Maze Solving Algorithms through Stackelberg Model-Based Reinforcement Learning

Giulia Ghisolfi, g.ghisolfi@studenti.unipi.it, Roll number: 664222
Master Degree in Computer in Science (Artificial Intelligence Curriculum), University of Pisa
Algorithmic Game Theory course (211AA), Academic Year: 2022-2023

Abstract

This work explores the application of model-based reinforcement learning algorithms for maze-solving tasks. The problem is formulated in its Stackelberg form, representing an asymmetric two-player game. Rational actors are identified as a leader and a follower, representing the policy and Markov Decision Process parameters of a reinforcement learning agent. The objective is to learn general rules that enable the discovery of solutions in previously unseen maze environments. The policy agent and the model agent operate in an adversarial dynamic, attempting to determine the Stackelberg equilibrium of the duopoly.

1 Introduction

In this work, we present our study of an asymmetric two-player game formulated in its Stackelberg form. We will identify our rational actors as a leader and a follower. The goal of this study is to determine the Stackelberg equilibrium of the Stackelberg duopoly using model-based reinforcement learning algorithms.

The algorithms are designed to solve maze-solving problems on randomly generated mazes that the agents have never encountered before. The agents lacks prior knowledge of the environment but constructs its representation through exploration.

The core concept revolves around stimulate cooperative learning within an adversarial dynamic, with both leader and follower working collaboratively towards a shared objective.

Due to the asymmetric formulation of the problem, two distinct versions of the algorithm are obtained. In one version, the policy agent assumes the role of the leader, with the model agent acting as the follower. This configuration is inverted in the alternate version.

2 Preliminaries

In this initial section, we present the theoretical groundwork for our study by introducing key concepts related to Stackelberg games and reinforcement learning. Our approach draws inspiration from the discussions presented in the books Tadelis [2013] and Sutton and Barto [2018].

2.1 Stackelberg Competition

The Stackelberg duopoly is a game of perfect information that is a sequential-moves variation on the Cournot duopoly model of competition. It illustrate that the order of moves matter, and rational agents will take this into account.

So in the Stackelberg duopoly, the two agents behave sequentially. One of the two agents, knowns as leader, makes its decision before the other agents, which is considered the follower. This means the leader has a strategic advantage in determining its own strategy before the follower makes its decision.

The Stackelberg model allows for the analysis of how the first-mover advantage influences agent behavior and final outcomes. Typically, the leader will choose to maximizes its rewards, taking into account the expected reaction of the follower. The follower will then respond optimally to the leader's decision.

2.1.1 Stackelberg equilibrium

A Stackelberg equilibrium is an extension of the traditional Nash equilibrium. In a standard Nash equilibrium, all players simultaneously make decisions without knowledge of the others' choices.

In contrast, in a Stackelberg equilibrium, one player, known as the leader, makes the first move. The other player, the follower, observes the leader's choice before making their own decision. This framework is common in sequential games, like the Stackelberg duopoly, where one player holds an advantage in timing, information, or resources.

Definition 1 (Stackelber equilibrium)**.** *$x_1^* \in S_1$, the set of strategies for the first player, is a Stackelberg solution if*

$$x_1^* \in argmax\{u_1(x_1, R_2(x_1) : x_1 \in S_1\}$$

$(x_1^*, R_2(x_1^*)$ is a Stackelberg equilibrium if x_1^* is a Stackelberg solution.

Where $R_2(x_1^*) \in S_2$ represents the best response of the follower to the leader's stategy.

2.2 Reinforce Learning

Reinforce learning is a machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment. The agent takes actions to maximize a cumulative reward signal over time.

2.2.1 Markov Decision Process

In our work, in order to be consistent with Rajeswaran et al. [2021], we modelized the environment as a Markov Decision Process. MDPs are a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the agent. Everything outside the agent is referred to as the environment, with which the agent interacts. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions. That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

Definition 2 (Markov Decision Process)**.** *A Markov Decision Process is a tuple $M = \langle S, A, \boldsymbol{P}, R, \gamma \rangle$*

S: finite set of states

A: finite set of actions a

***P**: state transition matrix, s.t.*
$P^a_{ss'} = P(S_{t+1} = s' | S_t = s, A_t = a)$

R: reward function, s.t.
$R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$

γ: discount factor, $\gamma \in [0, 1]$

The transition dynamics are described by $s' \sim P(\cdot|s, a)$, while $p(s|a, s') : S \times A \times S \to [0, 1]$ is transition dynamics density function, $r(s', a, s) : S \times A \times S \to \mathbf{R}$ defines the reward function. A policy is a mapping from states to a probability distribution over actions, i.e., $\pi : S \to P(A)$.

In Figure 1, we present a schematic representation of the agent-environment interaction in a Markov decision process. The figures are sourced from Sutton and Barto [2018].

2.2.2 Model Based RL

Model-Based Reinforcement Learning is a methodology in which the agent constructs a

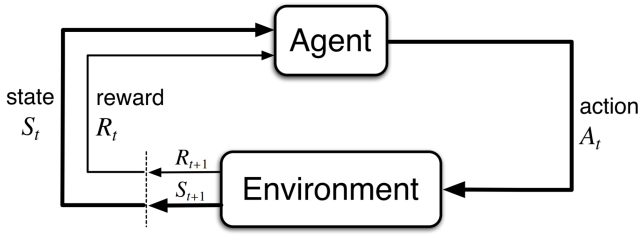

Figure 1: The agent-environment interaction in a Markov decision process.

model of the environment. This model predicts how the environment will respond to the agent's actions. The goal of MBRL is to enable the agent to make more informed decisions and optimize the learning process. MBRL is a field with numerous practical applications, including game playing.

2.2.3 Policy Player

A policy player is an agent in a reinforcement learning framework responsible for making decisions or selecting actions based on a learned policy. The policy player's role is to interact with the environment by taking actions according to a policy it has learned through past experiences. This agent aims to maximize cumulative rewards and, in a sequential decision-making process.

2.2.4 Model Player

A model player is an agent in a reinforcement learning context that utilizes a predictive model of the environment to simulate and plan for future actions. The model player does not directly interact with the environment but rather uses its internal model to simulate the outcomes of various actions. This simulation helps in making more informed decisions and optimizing the learning process. The model player in a Stackelberg-like setup, reacting to the policy player's actions.

3 Related works

Our work begins with an examination of the Stackelberg formulation for Model-Based Reinforcement Learning (MBRL) as presented in Rajeswaran et al. [2021], where the authors focus on designing stable and efficient algorithms for model-based reinforcement learning. They explore two scenarios: one where the policy player acts as the leader (PAL algorithm) and another where the model player takes the leader role (MAL).

In their algorithms, the leader chooses a conservative approach when updating its strategy, incorporating a regularization term to minimize the distance between the updated strategy and the previous one. In contrast, the follower adopts an aggressive update strategy, striving for optimal performance without considering the other player's actions. The authors demonstrate in their paper that their formulation for MAL and PAL algorithm outperforms non-game theoretic formulations.

In Table 1, we have presented the strategies depending on different scenarios of strategy choices made by model players and policy players in simultaneous games.

In their work, the authors analyze the strategies of MAL and PAL, but in a non-simultaneous game. In the scenarios they investigated, the leader chooses a conservative approach, while the follower selects an aggressive approach. The follower aims for their best performance without being concerned about the actions of other players.

		Model player	
		conservative	aggressive
Policy player	conservative	GDA	PAL
	aggressive	MAL	Best Response

Table 1: Strategies of Model Players and Policy Player in Different Scenarios (GDA: Gradient Descent Ascent, PAL: Policy As Leader, MAL: Model As Leader)

3.1 PAL

Policy As Leader. The policy player employs a conservative approach in order to maximize the reward within model M, in contrast to the aggressive strategy employed by the model player. The model player aims to explain policy data, specifically by minimizing prediction errors within the induced distribution of the policy in the world, denoted as W.

$$\max_{\pi}\{J(\pi, M^{\pi}) \text{ s.t. } M^{\pi} \in \operatorname*{argmin}_{M} l(M, \mu_W^{\pi})\}$$

$$\max_{\pi} J(\pi, M^{\pi}) := \mathop{\mathbb{E}}_{M,\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t)]$$

Here, the objective is to maximize the expected cumulative reward following the policy π in model M^{π}, with the expectation taken over both the model and the policy.

