Lab1 RF

Carlotta Hölzle
csholzle@kth.se
20020521-8167

Jannis Eden
jannise@kth.se
20010411-1653

November 2024

1 Basic maze

State Space $\mathcal{S}$:
The state space $\mathcal{S}$ in the code is defined as all possible combinations of positions of the player and the minotaur within the maze, along with two terminal states:
$\mathcal{S} = \{((x_1, y_1), (x_2, y_2)) \mid 0 \leq x_i \leq H, 0 \leq y_i \leq W, \text{ maze}[x_1, y_1] \neq 1\} \cup \{\text{Eaten}, \text{Win}\}$

- (x_1, y_1) represents the player's coordinates.
- (x_2, y_2) represents the minotaur's coordinates.
- W and L are the width and length of the maze.
- 'Eaten' and 'Win' represent terminal states when the player is caught by the minotaur or successfully exits, respectively
- Because the minotaur can be within a wall, only the player can not, the player's x and y coordinates can not be those where the maze value equals 1.

Action Space A:

- The action space $\mathcal{A}_p$ for the player can be formalized as followed:
  $\mathcal{A}_p = \{$stay, move left, move right, move up, move down$\}$
- The action space $\mathcal{A}_m$ for the minotaur can be formalized as followed:
  $\mathcal{A}_m = \{$move left, move right, move up, move down$\}$
  Because here the minotaur can not stand still.

Reward Function $R(s, a)$ We want to reward the player to find the exit, therefore any action that leads to reaching this goal is prositively rewarded. Any other action does not yield any reward. We assign a negative reward for when the minotaur catches the player such that the player learns to avoid the minotaur. Originally we also assigned a high negative reward for any not possible action

of the player e.g. the wall. However, we realized that we are not allowing moves which lead to impossible states, e.g. the wall, therefore this reward was never needed, thus we excluded it from the reward function. We also do not assign negative step reward, if it is not mandatory to find the exit quickly.
Terminal rewards:

- $r(\text{Eaten}, a) = \text{MINOTAUR\_REWARD} = -1$
- $r(\text{Win}, a) = \text{GOAL\_REWARD} = 1$

Non-terminal rewards:
For valid moves that do not result in winning or being eaten:

- $r(s, a) = \text{STEP\_REWARD} = 0$

Transition Probabilities $P(s\prime|s,a)$ Let $P(s\prime \mid s,a)$ be the probability of transitioning from state s to $s\prime$ after taking action a. The player has deterministic movement and the minotaur moves to any valid adjacent cell, and its probability of transitioning to a new state depends on the number of possible moves. Therefore, mathematically the transition probabilities for non-terminal states is:

$$P(((x\prime_1, y\prime_1), (x\prime_2, y\prime_2)) \mid ((x_1, y_1), (x_2, y_2)), a) = \tfrac{1}{N}$$

Where N is the number of possible cells the minotaur can move to, and:

- $x\prime_1, y\prime_1$ are the new coordinates of the player after applying action a.
- $x\prime_2, y\prime_2$ are the new coordinates of the minotaur after moving randomly.

For terminal states:

$$P(\text{Eaten} \mid ((x_1, y_1), (x_2, y_2)), a) = 1 \quad \text{if } (x\prime_1, y\prime_1) = (x\prime_2, y\prime_2)$$
$$P(\text{Win} \mid ((x_1, y_1), (x_2, y_2)), a) = 1 \quad \text{if } (\text{maze}[x\prime_1, y\prime_1] = 2)$$

Once these states are reached, they have a self-loop probability of 1, meaning:

$$P(\text{Win} \mid \text{Win}, \cdot) = 1, \quad P(\text{Eaten} \mid \text{Eaten}, \cdot) = 1$$

The transition probabilities assume that $\text{maze}[x\prime_1, y\prime_1] \neq 1$ because these are not defined as possible states in $\mathcal{S}$. Furthermore, the player will not move to non-adjacent cells as these are not reachable through any actions defined within $\mathcal{A}_p$.

1.1 b

If the player and minotaur move simultaneously the player's strategy needs to depend on anticipating the Minotaur's random walk probabilities to avoid potential collisions. Since the Minotaur moves randomly, its position might "jump" into the player's position unexpectedly, increasing the risk of being caught. If we have alternating moves the player has an advantage because they can always react to the Minotaur's last move before deciding their own. This reduces the risk of unexpected collisions since the player has a full step to avoid the Minotaur based on its new position. The differences in modelling the MDPs is only in the transition propabilities. If we have alternating moves and it's the player's turn: Only the player moves, and the minotaur stays stationary. And if it's the minotaur's turn: Only the minotaur moves, and the player stays stationary.

- On the player's turn, only the player moves:
  $P((x\prime_P, y\prime_P, t{+}1)|(x_P, y_P, x_M, y_M, t), a_{\text{player}}) = P((x\prime_P, y\prime_P, t{+}1)|(x_P, y_P, t), a_{\text{player}})$
- On the minotaur's turn, only the minotaur moves:
  $P((x\prime_M, y\prime_M, t+1)|(x_P, y_P, x_M, y_M, t)) = P((x\prime_M, y\prime_M)|(x_M, y_M))$

2 Dynamic Programming

2.1 c

To find the optimal path, it is enough to set the time horizon T=15. The player needs at least 15 steps to attain the exit B, if the starting point is at (0,0). If the time horizon less than 15, no path to the exit is found. For all $T \geq 15$ the path stays the same for the same movements of the minotaur. To illustrate the policy either run the code provided in the .ipynb version of this report or look at Figure 1 which manually painted the path taken by both player and minotaur.

2.2 d

For T = 1, . . . , 30 compute a policy that maximizes the probability of exiting the maze alive (in the shortest time possible) and plot the probability. Is there a difference if the minotaur is allowed to stand still? If so, why?

