Saarland University

Bachelor's Thesis

DSMC Evaluation Stages with Restart

Submitted by:

Nicola J. Müller

Submitted on:

October 31, 2022

Supervisors:

Univ.-Prof. Dr. Verena Wolf

Univ.-Prof. Dr. Jörg Hoffmann

Advisor:

M.Sc. Timo P. Gros

Abstract

Deep reinforcement learning (DRL) increasingly enables the automation of sequential decision-making problems encountered in numerous real-world tasks. Despite their many successes, applications of DRL often accept the possibility of agents learning risky behavior in favor of feasible training time. This safety negligence limits the applicability of DRL. A remedy to this dilemma is the utilization of DSMC evaluation stages, which focus the DRL agent's training on challenging state space regions while preserving adequate training time. However, the benefit of applying DSMC evaluation stages depends on how the given task's initial states are distributed across the state space. This thesis aims to address this limitation by extending DSMC evaluation stages to collect additional starting states throughout the training, which we call pseudo initial states. The resulting algorithms *DSMC evaluation stages with restart* dynamically regulate the utilization of pseudo initial states, such that they improve the agent's performance. Our experiments demonstrate that DSMC evaluation stages with restart can focus the training on beneficial state space regions, allowing agents to achieve high performance regarding safety metrics in the original task and when starting in various states throughout the state space.

Acknowledgments

I would like to thank Prof. Dr. Verena Wolf and Prof. Dr. Jörg Hoffmann for providing this exciting topic, for their supervision, and for reviewing this thesis. Furthermore, I thank my advisor Timo P. Gros for his guidance and feedback during all steps of conceptualizing and writing this thesis. My deepest thanks go to my friends and family who supported and motivated me throughout the last 3 years.

Contents

1. Introduction

Various applications of artificial intelligence require repeatedly making decisions where each may influence future situations, so-called sequential decision-making problems. For instance, an autonomous car will need to assess with every driving action whether it will later be able to stay on the road or avoid obstacles. Reinforcement learning (RL) is concerned with developing algorithms that solve sequential decision-making problems, making it a focal point of current innovation and research [8, 10, 25, 29].

The RL framework models sequential decision-making problems as Markov decision processes (MDP), called environments, in which an agent uses a policy π to execute actions that influence the environment's state. For each action taken, the agent receives a reward, which is a numerical feedback signal of the action's quality. The agent's objective is to learn an optimal policy π_* that allows it to take actions that maximize the accumulated rewards, called the return.

Value-based RL algorithms like Q-learning [34] attempt to solve sequential decision-making problems by learning for every possible state-action pair (s_t, a_t) an approximation of the return that can be earned when taking the action a_t in the state s_t, called the Q-value $Q(s_t, a_t)$. These algorithms then construct a policy π that returns the action with the highest Q-value for each state. However, learning the Q-values of all state-action pairs becomes intractable when dealing with MDPs that have gigantic state spaces. Thus, value-based deep reinforcement learning (DRL) algorithms like DQN [23] and DQNPR [28] use deep neural networks (DNN) to learn a function that computes the Q-values of any given state-action pair. Policy-gradient RL algorithms represent a different approach to solving sequential decision-making problems since they try to learn the parameters θ of a parameterized policy π_θ instead of constructing a policy from learned Q-values. Policy-gradient DRL algorithms like PPO [31] parameterize their policies using a DNN.

In practice, sequential decision-making problems often involve critical situations in which DRL agents must obey pre-defined safety constraints to ensure that damages to objects and people are avoided. However, approaches that train DRL agents to behave safely often fail to achieve satisfactory performance since they make the agent's objective too difficult to optimize [11]. Gros et al. developed a technique called DSMC evaluation stages (DSMC ES) [16], which circumvents this challenge by keeping the objective unaltered and instead focusing the training on state space regions where the agent exhibits the most extensive safety deficiencies, which forces it to improve its behavior. To do so, they partition the task's initial states into a manageable number of initial state regions, where the agent's policy is regularly evaluated. The results are then passed to an underlying training algorithm, and the training is focused on the initial state regions with the poorest evaluation.

The original case studies of DSMC ES were run on the Racetrack [12] benchmark, where the goal is to drive a car on a two-dimensional map from a starting point to a goal line without hitting a wall. The car's velocity defines in which direction and how far the car will drive between successive time steps, and the agent can adjust the velocity at each time step. The Racetrack benchmark is well suited for developing and evaluating DRL algorithms since it can simulate various scenarios using different maps and allows for insightful visualizations of the agent's behavior [4, 14, 15, 16, 17].

The disadvantage of DSMC ES is that it requires the initial state regions to cover a significant part of the state space. Otherwise, it may fail to focus the training on the state space regions where the agent's policy is most deficient. This thesis proposes to make DSMC ES generally applicable by storing certain visited states, which we call pseudo initial states, and using them to define additional initial state regions. This has the consequence that a portion of the training episodes starts in pseudo initial states instead of the task's initial states, and therefore we call the resulting method DSMC evaluation stages with restart (DSMC ESR). We will investigate selecting pseudo initial states according to different properties, such that the training can be focused on the state space regions where the agent can gather the most beneficial experiences. Another crucial part of DSMC ESR is that it dynamically adjusts how often pseudo initial states are utilized, such that the agent does not forget how to act when starting in the original initial states.

In our experiments, DSMC ESR agents were trained to drive on three Racetrack maps that vary in their challenges. DQN and PPO were used as the underlying training algorithms for DSMC ESR. We examine on which map areas DSMC ESR focused the training process and whether it could achieve higher safety performance than the DQN, DQNPR, and PPO baselines.

2. Background

This chapter establishes the mathematical foundation of this thesis and introduces a sequential decision-making problem called *Racetrack*.

2.1. Sequential Decision-making Problems

Sequential decision-making problems are characterized by the fact that every decision made at a particular time step t can influence situations in future time steps $t + k$. A multitude of real-life applications can be viewed as such problems, for instance, autonomous driving [20]. Figure 2.1 shows a simplified version of autonomous driving with two cars. The car at the top does not change its direction in the first time step, making a collision with the cat inevitable in the third time step, whereas the car at the bottom drives upwards, allowing it to later drive past the cat. Even though both cars execute the same drive action in the second time step, only the car that drove in the correct direction in the previous time step will be able to avoid a crash. This challenge of needing to make decisions with foresight motivates the development of algorithms that are specialized to solve sequential decision-making problems.

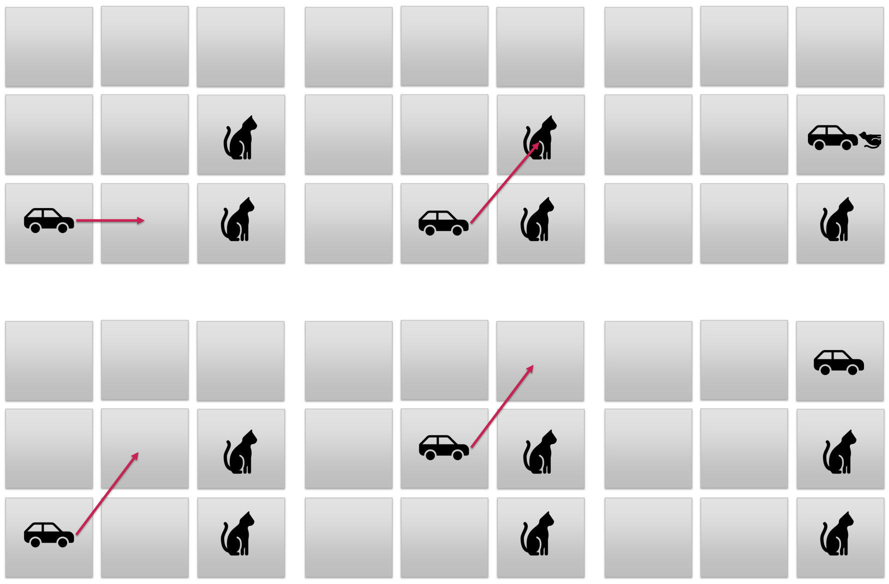


Figure 2.1 **Two autonomous cars.** The agents drive from left to right and may either steer upwards, downwards or not at all. The goal is to avoid crashing into a cat.

2.2. Markov Decision Processes

Markov decision processes (MDP) are mathematical constructs that can be used to model sequential decision-making problems.

Definition 2.1 (Markov Decision Process [16])**.** *Finite Markov decision processes are defined as tuples $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mu \rangle$, where $\mathcal{S}$ is the set of states s, $\mathcal{A}$ is the set of actions a, $\mathcal{T}$ is the transition probability function, and μ is the initial state distribution.*

The states $s \in \mathcal{S}$ model the different instances of the corresponding sequential decision-making problem, where the initial instances are represented by the initial states s_0, which are sampled from the initial state distribution $\mu \in \mathcal{D}(\mathcal{S})$. We will refer to the set of initial states as $\mathcal{I} = \{s \in \mathcal{S} | \mu(s) > 0\}$. Furthermore, a subset of states $s \in \mathcal{S}$ are terminal states from which no transitions to other states are possible. Thus, they represent the end of the sequential decision-making process, which is reached either by solving the problem or by failing to find a solution. The actions $a \in \mathcal{A}$ represent the decisions that must be made at each time step t. To model the effects of making decisions, we use the partial transition probability function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightharpoonup \mathcal{D}(\mathcal{S})$, which given a state-action pair $(s_t,\, a_t)$ allows us to sample from a probability distribution over the states $\mathcal{D}(\mathcal{S})$ the successor state s_{t+1}. For every state s_t the applicable actions *$a_t \in \mathcal{A}(s_t) \subset \mathcal{A}$ correspond to the actions for which $\mathcal{T}(s_t, a_t)$ is defined. Additionally, we define for an MDP $\mathcal{M}$ a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ that maps state transitions $s_t \underset{a_t}{\longrightarrow} s_{t+1}$ to rewards r_{t+1}, which are numerical values that specify the quality of the state transition with respect to the goal of solving the sequential decision-making problem.*

Since we only require the current state s_t and an action a_t to compute the next state s_{t+1} using $\mathcal{T}$, we can follow that states encountered before the time step t do not affect the state transition $s_t \underset{a_t}{\longrightarrow} s_{t+1}$. This feature is called the Markov property.

To decide which actions to take in each state, we use a policy π.

Definition 2.2 (Deterministic Policy [32])**.** *A deterministic (history-independent) policy π is a function $\pi : \mathcal{S} \to \mathcal{A}$ that, given a state s_t, returns an applicable action $a_t \in \mathcal{A}(s_t)$.*

Figure 2.2 shows how we can model a simple sequential decision-making problem as a finite MDP, by visualizing the MDP as a graph consisting of state and action nodes. The outgoing edges of state nodes are connected to action nodes, representing the applicable actions $a_t \in \mathcal{A}(s_t)$ in the corresponding state s_t. From these action nodes, outgoing edges to other state nodes represent the possible state transitions $s_t \underset{a_t}{\longrightarrow} s_{t+1}$ caused by executing the respective action a_t. Each edge is annotated with the earned reward r_{t+1} and, in the case of probabilistic state transitions, with the transition probability.

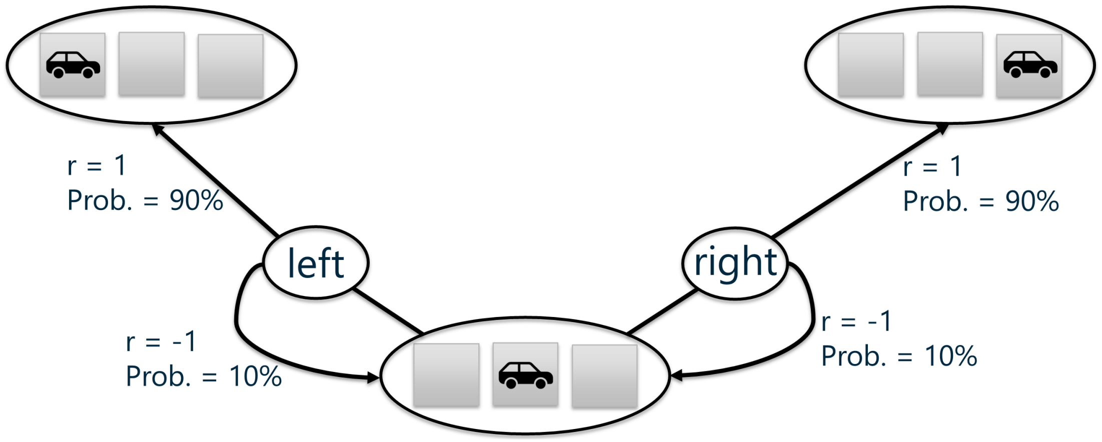


Figure 2.2 **Part of an MDP's graph.** In this exemplary task, the car may either drive to the left or the right. Both drive actions will fail with a probability of 10%.

2.3. The Reinforcement Learning Framework

The objective of RL is to train an entity, called the *agent*, to solve a given *task* corresponding to a sequential decision-making problem. The task is formalized as an MDP, called the *environment*, in which the agent starts in an initial state $s_0 \in \mathcal{I}$ and has to find a sequence of state transitions $s_t \underset{a_t}{\longrightarrow} s_{t+1}$ that earns a high accumulated reward, also called the *return*

$$G_t = \sum_{k=t+1}^{T} R_k,$$

which corresponds to solving the task. Figure 2.3 depicts how we can realize this process as an iterative agent-environment interaction. Based on the environment's current state s_t, the agent queries an action $a_t = \pi(s_t)$ from its policy π and executes it, which causes the environment to change its state. Afterward, the agent receives a new observation (s_{t+1}, r_{t+1}) from the environment, which contains the new state s_{t+1} and the reward r_{t+1} earned for executing the previous action a_t. This process then repeats until the agent reaches a terminal state, and the resulting sequence of states, actions, and rewards is called an *episode*. We conclude that to solve the task as well as possible, the agent needs to learn an *optimal policy* π_*, which enables it to take actions a_t that earn a maximum return G_t.

For every action they take, RL agents need to assess which future rewards it will enable since any decision made in a sequential decision-making problem can have long-lasting effects. However, we might want to reduce the importance of future rewards and put more weight on rewards earned in immediate time steps. To do so, we introduce the

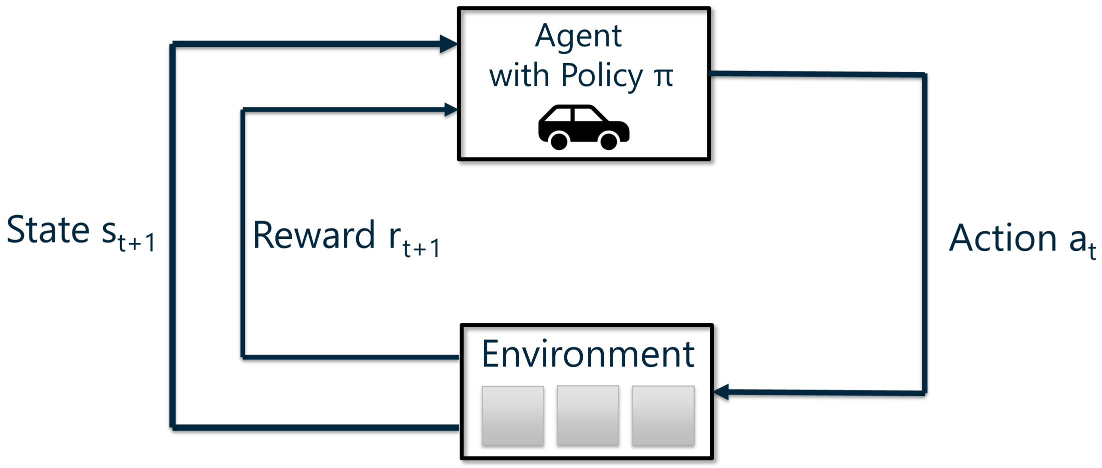


Figure 2.3 **Agent-environment interaction.**

discount factor γ into our definition of the return, giving us

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k.$$

Because γ^{k-t-1} gets smaller the larger k gets, rewards earned at later time steps will contribute less to the return G_t than rewards earned earlier. This effect increases the smaller we choose γ. Therefore, we can use γ to trade-off the importance of immediate and future rewards.

To evaluate an agent's expected return in a given state s_t, we can use a *state-value function* $v_\pi(s_t)$.

Definition 2.3 (State-value Function [32])**.** *The state-value function*

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right], \text{ for all } s \in S,$$

is a function $v_\pi : \mathcal{S} \to \mathbb{R}$ that computes the expected return when starting in the state s and then following the policy π. $v_\pi(s_t)$ is called the state-value of s_t.

Alternatively, we can evaluate an agent's expected return for a given state-action pair (s_t, a_t) using an *action-value function* $q_\pi(s_t, a_t)$.

Definition 2.4 (Action-value Function [32])**.** *The action-value function*

$$q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle| S_t = s, A_t = a\right], \text{ for all } s \in S,\ a \in A,$$

is a function $q_\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ that computes the expected return when starting in the state s, taking the action a and then following the policy π. $q_\pi(s_t, a_t)$ is called the action-value or q-value of (s_t, a_t).

The advantage of utilizing an action-value function q_π is that it indicates whether taking an action a in a state s_t leads to a higher return G_t than following our current policy π, allowing adjustments. For an optimal policy π_*, we can define the corresponding state-value and action-value functions as the *optimal state-value function*

$$v_*(s) = \max_\pi v_\pi(s),$$

and the *optimal action-value function*

$$q_*(s,a) = \max_\pi q_\pi(s,a).$$

2.4. The Racetrack Benchmark

Racetrack is a commonly used benchmark in the RL community [4, 14, 15, 16, 17, 32] that has originally been conceptualized as a pen and paper game [12]. The agent drives on two-dimensional maps consisting of driveable free cells and cells hosting walls, an example of which can be seen in Figure 2.4. Additionally, each map has a predefined starting and goal line. Depending on the setting, each game may either start with equal probability on any cell of the starting line or any free map cell (random restart setting). The goal is then to drive to a cell located on a goal line without hitting a wall or taking too many steps.

In the following, we will formalize Racetrack as an MDP such that we can apply RL to it. The states $s \in \mathcal{S}$ in Racetrack consist of the agent's coordinates $c = (x, y)$, its velocity $v = (vx, vy)$, and the distances to the nearest goal lines, and walls. Note that the coordinates and the current map determine the latter. The velocity represents a two-dimensional vector that defines the direction and distance the agent will travel between two consecutive time steps t and $t + 1$. So given the current coordinates c_t and the new velocity v_{t+1}, the agent's new coordinates c_{t+1} are computed by

$$c_{t+1} = (x_t + vx_{t+1}, y_t + vy_{t+1}).$$

The actions $a \in \mathcal{A}$ in Racetrack correspond to accelerating in any of 8 directions by at most one unit. Thus, at each time step, the agent chooses an acceleration

$$a_t = (ax_t, ay_t) \in \{-1, 0, 1\}^2,$$

which results in the new velocity

$$v_{t+1} = (vx_t + ax_t, vy_t + ay_t).$$

Furthermore, every action has a small probability of not affecting the velocity, in which case the acceleration is $a_t = (0, 0)$. This simulates slippery road conditions and allows for adjusting the task's difficulty by changing this noise probability.

Note that in Figure 2.4 the agent's transitions between coordinates are defined by two-dimensional vectors in $\mathbb{N}^2$ of variable length and direction, whereas the map consists of quadratic cells of the same size. This means each transition can cross several cells

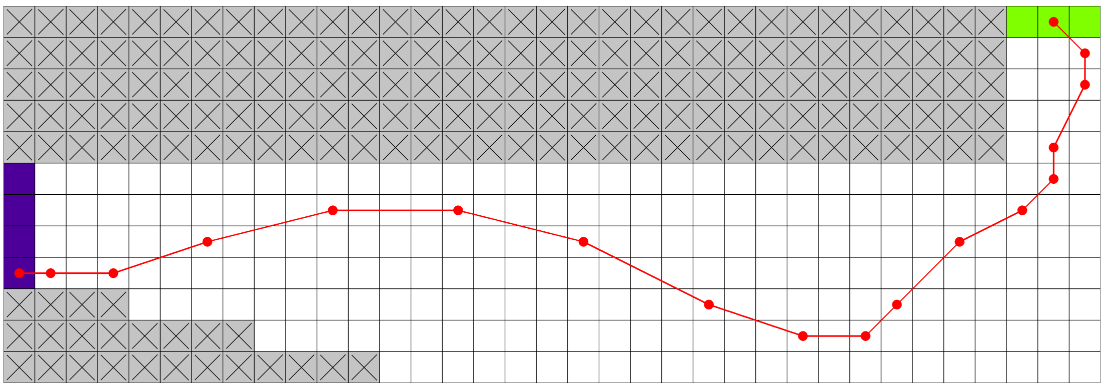


Figure 2.4 **Transitions of an agent on the Barto-small map.** The agent begins on the starting line (purple) and successfully drives to the goal line (green).

arbitrarily, making it difficult to recognize when the agent enters a cell hosting a wall or a cell of a goal line. Therefore, after every transition from coordinates c_t to c_{t+1}, we compute a discretized trajectory

$$T = \langle (x, y), (x_1, y_1), (x_2, y_2), ..., (x_k, y_k) \rangle$$

of crossed cell coordinates. We utilize the same discretization as Gros et al. [17], which is given by

$$T = \begin{cases} \langle (x,y) \rangle & \text{if } vx_{t+1} = 0 \wedge vy_{t+1} = 0 \\ \langle (x,y), (x+\sigma_x, y), (x + 2\cdot\sigma_x, y), ..., (x_k, y_k) \rangle & \text{if } vx_{t+1} \neq 0 \wedge vy_{t+1} = 0 \\ \langle (x,y), (x, y+\sigma_y), (x, y + 2\cdot\sigma_y), ..., (x_k, y_k) \rangle & \text{if } vx_{t+1} = 0 \wedge vy_{t+1} \neq 0 \\ \langle (x,y), (x+\sigma_x, \lfloor y + m_y \rceil), (x + 2\cdot\sigma_x, \lfloor y + 2\cdot m_y \rceil), ..., (x_k, y_k) \rangle & \text{if } vx_{t+1} \neq 0 \wedge vy_{t+1} \neq 0 \\ & \wedge\ |vx_{t+1}| \geq |vy_{t+1}| \\ \langle (x,y), (\lfloor x + m_x \rceil, y+\sigma_y), (\lfloor x + 2\cdot m_x \rceil, y + 2\cdot\sigma_y), ..., (x_k, y_k) \rangle & \text{if } vx_{t+1} \neq 0 \wedge vy_{t+1} \neq 0 \\ & \wedge\ |vx_{t+1}| < |vy_{t+1}| \end{cases},$$

where $\sigma_x = sgn(vx_{t+1})$, $\sigma_y = sgn(vy_{t+1})$, $m_x = \frac{vx_{t+1}}{|vy_{t+1}|}$, and $m_y = \frac{vy_{t+1}}{|vx_{t+1}|}$.

If the agent transitions from c_t to c_{t+1} on a vertical or horizontal line ($vx_{t+1} = 0$ or $vy_{t+1} = 0$) then T consists of all coordinates between c_t and c_{t+1}. If the vertical and horizontal velocities are non-zero, we compute the linear interpolation between c_t and c_{t+1}, and pick $n = \max(|vx_t|, |vy_t|)$ equidistant points on it. We then map each point to the closest coordinates on the map, resulting in T. Thus, after every state transition, we have a set of visited cell coordinates and can compute whether the episode continues or is finished due to the agent hitting a wall, reaching the goal, or exceeding the maximum number of time steps.