In pratice this became a first-order gradient approximation:

$$\text{aggressive model step:}$$
$$M_{k+1} \approx \operatorname*{argmin}_{M} l(M, \mu_W^{\pi})$$
$$\text{conservative policy step:}$$
$$\pi_{k+1} = \pi_k + \alpha \nabla_{\pi} J(\pi, M_{k+1})$$

The pseudo-code for the algorithm is provided in Algorithm .1 in Appendix A.

3.2 MAL

Model As Leader. In this scenario, the model player adopts a conservative approach, while the policy player takes on an aggressive one. The objective functions remain the same as in the previous case, but this time, we deal with a non-stationary goal.

$$\min_{M}\{l(M, \mu_W^{\pi^M}) \text{ s.t. } \pi^M \in \operatorname*{argmax}_{\pi} J(\pi, M)\}$$

$$l(M, \mu_W^{\pi}) = \mathop{\mathbb{E}}_{(s,a)\sim\mu}[D_{KL}(P(\cdot|s,a), \hat{P}(\cdot|s,a))]$$

where D_{KL} is the Kullback–Leibler divergence.

Definition 3 (Kullback-Leibler divergence)**.** *We can define KL divergence as the expectation of the logarithmic difference between the probabilities P and Q, where the expectation is taken using the probabilities P.*

$$D_{KL}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

So the Kullback–Leibler divergence is a measure of how one probability distribution P is different from a second, reference probability distribution Q.

In pratice, this became a first-order gradient approximation:

$$\text{aggressive policy step:}$$
$$\pi_{k+1} \approx \operatorname*{argmax}_{\pi} J(\pi, M_k)$$
$$\text{conservative model step:}$$
$$M_{k+1} = M_k - \beta \nabla_M l(M, \mu_W^{\pi_{k+1}})$$

The pseudo code for the algorithm is provided in Algorithm .2 in Appendix A.

4 Problem Formulation

In this section, we present the formalization of our problem and outline our approach to address it.

Our primary goal is to derive a generalized policy enabling efficient navigation of unexplored mazes. To achieve this, we train a learning agent capable of derive a policy that can generalize its knowledge for effective maze-solving.

4.1 Agents

The formulated learning agent has no prior knowledge of the environment but constructs its representation, the model, through exploration. The environment is fully observable.

Our agent is conceptualized as two distinct agents engaged in adversarial interaction. Both agents observe the environment from different viewpoints, gaining diverse information through their interaction with the surroundings.

The first agent, known as the Policy Agent, represents the policy, a probability distribution over the state and action space. This agent serves as a relative observer, describing explored states from its viewpoint. Each state is rendered as an array of walls (represented by 0) or free passages (rappresented by 1), starting from the agent's left and progressing clockwise. This representation ensures that the passage to the previous state, behind the agent, is consistently placed at the last position of the array. A visual representation of this concept is presented in Figure 2.

The policy is then stored and updated based on this state space representation crafted during training. This state formulation, independent of absolute position, empowers the agent to learn a generalized policy applicable to unknown environments.

The second agents, known as the Model Agents, are responsible for representing the parameters of the Markov Decision Process (MDP) model. These agents create a specific environment representation based on the environment they are exploring. However, unlike the Policy Agent, a Model Agent is unable to generalize its knowledge to unexplored environments. Its primary objective is to interpret the data generated by the Policy Agent and assess whether the generalized policy derived from the Policy Agent aligns well with the current environment.

This process allows for a comparison with the performance of a reinforcement learning algorithm specifically constructed for a given environment, offering valuable insights into the efficacy of the generalized policy.

4.2 Algorithms

We begin our work by implementing algorithms based on the fundamental concepts proposed in Rajeswaran et al. [2021]. The underlying idea is to facilitate cooperative learning between policy agents and model agents in an adversarial dynamic, aiming to achieve a common goal.

The game is formulated as a Stackelberg game, resulting in two asymmetric versions of the algorithm. In one version, the policy agent assumes the role of the leader, with the model agent as the followers, and vice versa in the alternate version.

Our approach to solve the problem involves approximate bi-level optimization. Policy players seek to maximize the expected cumulative reward, while model players aim to fit the data collected by the policy player during its execution in the environment.

In both algorithms, the leader tends to adopt a conservative approach, while the follower responds accordingly. In the case where the policy serves as the leader, the model dynamically adapts to explain the data collected by the policy. Conversely, in the case where the model assumes the leader role, the model updates itself while maintaining a conservative approach, and the policy adjusts to maximize

expected rewards concerning the model parameters.

4.2.1 Objective Functions' Choice

We chose to model our problem as an adversarial play between a maximization and minimization problem, following the strategy proposed in the reference paper Rajeswaran et al. [2021]. This formulation allows a cooperative dynamic between the model and the agent, where the objectives of the follower and the leader are not in conflict, facilitating the convergence of the algorithms.

The decision to formulate the problem in a way that the model tends to minimize its loss is also supported by the challenges highlighted in Huang et al. [2022]. In this paper, the authors demonstrate that in most adversarial dynamics, where both agents aim to maximize the return, the common assumption is a zero-sum game, and the follower tends to minimize the expected return of the leader. This leads to the generation of extremely difficult, if not unsolvable, environments that can arrested the learning process of the leader.

In response to this issue, Huang et al. [2022] proposes a robust formulation of the policy gradient for adversarial training, providing a solution to adversarial dynamics that tend to create unsolvable models. However this specific formulation was not explored in our work.

However, it is worth noting that it would have been possible to define a cost function to be maximized for the model as well, as the necessary parameters are defined, parameterized, and updated also for the model agent, as will be explained in the upcoming sections. However, this implementation choice was not made.

4.3 Policy as Leader

In our implementation of both Policy as Leader (PAL) algorithm for maze solving, we adhere to the algorithmic framework introduced in Rajeswaran et al. [2021].

4.3.1 Redefine objective function

The primary objective of the policy agent is to maximize the cost function, aiming to find a policy π such that:

$$\max \mathbb{E}[G_t|S_t = s]$$

Here, G_t represents the expected discounted return.

Definition 4 (Expected Discounted Return)**.** *The expected discounted return is the cumulative reward discounted of factor γ return at time t*

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

We can reformulate this problem as finding a policy π that maximizes the expected cumulative reward G_t, conditioned on following to that strategy through the game.

This problem is equivalent to maximizing the value function, i.e., the objective transforms into finding a policy π that maximizes the value function $v(s, \pi)$.

Definition 5 (Value function)**.** *The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter.*

$$\begin{aligned} v(s,\pi) = v_\pi(s) &:= \mathbb{E}[G_t|S_t = s, \pi] \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, \pi\right] \\ &\qquad \forall s \in S, \text{ under policy } \pi \end{aligned}$$

Similarly, we can also define the concept of the action-value function.

Definition 6 (Action-value function)**.** *The action-value function $Q(s, a)$ gives the expected*

return for taking action a in state s under policy π

$$q(s,a) := \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a, \pi\right]$$
$$= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a, \pi\right]$$
$$\forall s \in S, \forall a \in A, \text{ under policy } \pi$$

This allows us to quantify the effectiveness of an action a in a state s at time t, assuming that from time $t+1$ ahead, the agent will follow policy π.

In particular, the following relationship exists between value function and action-value function

$$v_\pi(s) = \sum_a Q_\pi(s,a)\pi(a|s)$$

An ordering relation can be defined on the set of policies. A policy π is considered better than or equal to a policy π_0 if its expected return is greater than or equal to that of π_0 for all $s \in S$. In other words,

$$\pi \geq \pi_0 \iff v_\pi(s) \geq v_{\pi_0}(s) \quad \forall s \in S$$

There is always at least one policy that satisfies this condition, known as an optimal policy. Although uniqueness is not guaranteed, all optimal policies are denoted by π^*, and they share the same state-value function, referred to as the optimal state-value function, denoted v^*.