The optimal policy that maximizes the probability of exiting the maze alive in the shortest time possible happens for T=15. For this time horizon the probability of exiting the maze alive is 100%. In the jupyter notebook you will find a function which prints this policy.

Looking at 2 one can see that there is a difference if the minotaur is allowed to stand still or not. If the minotaur is not allowed to stand still it is much simpler to win for the player. The player can just move to the state the minotaur was at previously and be certain it will not be there in the next move to eat the player. When the minotaur is allowed to move this strategy does not work. Especially with a high time horizont e.g. T ¿ 26 it becomes very likely that the minotaur finds the player at one point.

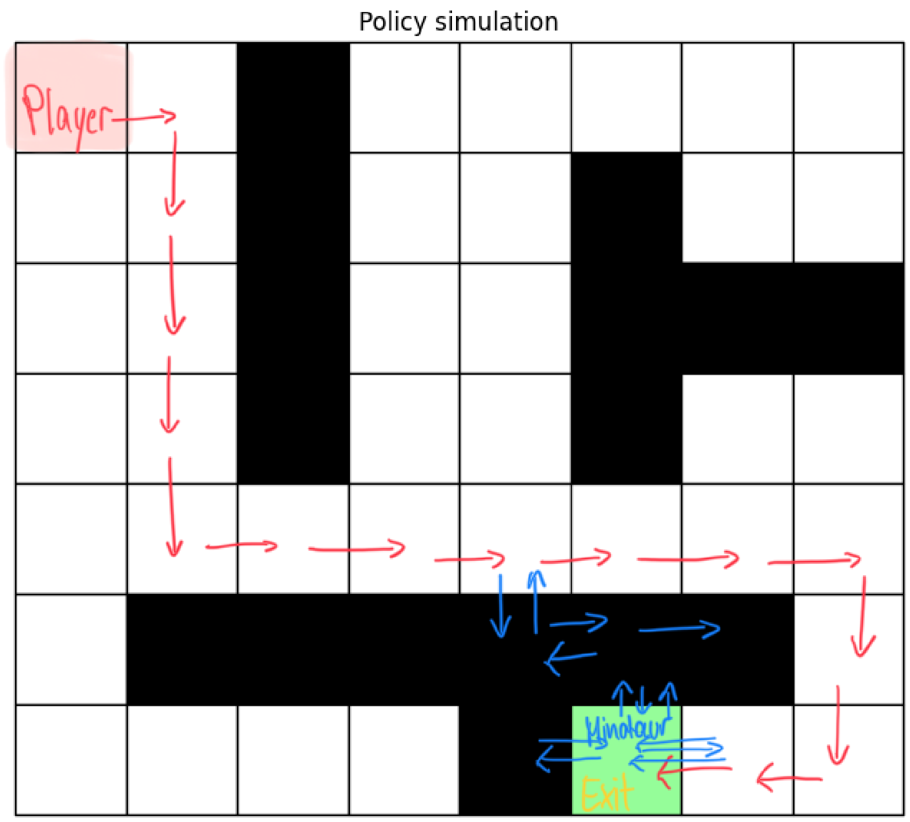

Figure 1: Visualisation of Policy

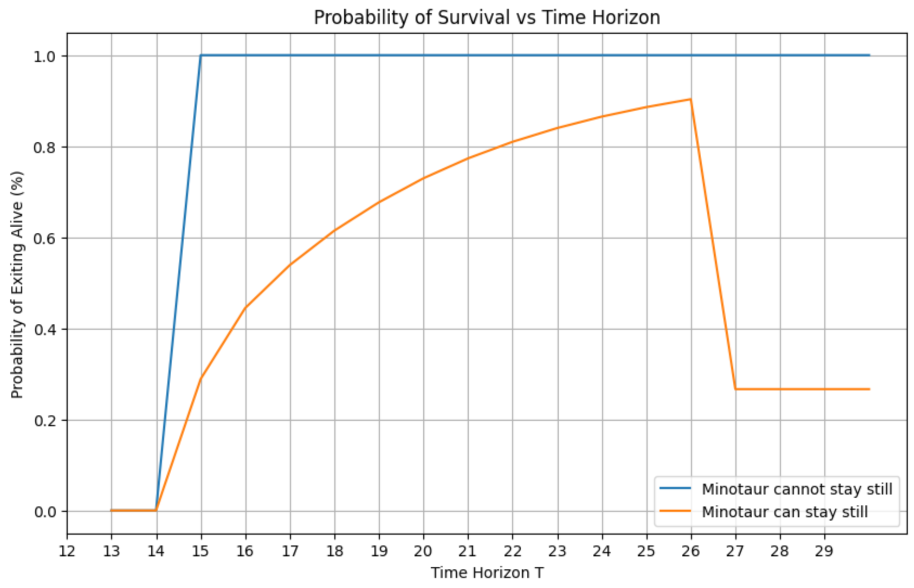

Figure 2: Visualisation of Policy

3 Value Iteration

3.1 e

As stated in the exercise we are assuming that the time is geometrically distributed with mean $\mu = 30$. The discount factor is set to $\lambda = 1 - \frac{1}{\mu}$.

The new problem formulation is derived through finding a stationary policy that minimizes the infinite horizon objective with the discount factor stated above.

