Efficient Maze Solving: A Dataset Driven Approach

By

Harshvardhan Rajpurohit - Faculty of Graduate studies (hrx407@uregina.ca)

Dhruv Patel - Faculty of graduate studies (dpf528@uregina.ca)

Akshat Sharma - Faculty of graduate studies (asp700@uregina.ca)

This project presents a methodology for maze solving by using techniques in deep learning and machine learning. The main goal is to develop a robust maze solver, capable of efficiently navigating through diverse maze structures using a dataset driven approach. Traditional maze solving algorithms often struggle with complex mazes, providing the need for more adaptive and intelligent solutions. In this study, we propose a method that uses neural networks and machine learning algorithms to efficiently better the maze solving process.

The proposed maze solver is set to be trained on a curated dataset of diverse mazes, having various complexities and dimensions. Deep learning models, such as CNNs and RNNs, will be employed to extract intricate patterns and spatial relationships within the maze structures and reinforcement learning is to be integrated to facilitate learning and decision making during navigation. We also aim to introduce constraints in the dataset which makes it difficult for the solver to reach the goal.

Experimental results would be in the form of efficacy and efficiency of the proposed maze solver in successfully navigating through the mazes. This project aims to the advancement of maze solving techniques by providing a foundation for the development of intelligent systems that are capable of autonomously navigating through complex environments represented by a maze in our case. The findings hold implications for various applications, including robotics, autonomous vehicles, and path planning in dynamic environments.

Keywords: Spatial relationships, Maze layouts, Intelligent systems, Reinforcement learning, Machine learning.

Contents

Introduction

In the last few decades, Artificial Intelligence and Machine Learning has made a remarkable contribution in field of computer science and robotics. Maze solving is a fundamental problem in Computer Science, there are algorithms like BFS, DFS, Flood Fill Algorithm, Dijkstra algorithm and few more. In this project we planned to solve a maze using Reinforcement Learning algorithm.

Maze solving, as a paradigmatic problem in reinforcement learning, has garnered considerable attention due to its relevance in understanding and advancing AI capabilities. This problem serves as a perfect testbed for assessing the effectiveness of reinforcement learning algorithms since it captures fundamental ideas like reward maximization, exploration-exploitation trade-offs, and policy optimization. Researchers examine novel strategies for dealing with problems in decision-making, learning, and adaptability through the lens of maze solving; the findings have implications for a wide range of applications, including robotics, gaming, and optimization.

Solving a maze, with some constraints acts as a base for making complex robots or drones, which can navigate through difficult paths by their own for example Mars Rover. For instance, Lin, Shih-Wei, Huang, Yao-Lin, and Hsieh, Wen-Keui (2019) presented a study on solving the maze problem using reinforcement learning by a mobile robot [1]. In their work, they demonstrated the practical implementation of reinforcement learning algorithms for maze navigation tasks, highlighting the potential for real-world applications in robotics and autonomous systems.

In this project, the objective is to generate a random maze using 3 algorithms namely backtracking, Kruskal's algorithm and Prim's algorithm and then solving it using reinforcement learning techniques. There are constraints like shortest-path and wall density which our agent would be dealing with.

1. Methodology

1.1 Reinforcement Learning

A machine learning paradigm known as reinforcement learning (RL) [4] [5] [6] teaches an agent to make successive decisions by interacting with its surroundings in a way that maximizes cumulative rewards. Unlike supervised learning, it gives the agent feedback for every action it takes in the form of rewards or penalties instead of directly instructing it on what to do with each input. Using previously acquired knowledge and environment exploration, reinforcement learning algorithms, such Q-learning and policy gradient approaches, seek to identify the best course of action. This survey paper by Kaelbling, Littman, and Moore (1996) provides a comprehensive overview of reinforcement learning techniques, highlighting key concepts, algorithms, and applications in artificial intelligence research.

