evaluation: Experiments with human participants could deepen our understanding of maze difficulty by introducing a cognitive and perceptual standpoint by acting like a distinct myopic heuristic algorithm. Comparing human performance with solving agents could be beneficial for comparison purposes and further pattern analysis.

Adaptive and controlled generation: With the outputs of this project such as PCA results and key metrics like dead-end depth future work could include generating mazes with different difficulty levels respective to different solving agents as the main purpose of the project. If

human solvers are to be introduced, this controlled generation process can have applications in educational tools and game level designs.

Scaling up and generalization: Extending the current study to larger mazes and mazes with different shapes or extending to 3d environments, could reveal new structural effects and solver behaviors.

Interactive visualization and explainability: Developing an interactive tool that visualizes maze metrics within the maze in real time during generation or solving could improve the coherency of the analysis, especially for educational or research purposes.

Expanding Deep Q-Learning: The main purpose of the DQN in this application is to train an agent capable of estimating the optimal policy in any type of maze. Therefore, it was implemented in a generalized manner. A natural extension of this work involves optimizing the parameters to minimize the number of training epochs required for different maze types. This can be achieved either by training the agent to tune its own parameters dynamically or by selecting appropriate parameters through trial and error beforehand.

References

[1] Maze. (2025, May 4). In Wikipedia. https://en.wikipedia.org/wiki/Maze

[2] J. (n.d.). The Winding History of Mazes: A Journey Through Time and Culture. Life Science Art. https://www.lifescienceart.com/art/architecture/the-winding-history-of-mazes-from-ancient-labyrinths-to-modern-day-masterpieces/

[3] Geiling, N. (n.d.). The Winding History of the Maze. Smithsonian Magazine. https://www.smithsonianmag.com/travel/winding-history-maze-180951998/

[4] Ramadhian, Fauzan. (2013). Implementation of Prim's and Kruskal's Algorithms' on Maze Generation.

[5] P. Gabrovsek, Analysis of Maze Generating Algorithms, University of Ljubljana, 2017.

[6] T. Kurokawa, Maze Construction by Using Characteristics of Eulerian Graphs, Aichi Institute of Technology, 2015.

[7] M. S. McClendon, The Complexity and Difficulty of a Maze, University of Central Oklahoma, 2001.

[8] M. Foltin, Automated Maze Generation and Human Interaction, Faculty of Informatics, Masaryk University, 2011.

[9] Yarkov, A. (2023, September 6). Crafting Mazes with Graph Theory. DEV Community. https://dev.to/optiklab/crafting-mazes-2dia

[10] Buck, J. (2011, February 7). Maze generation: Algorithm recap. Buckblog. https://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap

[11] Nolan, E,. (n.d.). Maze Generation Algorithms – An Exploration. GitHub Pages. https://professor-l.github.io/mazes/

[12] Kroudir, N.(2023, July2). Randomized Depth-First-Search Algorithm for Maze Generation.

https://medium.com/@nacerkroudir/randomized-depth-first-search-algorithm-for-maze-generation-fb2d83702742

[13] Milco, E. (n.d.). PathFinder: Algorithms for Making and Solving Mazes. GitHub Pages. https://emmilco.github.io/path_finder/

[14] Maze-solving algorithm. (2025, April 16). In Wikipedia. https://en.wikipedia.org/wiki/Maze-solving_algorithm

[15] (n.d.). Breadth First Search or BFS for a Graph. Geeks for Geeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[16] (n.d.). Depth First Search or DFS for a Graph. Geeks for Geeks. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

[17] A* search algorithm. (2025, May 27). In Wikipedia. https://en.wikipedia.org/wiki/A*_search_algorithm

[18] [Veritasium]. (2023, May 24). The Fastest Maze-Solving Competition On Earth [Video]. Youtube. https://www.youtube.com/watch?v=ZMQbHMgK2rw&ab_channel=Veritasium

[19] Kaya, E. (2023). DQN-on-maze [Source code]. GitHub. https://github.com/Ezgii/DQN-on-maze

[20] Nicoletti, G. (2020). deep_Q_learning_maze [Source code]. GitHub. https://github.com/giorgionicoletti/deep_Q_learning_maze

[21] Dhumne, S. (n.d.). Deep Q-Network (DQN). Medium. https://medium.com/@shruti.dhumne/deep-q-network-dqn-90e1a8799871

[22] Pandey, A. K. (n.d.). SiLU (Sigmoid Linear Unit) activation function. Medium. https://medium.com/@akp83540/silu-sigmoid-linear-unit-activation-function-d9b6845f0c81

[23] Bellman, R. (1957). A Markovian decision process. Journal of Mathematics and Mechanics, 6(5), 679–684.