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \lambda^{t-1} r(s_t^{\pi}, a_t^{\pi})\right].$$

Through the lecture we know the Bellman equation in case of a stationary policy π is:

$$\forall s \in \mathcal{S} \quad V^*(s) = \max_{\pi} \left\{ r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s' \mid s, \pi(s)) V^*(s') \right\}$$

Furthermore, through the lecture we have the VI algorithm

1. **Initialization:** Select a value function $(V_0 \in \mathcal{V}), (n = 0), \delta \gg 1$
2. **Value Improvement:** While $\delta > \frac{\epsilon(1-\lambda)}{\lambda}$ do

- (a)($V_{n+1} = \mathcal{L}(V_n)$, i.e., for all $s \in \mathcal{S}$

$$V_{n+1}(s) = \sup_{a \in \mathcal{A}_s} \left(r(s,a) + \lambda \sum_j p(j \mid s,a) V_n(j) \right)$$

- (b) $\delta = \|V_{n+1} - V_n\|, n \leftarrow n+1$

3. **Output:** $\pi = (\pi_1, \pi_2, \ldots)$ with

$$\forall s \in \mathcal{S}, \quad \pi_1(s) \in \arg\max_{a \in \mathcal{A}_s} \left(r(s,a) + \lambda \sum_j p(j \mid s,a) V_n(j) \right)$$

And we know that VI converges since $\mathcal{L}$ is a contraction mapping. When it stops, VI returns an ϵ-optimal policy

3.2 f

In the jupyter notebook version of this report you will find the code to run to calculate the probability of getting out alive of the maze using the constraints of being poised and the value iteration. Copying the output from there we can state that the probability of getting out alive with the minotaur not allowed to stay is around 56% while it is around 52% when the minotaur is allowed to stay.

4 Theoretical Questions

4.1 On-policy vs. Off-policy

On-policy learning means the method learns the value of the policy it is currently following. It explores and updates based on actions taken according to its current policy. Example: SARSA (State-Action-Reward-State-Action). Off-policy learning means the method learns the value of an optimal policy independently of the policy it uses to explore. It can update its value estimates using data from a different policy. Example: Q-learning.

4.2 Convergence conditions for Q-learning and SARSA

Q-learning:

- The learning rate α_t should decay over time such that $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$ (e.g., $\alpha_t = \frac{1}{t}$).
- The agent must visit all state-action pairs infinitely often.

SARSA:

- Same learning rate condition as Q-learning (α_t decays appropriately).

- The agent should explore sufficiently (e.g., use an ϵ-greedy policy to ensure all actions are tried).
- $\epsilon \to 0$ as $t \to \inf$

4.3 Exit Maze with Keys

The introduction of new transition probabilities and keys requires modification of the state space, rewards, and transition probabilities.

Because the life of the player is still geometrically distributed, "poised", we need to model the MDP as a discounted MDP. The discount factor is set to $\lambda = 1 - \frac{1}{\mu}$ with $\mu = 50$. We define a new state space $\mathcal{S}_\langle$: The state space is similar to the one previously defined with the additional constraint if the player has or has not the key. We define $\mathcal{S}$ as the state space from the beginning.

$$\mathcal{S}_\langle = \{((x_1, y_1), (x_2, y_2), k) \mid ((x_1, y_1), (x_2, y_2)) \in \mathcal{S},\ \text{k} \in \{1, 0\}\} \cup \{\text{Eaten}, \text{Win}\}$$

Where k = 1 indicates the player is in possession of the key and k=0 means he is not. Because all keys are in position C and we assume that once the player has one key they have all keys, we only define k=1 or k=0 (has all or has no key).

Next we need to define the new transition probabilities $\mathcal{P}_\langle$: With probability 0.35 the minotaur moves towards the player. Defining these actions gives:

- Minotaur is in the same row as the player: the minotaur moves along the x-direction towards the player
- Minotaur is in the same column as the player: the minotaur moves along the y-direction towards the player.
- Minotaur is neither in the same row nor column: the minotaur can move along the x- or y-direction towards the player.

With probability 0.65 the minotaur takes any OTHER valid action as defined previously, these action include the previous defined actions plus all actions which do not move the minotaur closer to the player.

Furthermore, we need to adapt the transition probability for exiting the game, e.g. winning the game. If k = 0 ($\text{maze}[x\prime_1, y\prime_1] = 2$) doesn't mean the player wins. But the state is just like any other state as previously defined and the terminal states are now:

$$P(\text{Win} \mid ((x_1, y_1, key = 1), (x_2, y_2)), a) = 1 \quad \text{if } (\text{maze}[x\prime_1, y\prime_1] = 2)$$

We adapt the rewards $\mathcal{R}_\langle$ to include a reward for grabbing the key by adding

$$r(s, a) = \text{KEY\_REWARD} = 1 \text{ to } \mathcal{R}$$

5 Problem 2

5.1 i.1

```
Class QLearning:
    Initialize:
        Input: environment (env), epsilon, discount (gamma),
            alpha, q_init
        Initialize Q-table (Q[state][action]) with q_init or 0 for
            terminal states
        Initialize action counts (_n) to zeros
        Set exploration (epsilon) and learning rate parameters
            (alpha)

    Function q(state, action):
        Return Q[state][action]

    Function v(state):
        Return max(Q[state])  # Maximum Q-value over all actions

    Function compute_action(state, explore=True):
        Get valid actions for the state
        If explore AND random(epsilon):
            Return random valid action
        Else:
            Return action with max Q-value for the state (break
                ties randomly)

    Function update(last_experience):
        Extract state, action, reward, next_state from
            last_experience
        Compute step_size = 1 / (count(state, action) ** alpha)
        Update Q[state][action] using the Bellman equation:
            Q[state][action] += step_size * (reward + gamma *
                max(Q[next_state]) - Q[state][action])
        Increment count(state, action)

    Function train(n_episodes):
        Initialize statistics for tracking rewards, lengths, and
            initial state values
        For each episode:
            Reset environment to the initial state
            While episode is not done:
                Select an action using compute_action()
                Perform the action, observe reward, next_state,
                    and done
                Record the experience
                Update Q-values using update()
                Accumulate reward and increment episode length
                Transition to next_state
            Log statistics (reward, length, initial state value)
        Return statistics and initial state value progression
```

Listing 1: QLearning Class

5.2 i.2

The plot in Figure 3 highlights the impact of ϵ on Q-learning performance and policy success. For $\epsilon = 0.1$, the agent achieves a moderate balance between exploration and exploitation, resulting in the highest exit probability (0.48) and relatively stable average rewards and lengths. When $\epsilon = 0.01$, the agent prioritizes exploitation more aggressively, leading to slightly lower exit probability (0.33) but longer episode lengths (28.8), indicating it focuses more on refining an optimal policy. Conversely, with $\epsilon = 1$, the agent is fully exploratory, achieving no meaningful rewards or policy success (exit probability = 0.0), as it fails to converge to a useful strategy within the episode limit. These results demonstrate that excessive exploration hinders learning, while moderate exploration offers the best trade-off for convergence and performance.