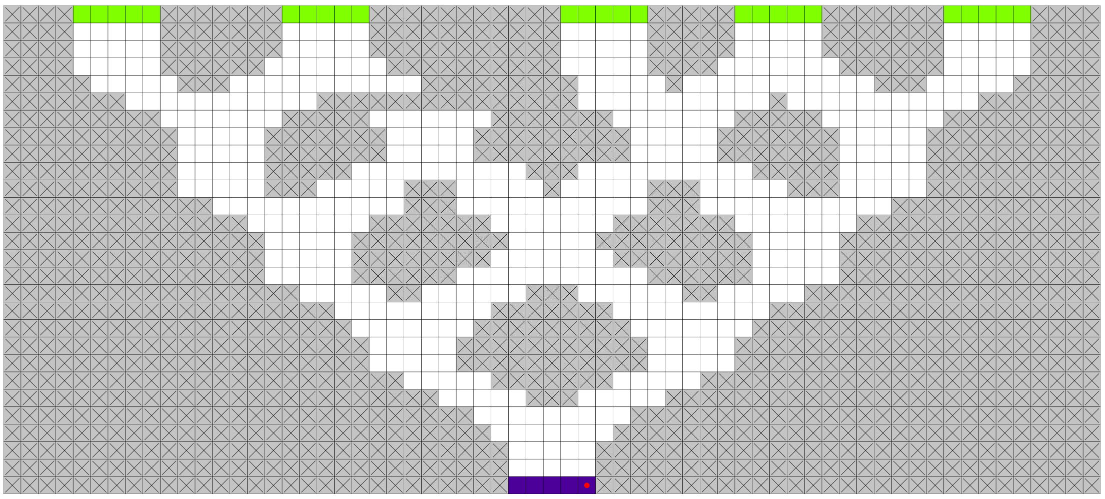
We will use one of the reward functions recommended by Gros [13], which is given by

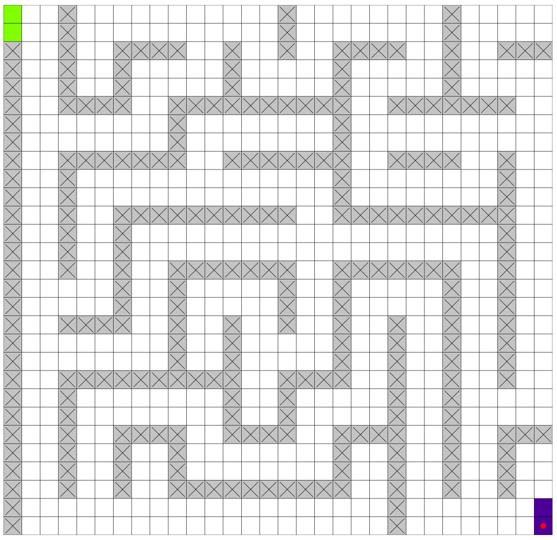
$$r(s_t, a_t, s_{t+1}) = \begin{cases} 100 & \text{if } T \text{ contains goal coordinates and no wall coordinates} \\ -20 & \text{if } T \text{ contains wall coordinates} \\ 0 & \text{otherwise} \end{cases}.$$

Furthermore, if the agent takes more than 100 steps, it earns a reward of 0, and the episode ends. The discount factor corresponds to $\gamma = 0.99$.

What makes Racetrack an appealing benchmark for the development and evaluation of RL algorithms is that it represents a simplification of autonomous driving, one of the most challenging RL applications [20]. Furthermore, it allows for testing a wide range of scenarios by utilizing different maps and noise probabilities. Racetrack can also be easily extended to accommodate more complex use cases such as multiple drivers, difficult weather conditions, or obstacles on the road.

Figure 2.5 shows the River-deadend, Maze, and Hansen-bigger maps, which vary in their difficulty and the challenges that they pose to the agents, making them ideal for our experiments in chapter 6. The River-deadend map has five goal lines that can be reached through various routes, allowing the agents to explore different paths to the goals. However, reaching a goal line requires only driving in a single direction on most routes, making it less challenging. The Maze map is characterized by its narrow paths and numerous dead ends, which put the agents at high risk of crashing once they deviate from the direct path to the goal line. The Hansen-bigger map is less intricate than the other maps, yet it requires agents to drive the farthest distance from starting line to the goal line. Furthermore, this map frequently alternates between straight and curvy passages, forcing agents to adapt their velocity constantly.





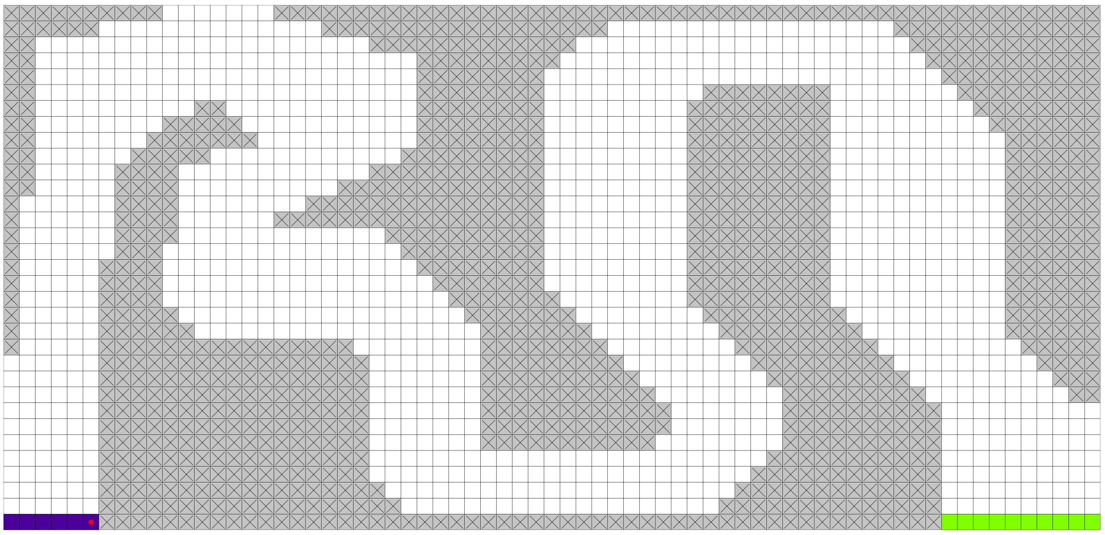


Figure 2.5 **The River-deadend (top), Maze (middle), and Hansen-bigger (bottom) maps.**

3. Reinforcement Learning Algorithms

Using the previously introduced RL framework, we can derive algorithms for learning policies that solve any given RL task. This chapter introduces *value-based algorithms* and how they can be improved using *deep neural networks* (DNN). Afterward, we will look upon an alternative to value-based algorithms, called *policy-gradient algorithms.*

3.1. Value-based Algorithms

Recall that an RL agent's goal is to learn an optimal policy π_*, enabling it to take actions a_t that maximize its return G_t. Given the optimal action-value function q_*, we can construct the policy

$$\pi_*(s_t) = \operatorname*{argmax}_{a \in \mathcal{A}(s_t)} q_*(s_t, a),$$

which is optimal since it returns the action a_t that leads to the highest expected return for every state s_t.

Since we cannot expect to have access to the optimal action-values $q_*(s_t, a_t)$, we learn approximations called *Q-values* using algorithms such as *Q-learning* [34].

Definition 3.1 (Q-value [32])**.** *The Q-value $Q(s_t, a_t)$ of a state-action pair (s_t, a_t) is an approximation of its action-value $q_\pi(s_t, a_t)$.*

Q-learning is based on the principle of *value iteration* [6], which is shown in Figure 3.1. The left diagram depicts how, in each iteration, Q-learning first makes the Q-value $Q(s_t, a_t)$ of the visited state-action pair (s_t, a_t) more accurate to its action-value $q_\pi(s_t, a_t)$ under the current policy π. After $Q(s_t, a_t)$ has been updated, another action a' might achieve a higher Q-value, indicating that a higher return can be earned by taking a' in s_t. Thus, in the second step, Q-learning changes π such that it is greedy with respect to the updated Q-values, resulting in the improved policy

$$\pi'(s_t) = \operatorname*{argmax}_{a \in \mathcal{A}(s_t)} Q(s_t, a).$$

The right side of Figure 3.1 illustrates how this cycle of alternating policy evaluation and policy improvement repeats until it reaches a fixed point where π is optimal.

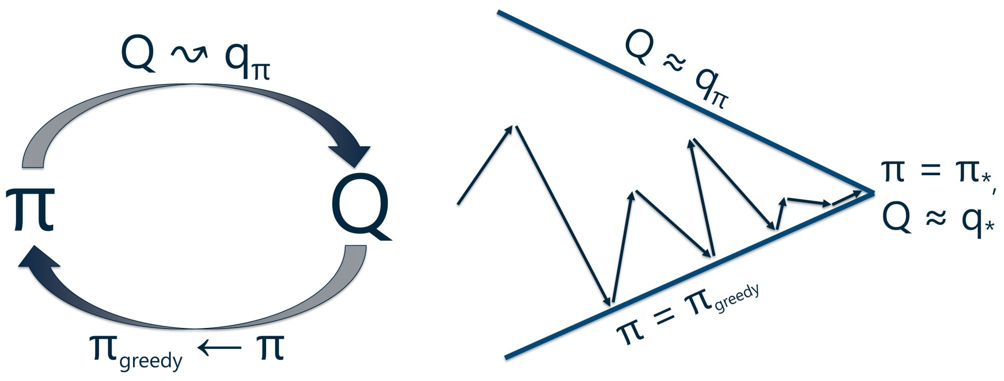


Figure 3.1 **Value iteration for Q-values.**

At the beginning of the training, Q-learning initializes the Q-values and stores them in a lookup table, called the *Q-table*. The accuracy of each Q-value can then be evaluated with the *temporal-difference error* (TD-error).

Definition 3.2 (Temporal-difference Error [32])**.** *The temporal-difference error*

$$TD(s_t, a_t) = r_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)$$

is a function $TD : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *that maps state-action pairs* (s_t, a_t) *to an approximation of*

$$q_\pi(s_t, a_t) - Q(s_t, a_t).$$

The term

$$r_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a')$$

estimates $q_\pi(s_t, a_t)$ by incorporating the earned reward r_{t+1} and then estimating the return's remainder assuming that the agent will take the *greedy action*

$$a = \operatorname*{argmax}_{a' \in \mathcal{A}(s_t)} Q(s_{t+1}, a')$$

in time step $t + 1$. This term is a more accurate estimate of $q_\pi(s_t, a_t)$ than $Q(s_t, a_t)$ since the latter estimates the return starting already at time step t. Because the TD-error computes the difference between these two estimates, it approximates how far off $Q(s_t, a_t)$ is from $q_\pi(s_t, a_t)$. Thus, a positive TD-error indicates $Q(s_t, a_t)$ should be increased and vice versa. Once we computed the TD-error of a state-action pair (s_t, a_t), we can improve $Q(s_t, a_t)$ using the update rule

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right],$$

which increases $Q(s_t, a_t)$'s accuracy by using the *step size* α to compute a fraction of $TD(s_t, a_t)$, and adding it to $Q(s_t, a_t)$.

Figure 3.2 shows an example of how Q-learning updates Q-values in the simplified autonomous driving task from Figure 2.1. The agent takes in the initial state s_0 the action a_1 since it achieves the highest Q-value. This causes the agent to transition to the state s_1, in which crashing into a cat is inevitable, so all applicable actions have a Q-value of -1. Afterward, Q-learning updates $Q(s_0, a_1)$ as

$$\begin{aligned} Q'(s_0, a_1) &= Q(s_0, a_1) + \alpha \left[r_{t+1} + \gamma \cdot Q(s_1, a_2) - Q(s_0, a_1)\right] \\ &= 1 + 0.8\left[0 + 0.9 \cdot (-1) - 1\right] \\ &= -0.52 \end{aligned},$$

meaning that now $Q(s_0, a_1) < Q(s_0, a_2)$ holds. Therefore, the agent will take in the next episode the action a_2 in the state s_0, making it possible to drive past the cats in the subsequent time step.

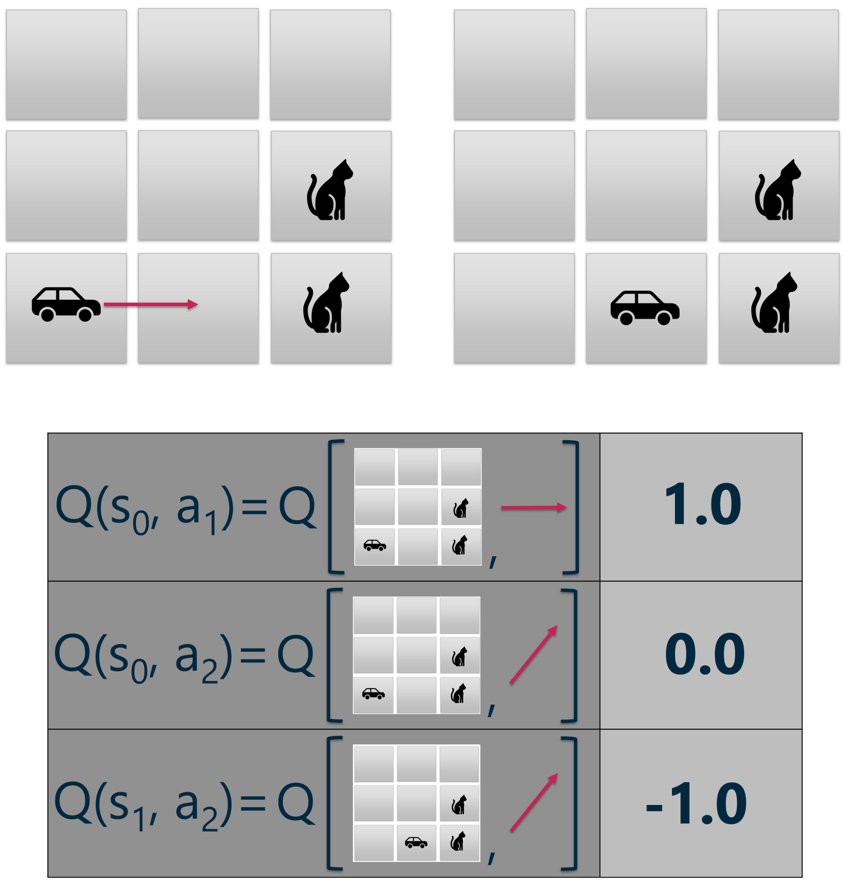


Figure 3.2 **State transition (top) and partial Q-table (bottom) from the simplified autonomous driving task.** The agent receives a reward of -1 for crashing into a cat, a reward of 1 for driving past the cats, and 0 otherwise. The discount factor γ is 0.9 and the step size α is 0.8.

If the agent always takes greedy actions, it will not discover optimal strategies requiring taking actions that initially seem unbeneficial. To ensure that the agent sufficiently explores the state space, Q-learning utilizes an *ϵ-greedy policy*

$$\pi(s_t) = \begin{cases} \underset{a \in \mathcal{A}(s_t)}{\operatorname{argmax}} Q(s_t, a) & \text{with probability } 1 - \epsilon \\ \text{random } a \in \mathcal{A}(s_t) & \text{with probability } \epsilon \end{cases},$$

meaning that the agent will execute a random action with probability ϵ. We will exponentially decay ϵ throughout the training [16], meaning we update ϵ as $\epsilon = \epsilon \cdot \epsilon_{decay}$ with $\epsilon_{decay} < 1$ after every episode until we reach $\epsilon = \epsilon_{end}$. This way, the agent will explore taking different actions at the beginning of the training, and after sufficient exploration, it will focus on taking greedy actions.

Note that Q-learning gathers samples using an ϵ-greedy policy but updates its Q-values as if following a greedy policy. This way, we can enforce exploration while ensuring that the final policy will not take random actions. RL algorithms that do not gather samples according to the policy they optimize are called *off-policy algorithms*, as opposed to *on-policy algorithms* that gather samples using the policy they optimize [32].

Algorithm 1 shows the basic Q-learning algorithm that we can construct from all the previously explained techniques. Every episode begins, in line 2, with sampling the initial state $s_0 \in \mathcal{I}$ from the initial state distribution μ. At every following time step t, from lines 4 to 6, the agent acts in the environment using its ϵ-greedy policy, and gathers new observations. In line 7, Q-learning updates the Q-value of the current state-action pair according to its TD-error, and the next iteration begins.

Algorithm 1 Q-learning

```
1: for episodes $e = 0$ to $E - 1$ do
2:     sample $s_0 \in \mathcal{I}$ from $\mu$
3:     for steps $t = 0$ to $T - 1$ do
4:         with probability $\epsilon$ select random action $a_t \in \mathcal{A}(s_t)$
5:         otherwise with probability $1 - \epsilon$ select $a_t = \underset{a \in \mathcal{A}(s_t)}{\operatorname{argmax}} Q(s_t, a)$
6:         execute $a_t$; observe $s_{t+1}$ and $r_{t+1}$
7:
           $$Q'(s_t, a_t) = \begin{cases} Q(s_t, a_t) + \alpha \left[ r_{t+1} - Q(s_t, a_t) \right] & s_{t+1} \text{ terminal} \\ Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right] & \text{else} \end{cases}$$
8:     end for
9: end for
```

3.2. Deep Reinforcement Learning

In practice, most sequential decision-making problems are so complex that we require MDPs with gigantic state and action spaces to model them accurately. This leads to tremendous memory requirements for storing the Q-values of all state-action pairs, making value-based RL algorithms, such as Q-learning, inapplicable. Even with sufficient memory, value-based RL algorithms are still undesirable since, in large MDPs, many state-action pairs will never be visited during training, which prevents agents from learning their Q-values. We can address these problems by exchanging the Q-table in value-based algorithms with *function approximators* that map given state-action pairs to their Q-values. Function approximators have the advantage of requiring drastically less memory than lookup tables. Additionally, they can generalize to unseen data, allowing agents to learn Q-values for state-action pairs that were not encountered during training.

In general, we will refer to RL algorithms that utilize function approximators belonging to the class of *deep neural networks* (DNN) as *deep reinforcement learning* (DRL) algorithms. DNNs consist of multiple layers of neurons that compute non-linear functions of their inputs. Figure 3.3 shows a neuron, which receives the outputs of the previous layer's neurons as input. The neuron then multiplies each input I_i by a weight w_i, takes the sum, and adds a bias term b to the weighted sum. The result is a linear function of the neuron's inputs. To enable the neuron to model non-linear relationships, its computation is passed to a non-linear activation function. The output is then given to the neurons in the next layer, and the process repeats. We will call the entirety of the weights and biases of a DNN its parameters θ.

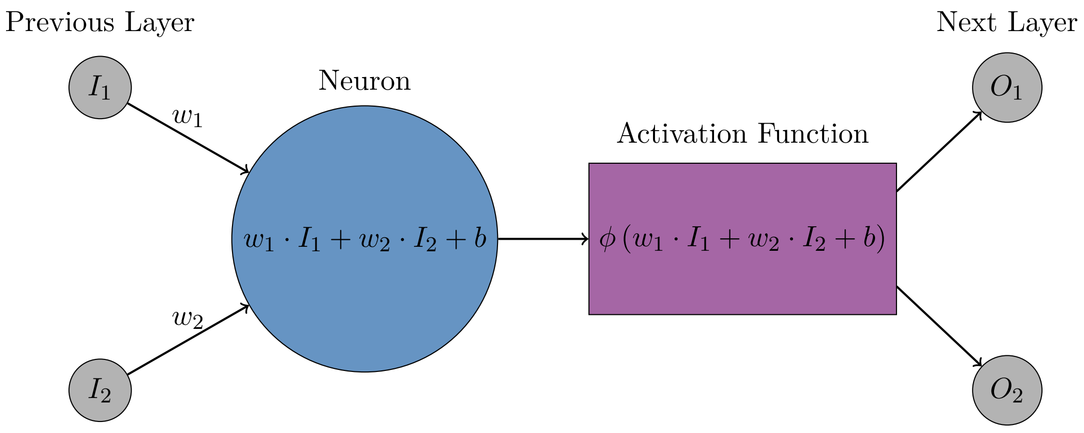


Figure 3.3 **A neuron's computation.**

In value-based DRL algorithms, we replace the Q-table with a DNN, called the *Q-network*, so we designate the Q-values as Q_θ to express that they are computed by a DNN with parameters θ. Figure 3.4 depicts how we can stack multiple layers of neurons to build a Q-network for a Racetrack agent. First, the current state is passed to the input layer, the output of which is then given to the first hidden layer. This process repeats for the second hidden layer. In the last step, the output layer receives the second hidden layer's output and computes the Q-values for every action. This Q-network belongs to the class of fully connected DNNs since each neuron is connected to every neuron in the previous and succeeding layer [2].

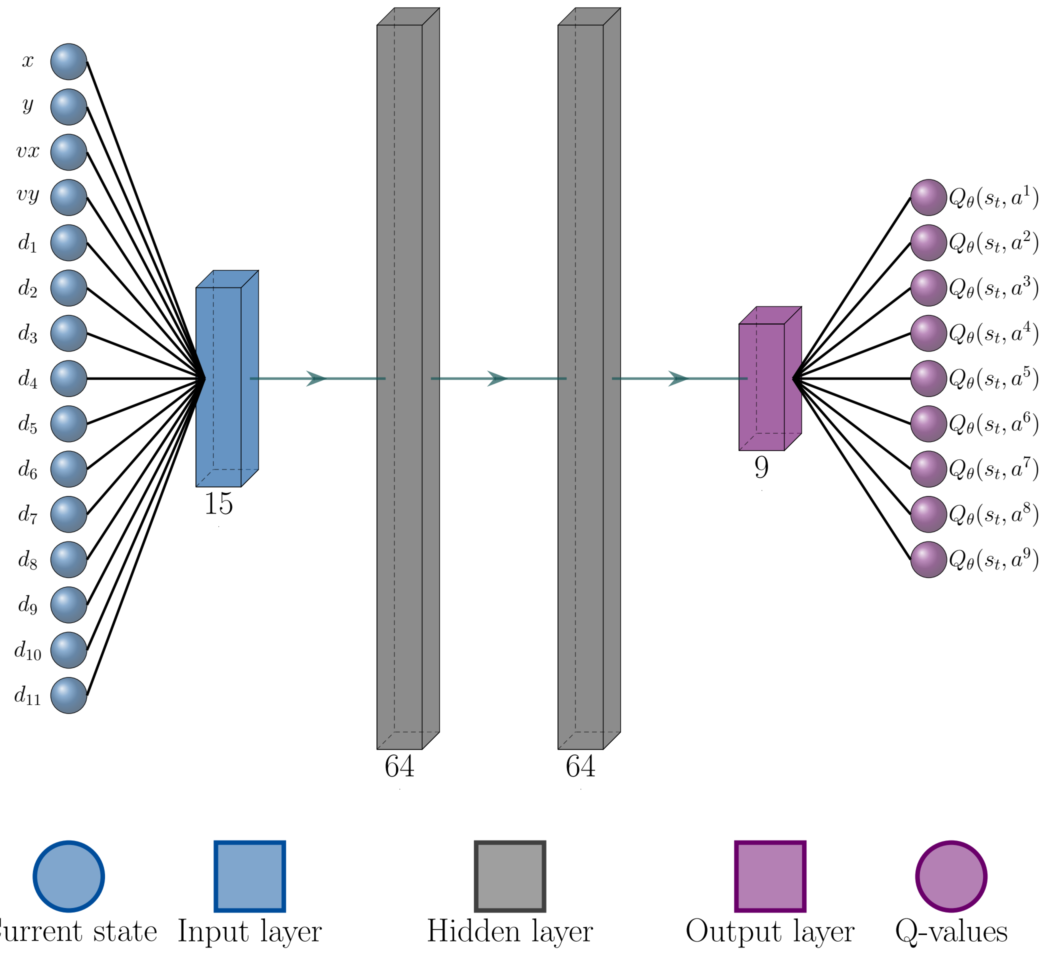


Figure 3.4 **A Q-network for Racetrack.** Each layer is annotated with its number of neurons.