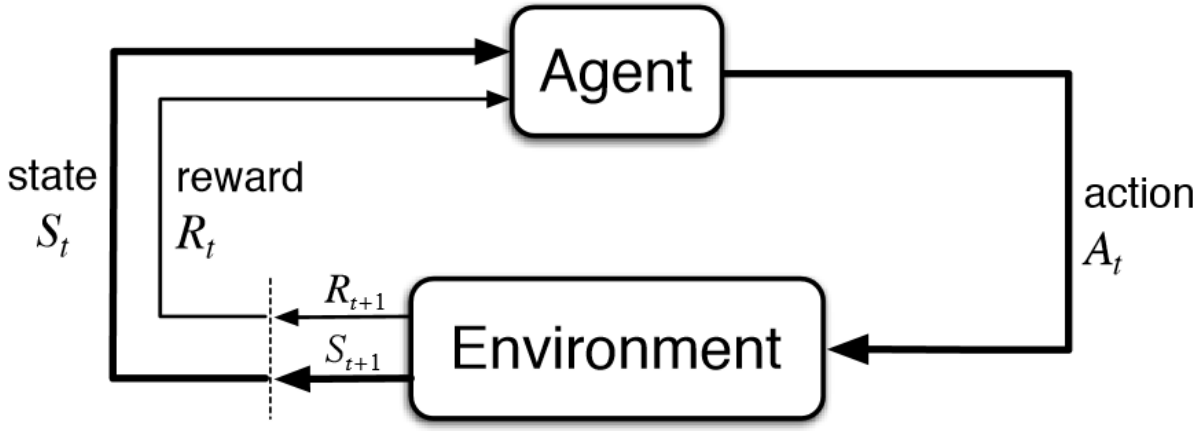


Figure 1: Reinforcement Learning

In above figure, agent can be considered as our computer program, it will perform a certain action for going from current state to next state. Reinforcement learning is a supervised learning algorithm, when an agent reaches the correct next state as intended, we reward them and when it reaches a wrong state we would penalize it with a negative reward value, this way the agent learns through the idea of obtaining maximum reward with every steps it takes when traversing the maze

Reinforcement Learning algorithm are built on adaptive framework, they learn more as they interact with the environment. If environment is changed RL algorithm adapts itself accordingly [9].

1.2 Q-Learning

By making the correct decisions over time, a model can learn repeatedly and improve thanks to a machine learning process called Q-learning. Q-learning is a type of reinforcement learning that uses a Q-matrix, for the representation of possible pathways, and a function that essentially drives the calculation of rewards based on the current state, the previous state and the final destination state

Machine learning models are trained using reinforcement learning to learn in a manner similar to that of children or animals. Positive behavior is rewarded or reinforced, while negative behavior is discouraged and penalized.

The state-action-reward-state-action method of reinforcement learning is used in the training regimen to assist the learners in imitating the appropriate actions. Q-learning provides a model-free approach to reinforcement learning. The absence of an environment model does not affect reinforcement learning. Based on interactions from the AI component interacting with the environment, called the agent, it develops its own hypotheses about the world.

Q-learning also approaches reinforcement learning from an off-policy perspective. Selecting the best course of action given the current situation is the aim of a Q-learning technique. This can be accomplished using the Q-learning strategy by either creating new guidelines of its own or by

departing from the recommended path of action. A precise policy is not required because Q-learning has the ability to stray from instructions.

1.3 Working of Q-Learning

Q-Learning is an iterative approach [7], where multiple components work together to help train a model. We define an epoch value that essentially highlights the number of iterations the agent will take in a single training session where in the agent is dropped at a random location in the maze and aims to find a path to the destination node.