To test the impact of initalization on the convergence, we use the best performing epsilon value from the before experiement, $\epsilon = 0.1$. We plotted the convergence of different initialization values for Q in Figure **??**. When Q-values are initialized optimistically (e.g., $q\_init = 1$), the algorithm initially overestimates the values, resulting in faster exploration but a slower decline before stabilizing. Conversely, lower initial Q-values (e.g., $q\_init = 0.01$) lead to more cautious updates, with slower convergence but more stable growth.

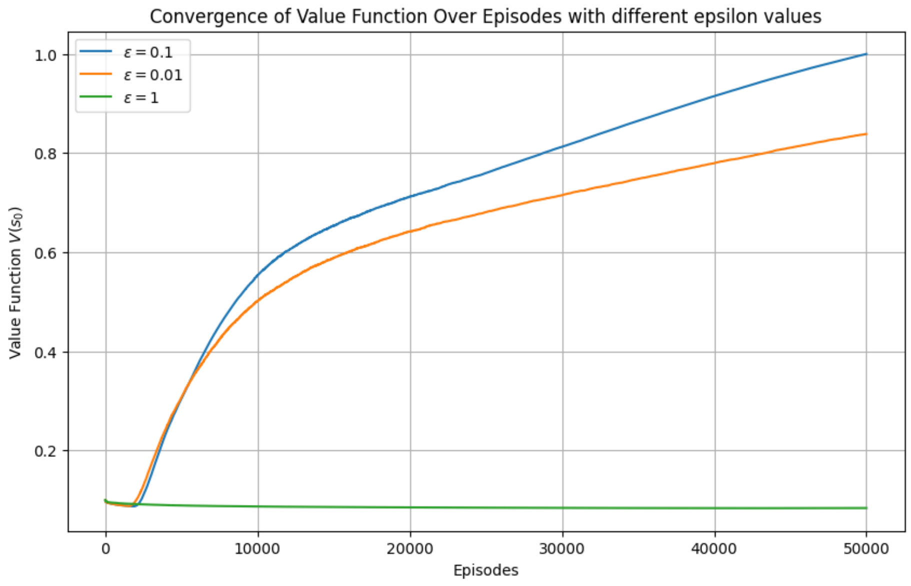

Figure 3: Comparison of Different ϵ Values

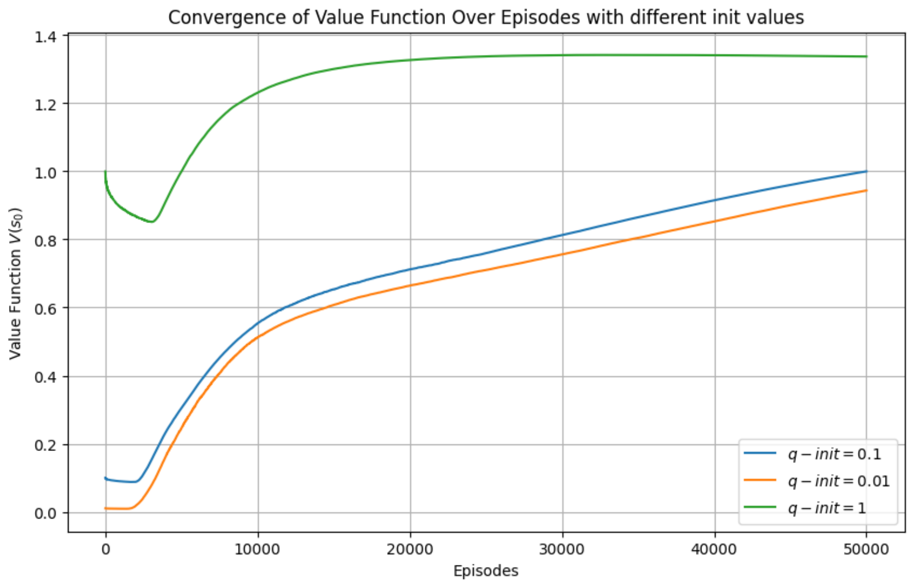

Figure 4: Comparison of Different Initialization Values for Q

5.3 i.3

The plot in Figure 5 visualizes that the choice of α significantly impacts the convergence speed and performance of Q-learning. For $\alpha = 1.0$: The value function remains flat at 0 throughout the episodes, indicating that the Q-values of s_0 are not being updated effectively. Despite the flat trajectory of $V(s_0)$ for $\alpha = 1$, the exit probability is non-zero (0.4). This discrepancy suggests that some learning and successful trajectories occur, but they are not reflected in the initial state's value $V(s_0)$. An explanation for this is that for $\alpha = 1$, the step size becomes constant ($1/n^{\alpha}$ is always 1), making the Q-values entirely dependent on the most recent reward and future estimates. The constant learning rate prevents Q-values from stabilizing over time, making convergence slower or nonexistent, especially for states like s_0 that require multiple updates to reflect accurate values.

$\alpha = 0.55$ shows a steep and consistent growth of $V(s_0)$ after around 2000 episodes. The overall slower decay of the step size, compared to $\alpha = 0.77$ allows for continued learning and refinement of the policy, which results in higher convergence and better overall performance in terms of maximizing the value function. While the faster decay of the step size with $\alpha = 0.77$ limits the agent's ability to make significant updates to Q-values as training progresses it might be preferable in scenarios where faster stabilization of Q-values is required.