Assuming that our Q-network's architecture has a sufficient number of neurons and layers, we can compute accurate Q-values $Q_\theta \approx q_\pi$ by finding the corresponding parameters θ. We can achieve this by iteratively optimizing our parameters θ_i with respect to a *loss function* $L(\theta_i)$ that measures the expected difference between the Q-network's current function and the target function q_π. In each iteration, the loss function is then given by

$$L(\theta_i) = \mathbb{E}_{\theta_i}\left[\left(y_{\theta'}(S_t, A_t) - Q_{\theta_i}(S_t, A_t)\right)^2\right], \tag{I}$$

where the target y_θ corresponds to

$$y_\theta(s,a) = \mathbb{E}_\theta\left[R_{t+1} + \gamma \cdot \max_{a'} Q_\theta(S_{t+1}, a')\middle| S_t = s, A_t = a\right] \text{ [23].}$$

The expectation in $L(\theta_i)$ is taken over the transitions that were gathered by the policy π, induced by the Q-network with parameters θ_i.

Algorithm 2 depicts the DRL version of Q-learning, called *deep Q-learning* (DQN) [23]. At the beginning of each Q-network update, DQN needs to sample a minibatch of samples on which it can compute the loss. However, popular algorithms for the optimization of DNNs, such as stochastic gradient descent [26] or Adam [19], assume that these samples are identically and independently distributed (i.i.d.), yet subsequent transitions collected by a DRL agent are highly correlated. In line 7, DQN resolves this by storing encountered transitions in a replay buffer D and then sampling the transitions with uniform probability from the buffer during each update. Additionally, this allows the same transition to be used several times to update the Q-network, potentially decreasing the number of individual samples needed to fit its parameters. After sampling the minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1})$ in line 9, DQN computes the loss

$$L(\theta_i) = \left(r_{j+1} + \gamma \cdot \max_{a'} Q_{\theta'}(s_{j+1}, a') - Q_{\theta_i}(s_j, a_j)\right)^2, \tag{II}$$

which approximates the loss function (I). Next, DQN computes the Q-network's new parameters θ_{i+1} by performing a gradient descent step on $L(\theta_i)$. We will utilize the Adam optimizer for this since it has empirically been shown to achieve high performance [19].

DQN is known to suffer from unstable performance [33], and this can primarily be attributed to the fact that updating $Q(s_t, a_t)$ will likely also change $Q(s_{t+1}, a_{t+1})$ for succeeding state-action pairs. This means a destructive Q-network update will make both $Q(s_t, a_t)$ and $Q(s_{t+1}, a_{t+1})$ inaccurate, thus making the TD-error $TD(s_t, a_t)$ less informative in future updates. This severely limits the agent's ability to recover accurate Q-values, meaning more destructive updates are likely to follow, creating a cycle of decreasing performance. To facilitate this problem, we utilize in DQN's loss function (II) a local Q-network with parameters θ_i that we directly optimize and a target Q-network with parameters θ' for which the update rule is given by

$$\theta' = (1-\tau)\cdot\theta_i + \tau\cdot\theta',$$

where $\tau \in (0,1)$. In line 10, the target network is then used to compute the target

$$r_{j+1} + \gamma \cdot \max_{a'} Q_{\theta'}(s_{j+1}, a'),$$

whereas the local network computes $Q_{\theta_i}(s_j, a_j)$ in line 11. In the case of a destructive update, the local network will become inaccurate, whereas the target network is only partially affected due to the soft update to θ' [21] in line 12. This means the target will remain sufficiently accurate for multiple iterations, which preserves the informativeness of the TD-error.

Algorithm 2 Deep Q-learning

```
 1: for episodes $e = 0$ to $E - 1$ do
 2:     sample $s_0 \in \mathcal{I}$ from $\mu$
 3:     for steps $t = 0$ to $T - 1$ do
 4:         with probability $\epsilon$ select random action $a_t \in \mathcal{A}(s_t)$
 5:         otherwise with probability $1 - \epsilon$ select $a_t = \underset{a \in \mathcal{A}(s_t)}{\operatorname{argmax}} Q_{\theta_i}(s_t, a)$
 6:         execute $a_t$; observe $s_{t+1}$ and $r_{t+1}$
 7:         store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay buffer $D$
 8:         every $K$ steps do
 9:             sample minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1})$ from $D$
10:             set target $y_j = \begin{cases} r_{j+1} & s_{j+1} \text{ terminal} \\ r_{j+1} + \gamma \cdot \max_{a'} Q_{\theta'}(s_{j+1}, a') & \text{else} \end{cases}$
11:             perform gradient descent step on loss $(y_j - Q_{\theta_i}(s_j, a_j))^2$
12:             soft-update the network weights $\theta' = (1 - \tau) \cdot \theta_i + \tau \cdot \theta'$
13:         end every
14:     end for
15: end for
```

A popular extension of the DQN algorithm, called *deep Q-learning with prioritized experience replay* (DQNPR) [28], is shown in Algorithm 3. It is based on the idea that transition samples with a low TD-error do not contribute as much to the learning process as samples with a high TD-error since they carry less relevant information. Therefore, in line 8, DQNPR stores every transition in the replay buffer along with a sampling priority

$$\delta = \left(\left(Q_{\theta'}(s_t, a_t) - \left(r_{t+1} + \gamma \cdot \max_{a'} Q_{\theta_i}(s_{t+1}, a') \right) \right) + \epsilon_p \right)^{\alpha},$$

that is based on the TD-error. Note that DQNPR utilizes two hyperparameters: $\epsilon_p > 0$ ensures that every sample will have a non-zero sampling priority δ, preventing the agent from neglecting low TD-error samples completely. $\alpha \in (0, 1)$ controls the amount of prioritization and the closer it is to 1, the stronger the prioritization of high TD-error samples. During the Q-network update, in line 10, the minibatch's transitions are then sampled with probabilities that are proportional to their priority δ. This ensures that high TD-error samples are used more frequently to update the Q-network. By not sampling the transitions randomly but according to their priority δ, we introduce bias into the estimation of the Q-network's loss. To reduce this bias, in line 13, DQNPR multiplies the current loss by importance sampling weights

$$w_i = \left(\frac{1}{|D|} \cdot \frac{1}{P(i)} \right)^{\beta} \cdot \frac{1}{\max_i w_i},$$

where $|D|$ corresponds to the number of currently stored samples and $P(i)$ corresponds to the sampling probability. Thus, the higher the sampling probability $P(i)$, the more we decrease the sample's impact on the Q-network update. β controls how much we compensate for the altered sampling probabilities, so we achieve full compensation for $\beta = 1$. The authors proposed to start with $\beta < 1$ and gradually increase it towards 1 during the training since annealing the bias becomes more important as the Q-values start to converge towards the action-values. To further increase stability, they divide each weight w_i by the largest weight $\max_i w_i$ such that all w_i are at most 1, meaning that the Q-network updates can only be scaled downwards.

Algorithm 3 Deep Q-learning with Prioritized Experience Replay

```
1: for episodes $e = 0$ to $E - 1$ do
2:     sample $s_0 \in \mathcal{I}$ from $\mu$
3:     for steps $t = 0$ to $T - 1$ do
4:         with probability $\epsilon$ select random action $a_t \in \mathcal{A}(s_t)$
5:         otherwise with probability $1 - \epsilon$ select $a_t = \underset{a \in \mathcal{A}(s_t)}{\operatorname{argmax}} Q_{\theta_i}(s_t, a)$
6:         execute $a_t$; observe $s_{t+1}$ and $r_{t+1}$
7:         compute $\delta$ using the TD-error
8:         store $(s_t, a_t, r_{t+1}, s_{t+1}, \delta)$ in replay buffer $D$
9:         every $K$ steps do
10:            sample minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1}, \delta)$ from $D$ w.r.t. $\delta$
11:            compute importance sampling weight $w_j$
12:            set target $y_j = \begin{cases} r_{j+1} & s_{j+1} \text{ terminal} \\ r_{j+1} + \gamma \cdot \max_{a'} Q_{\theta'}(s_{j+1}, a') & \text{else} \end{cases}$
13:            perform gradient descent step on loss $w_j \cdot (y_j - Q_{\theta_i}(s_j, a_j))^2$
14:            soft-update the network weights $\theta' = (1 - \tau) \cdot \theta_i + \tau \cdot \theta'$
15:        end every
16:    end for
17: end for
```

3.3. Policy-gradient Algorithms

We introduced value-based algorithms based on the principle that we can learn Q-values to construct a policy. The motivation behind *policy-gradient algorithms* is that we can instead directly learn a *stochastic policy* π_θ, which is parameterized by θ.

Definition 3.3 (Stochastic Policy [31]). *A stochastic policy π_θ is a function $\pi_\theta : \mathcal{S} \to \mathcal{D}(\mathcal{A})$ that returns a probability distribution $\pi_\theta(s_t)$ over all actions $a \in \mathcal{A}$ for a given state s_t.*

We can learn an optimal stochastic policy $\pi_*(s_t)$ by optimizing our parameters θ according to the policy-gradient

$$g_\theta = \nabla_\theta v_{\pi_\theta}(S_0),$$

corresponding to maximizing the expected return that the policy π_θ achieves. However, we cannot expect to have access to the state-value function v_{π_θ}, so policy-gradient algorithms utilize surrogate objectives whose gradient approximates g_θ. In the context of DRL, we parameterize the policy π_θ with a DNN and base our loss function on such a surrogate objective. One of the most successful policy-gradient DRL algorithms is called *proximal policy optimization* (PPO) [31], and it maximizes a surrogate objective that is based on the objective

$$L_t^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right].$$

In L_t^{CPI}, $\hat{\mathbb{E}}_t$ corresponds to the sample mean over a trajectory of k transitions, generated by the policy $\pi_{\theta_{\text{old}}}$. L_t^{CPI}'s first term is the probability ratio $r_t(\theta)$, which quantifies how much the policy π_θ deviates from the previous policy $\pi_{\theta_{old}}$. The second term $\hat{A}_t$ estimates the *advantage function.*

Definition 3.4 (Advantage Function [30]). *The advantage function*

$$A_\pi(s_t, a_t) = q_\pi(s_t, a_t) - v_\pi(s_t)$$

is a function $A : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that computes the difference in expected return between following the policy π after taking the action a_t in the state s_t and following π already in s_t.

We can improve the expected return of a stochastic policy π_θ in the state s_t, by increasing the sampling probabilities $\pi_\theta(a_t|s_t)$ of actions a_t that achieve a positive advantage $A_{\pi_\theta}(s_t, a_t) > 0$, and by decreasing $\pi_\theta(a_t|s_t)$ of a_t that achieve a negative advantage $A_{\pi_\theta}(s_t, a_t) < 0$. Therefore, we compute in L_t^{CPI} for each of the k given samples an estimate of the advantage function. To do so, we utilize an approximation of the state-value function v_{π_θ}, which is called the critic V. We can then compute

$$Q(s_t, a_t) \approx r_{t+1} + \gamma V(s_{t+1}),$$

which allows us to approximate $A_{\pi_\theta}(s_t, a_t)$ as

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t).$$

However, δ_t induces bias if $V \approx v_{\pi_\theta}$ does not hold. We can reduce this bias by incorporating more earned rewards r_{t+i} in our approximation of $Q(s_t, a_t)$. If we use all of the current trajectory's remaining rewards $\langle r_{t+1}, ..., r_{t+k} \rangle$, we get the estimator

$$\begin{aligned}\hat{A}_t &= r_{t+1} + \gamma r_{t+2} + ...\gamma^{k-1} r_{t+k} + \gamma^k V(s_{t+k}) - V(s_t) \\ &= \sum_{i=0}^{k-1} \gamma^i \delta_{t+i}\end{aligned},$$

which has a lower bias than δ_t but a higher variance due to using more samples. We can trade-off bias and variance in $\hat{A}_t$ by introducing the hyperparameter $\lambda \in (0, 1)$, giving us the advantage estimator called *generalized advantage estimation* (GAE) [30]

$$\hat{A}_t = \sum_{i=0}^{k-1} (\gamma\lambda)^i \delta_{t+i},$$

where for $\lambda = 0$, we minimize variance but attain high bias, and for $\lambda = 1$, we minimize bias but attain high variance.

If we would use L_t^{CPI} to train a PPO agent, we may experience destructively large policy updates, meaning that π_θ deviates from $\pi_{\theta_{\text{old}}}$ significantly, which leads to the probability ratio $r_t(\theta)$ being much larger or smaller than 1. To penalize such large policy updates, we first clip $r_t(\theta)$ to a small interval around 1, giving us the clipped objective

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t \right].$$

Additionally, we take the minimum over the clipped and unclipped objectives, giving us the objective

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t) \right],$$

which represents a pessimistic lower bound on the unclipped objective L_t^{CPI}. This means we restrict how much a single update can improve the maximization objective L_t^{CLIP} by increasing the probability $\pi_\theta(a_t|s_t)$ of actions a_t with $\hat{A}(s_t, a_t) > 0$. Conversely, we restrict how much L_t^{CLIP} can be improved by decreasing the probability $\pi_\theta(a_t|s_t)$ of actions a_t with $\hat{A}(s_t, a_t) < 0$. However, we do not limit how much the maximization objective L_t^{CLIP} can be decreased by increasing the probability of unbeneficial actions, or by decreasing the probability of beneficial actions. Thus, we remove the incentives for making policy updates that make the probability ratio $r_t(\theta)$ go beyond the interval $[1-\epsilon, 1+\epsilon]$.

Since PPO uses a critic V to compute its objective, it belongs to the class of *actor-critic algorithms* [32]. In this context, the policy π_θ is also referred to as the actor or, in the case of DRL, the DNN that computes π_θ is called the actor network. To compute the critic V, we use a DNN, called the critic network, and optimize it using an objective that measures the difference between V and our target function v_{π_θ}

$$L_t^{VF}(\theta) = \left(V(s_t) - V_t^{targ}\right)^2,$$

where we can compute the target $V_t^{targ} = \sum_{i=t}^{k} \gamma^{i-t} r_{i+1}$ using the remaining rewards in our trajectory. Furthermore, we can reduce the total number of parameters by combining the policy and critic network into one DNN with separate output layers, allowing us to optimize both using a single objective, given by

$$L_t^{CLIP+VF}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta)\right].$$

To enforce exploration, we add to $L_t^{CLIP+VF}$ an entropy bonus $S[\pi_\theta](s_t)$, which measures the randomness of π_θ in the state s_t. This way, the agent attempts to learn one of the most random policies among all optimal policies, meaning it achieves both exploration and exploitation. PPO's final maximization objective is then given by

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right].$$

Exploration can be further increased by using multiple agents that share the same policy and gather samples in parallel. However, this version of PPO requires significantly more computing resources, so it will not be used in this thesis.

The pseudo-code of PPO is shown in Algorithm 4, and it follows the same basic structure of alternating between gathering samples and updating the policy as the previously introduced value-based algorithms. The most notable difference is that the policy update computes M updates on a single minibatch, called epochs. To do so, PPO first loads transitions gathered by the most recent policy π_{θ_i} in line 8, and then computes their advantage estimates. Afterward, it computes the loss, takes a gradient descent step, and repeats it M times from lines 11 to 13.

One of the advantages of policy-gradient algorithms like PPO is that by updating their policy's parameters θ with gradient steps, they make smooth changes to the action probabilities $\pi_\theta(a_t|s_t)$. This is in contrast to value-based algorithms like DQN and DQNPR, where a slight change in Q-values can change which actions achieve the highest Q-value, causing drastic changes to the policy. Therefore, policy-gradient algorithms have better convergence guarantees than value-based algorithms [5, 32]. However, making smooth gradient-based updates increases the likelihood of getting stuck in local optima. Furthermore, estimates of the policy-gradient g_θ tend to suffer from high variance [3]. PPO is an on-policy algorithm since $L_t^{CLIP+VF+S}$ assumes that the given samples are generated by the previous policy $\pi_{\theta_{old}}$. Thus, PPO cannot use a replay buffer, meaning that gathered samples can be considered only for one epoch of policy updates. This may increase the number of samples required to fit the policy's parameters θ.

Algorithm 4 Proximal Policy Optimization (single agent)

```
1: for episodes $e = 0$ to $E - 1$ do
2:     sample $s_0 \in \mathcal{I}$ from $\mu$
3:     for steps $t = 0$ to $T - 1$ do
4:         sample $a_t \sim \pi_{\theta_i}(s_t)$
5:         execute $a_t$; observe $s_{t+1}$ and $r_{t+1}$
6:         store transition $(s_t, a_t, r_{t+1}, s_{t+1})$
7:         every $K$ steps do
8:             Load minibatch of samples gathered by $\pi_{\theta_i}$
9:             Compute advantage estimates $\hat{A}_t$ using GAE
10:            $\theta_i^1 = \theta_i$
11:            for steps $m = 1$ to $M$ do
12:                perform gradient descent step on loss $-L_t^{CLIP+VF+S}(\theta_i^m)$ to get $\theta_i^{m+1}$
13:            end for
14:            $\theta_{i+1} = \theta_i^M$
15:        end every
16:    end for
17: end for
```

4. DSMC Evaluation Stages

A multitude of DRL applications require agents not only to solve the given task but do so in a safe manner. However, training DRL agents to act safely often involves comprising in performance. This chapter introduces the field of *safe reinforcement learning* and common causes for safety issues in DRL. We will then present how an agent's behavior can be evaluated using *deep statistical model checking* (DSMC). Afterward, we will utilize DSMC's evaluations in the *DSMC evaluation stages* (DSMC ES) algorithms, which allow us to achieve safe behavior without sacrificing performance.

4.1. Safe Reinforcement Learning

Safe reinforcement learning aims to train agents that solve a given task while obeying safety constraints [11]. These measures are necessary for *safety-critical applications* where violating such a constraint might have catastrophic consequences. In the context of this thesis, we will assume that an agent disregarding any such constraint corresponds to a safety error that leads to the immediate termination of the current episode.

Assume we have a real-life model of the Racetrack map Barto-big, including a racecar, and our goal is to make it drive autonomously using DRL. Recall that a Racetrack agent receives a reward of 100 for reaching the goal line and that we use a discount factor of $\gamma = 0.99$. Thus, when reaching the goal line, an Racetrack agent's return corresponds to $0.99^{T-1} \cdot 100$, where T is the number of steps taken. Figure 4.1 compares the trajectories of two agents. Agent A achieved a return of 75.5 by reaching the goal line in 29 steps, whereas agent B achieved a lower return of 69.6 due to taking 37 steps. Nevertheless, agent A's behavior is undesirable since it drives with a high velocity and close to the map's walls, resulting in a high chance of crashes that could destroy the racecar. Thus, we would deploy agent B, which slowly drives in the middle of the racetrack, giving it a high probability of reaching the goal line. This example illustrates that even though the return and the *probability of reaching a goal state* (GRP) are correlated, maximizing the return often does not correspond to maximizing the GRP. This is one of the root causes of safety problems in DRL, and, in theory, we can prevent it by using a binary reward function

$$r_{binary}(s_t, a_t, s_{t+1}) = \begin{cases} 1 & \text{if task solved} \\ 0 & \text{otherwise} \end{cases}.$$

Given this reward function and no discounting, the average return corresponds to the fraction of episodes ending in goal states and, thus, to the probability of solving the task. However, maximizing this objective is extraordinarily difficult since it does not provide any feedback to the agent unless it reaches a goal state. To increase the learning

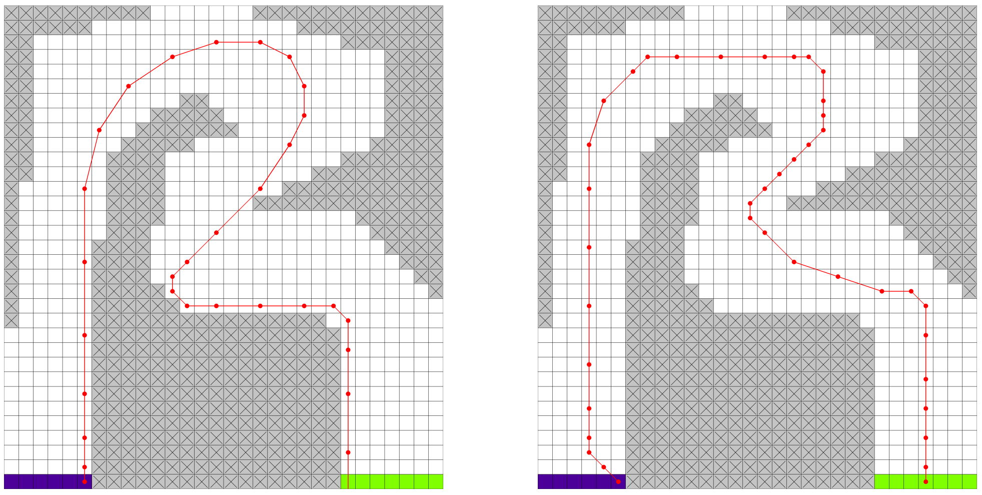


Figure 4.1 **Trajectories of agent A (left) and agent B (right) on the Barto-big map.**

speed, we can utilize a more informative reward function that provides the agent with intermediate feedback. For instance, the reward function that we use in Racetrack penalizes the agent with negative rewards whenever it crashes into a wall. Nevertheless, using a reward function other than r_{binary} means the agent's maximization objective does not correspond to the GRP (Problem 1). One way to counteract this safety negligence is to use an informative reward function that also enforces safe behavior. Despite resulting in shorter training times than r_{binary}, these functions are known to cause substantially poorer performance than reward functions designed for quick learning [11]. This is due to the challenging maximization objectives that result from safety-focused reward functions. For example, if we give a Racetrack agent a reward of -1000 for crashing, it is likely to stop driving since it tries to avoid the large negative reward but cannot explore the map without crashing.

Another aspect that reduces safety in DRL is that many algorithms, such as DQN,DQNPR, and PPO, maximize the average return. This means safety errors leading to low returns are disregarded when they rarely occur since they do not affect the average return noticeably (Problem 2). However, even a low probability of error is unacceptable in safety-critical applications.

4.2. Deep Statistical Model Checking

Since a high average return does not ensure safe behavior, we must evaluate an agent's policy according to metrics relevant to safety. To do so, we utilize *deep statistical model checking* (DSMC) [15], which is a technique that analyzes properties of DNN-induced policies using *statistical model checking* [18].

Starting at an initial state s_0, DSMC repeatedly queries from the policy's DNN an action a_t in order to transition to the next state s_{t+1}, until a terminal state is reached. Thus, the DNN acts as a black-box oracle, which resolves the non-determinism of the task's MDP at every visited state. This process results in a sequence of state transitions corresponding to a deterministic *Markov chain*.

Definition 4.1 (Oracle-induced Markov Chain [15])**.** *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mu \rangle$ be an MDP and $\pi : \mathcal{S} \to \mathcal{A}$ an action oracle given by a DRL agent's policy. Then π induces in $\mathcal{M}$ the Markov chain $C^\pi = \langle \mathcal{S}, \mathcal{T}', s_0 \rangle$ that comprises a set of states $s \in \mathcal{S}$, an initial state s_0, and a transition probability function $\mathcal{T}' : \mathcal{S} \to \mathcal{D}(S)$, for which $\mathcal{T}'(s) = \mathcal{T}(s, \pi(s))$ holds.*

Next, DSMC analyzes the deterministic Markov chain with respect to the desired property q, giving us the estimate q'. The model executions and Markov chain evaluations are then repeated until the error $|q' - q|$ is within the statistical confidence bound

$$P(|q' - q| > \epsilon) < \kappa,$$

meaning that the probability of the error in q' being larger than ϵ is smaller than κ.

Consider we trained a Racetrack agent on the Barto-big map with the random restart setting activated and want to know if it has learned safe behavior. We can evaluate this by computing with DSMC the agent's GRP when starting on each initial state. Additionally, we evaluate the agent's expected return in each initial state. The left side of Figure 4.2 visualizes the results of DSMC as a heatmap, where the greener a cell is, the higher the agent's GRP when starting on it. It shows that the agent lacks safety in the upper and left areas of the map since these areas are colored yellow and orange. The right side of Figure 4.2 shows the results of evaluating with DSMC the agent's expected return when starting on each map cell. Comparing this heatmap to the former, we can observe a correlation between GRP and expected return. In the following, we will call such heatmaps global GRP and global return heatmaps.