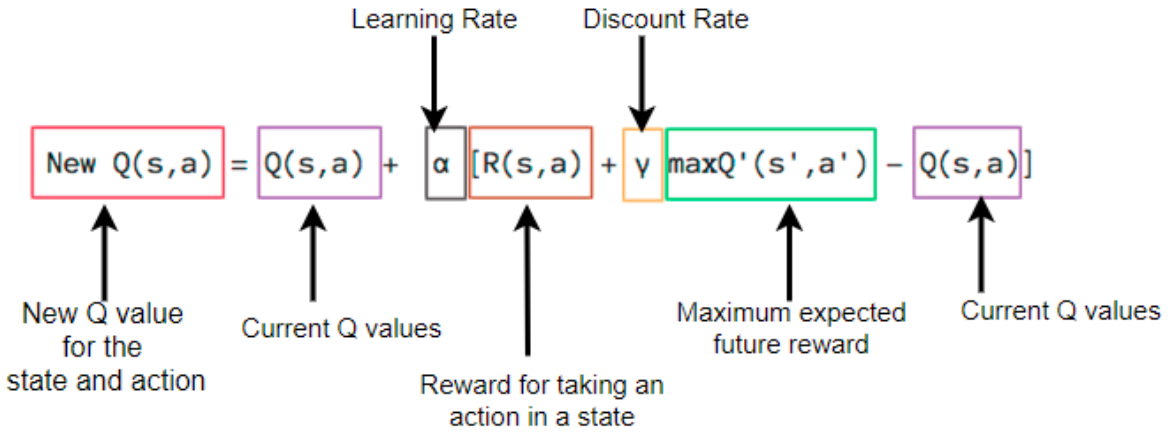
Components of Q-Learning:

A. Agent – It is an entity that acts or operates in the environment.
B. State – It is the current position of agent in the environment.
C. Actions – It is the operation performed by Agent in environment to go from one state to another.
D. Rewards – It is a value that an agent receives after going from one state to another. If agent goes to a wrong state, we assign a negative value and if goes to right state, we assign a positive reward.
E. Q-Value – It is a metric used to measure an action at a particular state. Q-value is determined by Bellman's equation.

Q-learning models learn the best behavior for a task through a series of trial-and-error encounters. Learning an optimal action value function, or Q-function, is the first step in the Q-learning process, which involves modeling optimal behavior. In states, this function represents the optimal long-term value of action a; it then behaves optimally in all following states.

Bellman's equation

The expected value of a state is represented by the value function. The value function of a state is determined by the current agent's policy and is the total of the rewards that will be earned when the agent advances to the next one. The relationship between the "value function of the present state and the value function of the next state" is expressed by the Bellman equation, which stands in for the policy [8].



The equation breaks down as follows:

- *Q(s, a)* represents the expected reward for taking action *a* in state *s*.
- The actual reward received for that action is referenced by *r* while *s′* refers to the next state.
- The learning rate is α and γ is the discount factor.
- The highest expected reward for all possible actions *a′* in state *s′* is represented *by max(Q(s′, a′))*.

Q-Table:

Q-table essentially highlights the rewards that the agent will receive when performing an action. The rows highlight the value that agent might encounter during its searching/traversing phase while the columns highlight the possible actions that the agent can choose through. Initially, all the cells of the Q-table are assigned 0 values, as the agent learns traversing through the maze, the values in the cells are updated. At the end of the training phase, the values of the Q-table essentially highlights all the best possible actions for the possible states that the agent can take when put in that situation.

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 2: Q-Matrix

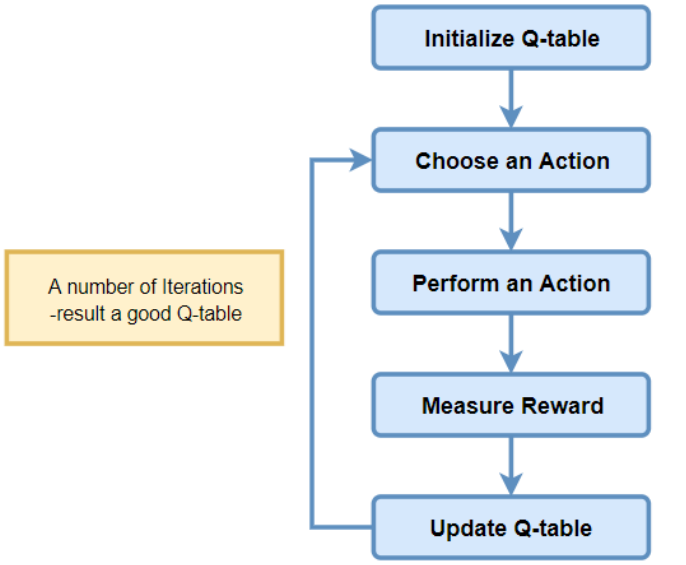


Figure 3: Q-Table

F-matrix:

F-matrix is a n^2 x n^2 matrix, where n is the number of rows or number of columns in input maze/array. Each cell in the F-matrix represents whether or not there exists a path between the cells highlighted by the corresponding row and the column. For example, F[1][2] highlights the relation between the cells 1 and 2. If the value of this F[1][2] = 1, this indicates the existence of a path between the cells.

