Q-Learning based Maze Navigation

Benjamin Kraljušić
School of Innovation Design and Engineering
Mälardalen University
Västerås, Sweden
bkc23001@student.mdu.se

Dženan Kreho
School of Innovation Design and Engineering
Mälardalen University
Västerås, Sweden
dko23001@student.mdu.se

Enesa Hrustić
School of Innovation Design and Engineering
Mälardalen University
Västerås, Sweden
ehc23002@student.mdu.se

***Abstract*—Maze navigation presents a pertinent problem in various scientific fields, including automation and robotics. Different algorithms related to artificial intelligence can be deployed to address such challenges. This paper utilizes Q-Learning as a Reinforcement Learning approach to tackle maze navigation. The agent is tasked with navigating mazes of varying sizes and accumulating rewards associated with each state. Mazes and the actions that the agent may take are represented using Q-Learning terminology and methods. Experimental evaluation of the effectiveness of the learning process is conducted, along with subsequent conclusions. The findings illustrate the agent's capability to learn and adapt its navigation strategy, in order to successfully traverse mazes to reach predefined maze exits. This project contributes to a broader understanding of Reinforcement Learning techniques for navigating unknown environments, which may prove valuable for various autonomous applications.**

***Index Terms*—Reinforcement Learning, Q-Learning, Maze, State, Action**

I. Introduction

Mazes present intricate structures that have captured human attention since the early ages, such as the Cretan Labyrinth. The purpose of investigating and solving mazes has shifted from mythological narratives to a more modern scientific approach. The necessity for considering mazes nowadays lies in the rising need and development of autonomous agents, such as autonomous cars, rovers, and robotic manipulators. More often than not, these manipulators have the task of navigating through unknown environments while conserving energy.

The idea of this project is to investigate the possibilities of adapting Reinforcement Learning (RL) to an agent whose task is to learn how to navigate through mazes of variable sizes. For this purpose, the RL technique will be limited to Q-Learning. The justification for choosing Q-Learning lies in the fact that the maze learning process can be intuitively described through a reward and feedback paradigm. The agent is rewarded for reaching the maze exit and consequently for choosing the path that leads to the exit.

Reinforcement Learning, in its basic principles, was familiar to humans long before the advent of computers. Reinforcing positive learning outcomes by providing rewards has been used for a considerable period in animal domestication and training. On a more formal note, the mathematical base which prompted its use in computer science was provided by Ivan Pavlov in 1911 [1].

The result of this project is the successful implementation of an intelligent agent based on the Q-learning approach. The agent can effectively navigate complex mazes of various sizes, providing a path from the initial (entry) state of the maze to the final (exit) state. In addition to the standard (default) Q-learning approach, several heuristic variations were implemented to compare the effectiveness of the learning process for each approach. Depending on how these approaches affect the learning process, they can be divided into two groups. The first group consists of the default approach and three additional approaches based on heuristics:

- Default learning process - complete all steps in the RL process to learn the optimal path
- Greedy learning process - stop the learning process when the agent finds one successful path
- Dead-end detection learning process - eliminate the dead-ends as a possible part of the path to boost the learning process
- Greedy with dead-end detection learning process - incorporate the two previous approaches into a single approach

The second group of approaches pertains to how the initial state is handled during the learning stage. There are two possible options:

- Fixed initial state learning - always use the same initial state during the learning process
- Varied initial state learning - randomly vary the initial state during the learning process

The rest of the report is structured as follows: Section II presents the background, Section III summarizes the related work on this topic, Section IV describes the approach and method, Section V discusses the results and contributions, and the report concludes with the conclusion in Section VI.

II. Background

Before diving deeper into the topic, several terms will be introduced with respect to the maze solving that was done in this project:

- State - Any field in the maze that agent can visit, also sometimes denoted as *s*
- Action - For maze navigation set of actions are: up, down, left and right. The agent performs these actions in order to move from one state to another within the maze, also sometimes denoted as *a*
- Environment - The maze presents the environment in this case. It is consisted of states and the agent performs some actions within the environment to change between the states
- Reward - The agent receives a reward if, by taking a certain set of actions, it manages to find the solution to the maze. The nature of the rewards is such that they accumulate over the learning period, reinforcing the agent to choose the states that are most likely to lead to the final state.