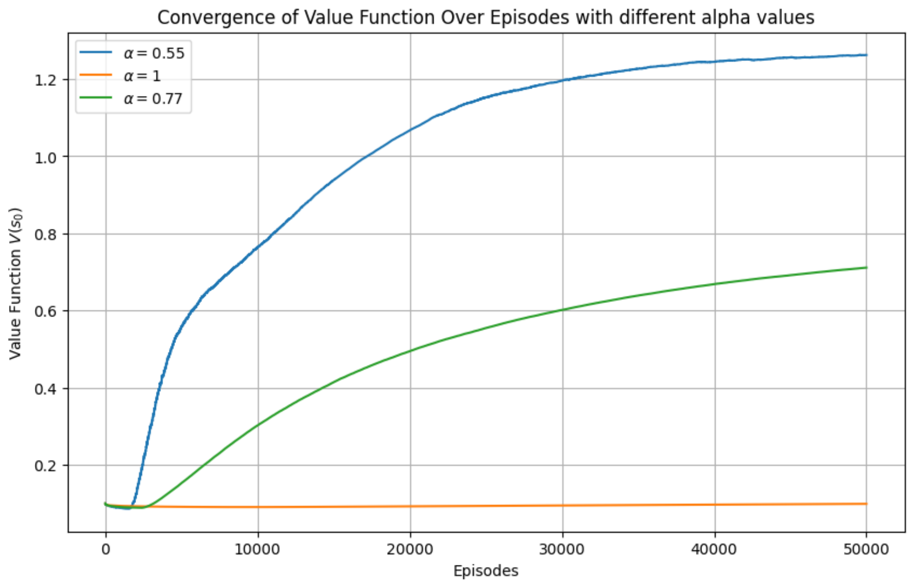

Figure 5: Comparison of Different α Values

5.4 j.1

The only difference to the implementation of q-learning function is the update function. We are still trying to solve the same problem (first finding the key, being poised, having a minotaur chaising us with probability 35%, needing to find the exit withe the keys), thus the logic of iteration is the same just how the q-function is updated differs. We no longer multiply the discount by the difference between the estimated optimal future value and the current Q-value for a given state-action pair but we update the q-values based on the next action chosen by the current policy. The main difference thus is that we move from an off-policy algorithm to an on-policy one.

5.5 j.2

With $\epsilon = 0.1$, the agent converges more quickly and achieves a higher overall value for $V(s_0)$ by the end of training, as seen in Figure 6. Here the exploration is limited, allowing the agent to focus on refining its policy. In contrast, for $\epsilon = 0.2$ we see a much slower convergence and a lower value of the Q-function. This might be explained by the increased exploration that could delay the exploitation of learned strategies. The exit probabilities also reflect the observations in the plot, where $\epsilon = 0.1$ achieves a higher success rate (46%) compared to $\epsilon = 0.2$ (27%), demonstrating the trade-off between exploration and convergence in SARSA.

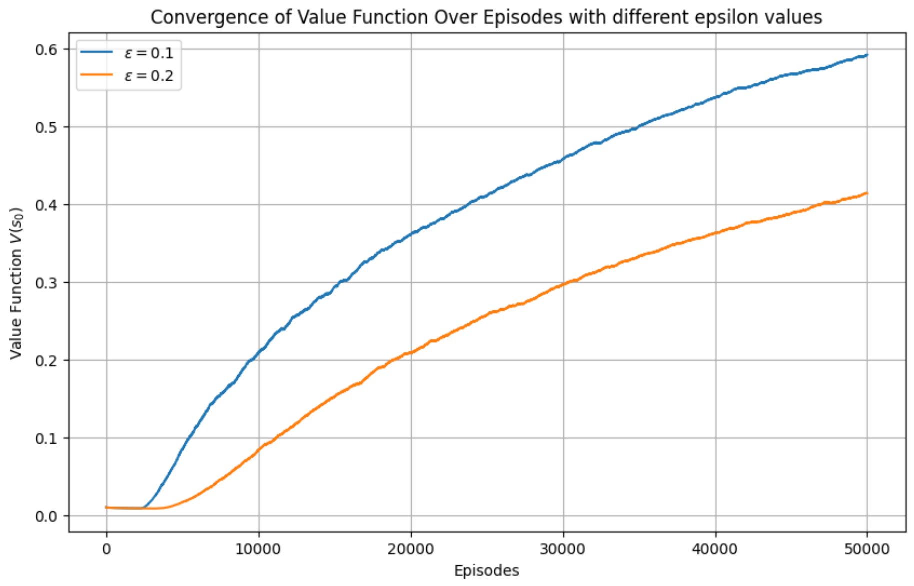

Figure 6: Comparison of Different α Values

5.6 j.3

The experiment results, plotted in Figure 7, demonstrate how dynamically adjusting the exploration rate (ϵ) using different delta values (δ) affects SARSA learning in terms of the Q-value function and the exit probability. With $\delta = 1$, the Q-value function $V(s_0)$ grows faster, indicating strong exploitation of learned values, but the exit probability is lower (0.38), suggesting insufficient exploration limits the discovery of optimal paths to the exit. On the other hand, $\delta = 0.3$ results in slower growth of $V(s_0)$ which can be attributed to the increased exploration of the function. The higher exit probability for this delta value (0.53) shows that this exploratory behavior allows the agent to discover better strategies for survival in the end. IThe highest exit probability is given with $\delta = 0.3$, the highest q-value function estimate with $\delta = 1$ and the lowest q-value function estimate is given when no dynamic adaption of epsilon is deployed.

We would hypothize that prioritizing exploration (as seen with $\delta = 0.3$), thus $\alpha < \delta$ may lead to better long-term outcomes, even if the Q-value function has a lower estimate. With δ close to α giving the worst results, both in terms of q-value function estimate and exit probability we would advise to not take this one.

5.7 k

The exit probabilities for both Q-learning and SARSA (0.5) are not directly close to the Q-values of the initial state (0.71 for Q-learning and 0.69 for SARSA).