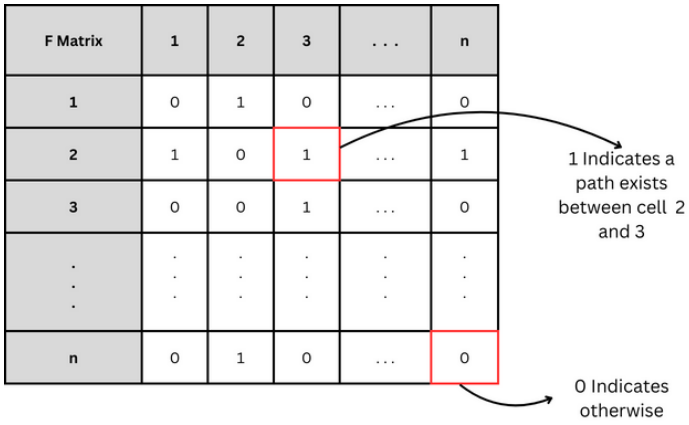


Figure 4: F-Matrix

2. Generating Maze:

This project propose the generation of the maze before using the Q-learning algorithm to bring a solution to it. While predefined mazes can be considered for the project, additional steps to convert the provided maze into the required representation are required. To keep us from all the hassle, we decided to generate our own mazes and providing the user the freedom of choosing the algorithm for their generation. 3 algorithms namely, Backtracking algorithm, Prim's algorithm and Kruskal's algorithm were incorporate to achieve the same.

Starting point can be any random maze cell from first 2 rows of maze, and ending point can be any random cell from last two rows of maze.

2.1 Backtracking algorithm for Maze Generation:

The backtracking algorithm [10] reverses direction while carefully examining every possible path in a labyrinth grid and coming into a dead end. Starting from a starting cell, the algorithm decides at random which directions to move in, breaking down walls to form tunnels between adjacent cells. It proceeds until it encounters an obstacle where it cannot proceed any further without breaching certain guidelines, such as creating loops or severing portions of the maze. This type of limitation is detected by the algorithm, which then returns to the previous intersection where alternatives paths were present and explores a different route. Once every cell, that can be reached, has been visited through this process of exploring and retracing your steps, there is only one way to navigate the maze from start to finish. Walls are denoted by the value 1 while passage or pathways are denoted by 0 values.

```python
# Function to generate the maze using bt
def generate_maze_bt(width, height, start, end, wall_density=0.3):
    maze_grid = [[0 for _ in range(width)] for _ in range(height)]

    start_x, start_y = start
    end_x, end_y = end

    maze_grid[start_y][start_x] = 1
    maze_grid[end_y][end_x] = 1

    def backtrack(x, y):
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        random.shuffle(directions)

        for dx, dy in directions:
            new_x, new_y = x + 2 * dx, y + 2 * dy

            if 0 <= new_x < width and 0 <= new_y < height and maze_grid[new_y][new_x] == 0:
                maze_grid[y + dy][x + dx] = 1
                maze_grid[new_y][new_x] = 1
                backtrack(new_x, new_y)
```