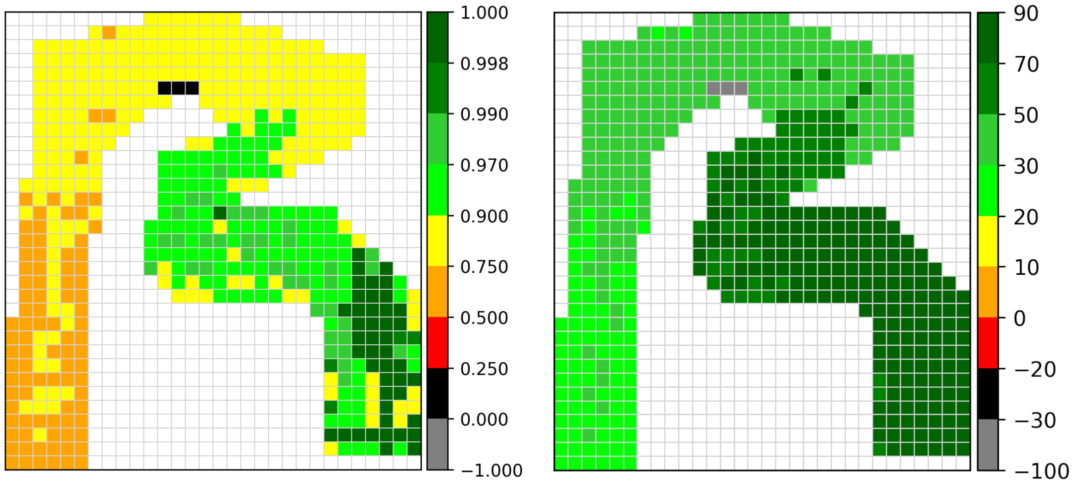


Figure 4.2 **Global GRP (left) and global return (right) heatmap of the Barto-big map.**

4.3. DSMC Evaluation Stages

Although DSMC can evaluate a DRL agent's behavior, it does not tell us how to train safe agents. However, we can use DSMC to analyze certain properties of the agent's policy throughout the training and then focus the training on the initial states in which these properties are the most deficient. The agent will then attempt to increase its return in these difficult regions, which likely addresses the deficient properties. This is the basic principle that allows *DSMC evaluation stages* [16] (DSMC ES) to train safe DRL agents. To apply DSMC ES, we first need to compute a partition of the task's set of initial states

$$\mathbb{P} = \{J_1, J_2, ..., J_k\} \text{ where } \forall i \in 1, 2, ..., k : J_i \neq \emptyset \text{ and } \bigcup_{i \in 1,2,...,k} J_i = \mathcal{I},$$

and then choose for each initial state region J_i a representative state $s_i \in J_i$. DSMC ES then executes an evaluation stage by running DSMC on the representatives of each region. The resulting evaluation values $E(J_i)$ are passed to an underlying training algorithm and utilized accordingly. The evaluation stages are repeated at regular intervals since the evaluation values $E(J_i)$ may become inaccurate as the agent's policy changes.

Gros et al. [16] utilized two configurations of DSMC ES: The first configuration computes the evaluation values $E(J_i)$ as the agent's GRP, and the second one computes $E(J_i)$ as the agent's expected return, which is linearly interpolated between 0 and 1. The first configuration addresses Problem 1 by focusing the training on initial state regions where the agent's GRP is low, such that the agent may learn to increase its GRP. Thus, we can incorporate the safety objective of increasing the GRP into the training process without requiring a reward function that decreases performance. The second configuration addresses Problem 2 since it allows us to focus the training on initial state regions where low returns occur when starting in them, such that ameliorating these deficiencies becomes significant to maximizing the average return.

4.3.1. Evaluation-based Initial Distribution

Evaluation-based initial distribution (EID) is one of two algorithms that utilize DSMC ES during training. EID makes the probabilities of starting in an initial state $s_0 \sim J_i$ inversely proportional to the evaluation value $E(J_i)$ of the corresponding initial state region J_i. Thus, the agent will start more frequently in regions J_i where $E(J_i)$ is low, allowing it to gather samples from which it can learn to increase its performance. Given the MDP's initial state distribution μ, EID first defines an initial probability distribution

$$\beta(J_i) = \frac{\mu(s_i)}{\sum_{j=1}^{k} \mu(s_j)},$$

over the regions J_i, using their representatives s_i. After the first evaluation stage, EID shifts β such that J_i with a low $E(J_i)$ have an increased probability of being sampled. Additionally, a minimum priority ϵ_p is used to ensure that every J_i has a non-zero probability of being sampled

$$p(J_i) = \frac{(1 - E(J_i) + \epsilon_p) \cdot \beta(J_i)}{\sum_j (1 - E(J_j) + \epsilon_p) \cdot \beta(J_j)}.$$

This measure stops EID from ignoring regions J_i in which $E(J_i)$ is high. In the last step, the initial state s_0 is uniformly sampled from the sampled region J_i.

Algorithm 5 shows the pseudo-code of an EID variant that uses DQN as its basis. The first difference between this EID variant and DQN is that, from lines 2 to 3, it samples the initial state according to the results of the previous evaluation stage. The second difference is that, from lines 16 to 20, EID updates the evaluation values $E(J_i)$ by computing a new evaluation stage. Note that the first $E(J_i)$ are only computed after W episodes since an agent that did not receive minimum training likely performs poorly in all regions J_i, making it redundant to evaluate its policy. Since EID neither changes how the agent gathers samples nor how the policy is updated, it can be used in conjunction with any other DRL algorithm, such as PPO.

Algorithm 5 Evaluation-based Initial Distribution (DQN version)

```
1:  for episodes $e = 0$ to $E - 1$ do
2:      sample $J_i$ w.r.t. $p(J_i)$
3:      sample $s_0 \sim J_i$ uniformly
4:      for steps $t = 0$ to $T - 1$ do
5:          with probability $\epsilon$ select random action $a_t \in \mathcal{A}(s_t)$
6:          otherwise with probability $1 - \epsilon$ select $a_t = \operatorname*{argmax}_{a \in \mathcal{A}(s_t)} Q_{\theta_i}(s_t, a)$
7:          execute $a_t$; observe $s_{t+1}$ and $r_{t+1}$
8:          store $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay buffer $D$
9:          every $K$ steps do
10:             sample minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1})$ from $D$
11:             set target $y_j = \begin{cases} r_{j+1} & s_{j+1} \text{ terminal} \\ r_{j+1} + \gamma \cdot \max_{a'} Q_{\theta'}(s_{j+1}, a') & \text{else} \end{cases}$
12:             perform gradient descent step on loss $(y_j - Q_{\theta_i}(s_j, a_j))^2$
13:             soft-update the network weights $\theta' = (1 - \tau) \cdot \theta_i + \tau \cdot \theta'$
14:         end every
15:     end for
16:     if $e > W$ then
17:         every $L$ episodes do
18:             compute $E(J_i)$ for all $J_i \in \mathbb{P}$
19:         end every
20:     end if
21: end for
```

We will represent whether we use the GRP or expected return configuration of EID using the subscripts G and R, and we will omit the subscript whenever the configuration is not relevant.

Like Gros et al. [16], we apply DSMC ES to the Racetrack task by defining that every initial state $s_i \in \mathcal{I}$ forms an initial state region $J_i = \{s_i\}$. To assess how DSMC ES affects the training in Racetrack, we can count for each map cell how often a state with the corresponding coordinates was considered for a Q-network update and then visualize these counts as a heatmap, which we call a considered states heatmap.

Figure 4.3 shows an example in which DQN and EID$_G$ agents were trained on the Maze map with the random restart setting activated. Comparing the global GRP heatmaps, we can see that DQN attained deficient GRPs on the rightmost area of the map, whereas EID$_G$ achieved significantly higher GRPs. Looking at the considered states heatmaps, we see that EID$_G$'s heatmap shows more bright yellow cells in this area, which means EID$_G$ considered it more for training than DQN, leading to a higher GRP. EID$_G$ also achieved with 72% a noticeably higher average GRP in the initial states than DQN, which only achieved 62%.

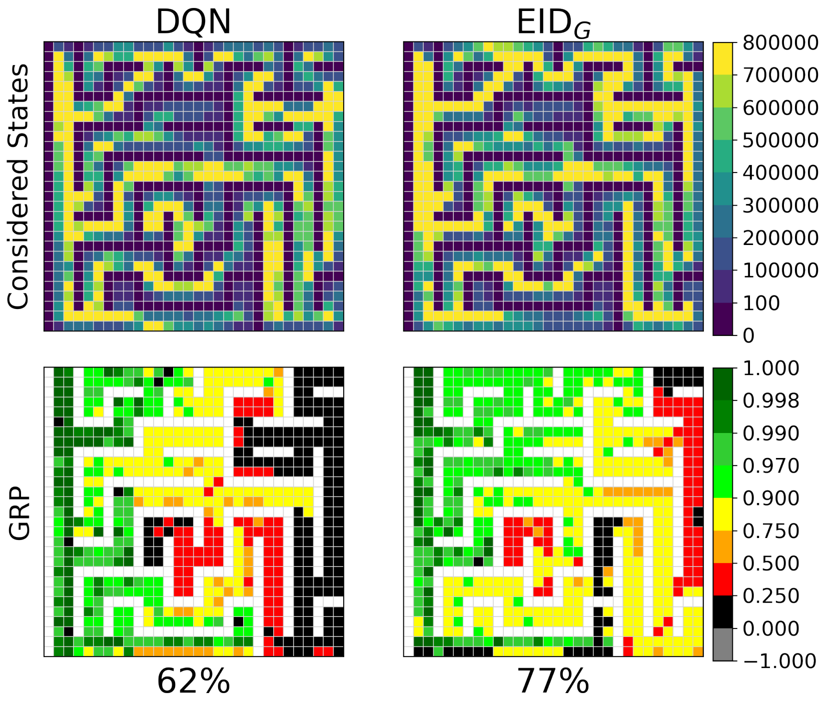


Figure 4.3 **Comparison of considered states and GRPs between DQN and EID$_G$ on the Maze map.**

4.3.2. Evaluation-based Prioritized Replay

The second DSMC ES algorithm is called *evaluation-based prioritized replay* (EPR). Contrary to EID, EPR leaves the initial state distribution μ unchanged, but during the Q-network update, it prioritizes samples from episodes starting in initial state regions J_i with a low evaluation value $E(J_i)$. This focuses the updates on using experiences from starting in regions J_i where $E(J_i)$ is low, such that the agent's performance may improve. So instead of generating more transitions from starting in problematic regions like EID, EPR uses the already gathered transitions more. EPR computes for every transition a sampling priority

$$\delta = (1 - E(J_i) + \epsilon_p)^{\alpha},$$

which is based on the evaluation value $E(J_i)$ of the current episode's initial state $s_0 \in J_i$. The hyperparameter ϵ_p ensures that transitions from starting in regions J_i where $E(J_i)$ is high are not neglected, and $\alpha \in (0, 1)$ controls the amount of prioritization.

Algorithm 6 corresponds to an EPR variant based on DQNPR. Unlike EID, in line 2, EID samples the initial states according to the initial state distribution μ, but it also periodically updates the evaluation values $E(J_i)$ from lines 16 to 20. What differentiates EPR from DQNPR is that, in line 7, it computes δ according to the results of the previous evaluation stage and not with respect to the TD-error. So when updating the Q-network, transitions from episodes starting in initial state regions J_i with a low evaluation value $E(J_i)$ have a high probability of being sampled. Another difference to DQNPR is that EPR does not use importance sampling weights w_i to compensate for the altered sampling probabilities. Since EPR works by changing how transitions are sampled from the replay buffer, it can only be used in conjunction with DRL algorithms that also use replay buffers.

Algorithm 6 Evaluation-based Prioritized Replay (DQNPR version)

```
 1: for episodes $e = 0$ to $E - 1$ do
 2:     sample $s_0 \in J_i$ w.r.t. $\mu$
 3:     for steps $t = 0$ to $T - 1$ do
 4:         with probability $\epsilon$ select random action $a_t \in \mathcal{A}(s_t)$
 5:         otherwise with probability $1 - \epsilon$ select $a_t = \underset{a \in \mathcal{A}(s_t)}{\text{argmax}}\, Q_{\theta_i}(s_t, a)$
 6:         execute $a_t$; observe $s_{t+1}$ and $r_{t+1}$
 7:         compute $\delta = (1 - E(J_i) + \epsilon_p)^\alpha$
 8:         store $(s_t, a_t, r_{t+1}, s_{t+1}, \delta)$ in replay buffer $D$
 9:         every $K$ steps do
10:             sample minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1}, \delta)$ from $D$ w.r.t. $\delta$
11:             set target $y_j = \begin{cases} r_{j+1} & s_{j+1} \text{ terminal} \\ r_{j+1} + \gamma \cdot \max_{a'} Q_{\theta'}(s_{j+1}, a') & \text{else} \end{cases}$
12:             perform gradient descent step on loss $(y_j - Q_{\theta_i}(s_j, a_j))^2$
13:             soft-update the network weights $\theta' = (1 - \tau) \cdot \theta_i + \tau \cdot \theta'$
14:         end every
15:     end for
16:     if $e > W$ then
17:         every $L$ episodes do
18:             compute $E(J_i)$ for all $J_i \in \mathbb{P}$
19:         end every
20:     end if
21: end for
```

As with EID, we will use subscripts to indicate whether we use EPR's GRP or return configuration whenever it is relevant.

In Figure 4.4, we can see an example that compares a DQNPR agent and an EPR$_R$ agent, which were trained on the Maze map with the random restart setting activated. DQNPR's global return heatmap shows that it only achieved a sufficient return in the upper left area of the map since it concentrated the training too intensely on this area, which can be seen in its considered states heatmap. On the other hand, EPR$_R$ focused less on this area, which balanced the training across the map, and resulted in a significantly higher average expected return in the initial states of 47 compared to

DQNPR, which only achieved an average return of 0.

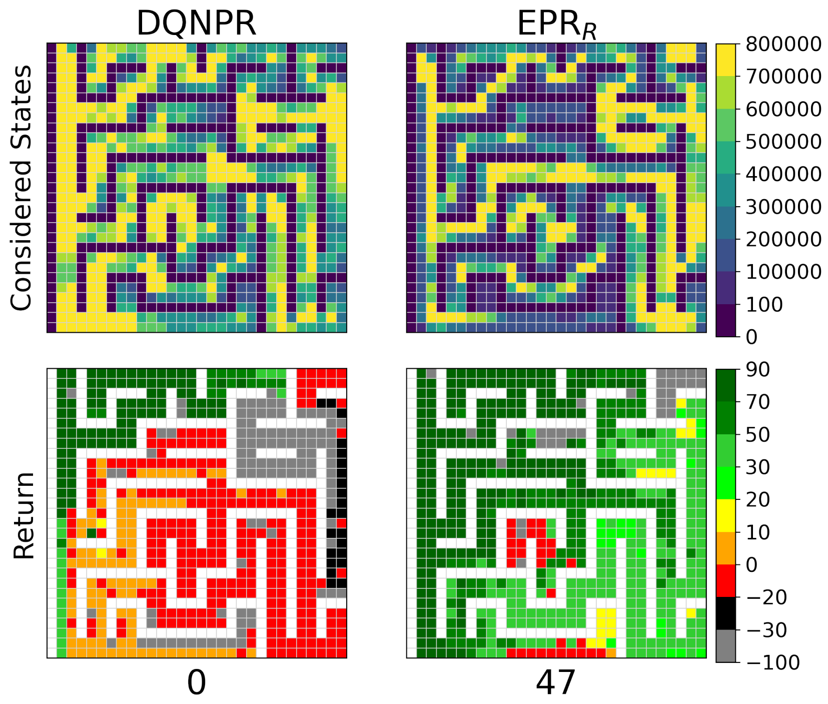


Figure 4.4 **Comparison of considered states and returns between DQNPR and EPR$_R$ on the Maze map.**

5. DSMC Evaluation Stages with Restart

The EID and EPR algorithms allow us to train safe DRL agents by focusing their training on initial state regions J_i where they have low evaluation values $E(J_i)$. However, both methods require that the regions J_i cover large parts of the state space to achieve noticeable increases in $E(J_i)$. This chapter introduces the concept of *pseudo initial states*, how we can use them to address EID and EPR's limitations, and how we can find them.

5.1. Limitations of DSMC Evaluation Stages

DSMC ES analyzes initial state regions J_i to identify deficiencies in the agent's policy and concentrate the training on the problematic regions such that the agent improves its behavior. However, this principle breaks down if the task's initial states only cover a small part of the state space. In such cases, the state space regions where deficient behaviors occur might be so distant from the initial state regions J_i, that focusing the training on regions J_i with low evaluation values $E(J_i)$ does not lead to sufficient training in the relevant state space regions. Since the MDPs of complex tasks tend to have gigantic state spaces, we cannot always expect that their initial states are distributed such that we can successfully apply DSMC ES.

We can simulate a lack of initial states by training a Racetrack agent on the River-deadend map using EID_G and restricting the initial states to the starting line, meaning we deactivate the random restart setting. After the training is completed, we find that the agent has only an 80% probability of reaching a goal line from the starting line. To assess in which parts of the map the agent has difficulty driving, we computed a global GRP and considered states heatmap, which can be seen in Figure 5.1. The heatmap at the top shows that the agent's GRP is only sufficient when starting on the leftmost area of the map, suggesting that its low GRP at the starting line is due to the inability to reach a goal line whenever it enters the map's right areas. We see in the heatmap at the bottom that the training disproportionally concentrated on the leftmost map area, despite the agent's deficiencies on the remaining parts. Thus, it is likely that if the training had been focused to some extent on the neglected map areas, the agent would have learned to drive in them, resulting in higher GRPs at the starting line.

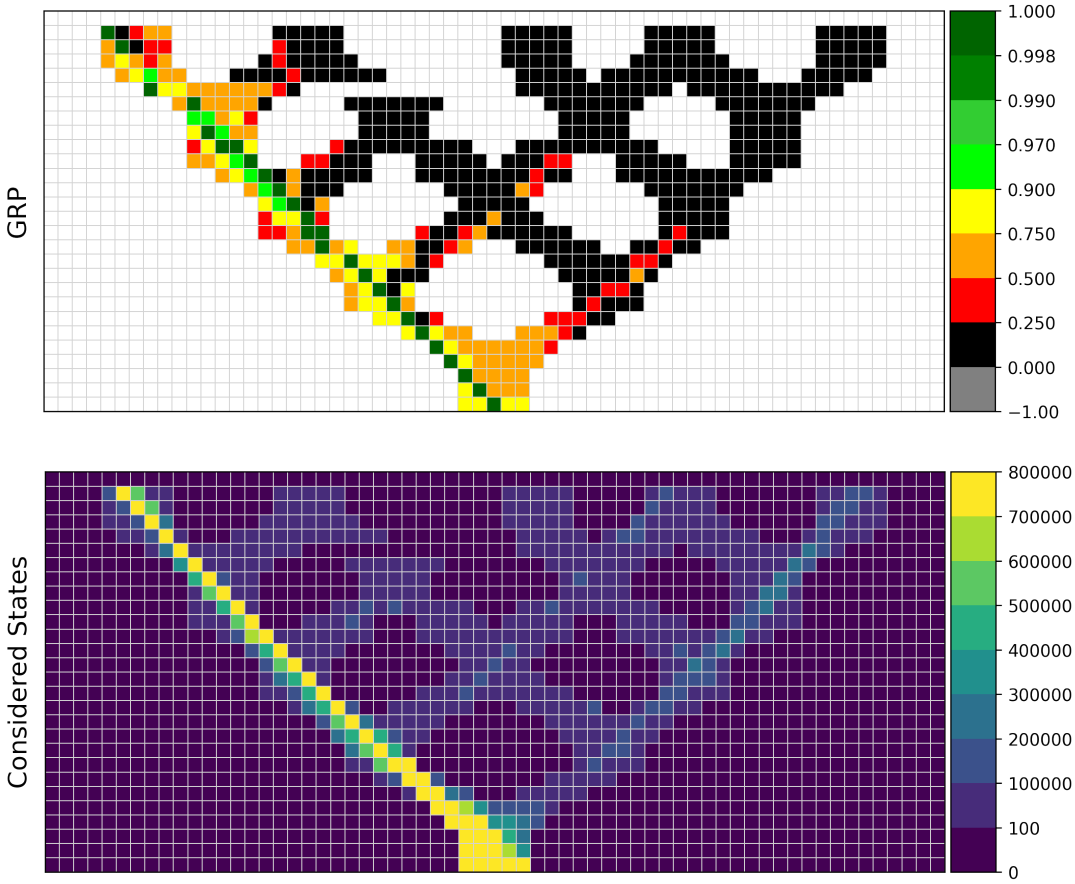


Figure 5.1 **Global GRP (top) and considered states heatmap (bottom) of the River-deadend map.**

5.2. Pseudo Initial States

This thesis proposes that we can remove DSMC ES' dependence on appropriately distributed initial state regions J_i by collecting visited states during training and using them to define additional *pseudo initial state regions* J_i^P. This means that between each evaluation stage, we collect a set of states P_{j+1} that will be given to the following evaluation stage. These states will be evaluated like the representatives s_i of the initial state regions J_i, so we call them *pseudo initial states* s_i^P, and each will form a pseudo initial state region $J_i^P = \{s_i^P\}$. From now on, we will refer to the initial state regions J_i as *original initial state regions* J_i^O, and to the initial states $s_i \in \mathcal{I}$ as *original initial states* s_i^O. If chosen correctly, the pseudo initial state regions J_i^P will enable the DSMC ES algorithms to focus the training on state space regions where deficient behaviors occur, such that it increases the agent's performance in the original initial states. Because we add to the DSMC ES framework the collection of pseudo initial states s_i^P in which subsequent episodes will restart, we call the resulting technique *DSMC evaluation stages* *with restart* (DSMC ESR), and the corresponding algorithms will be designated EID^P and EPR^P. The general approach of DSMC ESR can be divided into 3 steps:

1. The first step is to store certain visited states as candidates $c \in C$ and evaluate how qualified they are to become pseudo initial states. This qualification measure should reflect whether the resulting pseudo initial state regions J_i^P are suitable for focusing the training on them.

2. In the second step, we select from the set of candidates C the set of most qualified pseudo initial states $P_{j+1} = \left\{s_1^P, s_2^P, ..., s_B^P\right\}$, where $|P_{j+1}|$ is upper bounded by B to avoid an excessively high computational cost of the evaluation stage. Additionally, we enforce a minimum distance limit between the pseudo initial states s_i^P to ensure they cover a substantial portion of the state space. In general, this can be accomplished by defining a norm over the s_i^P and a corresponding distance measure.

3. The last step is to define for each $s_i^P \in P_{j+1}$ a pseudo initial state region J_i^P, and pass them and the original initial states regions J_i^O to the evaluation stage. Until the next evaluation stage, the representatives s_i^P of the pseudo initial state regions J_i^P will be used as additional initial states. The collection process repeats for the following evaluation stage, meaning we compute a new set P_{j+2}. This ensures that the most qualified pseudo initial states will be used in each iteration. The only exception is that $|P_{j+2}|$ may not exhaust the upper bound B. In this case, the most qualified pseudo initial states from the previous sets $P_{j+1}, P_j, ..., P_0$ are added to P_{j+2} until $|P_{j+2}| = B$ holds.