A. Reinforcment Learning

Reinforcement learning is one of the methodologies/algorithms of machine learning. An agent, which is an operator performing actions and tasked with learning to solve a specific type of problem, receives rewards for actions that yield positive outcomes. This type of learning is thus termed reinforcement learning. Ultimately, the sequence of actions leading to the problem's solution results in the maximum total of rewards. However, reward collection does not occur instantaneously. The agent gains understanding of the environment through trial and error, gradually providing a set of states and their corresponding rewards that lead to the solution over time. These multiple instances of exploring the environment are referred to as episodes [2].

The agent can be programmed to adopt two approaches when learning about and from the environment: exploration and exploitation. Exploration may be more suitable during the initial stages of the RL algorithm when the agent has little to no knowledge about its environment. It assumes that the agent may visit all or the majority of possible states within the environment and learn about the rewards those states yield. Some states will yield higher rewards, which the agent will then remember [1].

Exploitation is more suitable for later stages of the RL process, as agent will now try to visit the states that yield higher reward and from there decide upon its further actions. Over the span of the given episodes the agent will most likely accumulate the highest reward for states that lead to the solution. The good approach to any problem that may be addresses using Reinforcement Learning should include both exploration and exploitation phases.

Q-Learning is one of the methods in Reinforcement Learning. It is characterized by a Q-Table, which comprises states that the agent can visit and actions that the agent can take with respect to the state it is in. Each field within the Q-Table is updated during the episode with its respective reward. After all the episodes are finished, it is possible to follow through all states and actions within the table that lead to the highest accumulated reward. Q-Table values are updated using the following formula [1]:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(s_t, a_t) \cdot \\ \cdot [r_t + \gamma \max_{a_t} \{Q(s_{t+1}, a_t)\} - Q(s_t, a_t)] \quad (1)$$

Equation (1) includes several parameters that must be defined by the programmer. Parameter $\alpha(s_t, a_t)$ defines the learning rate and can take values within the range $[0, 1]$. Parameter γ is a positive number representing a discount factor and should be smaller than 1. Equation (1) is more suitable for types of environments where the set of possible actions is not known for each state. However, for deterministic environments such as mazes, the Q formula can be updated by setting the learning rate equal to 1, which yields the equation (2):

$$Q(s_t, a_t) = r_t + \gamma \max_{a_t} \{Q(s_{t+1}, a_t)\} \quad (2)$$

B. Maze solving using RL

In terms of Reinforcement Learning, mazes represent one type of environment, with states being represented by fields that the agent can visit (non-walls). Moreover, the number of actions that the agent may take is finite - up, down, right, and left. However, it may be impossible to perform all actions from every state; some actions may be impossible in certain states due to the configuration of the maze. With this in mind, the Q-Table will have a number of rows equal to the fields that can be visited inside the maze and a number of columns equal to 4. The goal is to find the path that leads from the initial state to the final state of the maze. For the purpose of maze solving, Equation 2 can be used as mazes represent a deterministic type of environment. In this work, mazes of different sizes will be used. One of the possible maze configurations is presented in figure 2. More details about how Q-Learning is used in this project are presented in Section IV.

Besides Q-Learning there are other techniques that can be used for maze solving, or any environment navigation in general. Related work that considers these problems is presented in the next Section.

III. Related Work

Computational maze-solving encompasses a broad range of algorithms and techniques aimed at navigating through different complexity mazes. Traditional graph path-finding algorithms, such as depth-first search (DFS), breadth-first search (BFS), and the A* algorithm, have been foundational in establishing the principles of maze navigation for intelligent agents. [3].

A. Maze-Solving Algorithms

Even [4], and Russell and Norvig [1] provide comprehensive analyses of traditional graph traversing algorithms that have often been used for maze solving. These algorithms serve as benchmarks for evaluating the efficiency and effectiveness

of newer, more complex approaches. For example, the A* algorithm's heuristic approach to estimating the distance to the goal makes it particularly relevant for comparison with machine learning techniques that incorporate similar estimation strategies.