Figure 5: Generate Maze by Backtracking

2.2 Generate Maze Using Kruskal's Algorithm:

Kruskal's algorithm guarantees that the generated maze stays connected [12], in contrast to certain other maze generating techniques, offering a logical framework for investigation and resolution. The maze's cells are first treated by the algorithm as separate sets [11]. Next, it chooses walls at random between neighboring cells and determines whether or not adding these walls results in loops or disconnected parts. A wall is incorporated into the maze's construction if it can be inserted without affecting connection. This process is repeated until the required wall density is reached, creating a maze that tests people's ability to choose the best routes while making sure every area of the maze is reachable. Because of its ability to strike a balance between intricacy and navigability, Kruskal's algorithm is a popular choice for creating mazes for a variety of applications, including puzzle solving.

```python
def generate_maze_kru(width, height, start, end, wall_density=0.3):
    maze_image = Image.new("RGB", (width * 10, height * 10), "white")
    draw = ImageDraw.Draw(maze_image)

    maze_grid = [[0 for _ in range(width)] for _ in range(height)]

    start_x, start_y = start
    end_x, end_y = end

    maze_grid[start_y][start_x] = 1
    maze_grid[end_y][end_x] = 1

    class DisjointSet:
        def __init__(self, n):
            self.parent = list(range(n))

        def find(self, i):
            if self.parent[i] != i:
                self.parent[i] = self.find(self.parent[i])
            return self.parent[i]

        def union(self, i, j):
            root_i = self.find(i)
            root_j = self.find(j)
            self.parent[root_i] = root_j

    disjoint_set = DisjointSet(width * height)
```

```python
    walls = []
    for y in range(1, height - 1, 2):
        for x in range(1, width - 1, 2):
            walls.append((x, y))
    random.shuffle(walls)

    for wall in walls:
        x, y = wall
        neighbors = []

        if x > 1:
            neighbors.append((x - 2, y))
        if x < width - 2:
            neighbors.append((x + 2, y))
        if y > 1:
            neighbors.append((x, y - 2))
        if y < height - 2:
            neighbors.append((x, y + 2))

        random.shuffle(neighbors)
        for neighbor in neighbors:
            nx, ny = neighbor
            if disjoint_set.find(y * width + x) != disjoint_set.find(ny * width + nx):
                maze_grid[y][x] = 1
                maze_grid[ny][nx] = 1
                disjoint_set.union(y * width + x, ny * width + nx)
                break
```

Figure 6: Working of Kruskal's Algorithm

2.3 Generating maze using Prim's algorithm:

First, a cell is chosen at random as the starting point of the process. Next, it designates the nearby cells that have not yet been visited and labels this cell as visited. These nearby cells could be the locations of future wall additions, that will provide passageways inside the maze. The program then picks a wall at random from the list of possible walls and determines whether or not adding it will open a doorway to a cell that has not been visited before. In that case, the wall is taken down, and the newly created cell joins the maze. Once every cell has been visited, this procedure is repeated, creating a maze with a single connected path running from the beginning to the finish. Prim's method works well for creating mazes that strike a compromise between complexity and navigability, which makes it appropriate for a number of uses, including pathfinding algorithms and puzzle-solving games [12].

```
def generate_maze_prim(width, height, start, end, wall_density=0.3):
    maze_image = Image.new("RGB", (width * 10, height * 10), "white")
    draw = ImageDraw.Draw(maze_image)

    maze_grid = [[0 for _ in range(width)] for _ in range(height)]

    start_x, start_y = start
    end_x, end_y = end

    maze_grid[start_y][start_x] = 1
    maze_grid[end_y][end_x] = 1

    def get_neighbors(x, y):
        neighbors = []
        if x > 1:
            neighbors.append((x - 2, y))
        if x < width - 2:
            neighbors.append((x + 2, y))
        if y > 1:
            neighbors.append((x, y - 2))
        if y < height - 2:
            neighbors.append((x, y + 2))
        return neighbors

    walls = []
    visited = set()

    def prims_algorithm(x, y):
        visited.add((x, y))
```

Figure 7: Generating Maze using Prims's Algorithm

3. Working of Code:

There are few steps user has to go through to generate final output.

```
Enter the number of mazes you want to generate: 3
Enter the complexity of the maze (e, m, h, c): m
Enter the wall density (between 0 and 1): 0.1
Enter the algorithm to generate the maze (bt/kru/prim): kru
Training for maze number 1 completed. Time taken: 1.47 seconds
Training for maze number 2 completed. Time taken: 1.09 seconds
Training for maze number 3 completed. Time taken: 7.28 seconds
```

1) Number of Mazes user wants to generate.
2) Complexity of Maze: there are 4 options, Easy, Medium, High and custom.
    - Easy Maze are the one which have height and width ranging from 5 – 10 cells.
    - Medium Maze are the one which have height and width ranging from 10-20 cells.
    - Hard Maze are the one which have height and width ranging from 20-40 cells.
    - In a custom maze, user can input height and width according to them.