In Figure 5.2 we can see an example of a DSMC ESR agent collecting candidates (cells with crosses) and selecting pseudo initial states (colored cells) in the Racetrack task. The maximum number of pseudo initial states is $B = 5$, and the distance limit is that no pseudo initial states can be located on the same cells. Our qualification measure is the distance to the goal line. Thus, in each iteration, we need to select the 5 candidates that are closest to the goal line and located on different cells as pseudo initial states. Before the first evaluation stage, the agent can only begin episodes on the starting line. In the image at the top, we can see red crosses, which mark the location of the visited states that the agent stored as candidates. Now the first evaluation stage is due, so we add the 5 most qualified candidates to our first set of pseudo initial states P_1, which are shown as red cells in the center image. In the next iteration, episodes can start on the starting line or in states $s_i^P \in P_1$. However, the agent's policy suffered destructive updates, so the agent kept crashing into walls. Therefore, it could only collect 3 new candidates, which are represented by blue crosses. In this case, we add all candidates to the new set of pseudo initial states P_2, and, additionally, add to P_2 the states from P_1 that were closest to the goal line. This allows us to provide the agent with a diverse set of pseudo initial states P_2, despite the agent's inability to collect promising new candidates in the previous iteration. In the last picture, restarting in states $s_i^P \in P_2$, shown as blue and red cells, enabled the agent to recover its performance and collect highly qualified new candidates, which are depicted as orange crosses.

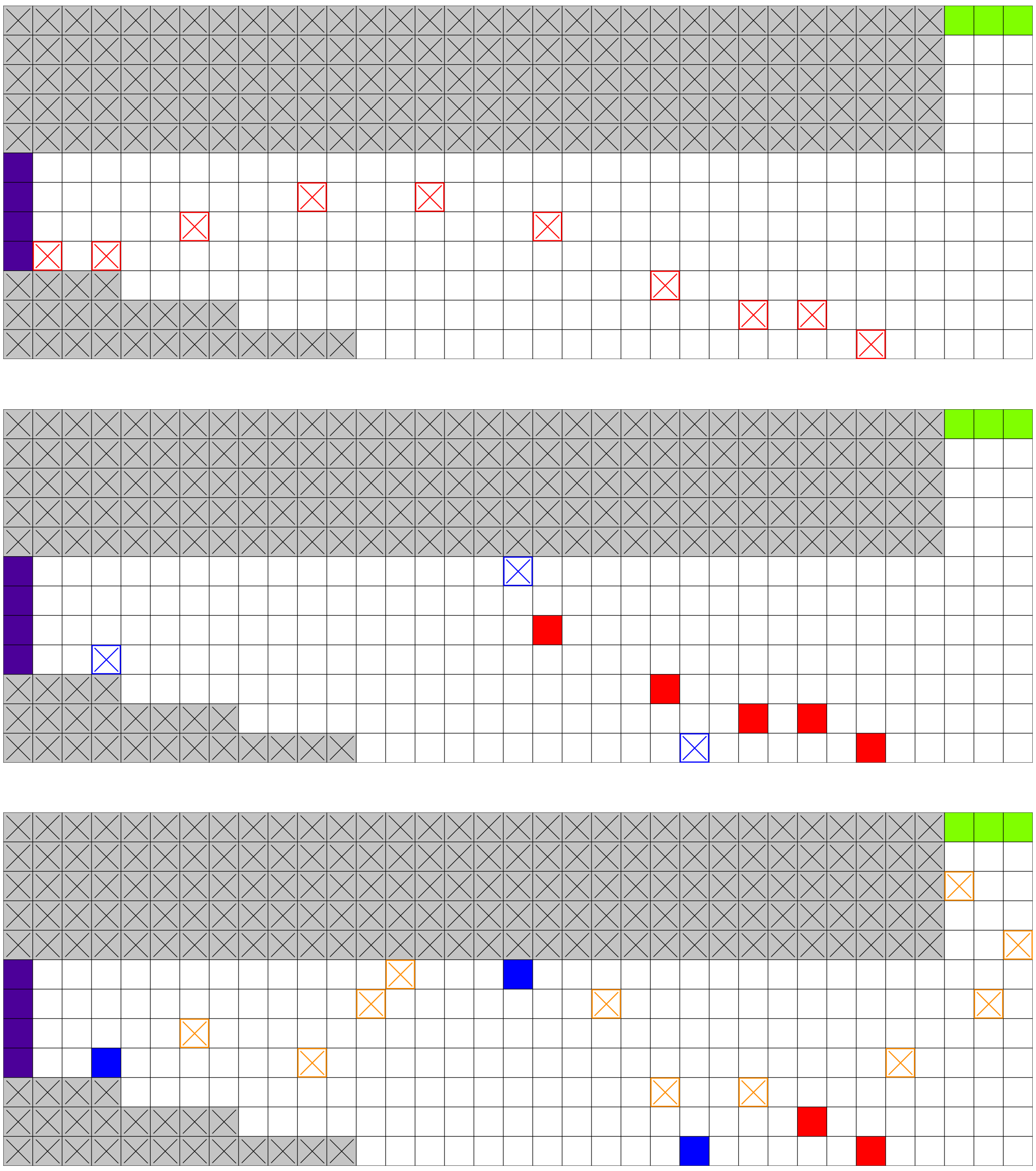


Figure 5.2 **Selection of pseudo initial states on the Barto-small map.** The agent collects the first candidates (top), collects the next candidates after selecting the first set of pseudo initial states P_1 (middle), and then collects further candidates after selecting the second set of pseudo initial states P_2 by reusing states from P_1 (bottom).

Early experiments showed that the disadvantage of introducing pseudo initial state regions J_i^P is that if we concentrate the training too heavily on them, the agent may forget how to act when starting in the original initial state regions J_i^O, meaning it cannot solve the original task. Thus, we need to ensure that we only utilize the regions J_i^P to the extent that they increase the agent's evaluation values $E(J_i^O)$ in the regions J_i^O. We achieve this by making the utilization of the pseudo initial state regions J_i^P proportional to the average evaluation value in the original initial state regions

$$\overline{E(J_i^O)} = \frac{\sum E(J_i^O)}{|\mathbb{P}|},$$

where $\mathbb{P}$ is the set of all original initial state regions J_i^O. So when $\overline{E(J_i^O)}$ decreases, we concentrate the training less on pseudo initial state regions J_i^P, allowing the agent to recover its performance in the original initial state regions J_i^O. If $\overline{E(J_i^O)}$ is very close to 0, the agent struggles to reach a goal state from the regions J_i^O, which might be due to insufficient exploration of the state space. In this case, the agent could benefit from starting episodes in pseudo initial state regions J_i^P since they might be closer to goal states. If $\overline{E(J_i^O)}$ is very close to 1, the agent has a high probability of reaching goal states from regions J_i^O, so we allow it to start in pseudo initial state regions J_i^P frequently. However, if the agent now starts too often in regions J_i^P, it will forget again how to act in the original initial state regions J_i^O, leading to unstable performance. To ensure exploration and avoid instability, we compute the *pseudo initial state region priority* as

$$\psi = clip\left(\overline{E(J_i^O)}, 1 - \psi_{max}, 0 + \psi_{min}\right),$$

by clipping $\overline{E(J_i^O)}$ to an interval between 0 and 1 using the *lower and upper pseudo initial state region priority bounds* ψ_{min} and ψ_{max}.

In Algorithm 7, we can see in the highlighted lines how to incorporate pseudo initial states in EID^P and EPR^P. Similar to the EID and EPR algorithms introduced in chapter 4, this algorithm is based on the DQN algorithm and only modifies how initial states are sampled and how transitions are sampled for the Q-network update. The main difference between this algorithm and the previously introduced EID and EPR algorithms is that we need to differentiate between episodes starting in original initial state regions J_i^O and episodes starting in pseudo initial state regions J_i^P. Furthermore, we store certain visited states and select pseudo initial states before computing an evaluation stage. In line 4, we control how many episodes start in pseudo initial state regions J_i^P by using initial state distributions that account for the pseudo initial state region priority ψ. In the case of EID^P, we sample the episode's initial state s_0 uniformly from J_i, which is sampled according to

$$p'(J_i) = \begin{cases} (1-\psi) \cdot p(J_i) & \text{if } J_i = J_i^O \\ \psi \cdot p(J_i) & \text{if } J_i = J_i^P \end{cases}.$$

This means we compute the sampling probability $p(J_i)$ of each region J_i according to the evaluation value $E(J_i)$, as in the EID algorithm, and then scale each $p(J_i)$ with regard to ψ to get the final sampling probabilities $p'(J_i)$. Therefore, the higher ψ, the more we increase the probability of episodes starting in pseudo initial state state regions J_i^P. In EPR^P, we sample with equal probability either an original initial state $s_i^O \in \mathcal{I}$ from μ or a pseudo initial state $s_i^P \in P_j$ uniformly from the current set of pseudo initial states P_j, giving us

$$\mu'(s_0) = \begin{cases} \frac{1}{2} \cdot \mu(s_0) & \text{if } s_0 \in \mathcal{I} \\ \frac{1}{2} \cdot \frac{1}{|P_j|} & \text{if } s_0 \in P_j \end{cases}.$$

The next modification is that every time the agent gathers a new transition, in line 9, we check whether the current state s_t should be stored as a candidate and, if so, add it to the set of candidates C. In the case of EPR^P, in line 10, we compute the current transition's sampling priority as

$$\delta' = \begin{cases} (1-\psi) \cdot (1 - E(J_i^O) + \epsilon_p)^\alpha & \text{if } s_0 \in J_i^O \\ \psi \cdot (1 - E(J_i^P) + \epsilon_p)^\alpha & \text{if } s_0 \in J_i^P \end{cases}.$$

Thus, we compute the sampling priorities, as in the EPR algorithm, and then scale them by ψ if the corresponding episodes started in pseudo initial state state regions J_i^P and by $1 - \psi$ otherwise. Therefore, the higher ψ, the more we increase the sampling probabilities of transitions gathered in episodes that started in pseudo initial state regions J_i^P. From lines 20 to 23, we update the original initial state regions' evaluation values $E(J_i^O)$ and ψ every L episodes. From lines 24 to 31, we select every U episodes the new pseudo initial states P_{j+1}, and if they do not exhaust the size limit B, we add pseudo initial states from $P_j, P_{j-1}, ..., P_0$ to P_{j+1}, until we have $|P_{j+1}| = B$. Afterward, we evaluate all pseudo initial state regions J_i^P and clear the set of candidates C to avoid excessive memory usage. Since we assume that the original initial state regions J_i^O only cover a small part of the state space, they may also be small in number. We can exploit this by updating the evaluation values $E(J_i^O)$ separately and at a higher frequency than $E(J_i^P)$, meaning we can update $E(J_i^O)$ every $L << U$ episodes without adding too much computational overhead. This allows us to update ψ frequently, so we can accurately balance the utilization of pseudo initial state regions.

Algorithm 7 EID^P/EPR^P (DQN version)

```
1: initialize set of candidates $C$
2: initialize pseudo initial state region priority $\psi = 0.0$
3: for episodes $e = 0$ to $E - 1$ do
4:     sample $s_0$ according to $\begin{cases} p'(J_i) & \text{EID}^P \\ \mu'(s_0) & \text{EPR}^P \end{cases}$
5:     for steps $t = 0$ to $T - 1$ do
6:         with probability $\epsilon$ select random action $a_t \in \mathcal{A}(s_t)$
7:         otherwise with probability $1 - \epsilon$ select $a_t = \underset{a \in \mathcal{A}(s_t)}{\text{argmax}}\, Q_{\theta_i}(s_t, a)$
8:         execute $a_t$; observe $s_{t+1}$ and $r_{t+1}$
9:         SELECTCANDIDATE$(s_t, s_{t+1})$
10:        compute $\delta = \begin{cases} \text{constant} & \text{EID}^P \\ \delta' & \text{EPR}^P \end{cases}$
11:        store $(s_t, a_t, r_{t+1}, s_{t+1}, \delta)$ in replay buffer $D$
12:        every $K$ steps do
13:            sample minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1}, \delta)$ from $D$ w.r.t. $\delta$
14:            set target $y_j = \begin{cases} r_{j+1} & s_{j+1} \text{ terminal} \\ r_{j+1} + \gamma \cdot \max_{a'} Q_{\theta'}(s_{j+1}, a') & \text{else} \end{cases}$
15:            perform gradient descent step on loss $(y_j - Q_\theta(s_j, a_j))^2$
16:            soft-update the network weights $\theta' = (1 - \tau) \cdot \theta_i + \tau \cdot \theta'$
17:        end every
18:    end for
19:    if $e > W$ then
20:        every $L$ episodes do
21:            compute $E(J_i^O)$ for all $J_i^O \in \mathbb{P}$
22:            update pseudo initial state region priority $\psi$
23:        end every
24:        every $U$ episodes do
25:            $P_{j+1} =$ SELECTPSEUDOS$(C)$
26:            if $|P_{j+1}| < B$ then
27:                add the $B - |P_{j+1}|$ most qualified $s_i^P \in P_j, P_{j-1}, ..., P_0$ to $P_{j+1}$
28:            end if
29:            compute $E(J_i^P)$ for all $J_i^P = \{s_i^P\}$ where $s_i^P \in P_{j+1}$
30:            clear $C$
31:        end every
32:    end if
33: end for
```

5.3. Pseudo Initial State Selection Strategies

So far, we have established how to utilize pseudo initial states $s_i^P \in P_j$ in the EID^P and EPR^P algorithms, but we have not yet considered which visited states should be stored as candidates $c \in C$ and how P_j should be chosen from C. Since we can only evaluate small parts of enormous state spaces, we want to ensure that P_j allows DSMC ESR to focus the training on the regions that enable the largest improvements in our safety metrics, i.e., the average GRP or the average expected return and return variance. This thesis introduces four *pseudo initial state selection strategies*, which select P_j according to different properties that indicate the potential for significant improvements in safety. To evaluate these properties, we require efficiently computable qualification measures since we will collect a large number of candidates. Furthermore, we will only evaluate the candidates' qualifications upon visiting them since recomputing the qualifications of all candidates before each evaluation stage may result in high computational costs. Thus, we assume that if a candidate was highly qualified at its visitation, it would remain sufficiently qualified until we select it as pseudo initial state.

5.3.1. Random Strategy

From lines 2 to 4, the random strategy's (Algorithm 8) selectCandidate method uniformly selects visited states as candidates if they are no original initial states. From lines 8 to 13, the selectPseudos method then uniformly selects B candidates as pseudo initial states. To ensure that they cover large parts of the state space, in line 10, the method checks for each candidate that it adds to P_{j+1} whether it has a sufficient distance to the already selected pseudo initial states $s_i^P \in P_{j+1}$. Since every visited state has the same probability of being selected, P_{j+1} will cover the visited parts of the state space evenly, allowing us to focus the training on a large variety of regions.

Algorithm 8 Random Strategy

```
1: function selectCandidate($s_t, s_{t+1}$)
2:     if $s_t \notin \mathcal{I}$ then
3:         with probability 50% add $s_t$ to $C$
4:     end if
5: end function
6:
7: function selectPseudos($C$)
8:     $P_{j+1} = \{\}$
9:     for candidate $c$ in $C$ do
10:        if distanceLimit($c, P_{j+1}$) and $|P_{j+1}| < B$ then
11:            with probability 50% add $c$ to $P_{j+1}$
12:        end if
13:    end for
14:    return $P_{j+1}$
15: end function
```

5.3.2. Crash Strategy

Instead of providing the evaluation stage with pseudo initial state regions J_i^P that represent large parts of the state space, we can only provide it with regions J_i^P where the agent is prone to take actions that lead to safety errors. Therefore, we force the agent to address its most deficient behaviors. Whenever the agent makes a safety error, the episode finishes, and the crash strategy's (Algorithm 9) selectCandidate method stores the state in which the agent was n time steps ago as a candidate from lines 2 to 7. If the episode was shorter than n time steps, the initial state s_0 is added to C, given that $s_0 \notin \mathcal{I}$. This way, pseudo initial states $s_i^P \in P_j$ will be considered as candidates for P_{j+1} if the agent kept making safety errors. Given that we set the hyperparameter n suitably, this method will select the states where the agent takes the actions leading to safety errors. We can find a suitable value of n by using domain knowledge or by starting at $n = 1$ and training agents with increasing values of n until we achieve adequate performance. The crash strategy's selectPseudos method selects as pseudo initial states the B candidates for which the most safety errors occurred after visiting them. However, we cannot simply count how often each candidate was encountered n steps before a safety error since the same state is rarely visited twice in a large state space. Therefore, from lines 11 to 15, we define groups of similar states and assign each candidate to its respective group. In general, we could achieve this by using clustering algorithms [35], but in Racetrack, we assign all candidates with the same x and y coordinates to the same group. From lines 16 to 22, selectPseudos then builds the set of pseudo initial states P_{j+1} by choosing a random candidate from each of the B groups containing the most candidates.

Algorithm 9 Crash Strategy

```
 1: function selectCandidate($s_t, s_{t+1}$)
 2:     if safetyError($s_{t+1}$) then
 3:         $n = \min(n, t+1)$
 4:         if $s_{t+1-n} \notin \mathcal{I}$ then
 5:             add $s_{t+1-n}$ to $C$
 6:         end if
 7:     end if
 8: end function
 9:
10: function selectPseudos($C$)
11:     initialize list of groups $G = [G_1, G_2, ..., G_k]$
12:     $P_{j+1} = \{\}$
13:     for candidate $c$ in $C$ do
14:         add $c$ to corresponding group $G_i$
15:     end for
16:     sort $G$ according to $|G_i|$ in descending order
17:     for group $G_i$ in $G$ do
18:         sample random candidate $c \sim G_i$
19:         if distanceLimit($c, P_{j+1}$) and $|P_{j+1}| < B$ then
20:             add $c$ to $P_{j+1}$
21:         end if
22:     end for
23:     return $P_{j+1}$
24: end function
```

5.3.3. Value Strategy

When the agent makes a safety error, it means that at some point, it deviated from its path to a goal state and transitioned into state space regions where it could not reach a goal state. If we use the states from which the agent enters these regions as pseudo initial states, we can focus the training on them such that the agent may learn to stay on the goal path. From lines 2 to 4, the value strategy's (Algorithm 10) selectCandidate method stores every visited state s_t as a candidate, along with the approximated drop in state-value

$$drop(s_t, s_{t+1}) = V(s_t) - V(s_{t+1}),$$

that occurred when the agent transitioned to the successor state s_{t+1}. From lines 8 to 14, the selectPseudos method then selects the B candidates with the highest *drop* as pseudo initial states since a significant drop in state-value indicates that the agent became unlikely to reach a goal state after transitioning to s_{t+1}.

Algorithm 10 Value Strategy

```
1: function selectCandidate($s_t, s_{t+1}$)
2:     if $s_t \notin \mathcal{I}$ then
3:         add ($s_t$, $drop(s_t, s_{t+1})$) to $C$
4:     end if
5: end function
6:
7: function selectPseudos($C$)
8:     sort $C$ according to $drop$ in descending order
9:     $P_{j+1} = \{\}$
10:    for candidate $c$ in $C$ do
11:        if distanceLimit($c, P_{j+1}$) and $|P_{j+1}| < B$ then
12:            add $c$ to $P_{j+1}$
13:        end if
14:    end for
15:    return $P_{j+1}$
16: end function
```

5.3.4. Novelty Strategy

We already established that agents perform poorly when they do not sufficiently explore the task's state space. If we use pseudo initial state regions J_i^P that are novel to the agent, we will focus the training on unexplored state space regions and increase exploration. From lines 2 to 4, the novelty strategy (Algorithm 11) uses its selectCandidate method to save all visited states as candidates along with their novelty. From lines 8 to 14, it uses its selectPseudos method to select the B most novel candidates as pseudo initial states. We measure novelty using *random network distillation* (RND) [7], which utilizes two randomly initialized DNNs with identical architectures that take as input the visited states and return a random output. One of these networks is designated as the target network, meaning its parameters will remain fixed, whereas the other is the predictor network. The goal of the latter is to predict the random outputs of the target network. During training, the current state will be passed to the predictor and target network at each time step. RND then computes the error in the predictor network's prediction and updates its parameters accordingly. If the agent visits similar states frequently, the predictor network's loss will decrease since it will learn the target network's outputs for the corresponding type of state. This is possible since DNNs generalize over their input data, meaning they will return similar outputs for similar inputs. Thus, the predictor network's loss encapsulates the agent's lack of experience in a given state, corresponding to the novelty. Note that, instead of RND, we could also utilize other novelty measures [1, 24].

Algorithm 11 Novelty Strategy

```
1: function selectCandidate($s_t, s_{t+1}$)
2:     if $s_t \notin \mathcal{I}$ then
3:         add ($s_t$, $novelty(s_t)$) to $C$
4:     end if
5: end function
6:
7: function selectPseudos($C$)
8:     sort $C$ according to novelty in descending order
9:     $P_{j+1} = \{\}$
10:    for candidate $c$ in $C$ do
11:        if distanceLimit($c, P_{j+1}$) and $|P_{j+1}| < B$ then
12:            add $c$ to $P_{j+1}$
13:        end if
14:    end for
15:    return $P_{j+1}$
16: end function
```

5.4. Application to Racetrack

Selecting a suitable distance limit is crucial for successfully applying EID^P and EPR^P. If the limit is too small, we may collect pseudo initial states that do not cover large parts of the state space since states from the same state space regions likely have similar qualifications, meaning that the most qualified candidates will be concentrated in the same regions. If the distance limit is too large, we may inadvertently reject the most qualified candidates during the selection process. Thus, choosing a distance limit corresponds to trading off the qualification of pseudo initial states and state space coverage. To find an appropriate distance limit, we can start with an initial value and train EID^P/EPR^P agents with increasing limits until we achieve satisfactory performance. However, a more efficient approach is to construct distance limits using domain knowledge, which is how we will define four limits to be used in the Racetrack task. The first distance limit is called the *single cell* limit, and it requires that no pseudo initial states may have the same x and y coordinates, meaning that for every map cell, we only consider the most qualified candidate as pseudo initial state. This way, we can avoid that pseudo initial states accumulate on single cells. We can achieve a larger distance limit by grouping single map cells into regions and only considering the most qualified candidates of each region as pseudo initial states. To do so, we lay a grid over the map so that it partitions it into square-shaped regions, among which we choose the best candidates. However, if we can only select less than B pseudo initial states $s_i^P \in P_{j+1}$ from the regions, we will identify the remaining most qualified candidates using the single cell distance limit and add them to P_{j+1} until we have added all candidates or $|P_{j+1}| = B$ holds. In our experiments, we will group the cells into 2 by

2 and 3 by 3 regions, an example of which is shown in Figure 5.3. We will call these two distance limits 2×2 *regions* and 3×3 *regions*.

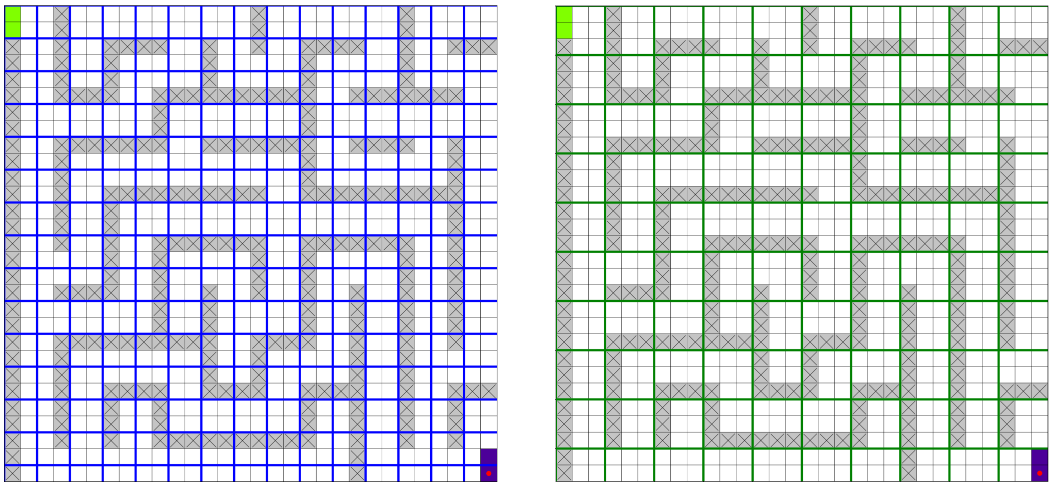


Figure 5.3 **2×2 (left) and 3×3 (right) regions on the Maze map.**

Our last distance limit is called the *max distance* limit and it aims to select pseudo initial states $s_i^P \in P_{j+1}$ that cover as much of the state space as possible. Therefore, we require that every candidate c we add to P_{j+1} must fulfill

$$c = \operatorname*{argmax}_{c \in C} \min_{s_i^P \in P_{j+1}} \left\| c, s_i^P \right\|_2 ,$$

meaning that c has the largest minimum euclidean distance to all of the already selected pseudo initial states $s_i^P \in P_{j+1}$. Note that we take the norm over the entire states, meaning that, unlike the previous distance limits, this limit considers the states' velocities in addition to their coordinates.