Definition 7 (Optimal value function)**.**

$$v^*(s) := \max_\pi v_\pi(s) \, \forall s \in S$$

Our objective is then redefined as

$$v^*(s) = \max_\pi v_\pi(s)$$
$$= \operatorname*{argmax}_\pi \mathbb{E}[G_t|S_t = s] \quad \forall s \in S$$

So we aim to find a policy that corresponds to this optimal value function.

Similarly, we can also define the optimal action-value function.

Definition 8 (Optimal action-value function)**.**

$$q^*(s,a) := \max_\pi q_\pi(s,a) \quad \forall s \in S, \quad \forall a \in A.$$

The optimal action-value function provides the expected return for taking action a in state s and thereafter following an optimal policy.

Furthermore, we can express q^* in terms of v^* as follows:

$$q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a]$$

At this point, we can conclude that in order to find a policy that optimizes our objective, we can seek a policy corresponding to the optimal action-value function.

4.3.2 Model Optimization

To achieve goal defined before, we need a way to estimate a policy that corresponds to this optimal action-value function. One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning Watkins and Dayan [1992], defined by:

$$Q(St, At) \leftarrow Q(St, At) +$$
$$\alpha\left(R_{t+1} + \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right)$$

In this case, the learned action-value function, Q, directly approximates q^*, the optimal action-value function, independent of the policy being followed.

The theorem demonstrating the convergence of Q to q^* as the number of steps $n \to \infty$ was providing in Watkins and Dayan [1992].

The pseudo code for the Q-learning algorithm is provided in Algorithm .3 in Appendix A.

By employing Q as a model parameter updated through episodes executed under the policy in the environment, we aim to optimize the model parameters.

To evaluate the model, we utilize the Mean Squared Error (MSE) as a loss function, quantifying the discrepancy between two sequential approximations of the quality function Q at time steps k and $k+1$ as $\hat{Q}^k$ and $\hat{Q}^{k+1}$, respectively.

Definition 9 (Mean Squared Error)**.**

$$MSE(\hat{Q}^k, \hat{Q}^{k+1}) = \frac{1}{N}\sum_{i=1}^{N}(\hat{Q}^k(s_i, a_i) - \hat{Q}^{k+1}(s_i, a_i))^2$$

where N represents the total number of state-action pairs in the dataset.

4.3.3 Policy Improvement

Having obtained the optimized approximation of the model, we proceeded to improve the policy with the aim of maximizing its quality function, adapting to the model parameters. To achieve this, we employed a Policy Gradient Method.

In this context, the goal is to maximize a reward function with respect to the policy parameters. The algorithm iteratively updates the policy parameters by following the gradient of the reward with respect to these parameters:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Here, θ represents the policy parameters, α is the learning rate, $J(\theta)$ is the objective function and $\nabla_\theta J(\theta)$ is the gradient with respect to the policy parameters.

This approach allows us to update the policy in the direction of the maximum increase in reward at each iteration, thereby contributing to the learning of an optimal strategy.

Theorem 1 (Policy Gradient Theorem)**.** *The policy gradient method guarantees convergence for policy gradient methods:*

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a),$$

where the distribution μ is the on-policy distribution under π.

The proof of the theorem for the episodic case is given in *Chapter 13: Policy Gradient Methods* in Sutton and Barto [2018].

Following from the policy gradient theorem, we can write

$$\begin{aligned}
\nabla J(\theta) &\propto \\
&\propto \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \theta) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \theta) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \theta)}{\pi(a|S_t, \theta)} \right] \\
&\quad \text{(replacing } a \text{ by the sample } A_t \sim \pi) \\
&= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] \\
&\quad \text{(because } \mathbb{E}_\pi[G_t|S_t, A_t] = q^\pi(S_t, A_t)) \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right].
\end{aligned}$$

So, we instantiate our stochastic gradient-ascent algorithm as

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha \sum_a \hat{q}_\pi(S_t, a) \nabla \pi(a|S_t, \theta) \\
&= \theta_t + \alpha \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}.
\end{aligned}$$

This updating is given by the Monte-Carlo Policy Gradient Algorithm, pseudo code is reported in Algorithm .4 in Appendix A.

From these updated parameters θ, we derive the policy π using a softmax function:

$$\pi(a|s, \theta) = \frac{e^{\theta^T \phi(s,a)}}{\sum_{a'} e^{\theta^T \phi(s,a')}}$$

$\phi(s, a)$ represents the state-action features function that capture relevant information for action selection in a specific state, encoding information about permissible actions for each state. This representation is necessary because not all state-action pairs are allowed. In our practical implementation, where both action

and state spaces are discrete and finite, the function $\phi(s, a)$ essentially takes the form of a binary mask, a matrix consisting of 0s and 1s. This matrix represents the possible actions a taken from state s, with 1 indicating feasibility and 0 otherwise.

This way, we obtain a stochastic policy. Actions corresponding to values of θ will have a higher probability of being selected, but other actions will still have non-zero probabilities. This allows for exploration and strikes a balance between exploiting current knowledge and exploring new possibilities.

4.4 Model As Leader

Also for the implementation of the Model As Leader (MAL) algorithm, we began with the version introduced by Rajeswaran et al. [2021] and adapted it to suit our environment and objectives.

4.4.1 Policy Optimization

Once again, following the same reasoning as before, we can express the policy agent's objective function in terms of the action-value function. As demonstrated in the PAL algorithm, finding a policy that maximizes our objective is equivalent to seeking a policy corresponding to the optimal action-value function.

Similarly, at each iteration, we can derive the optimal quality function. This involves starting from the action-value function modeled by the model agent through interaction with the data obtained from the interaction of the policy agent with the environment, mapping the state space from the model to that of the policy, and optimizing the obtained action-value function through the Q-learning algorithm.

Following the computation of the optimal action-value function Q^*, we proceed to derive a stochastic policy from the action-value function $q(s, a)$ by using a softmax policy:

$$\pi(a|s) = \frac{e^{q(s,a)}}{\sum_{a'} e^{q(s,a')}}$$

This policy introduces a level of stochasticity by treating the exponentiated action-values as probabilities. The action with a higher action-value will have a higher probability of being selected, but other actions still have non-zero probabilities. This approach facilitates exploration and strikes a balance between exploiting current knowledge and exploring new possibilities.

The optimized policy is then executed in the environment at each iteration to collect data for further model improvement.

4.4.2 Model Improvement

To improve the model using experience, we estimate the transition matrix between states, P, from the data collected during the execution of the policy in the environment. The estimation of the probability $P(s, a)$, i.e., the probability of choosing action a from state s, is computed as the ratio of the times action a was chosen from s during the episode to the total number of times in which s was visited during the episode.

From P, we aim to estimate q, the action-value policy, in order to use it for policy optimization. q is computed by decomposing it as defined in definition 6. We can express q as follows:

$$\begin{aligned} & q(s, a) \\ & \quad = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a, \pi\right] \\ & \quad = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \cdot \max_{a'} q(s', a') \end{aligned}$$

where $R(s, a)$ is the expected reward from taking action a in state s, and $P(s'|s, a)$ is the transition probability from state s to state s' under action a. Note that in our environment at each pair state-action correspond an unique

arrival state, so $P(s'|s,a)$ can be written as $P(s,a)$

4.5 Stopping Criteria

In the pursuit of understanding when and why to arrest the learning process, we explore the identification of equilibrium in the adversarial dynamic. Specifically, we aim to identify the Stackelberg equilibrium, providing insights into the optimal termination points for the adversarial interaction.

At the conclusion of each interaction, our goal is to determine whether the strategies independently chosen by each player, intending to minimize or maximize their respective objectives, are in equilibrium.

It's essential to note that, in all the subsequent considerations, we have treated the Stackelberg game with the formulation typically presented, where the objective for both players is to maximize utility. If we apply this framework to our case, we encounter a mixed optimization problem.

As previously observed, the objective function for the policy player is to maximize cumulative rewards, while the objective for the model player is to minimize the loss. To address this duality, we can transform the minimization objective into a maximization one. Therefore,

$$\underset{M}{\text{minimize}}\, l(M) = \underset{M}{\text{maximize}}\; -l(M)$$

This transformation aligns both objectives as maximization problems, providing a unified framework for analysis.