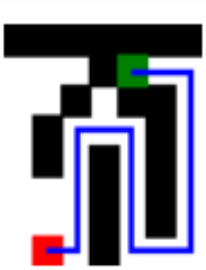


Figure 8: Easy complexity Maze

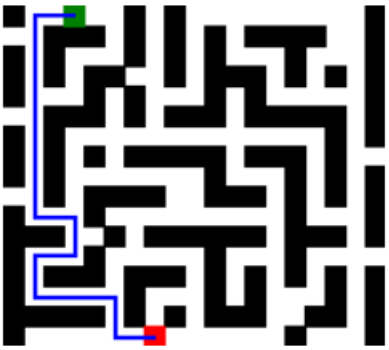


Figure 9: Medium Complexity Maze

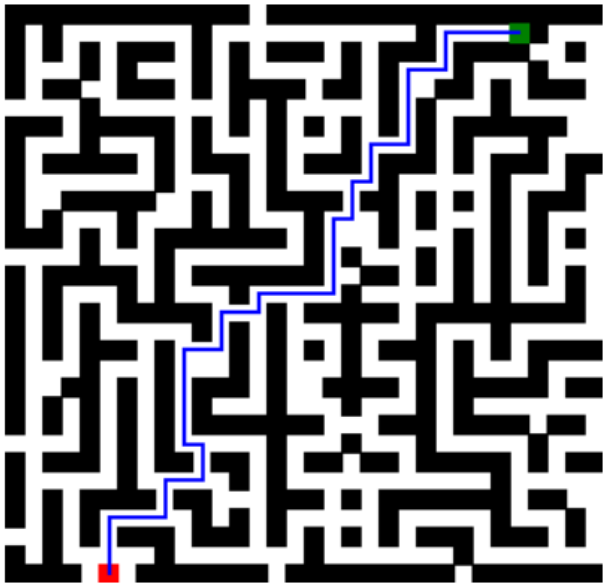


Figure 10: Hard Complexity Maze

3) Wall Density refers to how densely the walls are located in the maze. Its value ranges from [0,1], 0 indicating denser walls while 1 indicating sparse walls. The more the density is, the easier it is to find path for our Q-learning agent. The time taken is less when the wall density is high as compared to low wall density. Less dense or sparse walls means that the walls are more separated from each other thus there would be more paths for our agent to explore, which means that the agent would take more time to find best path. Below are images for maze having 0.5 wall density and 0.1 wall density respectively.

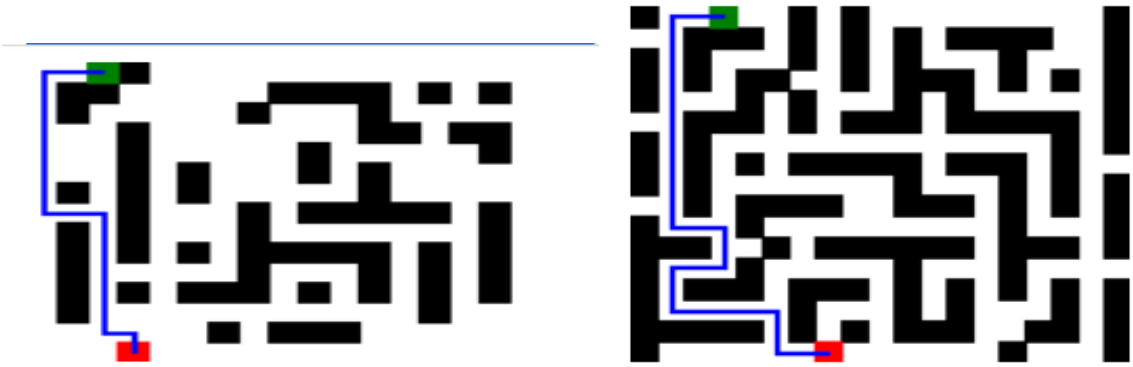


Figure 11: Maze with Wall Density 0.5 and with 0.1 Respectively

4) Algorithm which user wants to use to generate maze.
5) After everything is entered Q-learning starts to solve the maze. Initially all values in Q-table are 0. It will update matrix as it goes by.