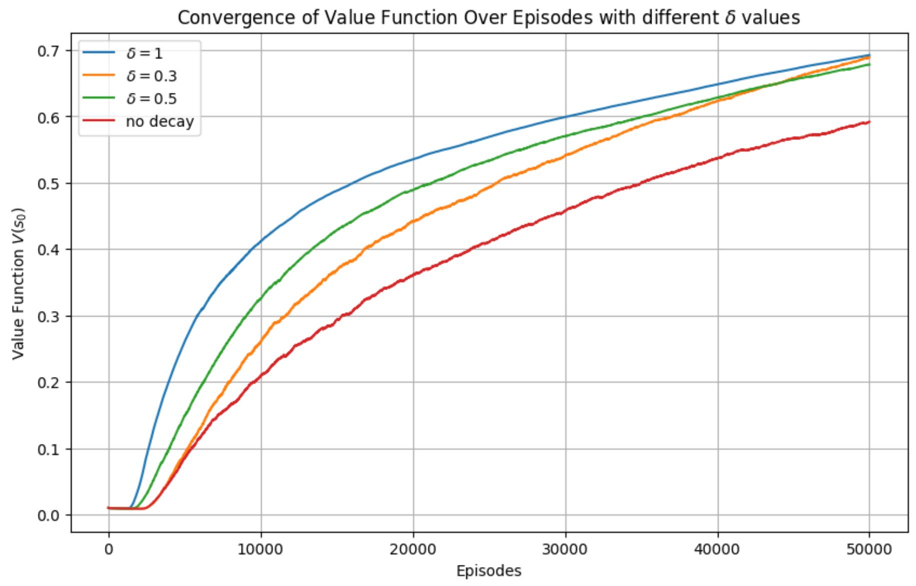

Figure 7: Comparison of Different δ Values

This is because the Q-value of the initial state represents the expected cumulative reward (including penalties or intermediate rewards) under the learned policy, while the exit probability only measures the likelihood of successfully leaving the maze, without accounting for the magnitude of rewards along the way.

Interestingly, if we only run training for 5000 episodes the Q-values of the initial state are reduced to 0.15 for Q-learning and 0.09 for SARSA, while the exit probabilities stay around 0.5 for both algorithms. This demonstrates the fact that the policy converges long before the values. Furthermore, it was very nice to discover that both algorithms have the same exit probability when tuned for best parameter setting.

5.8 Conclusion

All experiments can be rerun using the jupyter notebook version of this report, however running with 50000 episodes takes a considerable amount of time to finish, thus we recommend to run with 5000 episodes. While this will not give 100% the same results, they are comparable and have a better computation time vs. result trade-off.

6 Problem 2

6.1 c Describe the training process

In the beginning I trained for 1000 episodes and it was hard to find good parameters. Furthermore, the model is very sensitive to parameter changes. For this reason, I implemented all tips and tricks. This means implementing SGD with Nestorov Acceleration, clipping the eligibility trace, scaling the Fourier basis and reducing the learning rate during training.

Parameter	**Value**
N_{episodes}	200 (Number of episodes for training)
γ (discount factor)	1
λ (SARSA(λ) eligibility traces)	0.6
Base learning rate (α)	0.001
ϵ (exploration rate)	0.0001
Exploration decay rate	0.9
Learning rate decay rate	0.95
Momentum (SGD updates)	0.96

Table 1: Training Parameters

6.2 d.1: Analysis of episodic total reward across episodes

Figure 8 illustrates the learning process of the Sarsa(λ) algorithm with Fourier basis function approximation in solving the MountainCar environment. One can see that initially, the agent struggles with rewards near -200. However, within the first 25 episodes, there is a sharp improvement in total rewards, indicating the agent is quickly learning an effective policy. The rewards stabilize around -115, meeting the success threshold, with minor fluctuations due to the ϵ-greedy exploration strategy. Despite occasional performance dips, visualized through single low episode reward, the algorithm converges to a robust policy, achieving consistent and efficient performance by the end of training.

6.3 d.2: Analysis of value function of SARSA-Algorithm

The 3D plot 9 of the value function uses color gradients to represent the expected cumulative rewards across the state space, where warmer colors (yellow/green) indicate higher rewards and cooler colors (purple/blue) represent lower rewards. Important to note that the position values are scaled from [-1.2; 0.6] to [0.0; 1.0]. The value is highest in states where the agent can reach the goal quickly, such as positions near the goal, e.g., 1.0, or at the top of the left hill, position close to 0, where the car just needs to accelerate towards the right and will get over the valley. The reward is low around the position mirroring the valley, e.g., around 0.5, because here the car first needs to go on top of the left hill before it can

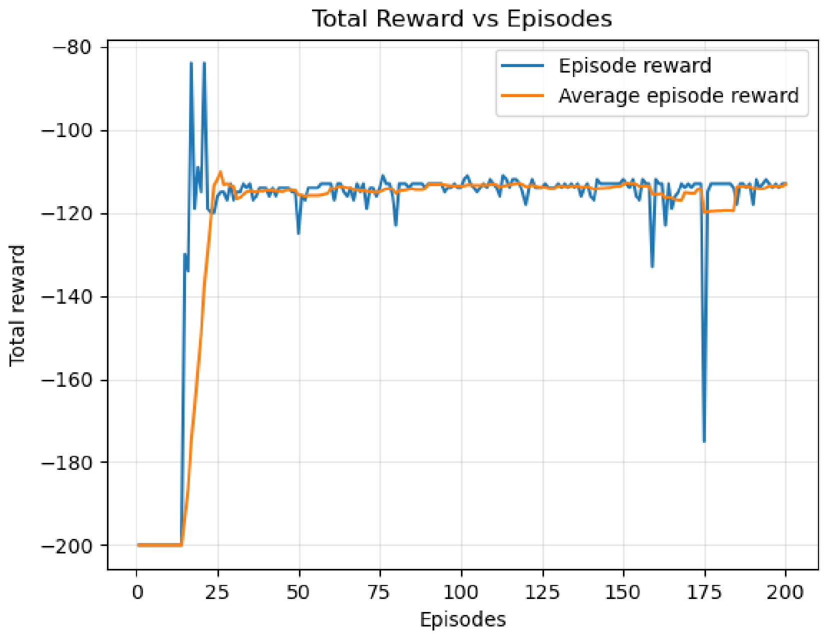

Figure 8: Episodic total reward across episodes

get to the goal on the right side. This distribution of values makes sense given the MountainCar environment's dynamics, where each step incurs a penalty of -1, and the agent's goal is to minimize the time required to reach the goal.