To solve the game, we employ a backward induction approach. We begin by selecting the strategy x_2 for the follower, given the leader's strategy x_1, such that

$$x_2 = \underset{x_2 \in S_2}{\text{argmax}}\; u_2(x_1, x_2)$$

This entails finding the best response $R_2(x_1)$. After the leader has chosen a strategy, we envisage that the follower consistently selects the best response.

It's worth noting that if $R_2(x_1)$ is not a singleton but rather a set of best responses, the utility for the leader, $u_1(x_1^*, R_2(x_1^*))$, is not well defined. Consequently, we are unable to establish a unique equilibrium, as defined in definition 1.

A suitable solution to this issue could involve formulating the problem as an Optimistic Stackelberg Problem, in order to extending the concept of the Stackelberg strategy for a nonzero-sum two-person game to accommodate a non-unique rational response from the follower.

In the optimistic formulation, the leader is interested in maximizing their own objective, assuming that the follower will respond optimally to the leader's actions.

The problem is thus presented as a bilevel optimization problem in x_1. Therefore, the search is for the optimistic Stackelberg equilibria, i.e., the maximum point (x_2^*, x_1^*) of

$$max\{u_1(x_1, x_2) : x_1 \in S_1, x_2 \in R_2(x_1)$$

An alternative approach was proposed in Leitmann [1978], introducing a Pessimistic Formulation of the Stackelberg problem that employs a security strategy for the leader. This strategy enables the leader to minimize the risk associated with follower responses and proves effective in scenarios where the follower can react efficiently to the leader's actions.

In Leitmann's proposed solution, the requirements are relaxed as follows: Let's define the set of optimal best responses of the follower to a given decision by the leader as

$$R_2^*(x_1) := \{x_2^* \in S2| \text{ given } x_1 \in S_1,\\ u_2(x_2^*, x_1) \geq u_2(x_2^*, x_1), \forall x_2 \in S_2\}$$

In the scenario where $R_2^*(x_1)$ is not a singleton, we can define a generalized Stackelberg strategy for the leader as follows:

Definition 10 (Generalized Stackelberg Strategy Leader)**.** *if* $\forall x_1 \in S_1$, $S_2^*(x_1) \neq \emptyset$, *and if* $\exists x_{1s1}^* \in S_1$ *s.t*

$$\inf_{x_2 \in S_2(u_{1s1}^*)} u_1(x_{1s1}^*, x_2) = \max_{x_1 \in S_1} \inf_{x_2 \in S_2^*(x_1^*)} u(x_1, x_2)$$

than $x_{1s1}^* \in S_1$ *is a generalized Stackelberg strategy for the leader(player 1).*

Despite considering the second approach based on the pessimistic formulation, the nature of the problem, in which an adversarial dynamic is established between the two agents, both trying to maximize their individual objective functions to achieve the common goal of generating a policy for solving novel maze problems, suggested the adoption of the Optimistic Stackelberg Problem solution in the implementation. This decision takes into account the pursuit of leadership by both agents to maximize their respective objectives.

5 Implementation

In this section, we document the formalization and implementation of our problem.

5.1 Environment

5.1.1 Maze Generator

The initial step toward applying our algorithms involves the generation of random mazes. Our approach employs a spanning tree algorithm, ensuring the creation of mazes that are both acyclic and fully connected. This process involves the creation of two-dimensional rectangular mazes, designed to operate as spanning trees. The key objective is to establish a distinct path from any node to every other node within the maze.

Definition 11 (Spanning Tree)**.** *A connected undirected graph that uses all the nodes in which there are no circuits.*

To execute this process, we leverage the algorithm developed by Kalle Saariaho[1], aligning with the theoretical approach outlined in Wilson [1996]. This algorithm serves as our foundational framework, adapted to ensure compatibility with the environment and the capability to seamlessly integrate with our subsequent algorithms.

The dimensions of the maze can be freely selected, with the graphical representation based on window size and maze block size in pixels. Additionally, the maze can be visualized using PyGame. A rendering of the maze is provided in Figure 3 in Appendix B.

5.1.2 MDP Environment

To represent the environment, we leverage a *Third-Party Environment* from the Gymnasium, an API standard for reinforcement learning featuring a diverse collection of reference environments. Gymnasium, a maintained fork of OpenAI's Gym library, *"provides a simple and pythonic interface capable of representing general reinforcement learning problems"*Gym.

Specifically, we utilize the **Matrix MDP gym** library implemented by Paul Festor[2] and explained in Festor [2023]. This library extends OpenAI's environment by formalizing it as a Markov Decision Process (MDP), enabling the agent to take actions in the environment and receive feedback in the form of current states and rewards.

The key parameters of the environment, as well as the input and output for the **step()** method, are summarized in Table 2 in Appendix B

This formalization allows interaction with the

[1]The **Maze Generator** implementation is detailed in this repository.

[2]The **Matrix MDP gym** library implementation is documented in this repository.

environment, establishing a standardized interface for integration with our maze generator. The maze generator plays a pivotal role in initializing the environment by providing the transition matrix between states, permissible actions for each state, and rewards associated with each state-action pair.

Within this environment, the terminal state is randomly generated by the library. On the other hand, the initial state is intentionally placed in one of the four corners of the rectangle containing the maze, specifically chosen to maximize the distance between the initial and terminal states.

This design choice is aimed at presenting the agent with a diverse and challenging set of maze-solving scenarios, enhancing the adaptability and robustness of the algorithms under examination. This ensures their efficacy across a spectrum of maze configurations.

5.2 Agents

Following the implementation presented in Rajeswaran et al. [2021] and the theoretical formulation of our problem discussed in Section 4, our agents were formulated as two distinct classes that interact with each other in an adversarial dynamic.

5.2.1 Model Agent

The **ModelAgent class**, extend the **Agent class**, is tasked with simulating an agent in an environment. It utilizes a model to approximate the environment and make decisions based on that model. The **Agent class** encapsulates methods for rendering and making the path followed by the agent visible using PyGame.

The parameters required for initializing the class are detailed in Table 6 in Appendix B.

The Model Agent constructs a representation of the environment without directly interacting with it. This is achieved through experience, a series of episodes collected via the policy agent's interaction with the environment. Each episode consists of steps: (state, action, reward, next state), indicating the initial state of the step, performed action, reward received from the environment, and resulting state. Using this information, the agent iteratively refines its knowledge of the environment by updating its parameters.

The action space A for the model agent is discrete and finite, known a priori. It consists of the four cardinal actions that the agent can take from each state.

The state space S is discrete and initially unknown. The agent constructs its representation through experience. Importantly, the agent remains unaware of its absolute position in the environment. Unless it comprehensively explores the state space, it will never gain complete knowledge. However, exhaustive exploration would be impractical, making the algorithm inefficient. The agent's primary objective is to deduce the path between the initial and terminal states.

To build its state space, the agent maps coordinates received from the environment during episodes. These coordinates are organized in an ordered space where each state maps to the next via the action connecting them.

The agent only recognizes actions explored from a given state as possible. Consequently, for each explored state, the set of possible actions represents only a subset of those available in the environment.

Similarly, rewards associated with each state-action pair are also stored.

Also, the state function and action-value function parameters are introduced, whose values are computed based on the executed algorithm.

5.2.2 Policy Agent

The **PolicyAgent class** represents the policy agent. It's implemented in a way that enables it to interact with the environment, selecting actions at each step based on the saved policy.

Its action space, analogous to that of the ModelAgent, is discrete and finite, as defined in Section 4. The state space, also discrete and not known a priori, is built through experience.

The agent initializes with an ϵ-greedy policy for every state added to the state space. Initially assigning the maximum value to the first possible action to the left of the agent and proceeding clockwise. The policy is then updated based on the algorithm.

Definition 12 (ϵ-greedy policy for PolicyAgent)**.**

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{|A|} + 1 - \epsilon & \textit{if } a \textit{ is the first possible} \\ & \textit{action agent's left} \\ \frac{\epsilon}{|A|} & \textit{otherwise} \end{cases}$$

Additionally, parameters for the value function and action-value function are present, and their values are computed based on the executed algorithm.