4. Constraints

1) Wall Density – It is one of the Biggest constraints in maze solving, If walls are dense then number of path a agent can take are also few, where as if walls are less dense there are more number of path agent can take. This increases complexity and time it takes for agent to solve a Maze.
2) Size of Maze – If size of maze is larger then time taken for agent to solve it is also more as compared to easy and small mazes.
3) Number of maze getting generated – More the number of maze, it going to take more time for generating as well as solving those maze.
4) There is also a constraint for starting and ending point in maze. Starting point should be in first two rows of maze and ending point should be in last two rows of maze.

5. Conclusion

To sum up, the goal of our project was to create a maze solver by utilizing a variety of algorithms and reinforcement learning strategies. We created successful mazes with diverse characteristics, such as different wall densities, sizes, and amounts, by implementing maze generation methods including backtracking, Prim's algorithm, and Kruskal's algorithm.

Our use of Q-learning, a type of reinforcement learning, worked well to teach the agent to identify the best routes through mazes. Our agent overcame obstacles presented by dense walls, vast sizes, and several maze instances by utilizing Q-learning to adapt to different maze layouts.

As we experimented and analyzed the data, we came to several important conclusions about maze solving problems. The maze's wall density has a big impact on how difficult and time-consuming it is for the agent to solve. Higher wall densities reduce the number of feasible routes, making the agent's task less challenging. In a similar way, a low maze density may cause the agent to explore a greater number of possible paths, which would similarly lengthen the solving time, and sometime go in infinite loop.

Furthermore, the agent's time to travel and complete the maze is directly impacted by its size. When compared to smaller, more compact mazes, larger mazes demand more processing power and traversal time, which leads to larger solving time.

Moreover, we identified how the number of mazes created affected overall performance. The computing load and time required to generate and solve several mazes at once rise, underscoring the scalability issues with maze-solving algorithms.

In conclusion, our project not only provided practical insights into maze generation and solving algorithms but also demonstrated the capabilities of reinforcement learning in addressing complex maze-solving tasks. Future work may involve optimizing algorithms for scalability, exploring advanced reinforcement learning techniques, and tackling additional constraints to further enhance maze-solving efficiency.

6. References

[1] Lin, Shih-Wei, Huang, Yao-Lin, & Hsieh, Wen-Keui. (2019). Solving Maze Problem with Reinforcement Learning by a Mobile Robot. pp. 215-217. 10.1109/ICCCE48422.2019.9010768

[2] Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research, 4, 237-285. https://doi.org/10.1613/jair.301

[3] Hai Nguyen & Hung La. (2023). "*Review of Deep Reinforcement Learning for Robot Manipulation*". 2019 Third IEEE International Conference on Robotic Computing (IRC). Retrieved from: *https://ieeexplore.ieee.org/document/8675643*

[4] Richard S. Sutton et al., Policy Gradient Methods for Reinforcement Learning with Function Approximation, 2000.

[5] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, Cambridge, MA:MIT Press, 1998.

[6] Hado van Hasselt, Arthur Guez and David Silver, "Deep Reinforcement Learning with Double Q-Learning", *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, pp. 2094-2100, 2016.

[7] B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," in IEEE Access, vol. 7, pp. 133653-133667, 2019, doi: 10.1109/ACCESS.2019.2941229.

[8] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition", *J. Artif. Intell. Res.*, vol. 13, pp. 227-303, Nov. 2000.

[9] D. Osmanković and S. Konjicija, "Implementation of Q — Learning algorithm for solving maze problem", vol. 30, no. 3, pp. 1619-1622, 2007.

[10] H.A. Priestley, M.P. Ward, "A Multipurpose Backtracking Algorithm," Journal of Symbolic Computation, 18(1), 1-40, 1994. DOI: 10.1006/jsco.1994.1035.

[11] Buck, Jamis. "Maze Generation: Kruskal's Algorithm," weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-salgorithm, January 3 2011

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