6. Results

This chapter will investigate the results of using DSMC ESR to train Racetrack agents. We examine EID$_G^P$/EPR$_G^P$ and EID$_R^P$/EPR$_R^P$ using DQN and DQNPR as their basis, respectively. Furthermore, we will discuss experiments using EID$_G^P$ with PPO as its basis. The experiments were conducted on the River-deadend, the Maze, and the Hansen-bigger maps, which were introduced in section 2.4. A noise probability of 50% was used on River-deadend, and a lower noise probability of 25% was used on the more challenging Maze and Hansen-bigger maps. In all experiments, the random restart setting was deactivated such that the initial states $\mathcal{I}$ were located only on the starting line, allowing us to show that the DSMC ESR algorithms work in tasks where $\mathcal{I}$ only covers a small portion of the state space.

The goal of the EID$_G^P$/EPR$_G^P$ experiments was to train agents that drive from the starting line to a goal line without crashing. This corresponds to having a high GRP in all initial states $\mathcal{I}$, so our primary performance metric will be the average GRP in $\mathcal{I}$, which we call the *initial GRP*. It is also interesting to assess the agent's GRP when starting in states that are not in $\mathcal{I}$ since DSMC ESR restarts episodes in various state space regions. Thus, we will use the average GRP across the whole state space as a secondary performance metric, which we approximate by taking the average over the agent's GRP when starting on each cell of the map. We will refer to this metric as the *global GRP*.

The EID$_R^P$/EPR$_R^P$ experiments aimed to train agents that achieve high average returns when driving from the starting line while only earning a minimum number of large negative returns, which are due to crashing. Thus, we will use the average expected return and average return variance in $\mathcal{I}$ as our primary performance metrics, which we call the *initial return* and *initial variance*. Similarly to the EID$_G^P$/EPR$_G^P$ experiments, we will use the agent's average expected return when starting on each map cell, called the *global return*, as our secondary performance metric.

Similar to Gros et al. [16], we saved the agents' policies during training at regular intervals and then, for each agent type, evaluated all stored policies with regard to the primary performance metrics.

6.1. EID$_G^P$ and EPR$_G^P$ Results

The first experiments incorporated training EID$_G^P$ and EPR$_G^P$ agents. Additionally, DQN and DQNPR agents were trained as performance baselines. The EID$_G^P$/EPR$_G^P$ agents were trained using the four pseudo initial state selection strategies and four distance limits, totaling 34 agent types, including the baselines. For each agent type, the training was run on each map for $E = 100.000$ episodes and repeated five times using different random seeds, resulting in a total of 510 agents. The EID$_G^P$/EPR$_G^P$ agents selected and evaluated the pseudo initial states every $U = 10.000$ episodes and evaluated the original initial states every $L = 100$ episodes. The maximum number of pseudo initial states in each iteration was chosen as

$$B = \frac{|\text{free map cells}|}{4},$$

and the lower and upper pseudo initial state region priority bounds were both set to $\psi_{min} = \psi_{max} = 0.2$. The DSMC ESR agents that utilized the crash strategy stored the state in which they were $n = 5$ steps ago as a candidate whenever they crashed. This hyperparameter was chosen by training agents with increasing values of n until we achieved adequate performance. The architecture of the Q-network was the same as depicted in Figure 3.4. A full list of hyperparameter choices can be found in the appendix A.1. The experiments were run on an Apple M1 Pro CPU with 10 cores and took approximately 224 hours (EID$_G^P$: 70 hours, EPR$_G^P$: 154 hours) to complete.

Figure 6.1 and Figure 6.2 show the initial GRPs of the best-performing baseline agents and best-performing EID$_G^P$/EPR$_G^P$ agents for each pseudo initial state selection strategy. A table containing the configurations of all best-performing agents can be found in the appendix A.2. We can see that EID$_G^P$ outperforms the DQN baselines in 9 out of 12 instances, and EPR$_G^P$ outperforms the DQNPR baselines in 10 out of 12 instances. Furthermore, EPR$_G^P$ attained equal or higher initial GRPs than EID$_G^P$ in all cases except for the value strategy agent on the Maze map. For both EID$_G^P$ and EPR$_G^P$, we can see that the improvements over the baselines are relatively small on the River-deadend map, which is likely due to this map being unchallenging, making it difficult to outperform baselines that already achieve very high initial GRPs. However, it is remarkable that in both cases, the random strategy agents achieved an initial GRP of 1.0, meaning that they learned perfectly safe driving behavior. On the more challenging Maze map, we see that EID$_G^P$ could only outperform DQN by 3% using the value or novelty strategy, whereas all EPR$_G^P$ agents achieved higher initial GRPs than DQNPR, with improvements of up to 15%. The fact that the EPR$_G^P$ agent that used the novelty strategy achieved the highest initial GRP suggests that focusing on unexplored state space regions is beneficial on the Maze map. On Hansen-bigger, it is striking that neither DQN nor DQNPR learned to reach the goal line, yet all EPR$_G^P$/EID$_G^P$ agents attained high initial GRPs.

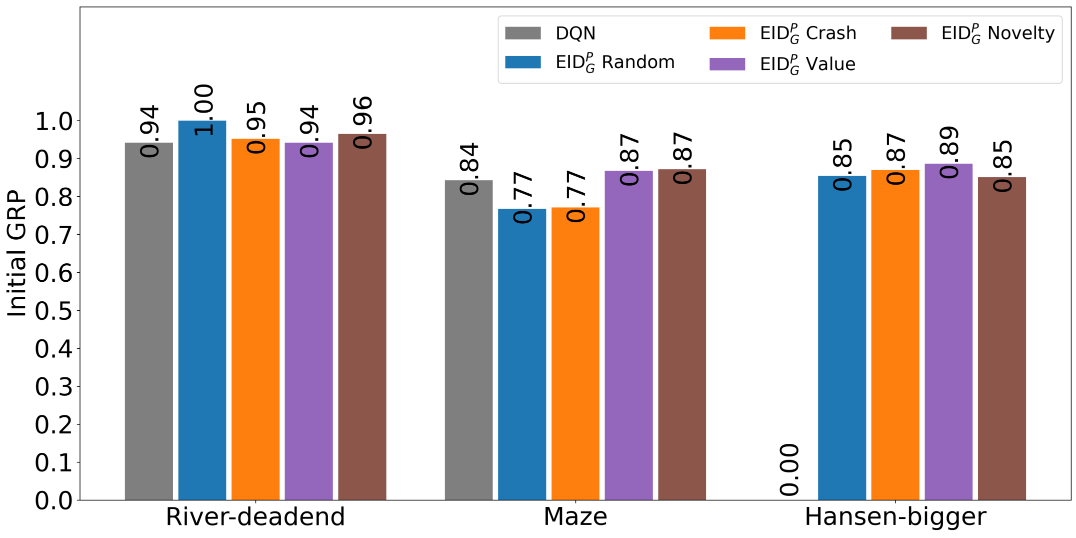


Figure 6.1 **Initial GRPs of the best-performing EID$_G^P$ and DQN agents.**

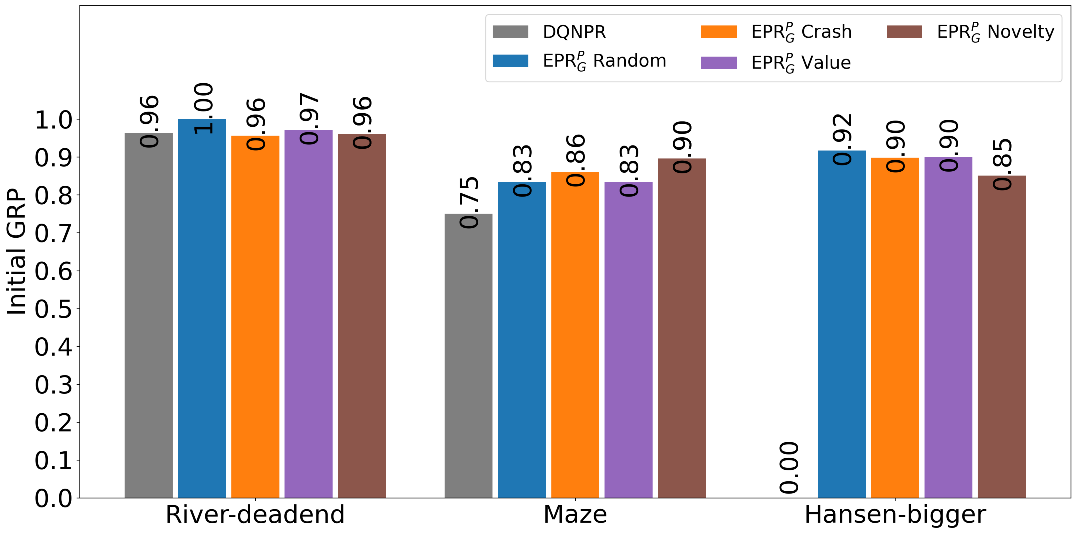


Figure 6.2 **Initial GRPs of the best-performing EPR$_G^P$ and DQNPR agents.**

We can illustrate which states the selection strategies selected as pseudo initial states by counting for each cell how often a pseudo initial state was located on it and then visualizing these counts as heatmaps, which we call pseudo heatmaps. We will also annotate each heatmap with the distance limit that the corresponding agent utilized. To gain further insights into how the pseudo initial state selection strategies influenced the training, we will compare the pseudo, considered states, and global GRP heatmaps of some of the best-performing DSMC ESR and baseline agents on all three maps.

Figures 6.3, 6.4, 6.5 show the pseudo, considered states, and global GRP heatmaps of the EID$_G^P$ and baseline agents, trained on the River-deadend map. On the River-deadend map, the random, crash, and novelty strategy agents outperformed DQN in terms of initial GRP. If we compare their pseudo initial states heatmaps to those of the value strategy agent, we can see that they selected significantly more states from the map's center areas. Subsequently, we observe in the considered states heatmaps of the random, crash, and novelty strategy agents that they considered these areas more intensely for training than the value strategy, DQN, and DQNPR agents. Therefore, we can see in the global GRP heatmaps that these agents attained higher GRPs in the central map areas, leading to higher global GRPs. The random, crash, and novelty agents likely achieved higher initial GRPs than the DQN agent because they could reach the goal line from more of the map's central areas. However, reaching a goal from the starting line only requires driving in a straight line, making it unlikely for agents to enter the central map areas, which could explain why the agents only achieved small improvements.

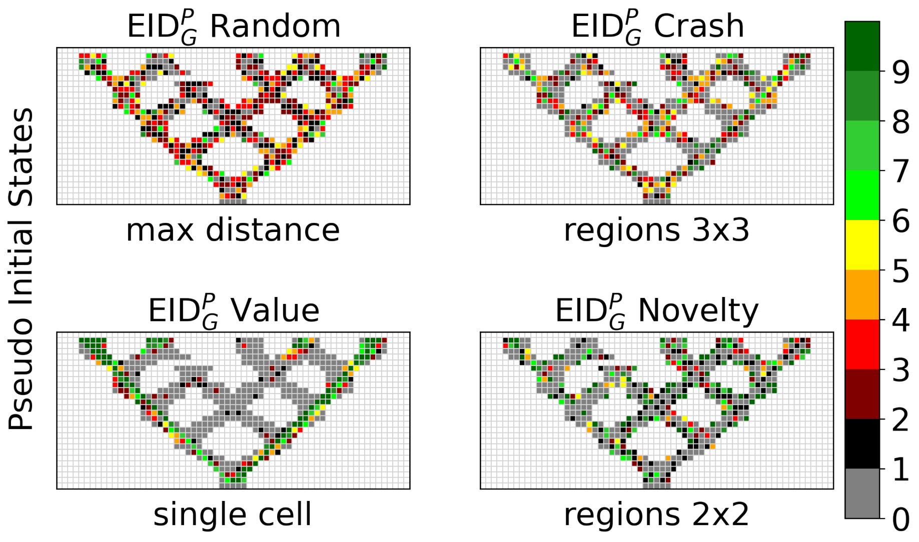


Figure 6.3 **Pseudo heatmaps of the best-performing EID_G^P agents on Riverdeadend.**

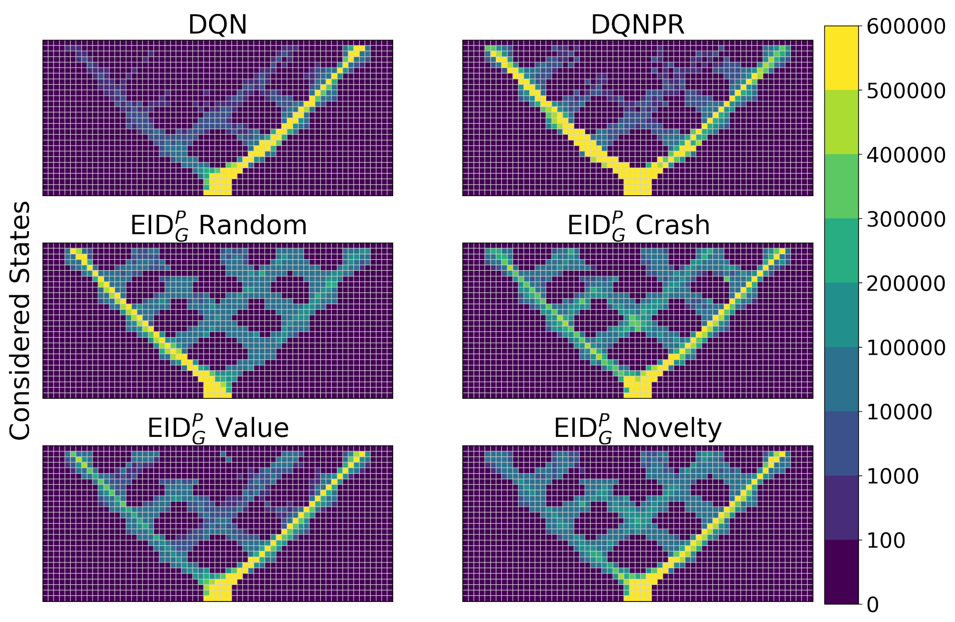


Figure 6.4 **Considered states heatmaps of the best-performing EID$_G^P$, DQNPR, and DQN agents on River-deadend.**

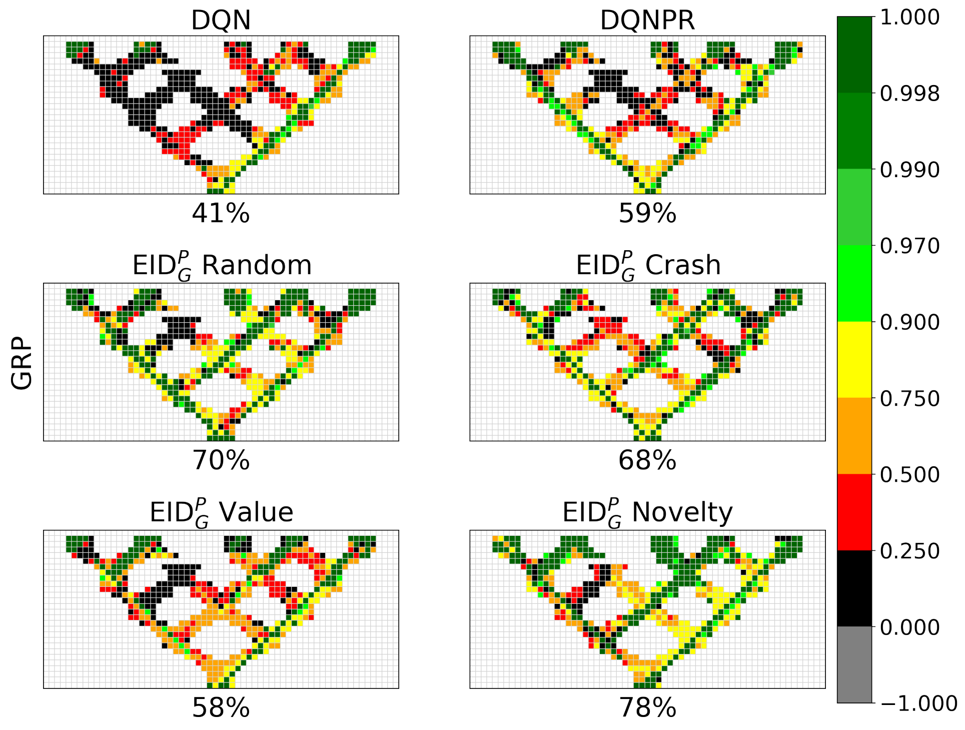


Figure 6.5 **Global GRP heatmaps of the best-performing EID$_G^P$, DQNPR, and DQN agents on River-deadend.**

Figures 6.6, 6.7, 6.8 show the pseudo, considered states, and global GRP heatmaps of the EPR_G^P and baseline agents, trained on the Maze map. The pseudo heatmaps show that the random, value, and novelty strategy agents evenly distributed the pseudo initial states across the whole Maze map, whereas the crash strategy agent neglected some of the upper and lower map areas. We observe in the considered states heatmaps that the EPR_G^P agents considered states on the direct path through the Maze map less than DQN and DQNPR, but focused noticeably more on the remaining map areas. The global GRP heatmaps show that the crash, value, and novelty strategy agents achieved drastically higher GRPs than the baseline agents in the map areas beyond the direct goal path, whereas the random strategy agent only attained higher GRPs in some areas. This suggests that the EPR_G^P agents that achieved higher initial GRPs than the DQN and DQNPR agents did so because they could reach the goal line from various map areas, whereas the DQN and DQNPR agents were unlikely to reach the goal once they deviated from the direct path to the goal.

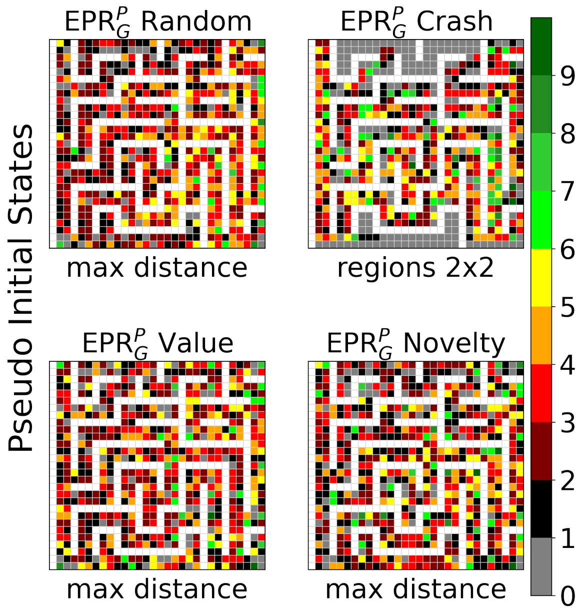


Figure 6.6 **Pseudo heatmaps of the best-performing EPR_G^P agents on Maze.**

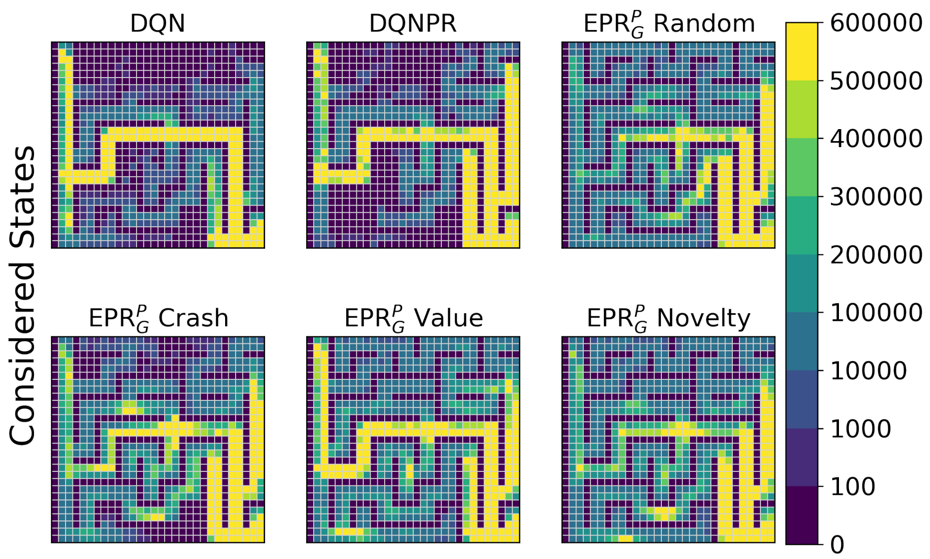


Figure 6.7 **Considered states heatmaps of the best-performing EPR$_G^P$, DQNPR, and DQN agents on Maze.**

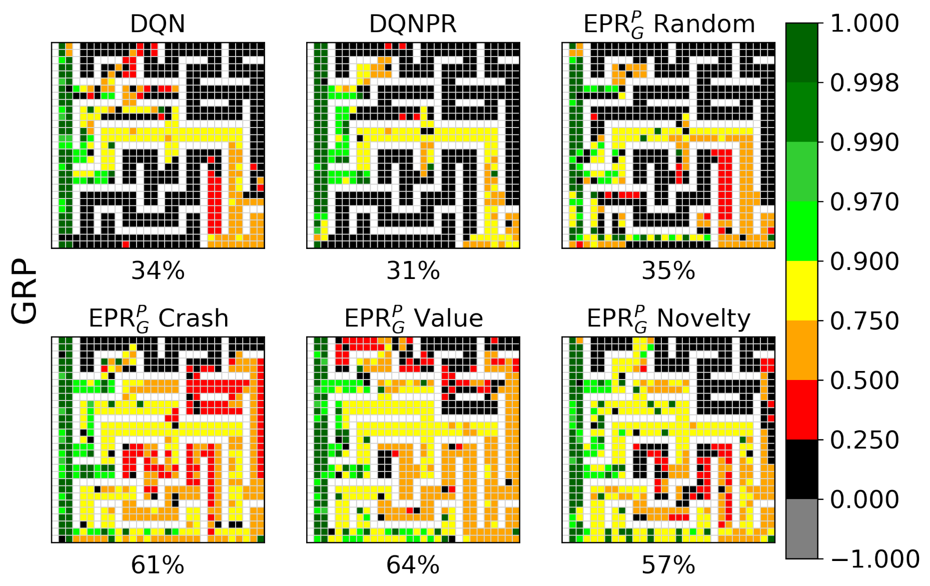


Figure 6.8 **Global GRP heatmaps of the best-performing EPR$_G^P$, DQNPR, and DQN agents on Maze.**

In Figures 6.9, 6.10, 6.11, we can see the heatmaps of the EPR_G^P and baseline agents, trained on the Hansen-bigger map. The first noticeable result is that the baselines' global GRP heatmaps show small average GRPs of 7% and 1%. We see in the corresponding considered states heatmaps that this is due to not reaching the goal line. However, all EPR_G^P agents reach the goal and achieve a high global GRP. This highlights the importance of using a lower bound ψ_{min} for the pseudo initial state region priority ψ since otherwise, the EPR_G^P agents would not have utilized pseudo initial state regions as long as they could not reach the goal from the starting line. The pseudo initial states collected by the random, value, and novelty strategy agents cover all map areas. On the contrary, the crash strategy selected pseudo initial states on the racetrack's center, forming a path from start to goal. This pattern translates to the considered states, which can also be seen in the considered states of the other EPR_G^P agents. Since the EPR_G^P agents attained high initial and global GRPs, we can conclude that focusing on the center areas of Hansen-bigger is more relevant than exploring its edges. Comparing the EPR_G^P agents' global GRP heatmaps, we observe that all achieve higher GRPs on the map's right areas than on its left areas. This indicates that navigating through Hansen-bigger's left part poses the greatest challenge for reaching the goal line.