B. Machine Learning in Maze Solving

With the development of machine learning, the strategies for maze solving have evolved from deterministic algorithms to adaptive, data-driven models. Early applications of neural networks to maze solving demonstrated the potential for machine learning to approximate complex decision-making processes in maze solving context [5]. Beyond neural networks, genetic algorithms have been applied to maze-solving tasks, offering alternative ways of learning and optimizing paths based on data [6].

Machine learning not only provides the ability to solve mazes but also to generate them. Generative adversarial networks (GANs) [7], for instance, have been utilized to create maze-alike structures in game development, presenting increasingly challenging scenarios for reinforcement learning agents to overcome.

C. Reinforcement Learning and Q-Learning

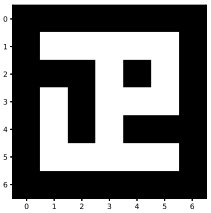
Reinforcement learning (RL) represents a paradigm within machine learning where agents learn to make decisions by interacting with an environment to achieve a goal. The agents learn from the consequences of their actions through a reward feedback mechanism, rather than from prior knowledge of the correct action to take [8]. Among various RL algorithms, Q-learning, introduced by Watkins, is particularly prominent for its nature of learning without model, allowing the agent to learn the value of the optimal policy independently of the agent's actions [2]. Q-learning has been instrumental in many areas where decision-making is sequential and the environment is initially unknown to the agent. In the context of maze solving, Q-learning helps an agent to navigate a maze by learning a policy that maximizes the cumulative reward, which is usually inversely proportional to the path length or time taken to reach the goal.

The development of deep learning has given rise to Deep Q-Networks (DQN), which combine Q-learning with deep neural networks. DQNs utilize high-capacity function approximators to handle environments with high-dimensional state spaces, such as those with visual inputs from the maze [9]. Variants of DQN have been proposed to address limitations of Q-learning. Double Q-learning, for example, tackles the overestimation bias of action values inherent in standard Q-learning by decoupling the selection and evaluation of actions [10]. Dueling DQN architecture improves upon DQN by separately estimating the state value and the advantages of each action, leading to more robust policy learning, especially in states where the choice of action does not significantly affect the outcome [11]. Lin integrated reinforcement learning with artificial neural networks in a way that made learning faster and enabled tackling non-Markovian environments, all of which made reinforcement learning more suitable in case of real robots solving mazes [12].

Overall, the integration of RL and Q-learning into maze solving represents a significant shift towards algorithms that are not only capable of finding a path through a maze but can also adapt to new, unknown mazes without reprogramming.

IV. Approach and Method

As previously mentioned, Q-learning, or more broadly, reinforcement learning, can be easily applied to the problem of maze solving, especially when the maze is represented as a grid. This means that a maze is represented by an $n \times m$ grid or matrix consisting of walls and free spaces. For this project, it has been chosen to use this representation and decided to represent walls as zeros and free spaces as ones. An examples of this is shown in Figure 1. Since the agent



(a) Graphical representation of the maze

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

(b) Grid representation of the maze

Fig. 1: Example maze

cannot move through walls, all the cells in the free space can be enumerated, with each cell representing a specific state. Using these states, a Q-table can be formed, with the rows representing the states and the columns representing the actions that can be performed in each state (up, down, left, and right). It's worth noting that the agent might not have every action available in each state, resulting in a ragged Q-table where each row may have a different number of columns.

Q-learning is conducted in episodes or epochs. Each episode involves the agent traversing the maze, starting from the specified initial state and aiming to reach the specified final state. Given that the agent may not always reach the final state, i.e., it may become lost in the maze, the number of moves per episode is restricted. An episode finishes when the agent reaches the final state or exceeds the maximum allowed number of moves. At each step or move, the Q-table is updated using Equation 2. Since Equation 2 is based on the Bellman optimality equation [1], the resulting Q-table will be optimal in the sense that it minimizes the number of steps needed for the traversal of the maze. Algorithm 1 presents the pseudocode of the standard Q-learning algorithm applied to the aforementioned maze-solving problem and maze