5.3 Algorithms

We implement two different classes, MAL and PAL, for their respective algorithms. Both classes extend the **MazeSolver class**, which is responsible for training a policy agent and a model agent in multiple environments.

The implementation structure for both classes is consistent. Each algorithm consists of three nested training loops. The outermost loop iterates over the environments. For each environment, the parameters of the model and the environment itself are initialized, and the model agent and policy agent engage in an adversarial dynamic until reaching equilibrium. Once equilibrium is achieved, the parameters of the policy agent are retained, while the rest are reinitialized.

This process is repeated for each environment, iterating until convergence. The third loop is introduced due to the execution of the policy in the environment. At each iteration of the algorithm, the policy is executed a fixed number of times to collect episodes. A maximum step limit is set to control the number of steps per episode.

Algorithms .5 and .6 in Appendix A reported the pseudo codes.

5.3.1 Maze Solver

This class is responsible for initializing the environment, policy agent, and model agents. The **MazeSolver class** solves a maze using a specified algorithm. The parameters taken by the class and implemented methods are reported in Tables 3 and 4 in Appendix B.

5.3.2 PAL Algorithm

PAL class extends the **MazeSolver class** and implements the algorithm explained in Section 4.3.

5.3.3 MAL Algorithm

MAL class extends the **MazeSolver class** and implements the algorithm explained in Section 4.4.

5.3.4 Stackelberg Equilibrium

In a theoretical context, finding the Stackelberg Equilibrium involves computing all possible strategies for both players, determining the payoffs, creating the payoff matrix, and identifying equilibria, as discussed in Section 4.5. We then evaluate whether the resulting equilibrium strategies align with individual strategies that would have minimized or maximized the objective functions of both players.

In the practical implementation of our algorithms, this process is simplified. In both algorithms, starting from the moment the policy is executed in the environment, the model is updated, and the subsequent policy is computed from this model update.

Taking the example of the MAL algorithm, in our practical implementation, all data collected at each iteration are used to update the model at the same time. Consequently, there is only one strategy to which the policy can respond, constituting a unique best response. Thus, equilibrium is maintained.

The decision to implement a single update was driven by the goal of expediting training, based on initial observations that using episodes collected separately to optimize multiple models and subsequently computing corresponding policies did not lead to a significant improvement. In these observations, the optimal policy consistently corresponded to the one computed using the parameters of the best model. Therefore, we opted to use all the collected data for updating the model in each iteration, aiming to expedite training by presenting more data. This choice is also justified, considering that data collection is computationally expensive due to the calls to the environment library.

It is important to note that this outcome is derived from practical experiments on relatively small maze dimensions and may not be generalized. For larger models, this implementation choice may not be justified.

In the case of the PAL algorithm, we computed all possible strategies for both the model and the policy. At each iteration, we checked whether the algorithm was in equilibrium or not.

6 Experiments

In our experiments, we utilized maze environments with a width of 101 and a height of 51. The discount factor γ was set to 0.995, reflecting the suggested value in Rajeswaran et al. [2021]. Learning rate of 0.01 was employed for both Monte Carlo updates and Q-learning. The experiments involved training on an number of different environments between 5 and 30, with a maximum of 10 iterations per environment to reach equilibrium. Each iteration of the algorithm collected data from 3 episodes, each with a maximum of 6000 epochs, after, if terminal state was not reached from the agent, the episode terminate. The epsilon value for epsilon-greedy policy initialization was set to 0.1. These parameter choices aimed to balance the trade-off between exploration and exploitation for effective maze-solving learning. These values define the configuration used to conduct the experiments, influencing the algorithm's behavior and performance in maze-solving tasks.

The choice of environment dimensions for testing was a necessary trade-off. A larger environment would have required a significantly greater number of steps per episode to approach to the terminal state, at least after an initial training phase, demanding more memory and computational power than available. On the other hand, smaller mazes tended to be too easy for the policy to solve efficiently in the first iteration, resulting in a policy only slightly deviated from epsilon-greedy. Subsequently, it tended to converge to a uniform distribution, exploring a substantial portion of the state space before reaching the terminal state.

As for the PAL algorithm, initial experiments involved training the model on 30 or 50 environments with learning rates between 0.2 and 0.1. Unfortunately, these trials did not yield satisfactory results, as the policy tended not to converge. An example of the learning curve for this set of experiments is provided in Figure 4 in Appendix C.

By lowering the learning rate and training the model on a smaller number of environments,

we managed to achieve slightly better results, although still unsatisfactory in terms of convergence. We conducted several repetitions, and not all the curves showed convergence. One of the instances that did converge is presented in Figure 5 in Appendix C.

It's worth noting that the number of iterations is higher than the number of exploring environments because, for most environments, equilibrium was not achieved during the initial iteration.

Given the lack of particularly promising results, we decided not to conduct further experiments. Our approach was predominantly theoretical, and achieving better results would have required an exhaustive grid search. The initial attempts with this implementation did not yield the expected outcomes.

The MAL algorithm consistently tended to converge towards a uniform distribution of probabilities among possible actions. In Figure 6, found in Appendix C, a heatmap illustrates the policy resulting from the training of the MAL algorithm after exploring 10 environments. During the training, we observe that the distribution tends toward a uniform distribution. This training was conducted utilizing all other default parameters described above.

In Figure 7 in appendix C are presented four mazes solved by policy agents with policies learned via the PAL algorithm. The mazes, measuring 53 units in width and 37 units in height, featured a policy corresponding to the 40th iteration (see Figure 4 in Appendix C).

Training was conducted with a learning rate of 0.1. The agents successfully navigated the maze, reaching the terminal state in under 6000 iterations—the maximum limit set during training. This achievement was consistent across all four cases.

The initial state is represented by a black square, while the agent is denoted by a blue square. In all four scenarios, the agent is positioned at the terminal state. The black line illustrates the path taken by the policy agent to reach the terminal state, mapping the visited state space en route to the goal.

It's important to note that these paths were not derived from training; rather, the saved policy was executed in newly generated environments, providing a demonstration of the learned behaviors.

7 Limitations

We use Q-learning to approximate the action-value function. This algorithm can derive the optimal policy independently of which policy was used to collect the data. Therefore, it is possible to save the executed episodes in an external buffer and use them at each iteration, combining them with new episodes obtained by executing the updated policy in the environment. This solution requires external memory to store an increasing number of data, providing a computationally less expensive solution, as executing the policy in the environment is costly.

The implementation choice of using Monte Carlo methods was made because its derivation was straightforward. However, exploring alternative options among those proposed in Rajeswaran et al. [2021] may yield improved results in this domain.

Another limitation is that the policy derived in this way is capable of learning rules only on individual states but not on sequences composed of pairs of previously executed actions and states. This could potentially limit the algorithms' performance.

8 Conclusions

In conclusion, the PAL algorithm shows some improvement but it fails to converge. On the

other hand, MAL converges to a uniform distribution.

These outcomes are consistent with the discussions in Section 4.2 and the findings in Rajeswaran et al. [2021]. The policy derived by the PAL algorithm, where the model adopts a more aggressive approach, results in non-smooth learning curves characterized by significant policy fluctuations. These fluctuations occur not only between environments but also within iterations of the same environment. This outcome is expected since the model parameters change drastically from one iteration to the next, and the policy must adapt accordingly. In contrast, in MAL, where the policy is acting as the follower and the model's parameters tend towards a conservative approach, convergence is achieved more readily. Analysis of MAL's training results reveals a convergence of the probability distribution to a uniform one. Exploring different parameter initializations could be intriguing to drive the policy towards a distinct convergence.

It is essential to note that, with careful policy selection, learning from the PAL algorithm can produce satisfactory maze-solving results. However, determining a suitable stopping criterion based on the learning curve is challenging due to rapid changes between iterations.

The same challenges apply to the Stackelberg equilibrium, preliminary training notes do not reveal significant correlations between equilibrium iterations and improved metric values. This is partly due to the metrics exhibiting similar ranges within the same environment.