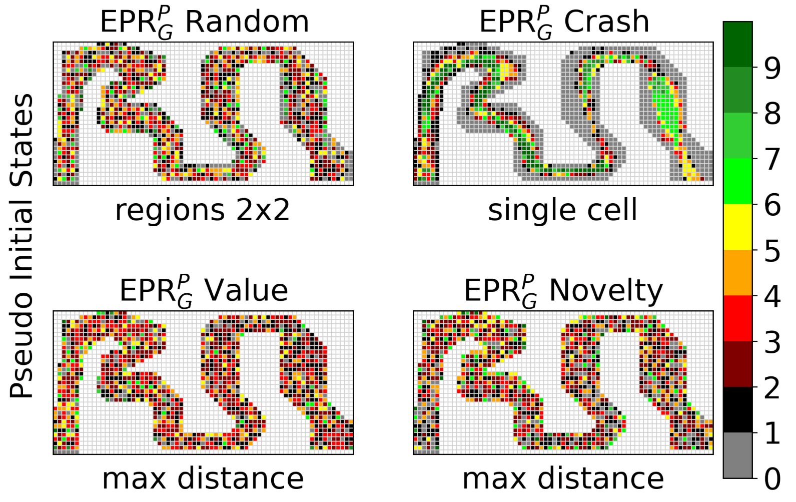


Figure 6.9 **Pseudo heatmaps of the best-performing EPR_G^P agents on Hansen-bigger.**

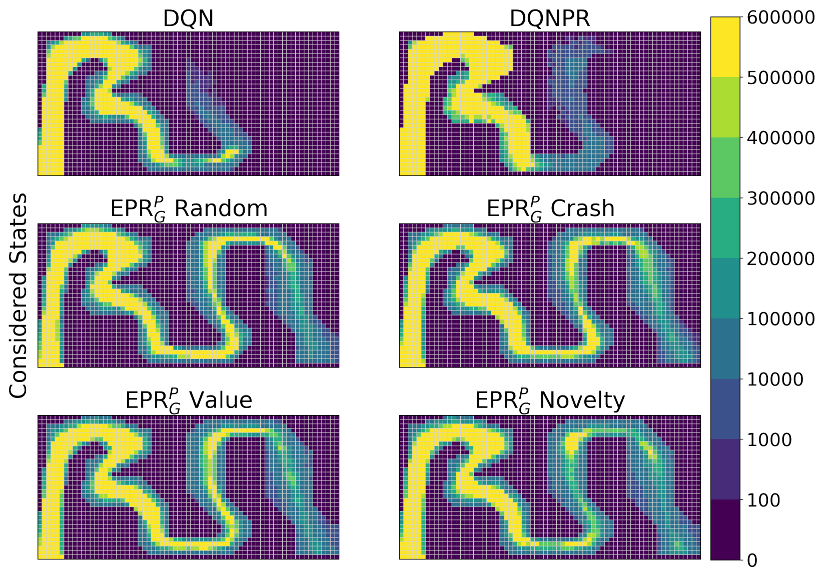


Figure 6.10 **Considered states heatmaps of the best-performing EPR$_G^P$, DQNPR, and DQN agents on Hansen-bigger.**

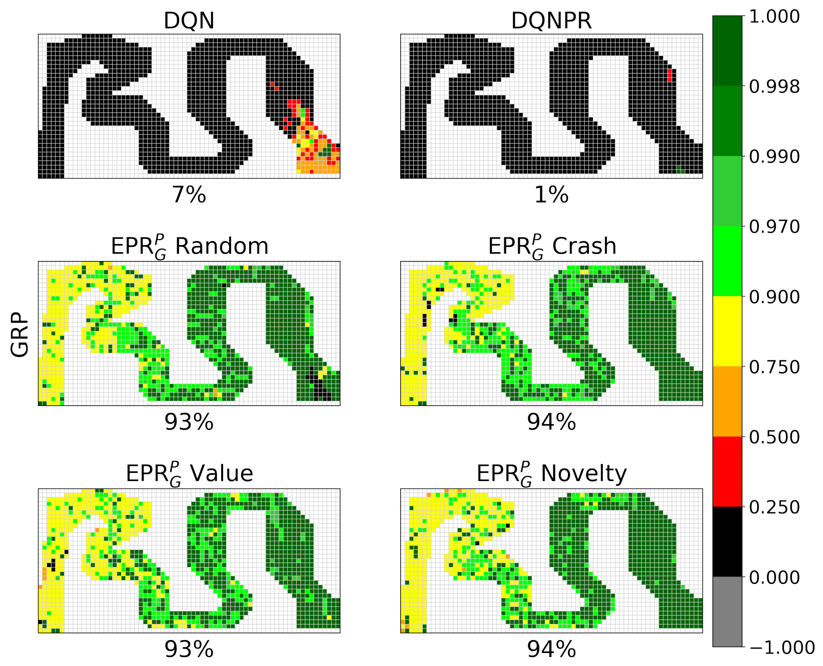


Figure 6.11 **Global GRP heatmaps of the best-performing EPR$_G^P$, DQNPR, and DQN agents on Hansen-bigger.**

6.2. EID_R^P and EPR_R^P Results

Here, we will discuss the results of training $\mathrm{EID}_R^P/\mathrm{EPR}_R^P$ agents. Initial experiments utilized the same training setup as the $\mathrm{EID}_G^P/\mathrm{EPR}_G^P$ experiments. However, they showed that the training progressed significantly slower when evaluating the expected return instead of evaluating the GRP since the evaluation stages required much more time to compute the evaluation values. Therefore, we made the following adjustments to the training setup: For each agent type, the training was repeated only 3 times with different random seeds, and the $\mathrm{EID}_R^P/\mathrm{EPR}_R^P$ agents only evaluated the original initial state regions every $L = 500$ episodes. The experiments took approximately 127 hours (EID_R^P: 50 hours, EPR_R^P: 57 hours) to complete.

Figure 6.12 shows the initial returns of the best-performing EID_R^P and DQN agents and Figure 6.13 shows their achieved initial variances. On the River-deadend map, we can see that all EID_R^P agents achieved higher initial returns and lower initial variances than DQN, indicating that the agents could reduce the number of situations in which they earned large negative returns. On the Maze map, only the EID_R^P agent that utilized the value strategy could match the baseline's initial return. However, the agent achieved a noticeably lower initial variance. Lastly, we see that the DQN agent failed to attain a suitable initial return on the Hansen-bigger map, whereas the EID_R^P agents achieved high initial returns and low variances.

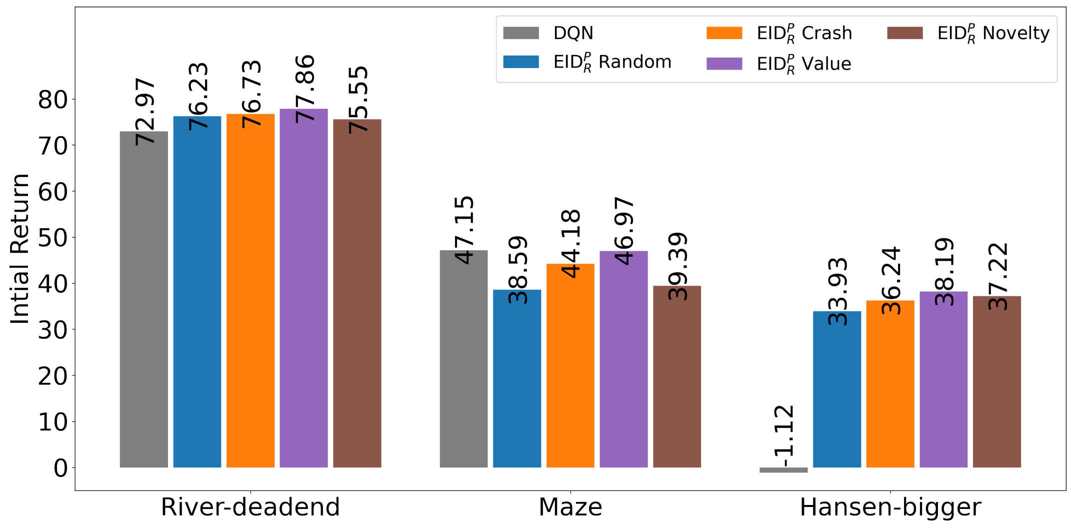


Figure 6.12 **Initial returns of the best-performing EID^P_R and DQN agents.**

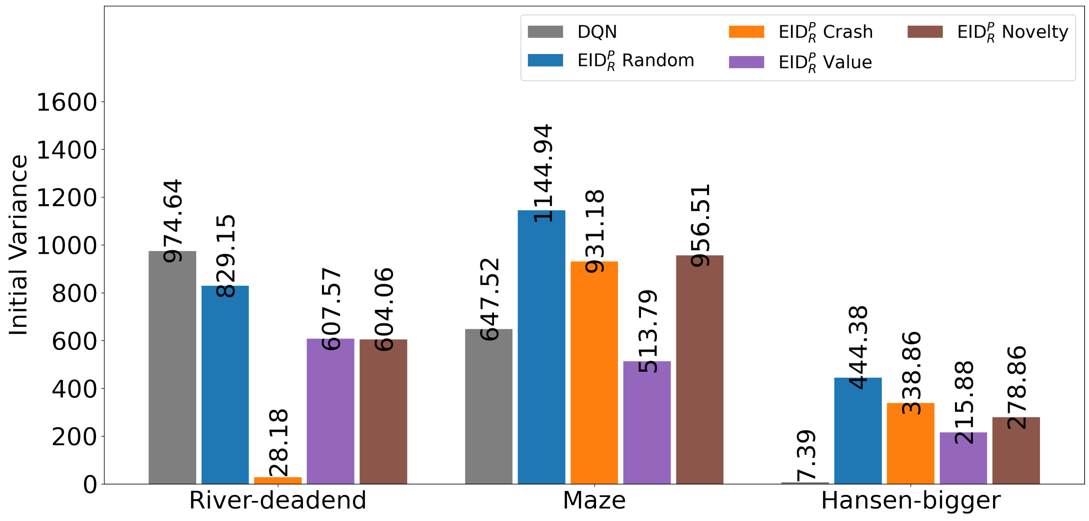


Figure 6.13 **Initial variances of the best-performing EID^P_R and DQN agents.**

In Figure 6.14 and Figure 6.15 we can see the initial returns and initial variances of the best-performing EPR_R^P and DQNPR agents. Contrary to the EID_R^P agents, the EPR_R^P agents could neither achieve higher initial returns nor lower initial variances than the DQNPR baseline on the River-deadend map. On the Maze map, the random, crash, and value strategy agents performed similarly to the DQNPR baseline, but the EPR_R^P agent that utilized the novelty strategy achieved a significantly higher initial return and lower initial variance. On the Hansen-bigger map, we observe the same behavior as in the EID_R^P experiments.

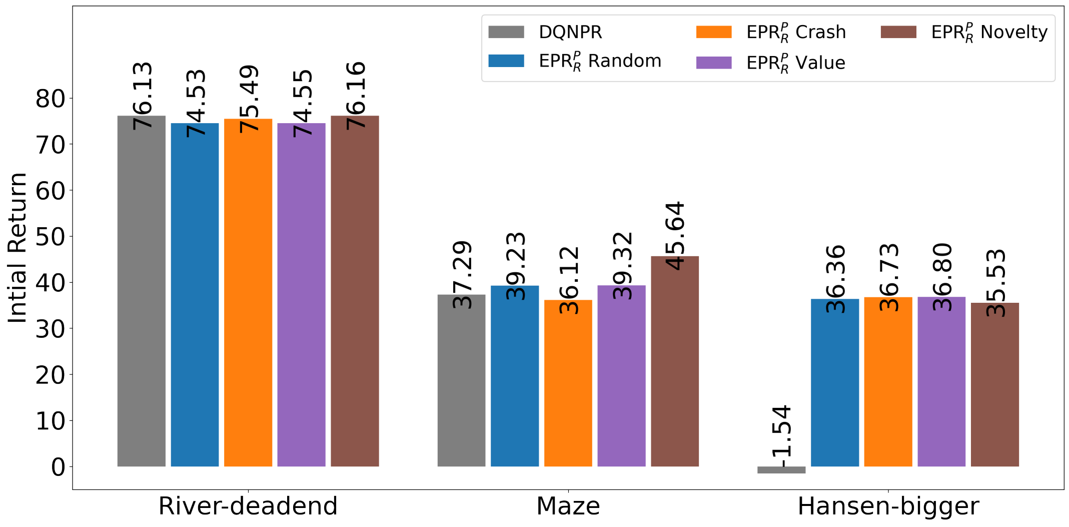


Figure 6.14 **Initial returns of the best-performing EPR_R^P and DQNPR agents.**

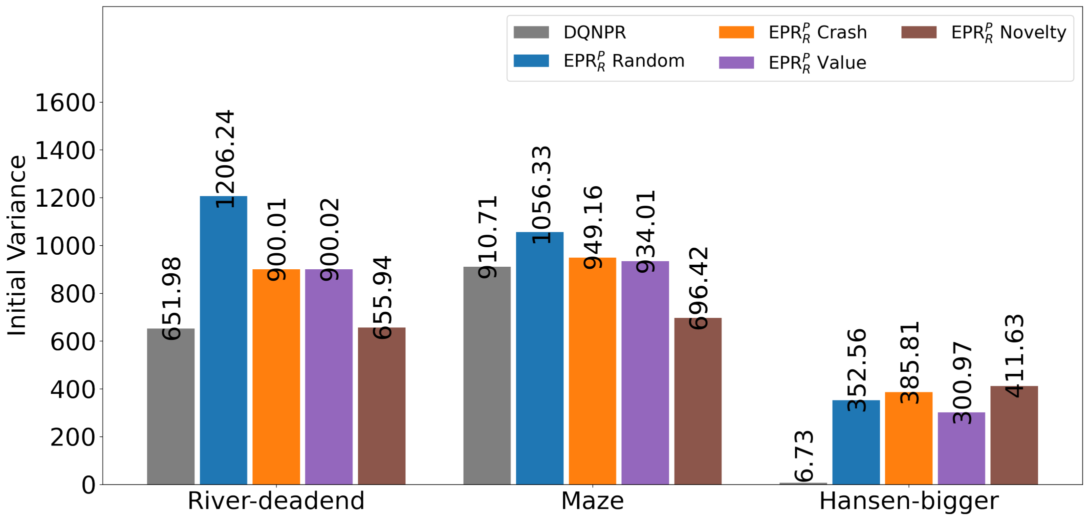


Figure 6.15 **Initial variances of the best-performing EPR_R^P and DQNPR agents.**

As in the previous section, we will investigate how some of the best-performing agents behaved during training. Figures 6.16, 6.17, 6.18 show the pseudo, considered states, and global return heatmaps of the best-performing EID^P_R and baseline agents on the River-deadend map. We can see in the pseudo heatmaps that the random, value, and novelty strategy agents selected pseudo initial states across the whole map, whereas the crash strategy agent solely selected states on the direct paths between starting line and goal lines. Thus, we observe in the considered states heatmaps that the agents explored the map more thoroughly than the DQN, DQNPR, and crash strategy agents. The global GRP heatmaps show that all EID^P_R agents achieved higher global returns than the DQN and DQNPR agents, which is likely why most of them also achieved higher initial returns and lower initial variances.

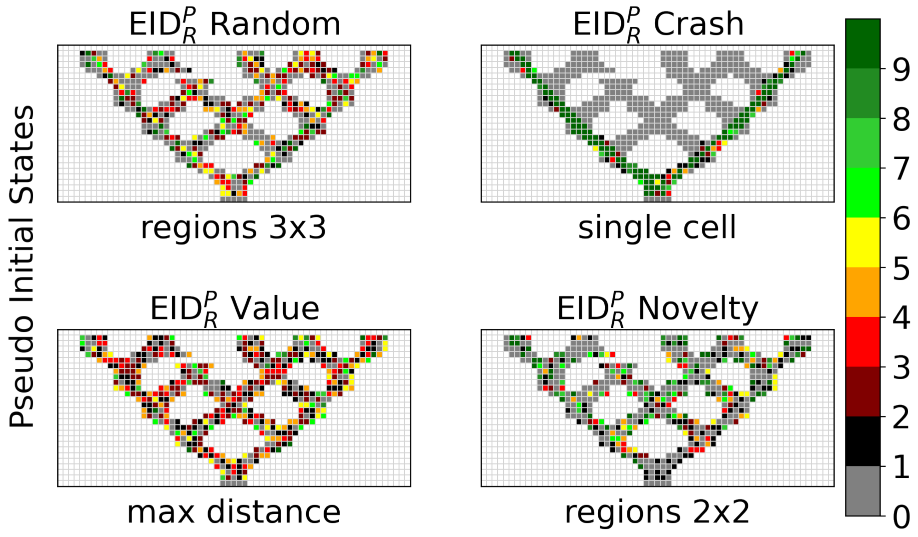


Figure 6.16 **Pseudo heatmaps of the best-performing EID^P_R agents on River-deadend.**

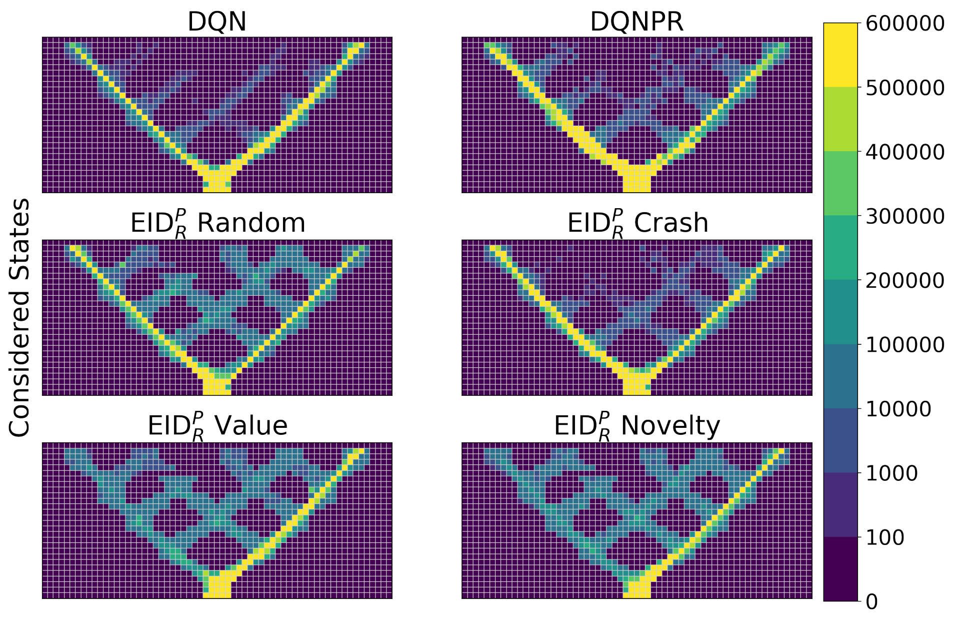


Figure 6.17 **Considered states heatmaps of the best-performing EID$_R^P$, DQNPR, and DQN agents on River-deadend.**

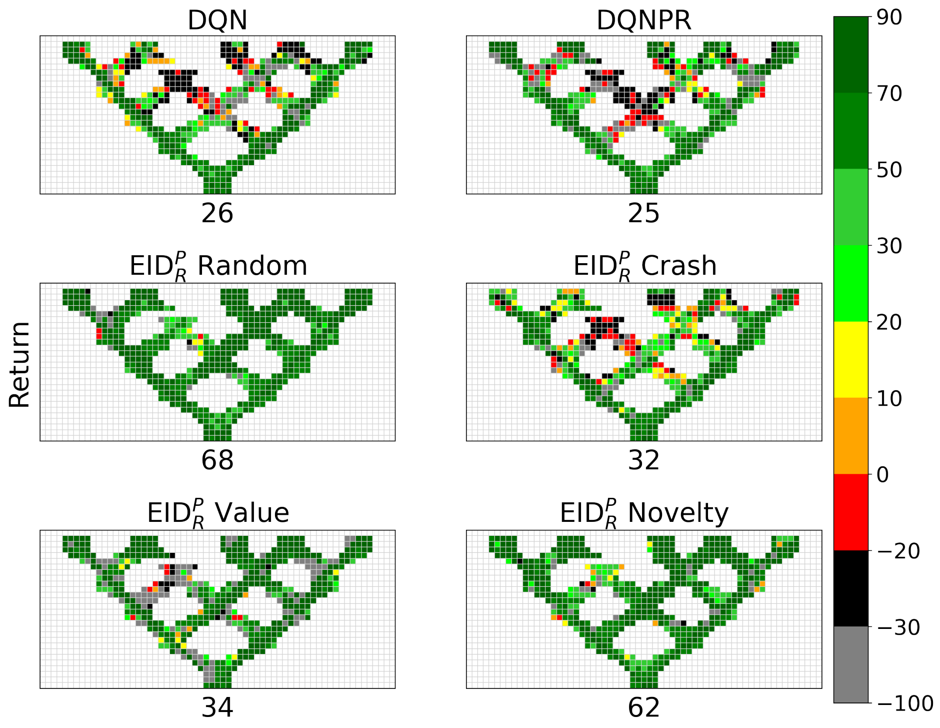


Figure 6.18 **Global GRP heatmaps of the best-performing EID$_R^P$, DQNPR, and DQN agents on River-deadend.**

In Figures 6.19, 6.20, 6.21, we see the pseudo, considered states, and global return heatmaps of the best-performing EPR$_R^P$ and baseline agents. The pseudo heatmaps show that the novelty strategy agent selected pseudo initial states that were mostly located on the same cells, yet the other EPR$_R^P$ agents selected states from a large portion of map cells. However, the only difference that we can observe in the considered states heatmaps is that the novelty strategy agent did not focus on states from the map's lower areas, whereas the other EPR$_R^P$ agents considered the areas beyond the direct path through the Maze map evenly. In general, all EPR$_R^P$ agents considered these map areas significantly more than the DQN and DQNPR agents. Therefore, the EPR$_R^P$ agents achieve much higher global returns than the DQN and DQNPR agents. Based on these results, it is remarkable that the only EPR$_R^P$ agent that achieved a higher initial return and a lower initial variance compared to a baseline agent was the novelty strategy agent, which outperformed the DQNPR agent.