Algorithm 1 Standard Q-learning Algorithm

Input: $maze$, $initState$, $finalState$
Output: $Q\text{-}Table$

$Q\text{-}Table \leftarrow InitializeQTable(maze)$
for $episode \leftarrow 0$ *to* $numOfEpisodes$ **do**
  $state \leftarrow initState$
  for $move \leftarrow 0$ *to* $maxNumOfMoves$ **do**
    if $state == finalState$ **then**
      break
    end if
    $action \leftarrow getAction(state)$
    $newState \leftarrow changeState(state, action)$
    if $newState == finalState$ **then**
      $reward \leftarrow 100$
    else
      $reward \leftarrow 0$
    end if
    $Q\text{-}Table[state, action] \leftarrow reward + discountFactor \cdot max(Q\text{-}Table[newState])$
  end for
end for
return $Q\text{-}Table$

representation. From Algorithm 1, it is evident that the agent selects an action based on the current state. Ideally, the agent will determine the new state based on the Q-value of that state. However, the issue with this approach is that initially, all Q-values, i.e., the entire Q-table, are set to zero. For this reason, at least initially, the agent must follow some other policy to navigate the maze. This means that the agent must explore the maze. After it has explored enough and the Q-table has started to contain non-zero Q-values it can start exploiting the knowledge, i.e., the Q-table. The problem is knowing when to switch from exploration to exploitation. In this project, the Epsilon-Greedy policy is employed to address this issue. The main idea behind the epsilon-greedy policy is as follows: Generate a random number, and if that number is lesser than a certain epsilon value, do exploration; otherwise, do exploitation [8]. It's worth noting that this policy doesn't directly address the problem of transitioning from exploration to exploitation, instead, it provides a mechanism for deciding between the two at each step. The issue can be fully addressed by implementing a mechanism or function that adjusts the value of the epsilon parameter as the episodes progress. This approach is known as the Decayed Epsilon-Greedy policy [13]. There are many good choices for this function, but it is important that the function decreases as the episode number increases. In this work, the exponential function was chosen. Algorithm 2 provides pseudocode for the action selection mechanism used, based on the decayed epsilon-greedy policy.

A. Heuristics

The approach explained previously is the de facto standard implementation of the Q-learning algorithm. However, modifications or extensions can be made by incorporating specific heuristics tailored for maze solving.

Algorithm 2 Decayed Epsilon-Greedy Policy

Input: $state$, $epsilonMin$, $epsilonMax$, $episodeNumber$, $decayRate$, $Q\text{-}Table$
Output: $action$

$epsilon \leftarrow epsilonMin + (epsilonMax - epsilonMin) \cdot exp(-decayRate \cdot episodeNumber)$
$number \leftarrow generateRandomNumber()$
if $number < epsilon$ **then**
  $action \leftarrow generateRandomAction(state)$
else
  $action \leftarrow \text{argmax}\{Q\text{-}Table[state]\}$
end if
return $action$

One such heuristic is performing dead-end detection. Inspired by the way a human might solve a maze, the idea is to check whether the current state is a dead-end. If so, it is marked as a dead-end and is not visited again in this episode or subsequent ones. In the context of the previously presented algorithm, this detection can be performed by examining the number of available actions at the current state. If only one action is available, then that state is a dead-end and can be excluded from further consideration by removing the action that leads from the previous state (which will necessarily also be the next state) to the current state. By repeatedly performing this detection, the agent will avoid traversing paths that lead to dead-ends. This increases the likelihood of reaching the final state sooner, thereby reducing the required learning time.

Up to this point, it has been assumed that all available episodes are used for learning. With a sufficiently large number of episodes, this will result in optimal Q-values, allowing traversal from the initial to the final state using these Q-values to be accomplished in the minimum number of moves. However, if one is not interested in finding the optimal solution but rather in just finding a solution, the maximum number of episodes might not be required. This leads to the use of the greedy heuristic. The main idea is as follows: At the end of every episode, check if it is possible to traverse the maze from the initial to the final state using the Q-table learned so far; if possible, stop the learning process. This will reduce the learning process but comes with the trade-off of not guaranteeing an optimal traversal path.