Despite practical results leaving room for improvement, this work lays a solid foundation for subsequent implementations. The adversarial dynamics established between the policy and the MDP-approximating agent, incorporating the OpenAI environment, provide a groundwork for future enhancements. Moreover, the coherence between the obtained results and the theoretical framework, starting from Rajeswaran et al. [2021], instills confidence in the potential for refinement and improvement in subsequent studies.

9 Future Works

In future implementations to improve the performance of our algorithm, our goal is to introduce a policy agent capable of learning not only on individual states but also on sequences composed of pairs of previously executed actions and states.

To achieve this, we plan to implement a Policy Agent capable of extending the state space to sequences of states. This implementation will involve changes in the definition of the state space of the Policy Agent, allowing it to store sequences of (state-action-next state) tuples and subsequently extending this list.

Additionally, we will need to incorporate a buffer into the Model Agent to store pairs of state-action explored during the last n steps. This buffer will be utilized to update a specific policy, taking into consideration these recent state-action pairs.

We believe that this upgrade has the potential to significantly enhance the performance of our model.

References

Gymnasium documentation. `https://gymnasium.farama.org/`.

Paul Festor. Matrix mdp gym. `https://www.paul-festor.com/post/i-created-a-python-library`, 2023.

Peide Huang, Mengdi Xu, Fei Fang, and Ding Zhao. Robust reinforcement learning as a stackelberg game via adaptively-regularized adversarial training, 2022.

G. Leitmann. On generalized stackelberg strategies. *Journal of Optimization Theory and Applications*, 26:637–643, 12 1978. doi: 10.1007/BF00933155.

Aravind Rajeswaran, Igor Mordatch, and Vikash Kumar. A game theoretic framework for model based reinforcement learning, 2021.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Steven Tadelis. *Game theory: an introduction*. Princeton university press, 2013.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, page 296–303. Association for Computing Machinery, 1996.

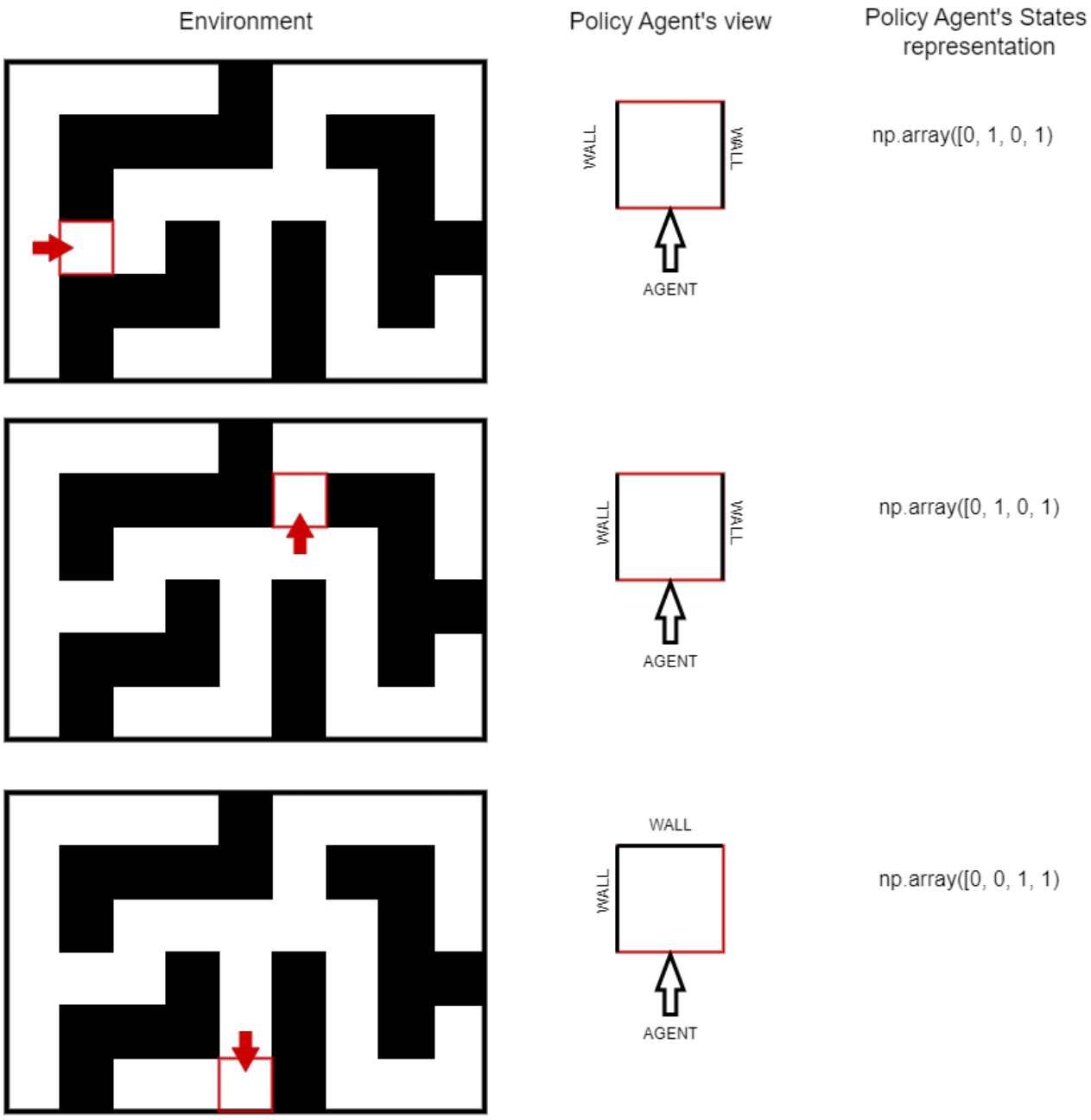

Figure 2: Policy Agent's view and representation of the states space. Each state is rendered as an array of walls (represented by 0) or free passages (rappresented by 1), starting from the agent's left and progressing clockwise. This representation ensures that the passage to the previous state, behind the agent, is consistently placed at the last position of the array.

A Algorithms

Algorithm .1: Policy as Leader (PAL) meta-algorithm

Initialize: policy π_0, model $\hat{M}_0$, data buffer D={}
for *iteration* $k = 0, 1, 2, \ldots$ **do**
  Collect data D_k by executing π_k in the environment
  Build local (policy-specific) dynamics model:

$$\hat{M}_{k+1} = \text{argmin} l(\hat{M}, D_k)$$

  Improve policy:

$$\pi_{k+1} = \pi_k + \alpha \nabla_\pi J(\pi_k, \hat{M}_{k+1})$$

  with a conservative algorithm like NPG or TRPO

Algorithm .2: Model as Leader (MAL) meta-algorithm

Initialize: policy π_0, model $\hat{M}_0$, data buffer D={}
for *iteration* $k = 0, 1, 2, \ldots$ **do**
  Optimize policy

$$\pi_{k+1} = \underset{\pi}{\text{argmax}} J(\pi, \hat{M}_k)$$

  using any algorithm
  Collect environment data D_{k+1} using π_{k+1}
  Improve model

$$\hat{M}_{k+1} = \hat{M}_k - \beta \nabla_{\hat{M}} l(\hat{M}, D_{k+1})$$

  using any conservative algorithm (mirror descent, data aggregation, ...)