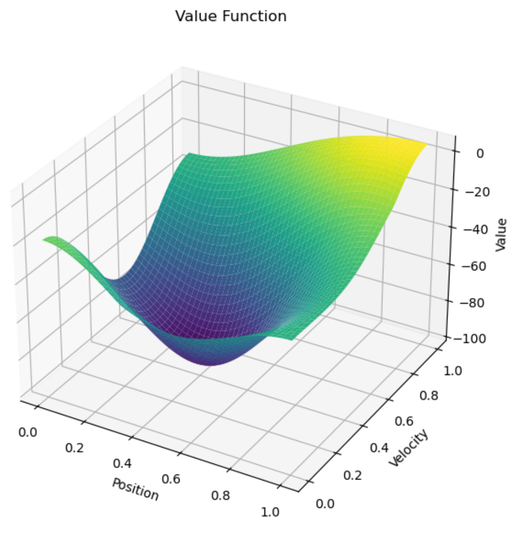

Figure 9: 3D plot of SARSA value function

6.4 d.3: Optimal policy over state space

The 3D plot 10 represents the optimal policy over the state space domain, where the action (e.g., accelerate left, no push, or accelerate right) is determined based on the car's position and velocity. The vertical axis encodes the selected action, with distinct bands (blue, orange, yellow) corresponding to different choices.

The plot aligns with the expected logic, close to the goal (position near 1), the car needs only a small positive velocity to succeed, so the policy favors accelerating to the right or not pushing, as these actions efficiently lead to the goal without overshooting. Conversely, when the car is far from the goal (positions closer to 0), the optimal policy considers whether the car has sufficient velocity. If the velocity is insufficient, the agent accelerates to the left to climb the left hill, gain momentum, and prepare for a strong push to the right. This is reflected in the transitions between action regions as velocity changes. The oscillating band structure in the middle of the state space (e.g., around position 0.4-0.6) suggests state dependent decisions where the policy dynamically adjusts between pushing left and right to optimize momentum gain. This reflects the challenge of getting out of the valley.

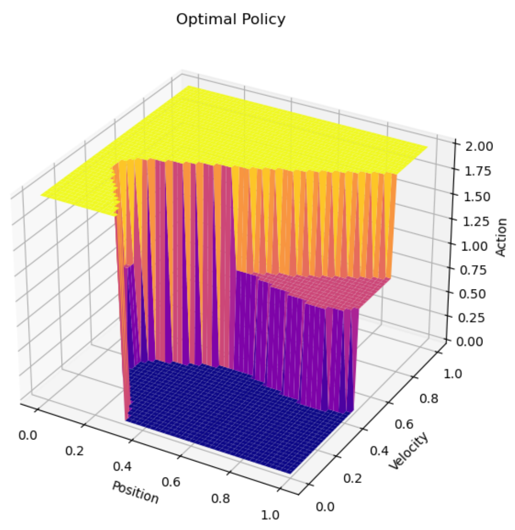

Figure 10: Optimal policy over the state space domain

6.5 d.4 Including $= [0, 0]$ in the basis

Including the constant term $= [0, 0]$ in the basis typically helps improve representation capability, but in, it can make the model less adaptive to specific features of the state space. In some instances the action 1 (no push) was not used in the optimal policy.

6.6 e.1

6.7 e.2

For me the best strategy was to initialise the q-values with zero. This meant the training process takes longer and less exploration. However, other strategies like random or optimistic initialisation introduced even more noise to the training process. Since the training was already relatively unstable, I decided to pursue with initialising the values with 0.

6.8 e.3

Since the training was noisy and random swings accord in the reward over the episodes, I decided to to implement epsilon decay. That means the exploration will be less during the end of the training and in the beginning the exploration will be higher. This also cooperates with the zero q-value initialisation. Through