B. Choice of initial state

Although not explicitly stated, the initial state in the previously presented approaches was considered to be fixed. This guarantees an optimal path, or just a path if the greedy heuristic is used, when traversing the maze from this initial state. With this in mind, it is possible that the maze could still be traversable from states other than the one used as the initial state during

the learning phase. If one desires to have a Q-table that can facilitate traversal of the maze from every possible state (assuming a fixed final state), learning from a fixed initial state might not be the best approach. To increase the likelihood of traversing the maze from all possible states, it is better to vary the initial state from episode to episode during the learning stage. By doing this, the agent might explore paths that wouldn't be explored if it always started from the same initial state. This, in turn, should increase the chance of the maze being traversable from all possible states. It is worth mentioning that for faster learning, this varied initial state learning can be combined with the previously mentioned dead-end detection heuristic. However, it cannot be combined with the greedy heuristic, as the latter requires a fixed initial state to know when to stop the learning.

C. Implementation details

All of the aforementioned approaches were implemented in a Q-learning-based maze solver using the Python programming language. The popular NumPy library was employed for efficient array and matrix manipulation. As mentioned earlier, mazes are represented as grids of ones and zeros, and they are stored in text files, which can be loaded into the solver. Additionally, the implementation includes a random maze generator, which will be further discussed in Section V.

V. Results and Evaluation

In this section, the results of the experimental evaluation performed on the implemented Q-learning maze solver will be presented. However, before performing the evaluation, it is necessary to create a set of mazes for testing. Given that manual creation of mazes is a daunting task, the use of a maze generation algorithm was employed. For this reason, the maze generation algorithm that was used will be presented first, followed by the testing results.

A. Maze generation

There exists a variety of options and variations of maze generation algorithms [14]. Considering the focus of this work is on maze solving using Q-learning rather than maze generation, a straightforward algorithm that is easy to implement was chosen. In particular, the recursive backtracking maze generation algorithm was utilized. Originally, this algorithm is designed to work with the graph representation of a maze, but it is easily adapted to the matrix representation. The algorithm starts with an empty maze and selects one of the cells/positions as the initial position. In this case, the initial position is the top-left cell of the maze. From this point it starts randomly moving and carving free space in the maze. This continues until it reaches either the edge of the maze or a cell that has already been explored. At this point, it begins backtracking to the first cell from which exploration can resume. This process repeats until further exploration is not possible [15], [16]. In the end, the outcome is a maze, an example of which is shown in Figure 2. It's worth mentioning that although the algorithm is recursive, it can be implemented by manually managing the stack instead of relying on actual recursion. This increases the speed and decreases the memory consumption, with the trade-off of a slightly more complex implementation. The pseudocode of this implementation is given in Algorithm 3.