Algorithm .3: Q-learning (Off-policy TD Control) for estimating $\pi \approx \pi^*$

Algorithm Parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize: $Q(s, a)$ for all $s \in S^+, a \in A(s)$, arbitrarily, except that $Q(\text{terminal}, \cdot) = 0$
for *each episode:* **do**
    Initialize: S
    for *each step of episode:* **do**
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \max_a Q(S', a) - Q(S, A)\right]$
        $S \leftarrow S'$
    until S is terminal

Algorithm .4: REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π^*

Algorithm Parameters:: step size $\alpha > 0$
Initialize: policy parameter $\theta \in \mathbb{R}^d$ (e.g., to 0)
for *each episode:* **do**
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
    for *each step of the episode* $t = 0, 1, \ldots, T-1$*:* **do**
        $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ Compute the return
        $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \ln \pi(A_t|S_t, \theta)$ Update policy parameters

Algorithm .5: Train Loop for PAL

for $k = 0, 1, 2, \ldots,$ *max environments:* **do**
    while *Stackelberg Equilibrium not found* **do**
        Loop on the environment
        Executing policy π in environment and collect data in D
        Optmizing model via Q-learning update
        Improve policy via Reinforce Monte Carlo (initial policy paramethers $\mathcal{U}[0, 1]$)

Algorithm .6: Train loop for MAL

for $k = 0, 1, 2, \ldots,$ *max environments:* **do**
    while *Stackelberg Equilibrium not found* **do**
        Loop on the environment
        Optmizing policy via Q-learning update
        Executing policy π in environment and collect data in D
        Improve model updating action-value function and transition probability

B Class Parameters and Methods

Parameter	
P_0	Prior distribution over the states implicitly containing initial state s_0
P	Transition distribution over the states, implicitly containing state space S and action space A
R	Reward function for each state of the environment
env.step() input	
action	Action taken by the agent
env.step() output	
agent state	Current state of the agent in the environment
reward	Reward from the environment

Table 2: **Environment**'s parameters and input-output for the **step()** method.

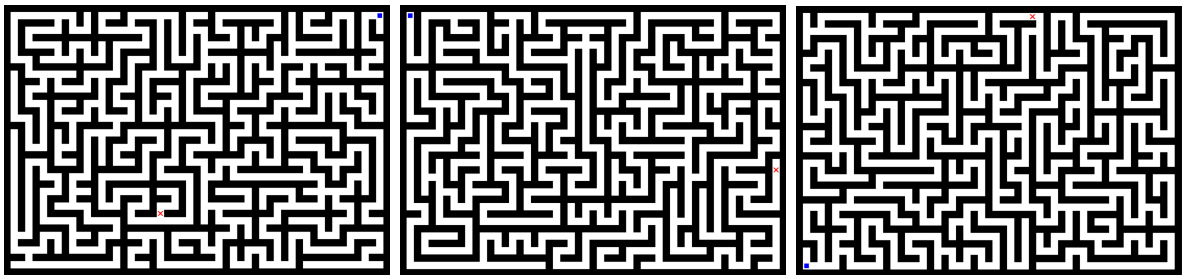

Figure 3: Render of the environment: the figures depict three randomly generated mazes created using the **MazeGenerator Class**. White cells represent the state space, while black cells indicate walls. The blue square represents the agent, initially positioned at the starting state, and the red cross denotes the terminal state.

Parameter	**Type**	
algorithm	(string)	PAL or MAL
number of environments	(int)	Number of different environments to train on
maximum iterations per environment	(int)	Number of iterations for each episode, attempting to reach the Stackelberg-Nash equilibrium of the game
number of episodes per iteration	(int)	Number of episodes to run for each iteration of the environment to train on
maximum steps per episode	(int)	Maximum number of steps in each episode to reach the terminal state
learning rate	(float)	Learning rate for policy improvement
alpha	(float)	Learning rate for value function update
gamma	(float)	Discount factor to compute expected cumulative reward
epsilon	(float)	Epsilon-greedy parameter

Table 3: **MazeSolver class** parameters.

Method	
executing policy	Method for executing the policy expressed in policy agents in the environment to solve the maze. Each execution is saved in a data buffer as an episode, a list of state-action-reward-next state
reset environment	Reset environment to a new one after agents reach equilibrium for the previous one
initialize environment and agents	

Table 4: **MazeSolver class** methods.

Parameter	**Type**	
gamma	(float)	discount factor used in reinforcement learning algorithms
agent state	(int)	current state of the agent
transition distribution	(np.darray)	matrix representing the probability distribution over actions for each state
next state function	(dict)	dictionaries representing various aspects of the environment and the agent's knowledge of it
reward function	(dict)	
states space	(dict)	
values function	(dict)	
quality function	(dict)	

Table 5: **ModelAgent class** parameters.

Parameter	**Type**	
gamma	(float)	discount factor used in reinforcement learning algorithms
policy	(np.darray)	matrix representing the probability distribution over actions for each state
states space	(dict)	dictionaries representing various aspects of the environment and the agent's knowledge of it
values function	(dict)	
quality function	(dict)	

Table 6: **PolicyAgent class** parameters.

C Training Outcomes

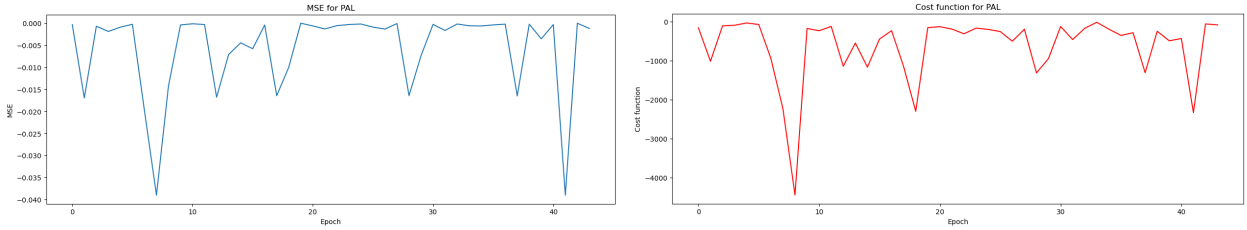

Figure 4: Learning curve for the PAL algorithm: the left plot (in blue) represents the Mean Squared Error (MSE), and the right plot (in red) illustrates the policy Cost Function. The algorithm was trained on 30 environments, with a learning rate α of 0.1 for Monte Carlo. Other parameters are set to their default values, as detailed in Section 6. Both curves exhibit significant fluctuations without clear convergence. Note that the y-axes of the two graphs have different scales.

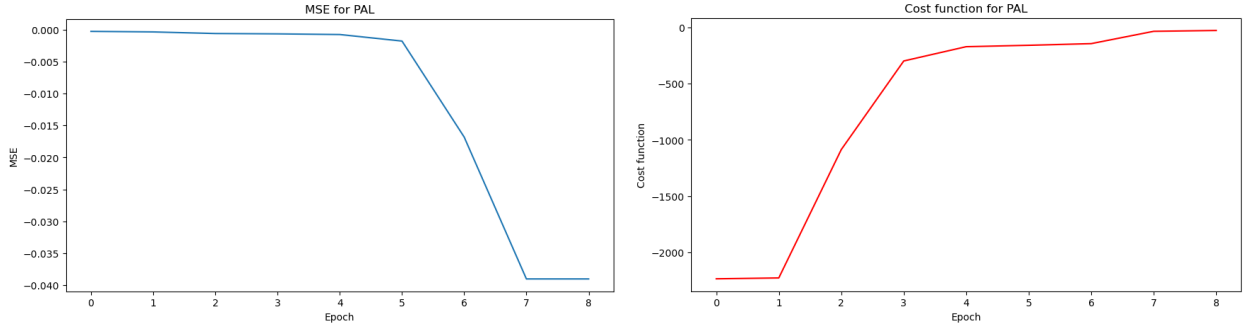

Figure 5: Learning curve for the PAL algorithm: the left plot (in blue) represents the Mean Squared Error (MSE), and the right plot (in red) illustrates the policy Cost Function. The algorithm was trained on 5 environments, with a learning rate α of 0.01 for Monte Carlo. Other parameters are set to their default values, as detailed in Section 6.