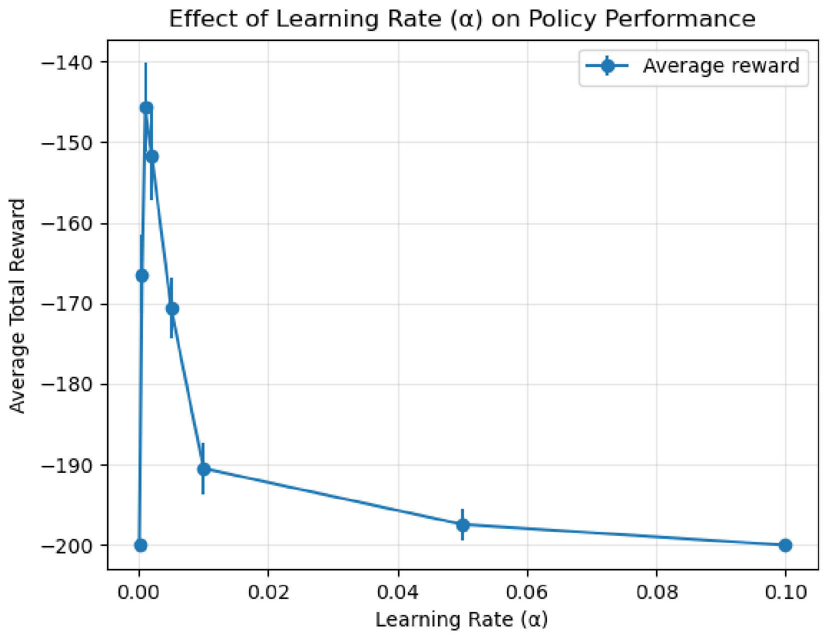

Figure 11: Average total reward of the policy as a function of the learning rate

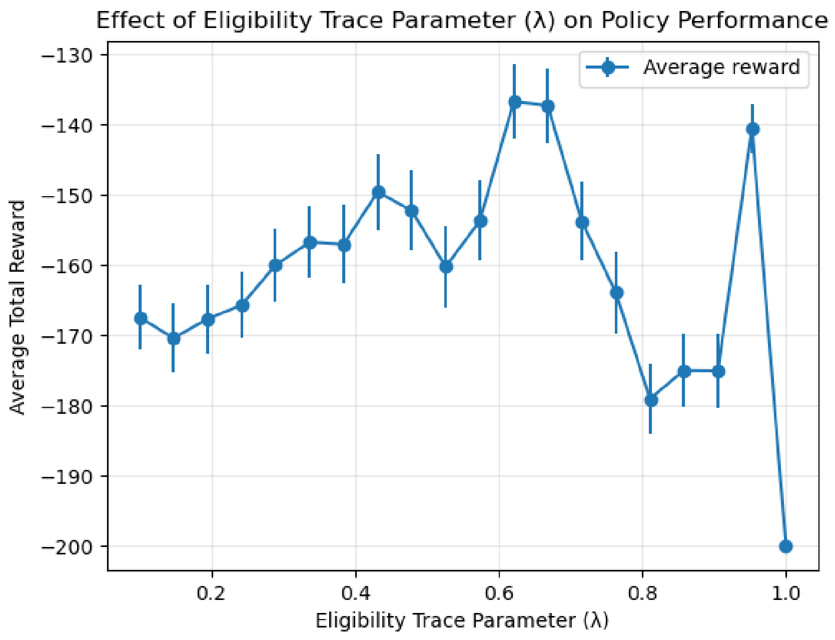

Figure 12: Average total reward of the policy as a function of the eligibility trace

this the training got more stable and less swings and noise occurred. Moreover, the reward raises later compared to not having epsilon decay implemented

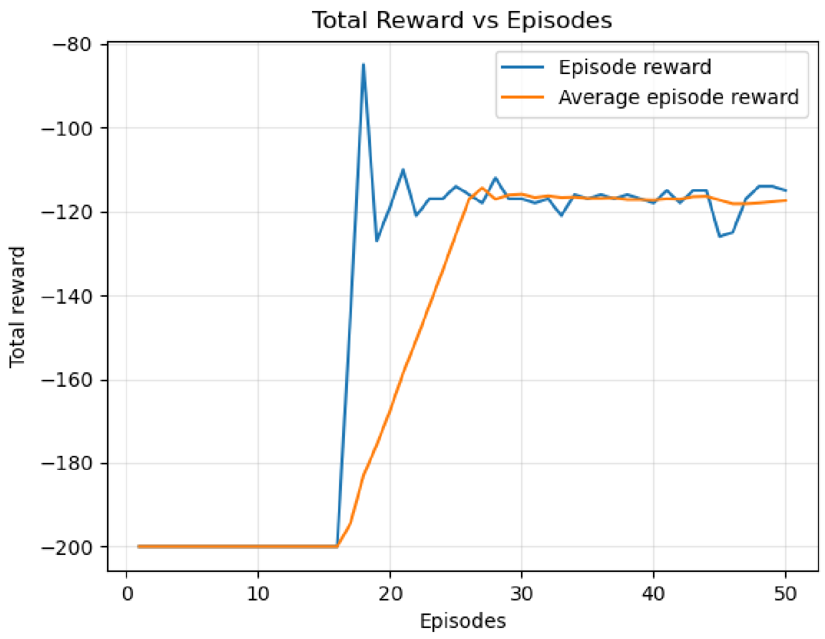

Figure 13: Average total reward of the policy with epsilon decay

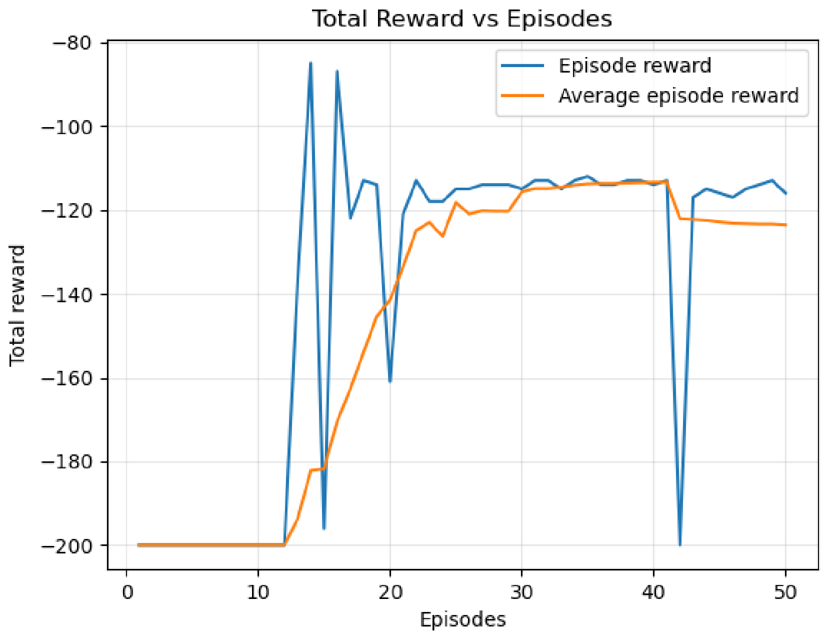

Figure 14: Average total reward of the policy without epsilon decay