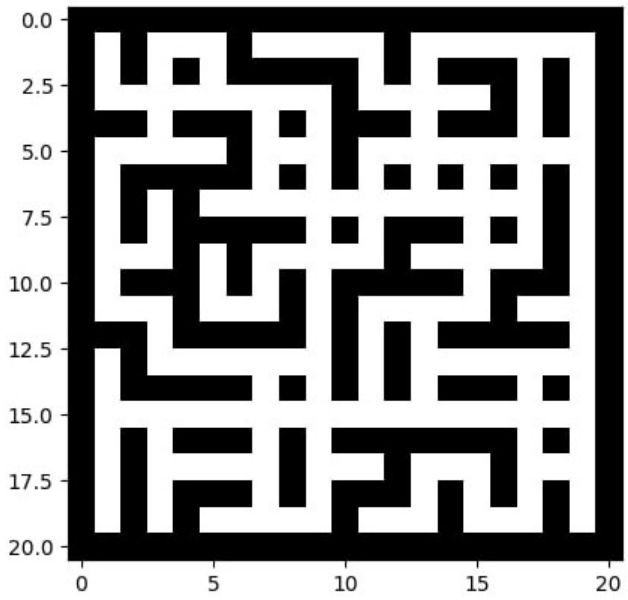


Fig. 2: An example maze generated using the recursive backtracking algorithm

B. Experimental setup and results

As previously mentioned, the implemented maze solver has various heuristics and learning options. Firstly, experimental evaluation was performed to see the effect of the heuristics on the learning times. The test set of mazes was made up of square mazes with a maze width/height ranging from 7 to 77. To ensure a fair comparison and that the solver yields solutions for all mazes, the Q-learning parameters were manually set. The used parameter values are presented in Table I. Additionally, to accommodate the varying episode lengths resulting from the maze exploration randomness, each maze was learned five times, and the average learning time was considered as the result. The results of the experimentation are shown in Figure 3, where the maze size (i.e., maze width/height) is represented on the abscissa,

Parameter	Value
Number of episodes	10000
Max. number of moves per episode	100000
Discount factor	0.9
Epsion max.	1
Epsion min.	0.05
Decay rate	0.0005

TABLE I: Q-learning parameters for test evaluating different heuristic options

Algorithm 3 Backtracking Maze Generation Algorithm

Input: $mazeWidth,\ mazeHeight$
Output: $maze$

$maze \leftarrow CreateMatrix((mazeHeight, mazeWidth), 0)$
$visited \leftarrow CreateMatrix((mazeHeight, mazeWidth), false)$
$stack \leftarrow CreateEmptyStack()$
$currentPosition \leftarrow (1, 1)$
$goForward \leftarrow true$
while true do
  if $goForward$ **then**
    $stack$.Push($currentPosition$)
    $visited[currentPosition] \leftarrow true$
    $maze[currentPosition] \leftarrow 1$
  else
    $goForward \leftarrow$ true
  end if
  $possibleMoves \leftarrow GetPossibleMoves(currentPosition, visited)$
  if $possibleMoves.isEmpty()$ **then**
    $stack.pop()$
    if $stack.isEmpty()$ **then**
      break
    end if
    $currentPosition \leftarrow stack.top()$
    $goForward \leftarrow False$
    continue
  end if
  $move \leftarrow getRandomMove(possibleMoves)$
  $addFreeSpaceToMaze(maze, currentPosition, move)$
  $currentPosition \leftarrow currentPosition + move$
end while
return $maze$

while the learning time in seconds is shown on the ordinate. From Figure 3, it can be observed that the learning times increase as the maze size increases. As expected, dead-end detection improves, i.e., decreases, the learning times, which is especially noticeable with larger mazes. A drastic decrease in learning times is observed with the use of the greedy heuristic. This aligns with expectations, as learning stops once a solution, although not necessarily optimal, is found. Further improvement is seen when the greedy and dead-end heuristics are combined.

Continuing with the experimental evaluations, the next objective is to determine whether learning from different initial states affects the maze-solving outcomes. A similar procedure was used to the previously described one with some differences. This time, the test set is comprised of square mazes ranging in size from 10 to 55. Considerations regarding the Q-learning parameters remained the same, although the

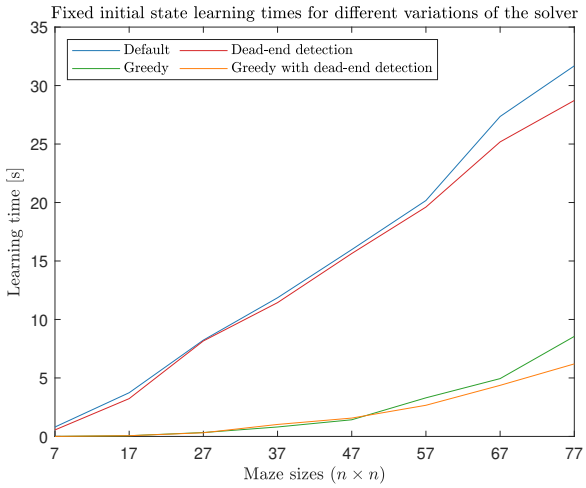


Fig. 3: Experimentation results from different heuristic variations of the solver with a fixed initial state

parameters were changed. The values of the used parameters are shown in Table II. Given that the aim of learning from

Parameter	Value
Number of episodes	5000
Max. number of moves per episode	50000
Discount factor	0.9
Epsion max.	1
Epsion min.	0.05
Decay rate	0.001

TABLE II: Q-learning parameters for test evaluating varied and fixed initial state learning

varied initial states is to expand the number of states from which a solution can be derived, the percentage of states from which a solution can be obtained after training was also measured, alongside the learning times. Figure 4 shows the results from this experimentation. Figure 4 demonstrates that learning times decreases when varied initial state learning is used. In addition, an increase in the percentage of states from which a solution is also observed. This is particularly noticeable with smaller maze sizes. This is in line with expectations when considering that larger maze sizes require more exploration regardless of the initial state.