[24] Hausknecht, M. J., & Stone, P. (2015). Deep recurrent Q-learning for partially observable MDPs. arXiv preprint arXiv:1507.06527. https://arxiv.org/abs/1507.06527

[25] Bellman, R. (1952). On the theory of dynamic programming. Proceedings of the National Academy of Sciences, 38(8), 716–719. https://doi.org/10.1073/pnas.38.8.716

Appendices

A. Code Snippets

```
import random
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

class RandomizedKruskalMaze:
    def __init__(self, width, height):
        self.maze_name = "Randomized Kruskal's Algorithm"
        self.width = width
        self.height = height
        self.graph = nx.grid_2d_graph(width, height)
        self.maze = nx.Graph()
        self.grid = np.ones((self.height * 2 + 1, self.width * 2 + 1))
        self.frames = []
        self.start = (0, 0)
        self.end = (self.width - 1, self.height - 1)
        self._generate_maze()

    def _generate_maze(self):
        edges = list(self.graph.edges())
        random.shuffle(edges)

        cell_sets = {cell: {cell} for cell in self.graph.nodes()}

        def find_set_root(cell):
            for root, cells in cell_sets.items():
                if cell in cells:
                    return root
            return None

        for edge in edges:
            cell1, cell2 = edge
            root1, root2 = find_set_root(cell1), find_set_root(cell2)

            if root1 != root2:
                self.maze.add_edge(cell1, cell2)
                cell_sets[root1].update(cell_sets[root2])
                del cell_sets[root2]

                self._update_grid(cell1, cell2)
                self.frames.append(self.grid.copy())
```

```
def _update_grid(self, cell, neighbor):
    x1, y1 = cell
    x2, y2 = neighbor
    grid_x1, grid_y1 = x1 * 2 + 1, y1 * 2 + 1
    grid_x2, grid_y2 = x2 * 2 + 1, y2 * 2 + 1
    self.grid[grid_y1][grid_x1] = 0
    self.grid[grid_y2][grid_x2] = 0
    self.grid[(grid_y1 + grid_y2) // 2][(grid_x1 + grid_x2) // 2] = 0

def draw(self):
    plt.figure(figsize=(8, 8))
    plt.imshow(self.grid, cmap="binary", interpolation="nearest")

    start_x, start_y = self.start
    end_x, end_y = self.end
    grid_start_x, grid_start_y = start_x * 2 + 1, start_y * 2 + 1
    grid_end_x, grid_end_y = end_x * 2 + 1, end_y * 2 + 1
    plt.scatter(grid_start_x, grid_start_y, color="red", s=100, zorder=10, label="Start")
    plt.scatter(grid_end_x, grid_end_y, color="green", s=100, zorder=10, label="End")

    plt.legend(loc="upper right", bbox_to_anchor=(1.15, 1))
    plt.xticks([]), plt.yticks([])
    plt.show()
```

```
def animate_maze(self, save=False):
    fig, ax = plt.subplots(figsize=(6, 6))

    fig.patch.set_facecolor("#B7D3DF")
    cmap = plt.cm.binary
    im = ax.imshow(self.frames[0], cmap=cmap, animated=True)

    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_frame_on(False)

    def update(frame):
        im.set_array(frame)
        return [im]

    ani = animation.FuncAnimation(fig, update, frames=self.frames, interval=50, repeat=False)

    if save or len(self.frames) > 1000:
        print("Saving the animation...")
        ani.save(f"Outputs/Generation of {self.maze_name}{self.width}_{self.height}.mp4",
                 writer="ffmpeg", bitrate=1000, fps=len(self.frames) / (len(self.frames) * 0.05))
        plt.close(fig)

    else:
        html_anim = HTML(ani.to_jshtml())
        plt.close(fig)
        return html_anim
```

```
def visualize_maze_as_graph(self):
    plt.figure(figsize=(6, 6))
    pos = {(x, y): (x, -y) for x, y in self.graph.nodes()}
    nx.draw(self.maze, pos, with_labels=True, node_color='lightblue', node_size=500, edge_color='black')

    plt.scatter(self.start[0], -self.start[1], color="red", s=100, zorder=10, label="Start")
    plt.scatter(self.end[0], -self.end[1], color="green", s=100, zorder=10, label="End")

    plt.legend(loc="upper right", bbox_to_anchor=(1.15, 1))
    plt.xticks([]), plt.yticks([])
    plt.show()

def draw_solution(self):
    plt.figure(figsize=(8, 8))
    plt.imshow(self.grid, cmap="binary", interpolation="nearest")

    start_x, start_y = self.start
    end_x, end_y = self.end
    grid_start_x, grid_start_y = start_x * 2 + 1, start_y * 2 + 1
    grid_end_x, grid_end_y = end_x * 2 + 1, end_y * 2 + 1
    plt.scatter(grid_start_x, grid_start_y, color="red", s=100, zorder=10, label="Start")
    plt.scatter(grid_end_x, grid_end_y, color="green", s=100, zorder=10, label="End")

    solution_path = nx.shortest_path(self.maze, self.start, self.end)

    if solution_path:
        for i in range(1, len(solution_path)):
            prev = solution_path[i - 1]
            curr = solution_path[i]
            path_x1, path_y1 = prev[0] * 2 + 1, prev[1] * 2 + 1
            path_x2, path_y2 = curr[0] * 2 + 1, curr[1] * 2 + 1
            plt.plot([path_x1, path_x2], [path_y1, path_y2], color="green", linewidth=2, zorder=5)

    plt.legend(loc="upper right", bbox_to_anchor=(1.15, 1))
    plt.xticks([]), plt.yticks([])
    plt.show()

def to_grid(self):
    return self.grid

def to_graph(self):
    return self.maze
```