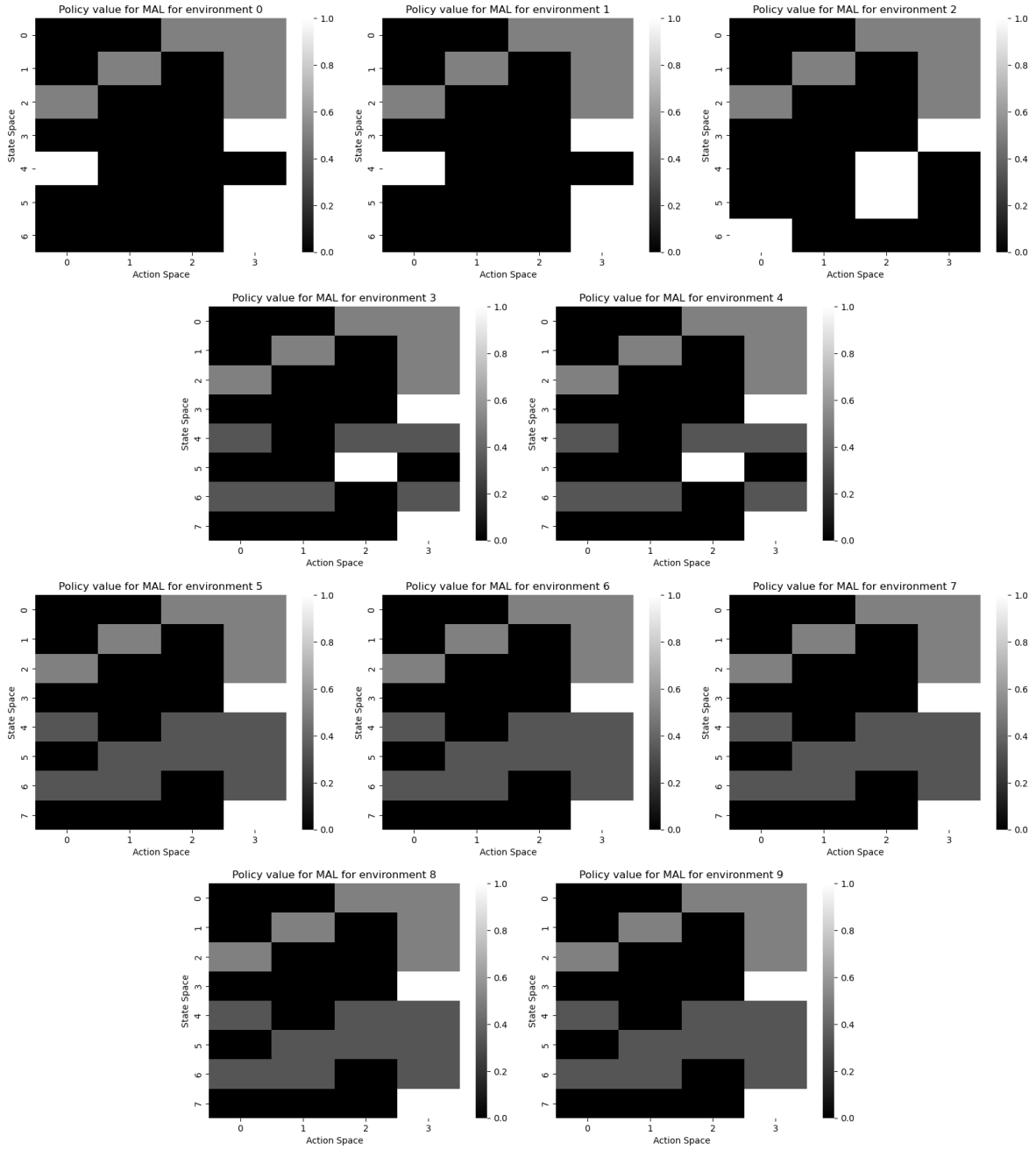

Figure 6: Heatmap of the policy resulting from the training of the MAL algorithm after exploring 1, 2, ..., 10 environments. White cells represent values closer to one, indicating higher probabilities to takes that action, while darker colors represent probabilities closer to zero. It is essential to note that states are defined such that action 0 is to the right of the agent, and subsequent actions proceed clockwise. Therefore, action number 3 corresponds to the agent returning to the previous state. As observed, after training on the initial environments, higher probabilities are consistently associated with action 3. With continued training, the distribution tends toward uniformity. Note that black cells, associated with a transition probability of 0, represent actions not permitted for that state, where walls are present.

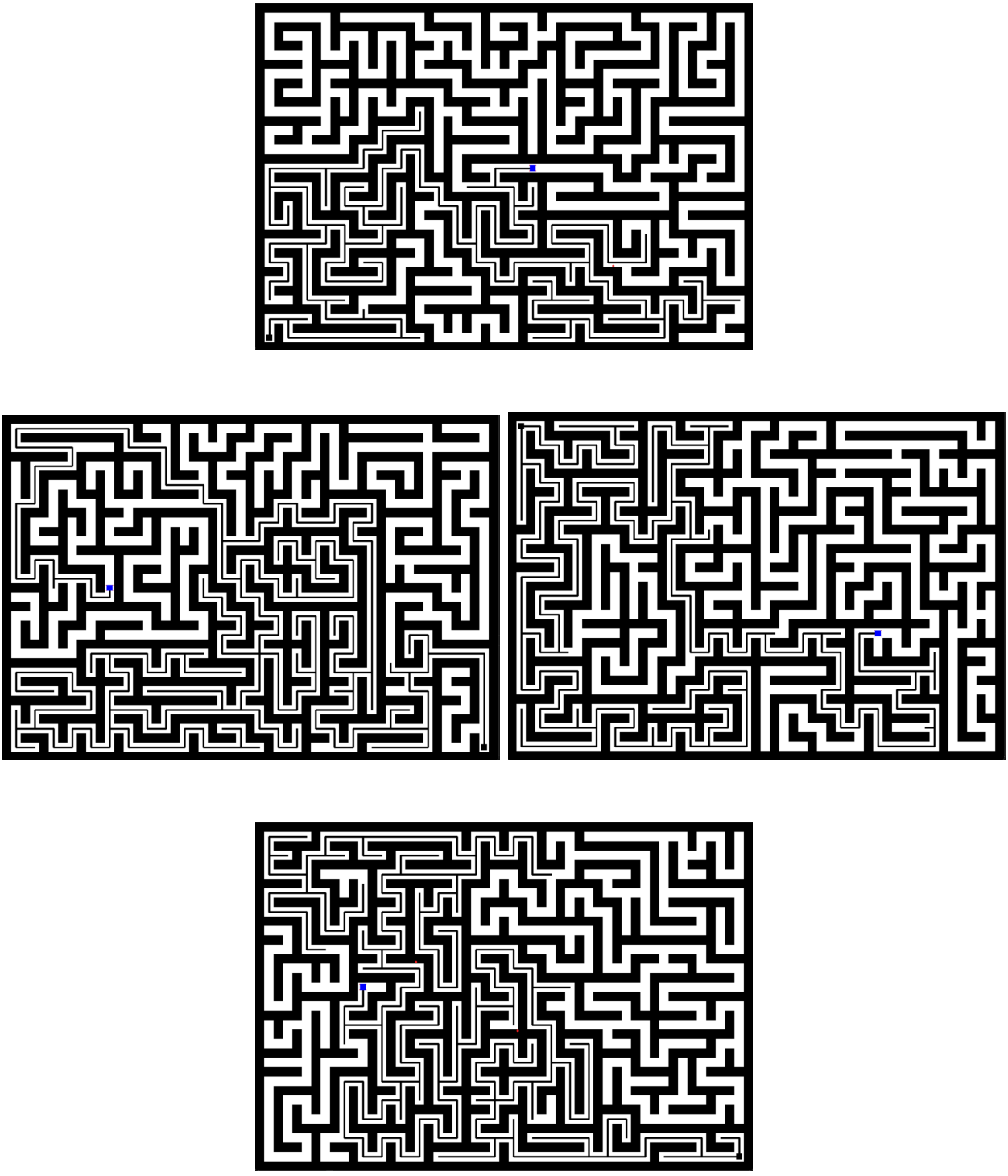

Figure 7: Mazes solved by policy agents with policies learned via the PAL algorithm. The maze, measuring 53 units in width and 37 units in height, featured a policy corresponding to the 40th iteration (see Figure 4 in Appendix C). Training was conducted with a learning rate of 0.1. The agents successfully navigated the maze, reaching the terminal state in under 6000 iterations—the maximum limit set during training. This achievement was consistent across all four cases. The initial state is represented by a black square, while the agent is denoted by a blue square. In all four scenarios, the agent is positioned at the terminal state. The black line illustrates the path taken by the policy agent to reach the terminal state, mapping the visited state space en route to the goal.