VI. Conclusion

This paper presents the use of Q-Learning, a reinforcement learning algorithm, for solving maze tasks with intelligent agents. It addresses two approaches to maze solving using Q-Learning. The first approach involves training the agent to navigate the maze in such a way that it can find an exit regardless of its starting position. The second approach focuses on solving the maze from a fixed starting position, employing Q-Learning as a conventional maze-solving algorithm. To improve the efficiency of Q-Learning in these contexts, two heuristics were introduced: dead-end detection and a greedy

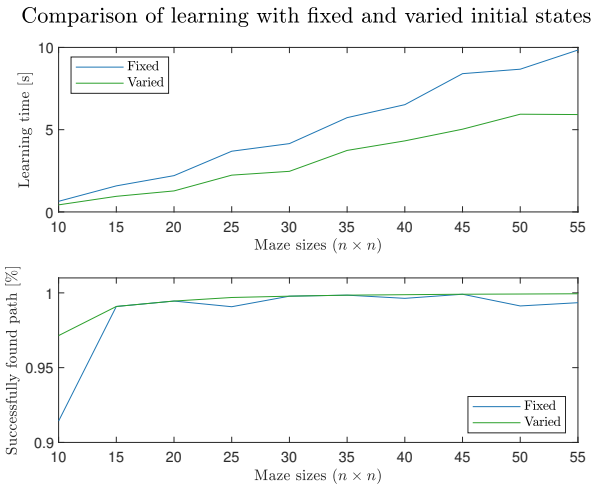


Fig. 4: Experimentation results from learning with with varied and fixed initial states

solving policy. Additionally, a backtracking algorithm was implemented for maze generation. The effectiveness of the Q-Learning algorithm in the context of maze solving was evaluated, highlighting the enhancements provided by the introduced heuristics. The findings demonstrate that Q-Learning is effective both as a method for learning to navigate maze environments optimally and as a traditional algorithm for maze solving.

REFERENCES

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[2] C. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 05 1992.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[4] S. Even, "Graph algorithms," *Computer Science Press*, 2011.

[5] S. Srinivasan, D. Mital, and S. Haque, "A novel solution for maze traversal problems using artificial neural networks," *Computers & Electrical Engineering*, vol. 30, pp. 563–572, 11 2004.

[6] N. Choubey, "A-mazer with genetic algorithm," *International Journal of Computer Applications (0975 – 8887)*, vol. 58, pp. 48–54, 11 2012.

[7] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning."

[10] H. V. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.

[11] Z. Wang, T. Schaul, M. Hessel, H. V. Hasselt, M. Lanctot, and N. D. Freitas, "Dueling network architectures for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1995–2003.

[12] L.-J. Lin, "Reinforcement learning for robots using neural networks," *PhD dissertation, Carnegie Mellon University, Pittsburgh*, 1993.

[13] S. Chinnamgari, *R Machine Learning Projects: Implement supervised, unsupervised, and reinforcement learning techniques using R 3.5*. Packt Publishing, 2019. [Online]. Available: https://books.google.se/books?id=4dKDDwAAQBAJ

[14] Wikipedia contributors. (2024) Maze generation algorithm — Wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Maze_generation_algorithm&oldid=1203215310

[15] G. Cope. Maze generation algorithm - depth first search. [Online]. Available: https://algosome.com/articles/maze-generation-depth-first.html

[16] P. Gabrovšek, "Analysis of maze generating algorithms," *IPSI Transactions on Internet Research*, vol. 15, no. 1, pp. 23–30, 2019.