Code 1. Randomized Kruskal's Algorithm Generation Code

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

class HandOnWallSolver:
    def __init__(self, maze):
        self.maze = maze.maze
        self.maze_name = maze.maze_name
        self.grid = maze.grid
        self.start = maze.start
        self.end = maze.end
        self.width = maze.width
        self.height = maze.height
        self.wall_frames = []
        self.visited = set()
        self.solution_path = self._solve()

    def _solve(self):
        current = self.start
        path = [current]
        visited = set()

        wall_grid = np.zeros((self.height * 2 + 1, self.width * 2 + 1, 3))
        wall_grid[self.grid == 1] = [0, 0, 0]
        wall_grid[self.grid == 0] = [1, 1, 1]

        while current != self.end:
            visited.add(current)

            wall_grid[current[1] * 2 + 1, current[0] * 2 + 1] = [0.0, 0.290, 0.678]
            if len(path) > 1:
                prev = path[-2]
                wall_grid[(current[1] + prev[1]) + 1, (current[0] + prev[0]) + 1] = [0.0, 0.290, 0.678]

            current_grid = wall_grid.copy()
            current_grid[current[1] * 2 + 1, current[0] * 2 + 1] = [1, 0, 0]
            self.wall_frames.append(current_grid)

            neighbors = [n for n in self.maze.neighbors(current) if n not in visited]
```

```
        directions = [(0, -1), (1, 0), (0, 1), (-1, 0)]
        current_to_directions = [(current[0] + d[0], current[1] + d[1]) for d in directions]

        if neighbors:
            if current_to_directions[0] in neighbors:
                current = current_to_directions[0]
            elif current_to_directions[1] in neighbors:
                current = current_to_directions[1]
            elif current_to_directions[2] in neighbors:
                current = current_to_directions[2]
            elif current_to_directions[3] in neighbors:
                current = current_to_directions[3]
            path.append(current)
        else:
            if len(path) > 1:
                path.pop()
                current = path[-1]
            else:
                break

    self.visited = visited
```

```
    solution_grid = np.zeros((self.height * 2 + 1, self.width * 2 + 1, 3))
    solution_grid[self.grid == 1] = [0, 0, 0]
    solution_grid[self.grid == 0] = [1, 1, 1]

    for node in path:
        solution_grid[node[1] * 2 + 1, node[0] * 2 + 1] = [0, 1, 0]
    for i in range(1, len(path)):
        prev = path[i - 1]
        curr = path[i]
        solution_grid[(curr[1] + prev[1]) + 1, (curr[0] + prev[0]) + 1] = [0, 1, 0]
    self.wall_frames.append(solution_grid)

    return path

def animate(self, save=False):
    fig, ax = plt.subplots(figsize=(6, 6))

    fig.patch.set_facecolor("#AACFC3")
    cmap = plt.cm.binary
    im = ax.imshow(self.wall_frames[0], cmap=cmap, animated=True)

    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_frame_on(False)

    def update(frame):
        im.set_array(frame)
        return [im]

    ani = animation.FuncAnimation(fig, update, frames=self.wall_frames, interval=50, repeat=False)

    if save or len(self.wall_frames) > 1000:
        print("Saving the animation...")
        ani.save(f"Outputs/Hand on Wall Solution of {self.maze_name}_{self.width}_{self.height}.mp4",
                 writer="ffmpeg", bitrate=1000, fps=len(self.wall_frames) / (len(self.wall_frames) * 0.05))
        plt.close(fig)
    else:
        html_anim = HTML(ani.to_jshtml())
        plt.close(fig)
        return html_anim
```

Code 2. Hand-on-Wall Solver Code

You can access the rest of the code from here.

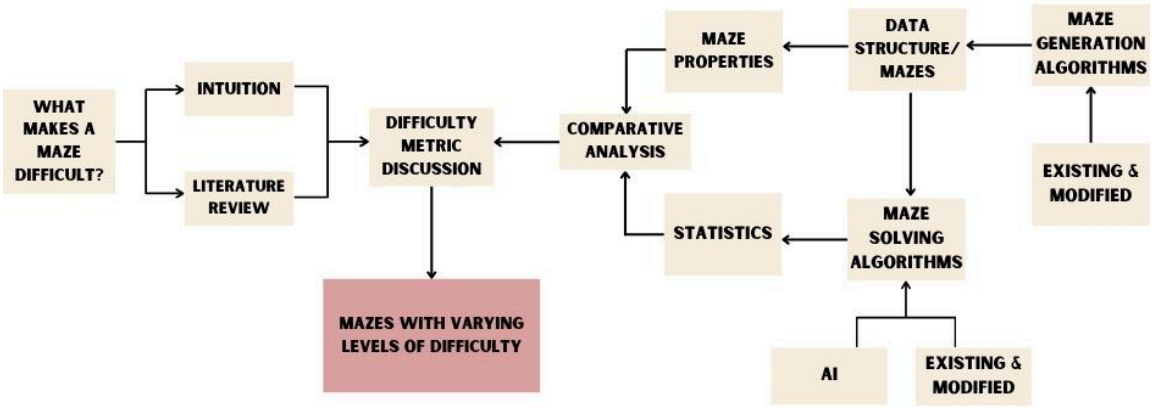
https://lmfaraday.github.io/Maze-Generation-Algorithms-Using-Graph-Theory/

B. Example Outputs

You can access an example visualization for each generator and solver through this link.

https://lmfaraday.github.io/Maze-Generation-Algorithms-Using-Graph-Theory/MazeAnimations/animation.html

C. Additional Figures or Tables



Appendix 1 : Context Diagram

Solver	A*	BFS	DFS	HandOnWall	HybridBFSDFS	RandomWalk
Maze Type						
Randomized Prim's	343.90	569.40	297.25	291.11	384.30	287.02
Depth First Prim's	290.26	568.21	298.01	280.45	387.91	269.43
Sidewinder	355.45	537.68	407.25	82.48	378.43	275.86
Stochastic Prim's	176.95	509.85	411.07	251.90	321.38	288.89
Aldous Broder	299.10	436.38	339.54	265.45	306.74	279.34
Randomized Kruskal's	291.11	458.05	289.68	278.48	300.89	296.08
Wilson's	291.10	430.31	353.63	273.76	291.66	269.38
Hunt and Kill	384.25	452.48	215.45	149.41	314.16	251.21
Randomized DFS	111.69	141.53	177.15	186.56	83.20	180.12
Recursive Backtracker	123.19	149.29	158.21	171.30	93.44	184.49

Appendix 2 : The Combination of Results Based on Excess Steps

Maze Type	# of Intersections	Solution Length	Shortest Possible Solution	Solution Path Tortuosity	# of Deadends	Deadend Crossroads	Avg Distance to Deadend(From Solution Path)	Max Distance to Deadend(From Solution Path)	Avg Distance to Deadend(From Intersection)	Tempting Count	Avg Intersection Degree	Avg Distance to Deadend(From Solution Path) Tortuosity
Aldous Broder	157.97	81.10	48.0	1.69	179.53	102.21	42.33	114.26	46.36	44.71	3.13	2.71
Depth First Prim's	198.27	51.10	48.0	1.06	248.31	171.82	31.37	84.30	34.43	45.75	3.25	2.11
Hunt and Kill	58.39	107.10	48.0	2.23	60.21	18.15	69.39	183.65	73.05	16.38	3.02	4.86
Randomized DFS	61.86	230.80	48.0	4.81	63.47	31.85	137.43	338.74	122.65	17.61	3.02	10.80
Randomized Kruskal's	163.39	75.16	48.0	1.57	187.85	107.12	38.09	99.01	41.30	44.78	3.15	2.49
Randomized Prim's	168.94	51.32	48.0	1.07	198.21	113.49	32.61	86.36	35.22	38.05	3.16	2.20
Recursive Backtracker	62.03	216.70	48.0	4.51	63.56	31.30	139.93	333.52	122.41	16.57	3.02	10.72
Sidewinder	153.60	59.36	48.0	1.24	171.00	90.54	39.06	99.52	42.29	30.59	3.11	2.44
Stochastic Prim's	174.94	48.02	48.0	1.00	196.78	112.74	38.15	100.53	42.10	25.69	3.12	2.02
Wilson's	160.10	77.88	48.0	1.62	182.68	103.80	41.24	111.34	45.53	43.72	3.13	2.64

Appendix 3: Maze Metric Results