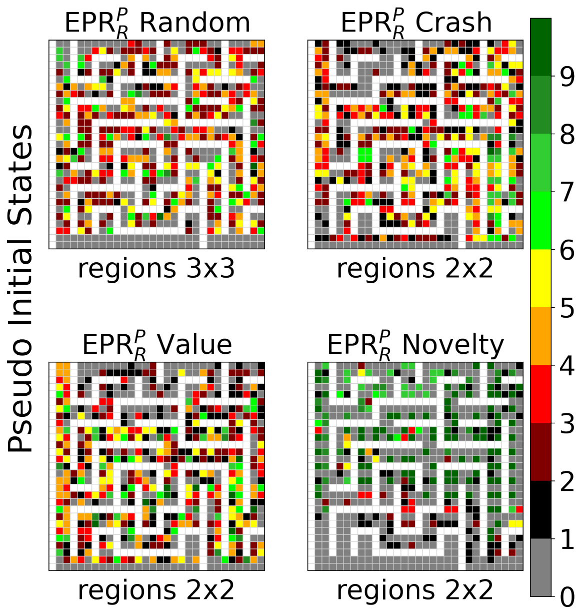


Figure 6.19 **Pseudo heatmaps of the best-performing EPR$_R^P$ agents on Maze.**

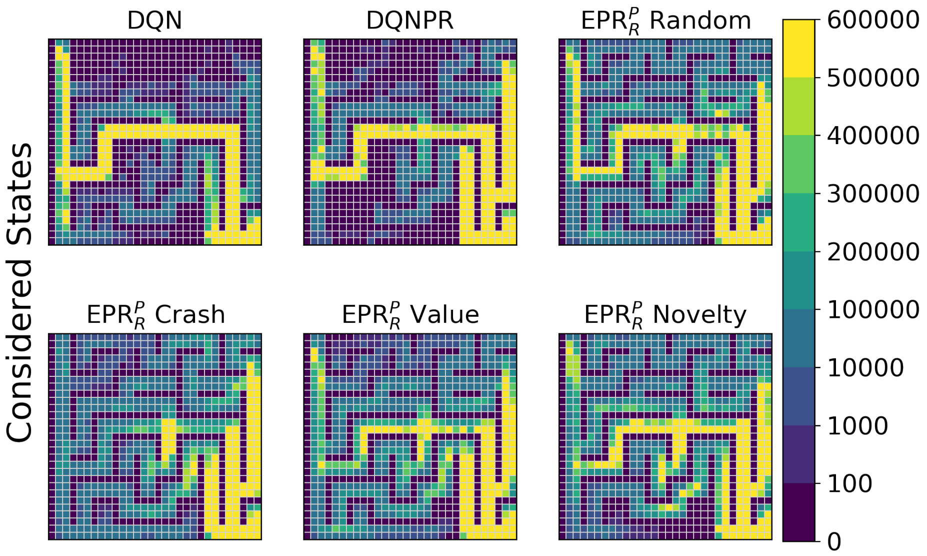


Figure 6.20 **Considered states heatmaps of the best-performing EPR$_R^P$, DQNPR, and DQN agents on Maze.**

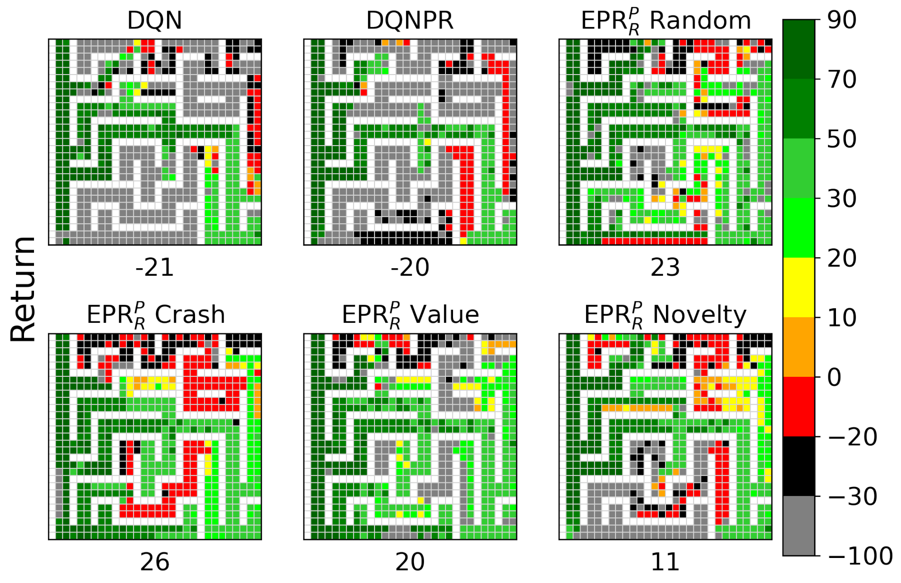


Figure 6.21 **Global GRP heatmaps of the best-performing EPR$_R^P$, DQNPR, and DQN agents on Maze.**

In Figures 6.22, 6.23, 6.24, we can observe that the best-performing EID$_R^P$ and baseline agents behaved on the Hansen-bigger map similarly to the best-performing EPR$_G^P$ and baseline agents in Figures 6.9, 6.10, 6.11. All EID$_R^P$ agents selected pseudo initial states across the whole map, allowing them to reach the goal line, whereas the DQN and DQNPR agents remained stuck in the map's left part. Thus, the EID$_R^P$ agents achieved high global returns, which enabled them to attain high initial returns and low initial variances.

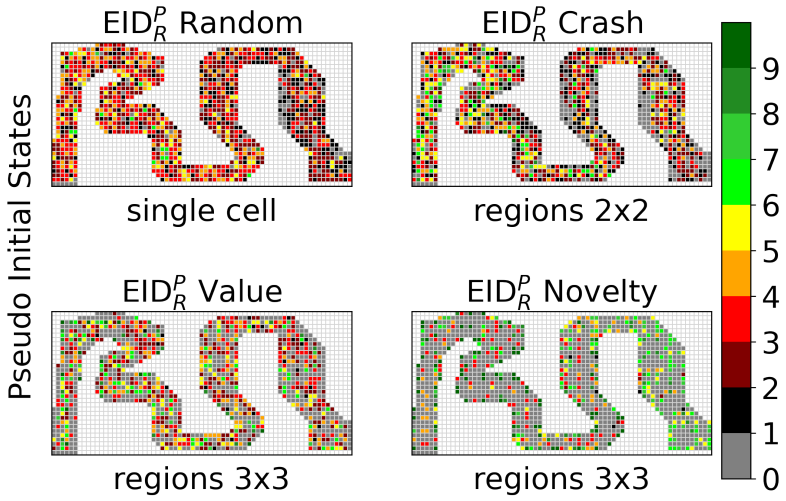


Figure 6.22 **Pseudo heatmaps of the best-performing EID$_R^P$ agents on Hansen-bigger.**

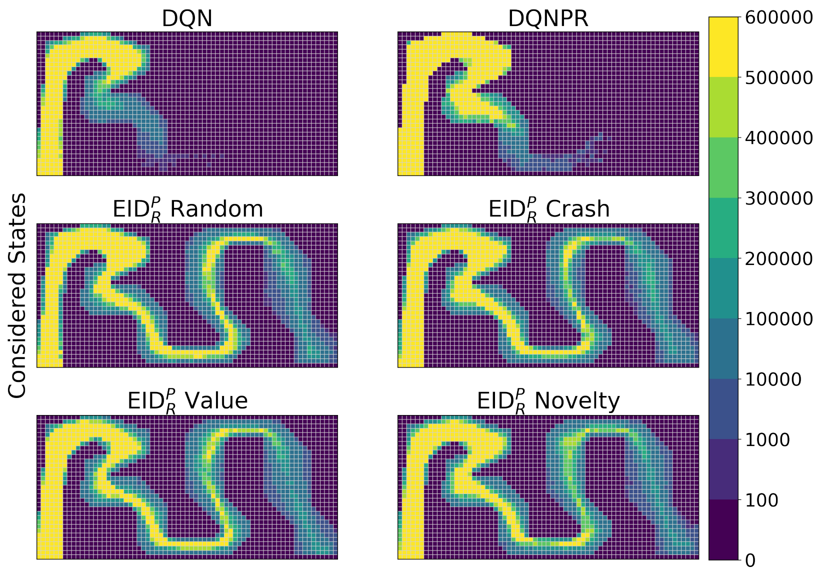


Figure 6.23 **Considered states heatmaps of the best-performing EID$_R^P$, DQNPR, and DQN agents on Hansen-bigger.**

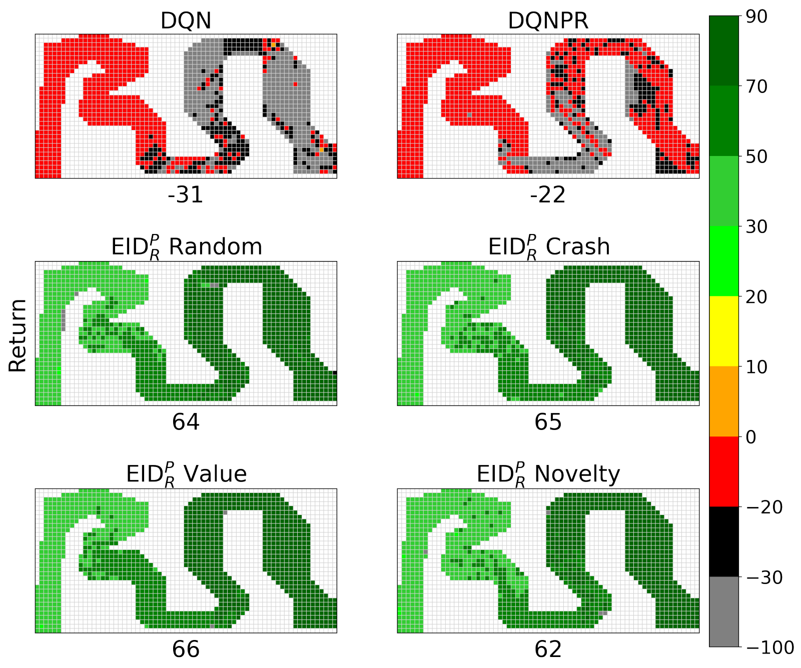


Figure 6.24 **Global GRP heatmaps of the best-performing EID$_R^P$, DQNPR, and DQN agents on Hansen-bigger.**

6.3. PPO as the Basis of EID$_G^P$

As we presented in chapter 3, policy-gradient algorithms, such as PPO, have a fundamentally different approach to learning policies than value-based algorithms like DQN and DQNPR. Therefore, it is interesting to investigate using PPO as the underlying training algorithm of DSMC ESR. Since PPO cannot utilize a replay buffer, we will only conduct experiments with EID^P.

The first experiments involved training EID$_G^P$ agents and utilized the same training setup as the DQN-based EID$_G^P$ experiments. However, we observed that the PPO-based agents learned very slowly since all of them either could not reach the goal line or achieved an insufficient initial GRP. To ensure that these behaviors were not due to inadequate hyperparameter choices, a grid search over the actor network's learning rate $\alpha_{Actor} \in \{5\text{e}-4, 8\text{e}-4\}$ and the entropy coefficient $c_2 \in \{1\text{e}-3, 4\text{e}-3\}$ was conducted in the following experiments. These showed to be the most influential hyperparameters, and their initial values $\alpha_{Actor} = 5\text{e}-4$ and $c_2 = 1\text{e}-3$ were taken from an existing implementation of the actor-critic algorithm A2C [22]. Furthermore, the number of episodes was increased to $E = 400.000$. These modifications significantly prolonged the required training time, so the experiments were not repeated for every distance limit. Instead, we experimentally selected a suitable distance limit for each EID$_G^P$ agent on each map. Furthermore, the lower pseudo initial state region priority bound ψ_{min} was increased to $\psi_{min} = 0.4$ to increase exploration. Two DNNs with the same architecture as in Figure 3.4 were used for the actor and critic networks, where the latter had only one output node. The remaining hyperparameters that are not specific to PPO had the same values as in the DQN-based experiments, and a complete list of hyperparameters can be found in the appendix A.1. The training took approximately 45 hours to complete.

We see in Figure 6.25 that the best-performing PPO and EID$_G^P$ agents could only achieve non-zero initial GRPs on the River-deadend map, where the value strategy agent was able to outperform the baseline, although with an insufficient initial GRP.

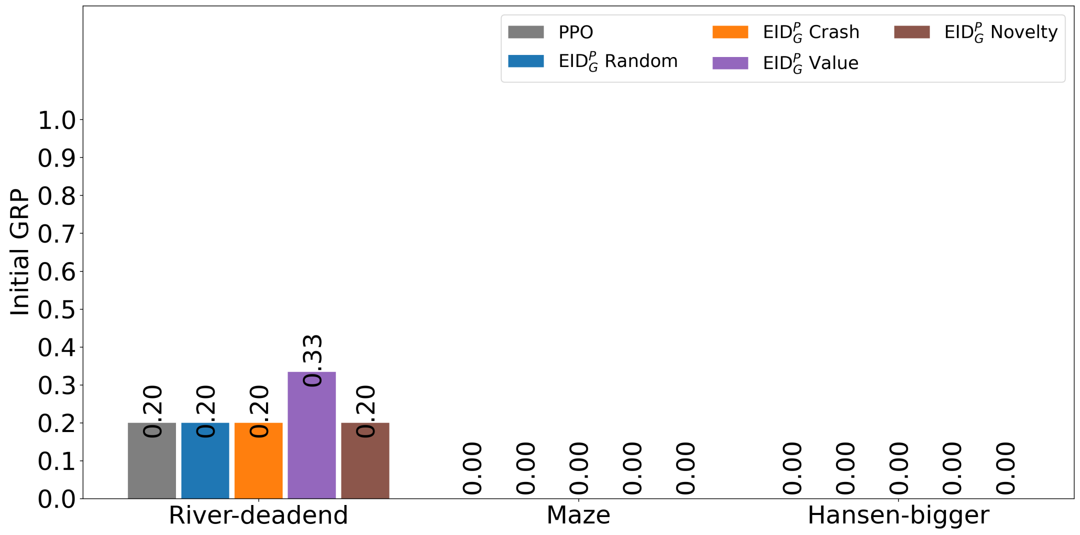


Figure 6.25 **Initial GRPs of the best-performing EID$_G^P$ and PPO agents.**

To understand why these agents performed worse than the agents in the DQN-based experiments, we will investigate which pseudo initial states the strategies selected, which states were considered for training, and how this affected the global GRP. Note that the bounds in the considered states heatmaps were lowered by a factor of 10.

Figures 6.26, 6.27, 6.28 show the pseudo, considered states, and global GRP heatmaps of the best-performing EID$_G^P$ and PPO agents on River-deadend. The pseudo heatmaps show similar patterns as those of the DQN-based EID$_G^P$ agents since the PPO-based EID$_G^P$ agents selected pseudo initial states in all map areas, except for the crash strategy agent. However, all EID$_G^P$ agents demonstrate a lack of exploration in their considered states heatmaps since they mainly considered cells close to the starting line. This deficiency is even more severe in the PPO agent's considered states heatmap. Comparing the global GRP heatmaps of the random and crash strategy with that of the PPO agent suggests that they all learned a similar behavior of driving right and in a straight line. The novelty strategy also learned such behavior, but it drove left. This leads to a smaller global GRP since the agent will more often get stuck in the map's dead end. The value strategy achieved the highest global GRP because its GRP exceeded 25% on more cells than the other agents. This suggests that the agent learned a more complex policy, allowing it to reach goal lines by driving in multiple directions, which also lead to a higher initial GRP. The better performance of the value strategy agent is likely due to higher exploration since it considered the most states from the map's center.

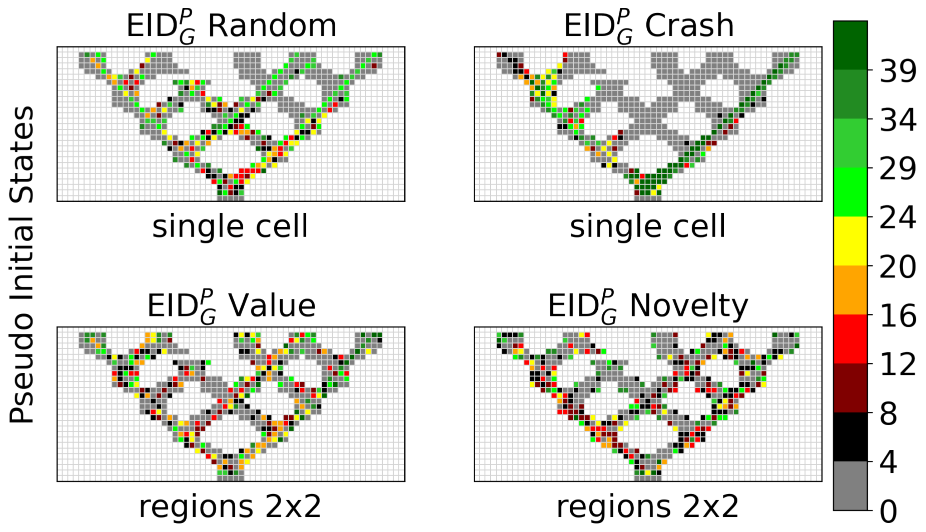


Figure 6.26 **Pseudo heatmaps of the best-performing EID$_G^P$ agents on River-deadend.**

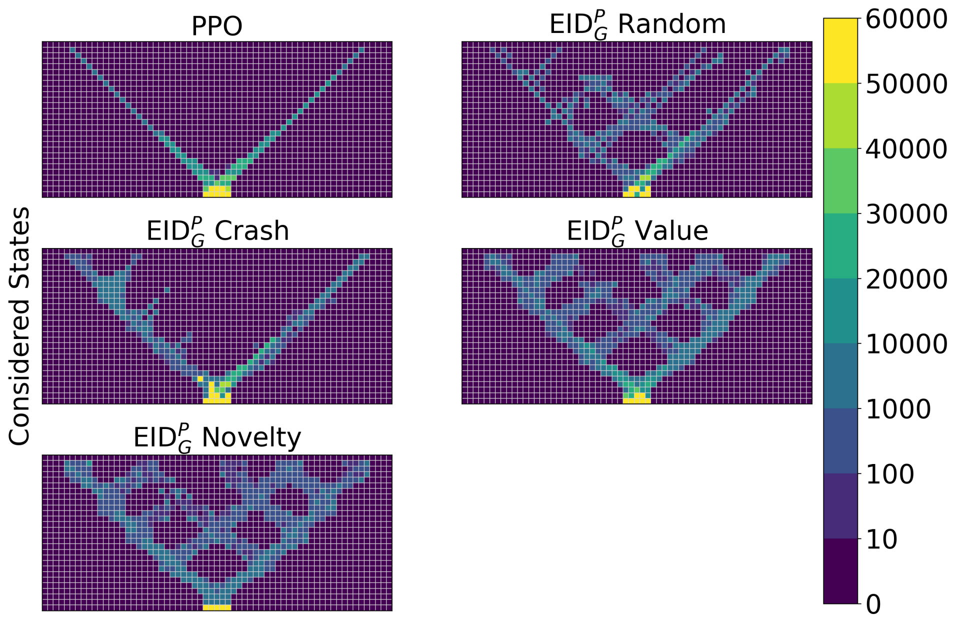


Figure 6.27 **Considered states heatmaps of the best-performing EID$_G^P$ and PPO agents on River-deadend.**

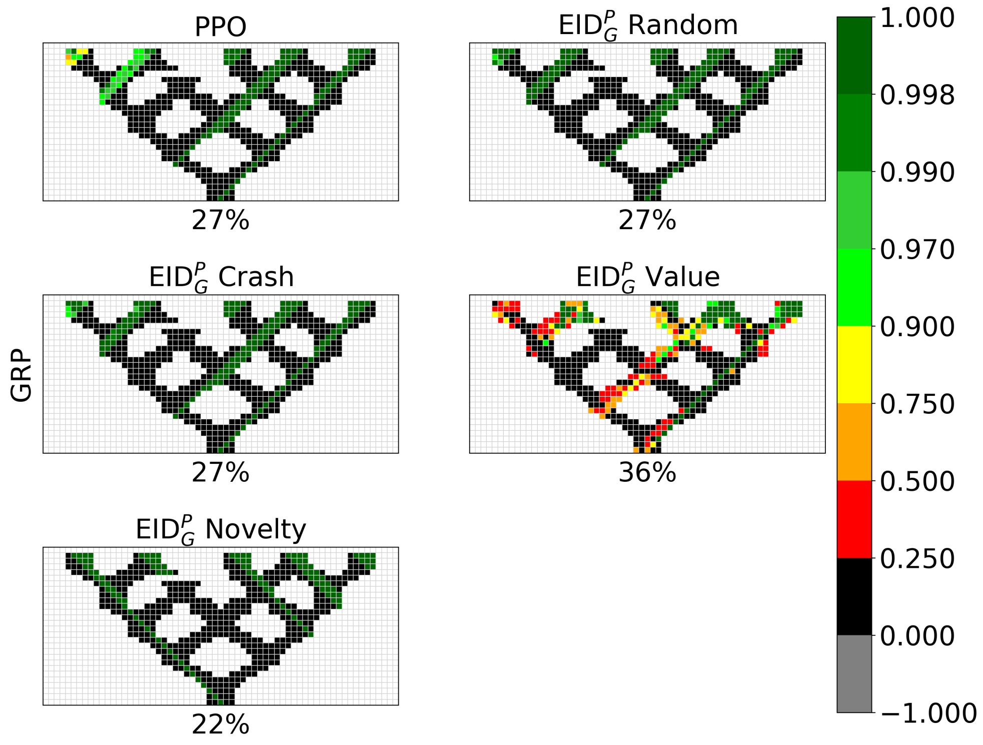


Figure 6.28 **Global GRP heatmaps of the best-performing EID$_G^P$ and PPO agents on River-deadend.**

Figures 6.29, 6.30, 6.31 depict the pseudo, considered states, and global GRP heatmaps of the Maze map. We observe that the EID$_G^P$ agents selected states from most parts of the map. However, we can see in none of their considered states heatmaps that they focused the training on a path from the starting line to the goal line. The random, crash, and value strategy agents' heatmaps show multiple small map areas with bright yellow cells, indicating that the agents were stuck in these areas since we cannot see any pattern resembling a path through the Maze. We see in the PPO and novelty strategy agents' heatmaps that they rarely visited areas beyond the starting line, although the novelty strategy agent explored the map significantly more than the PPO agent. The global GRP heatmaps show that no agents learned to reach the goal line from a considerable number of map areas.

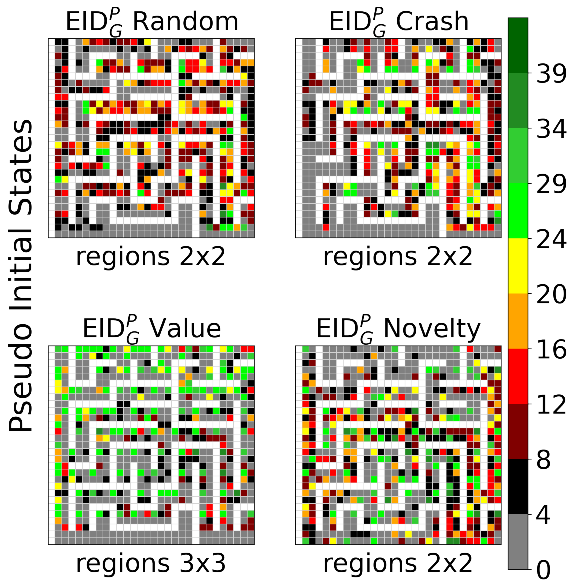


Figure 6.29 **Pseudo heatmaps of the best-performing EID$_G^P$ agents on Maze.**

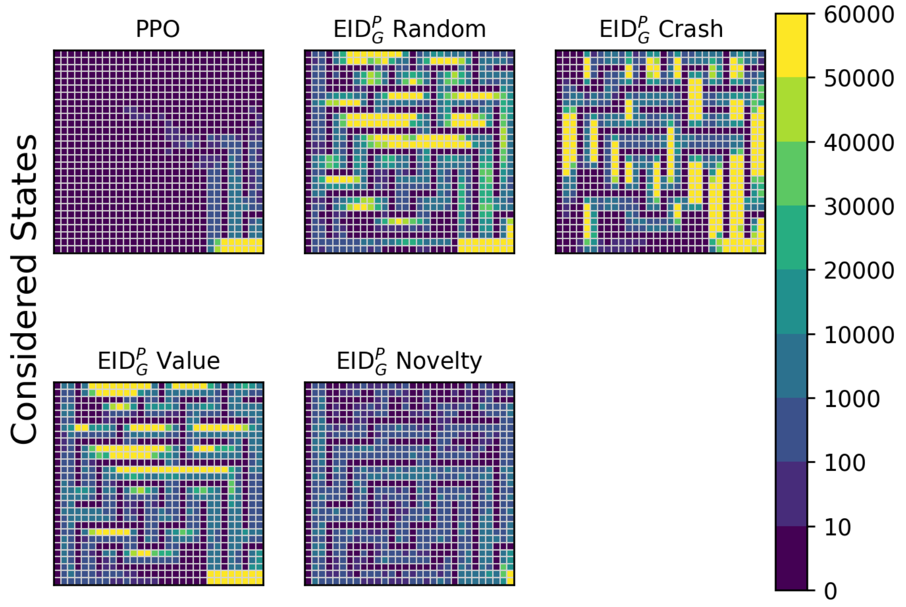


Figure 6.30 **Considered states heatmaps of the best-performing EID$_G^P$ and PPO agents on Maze.**

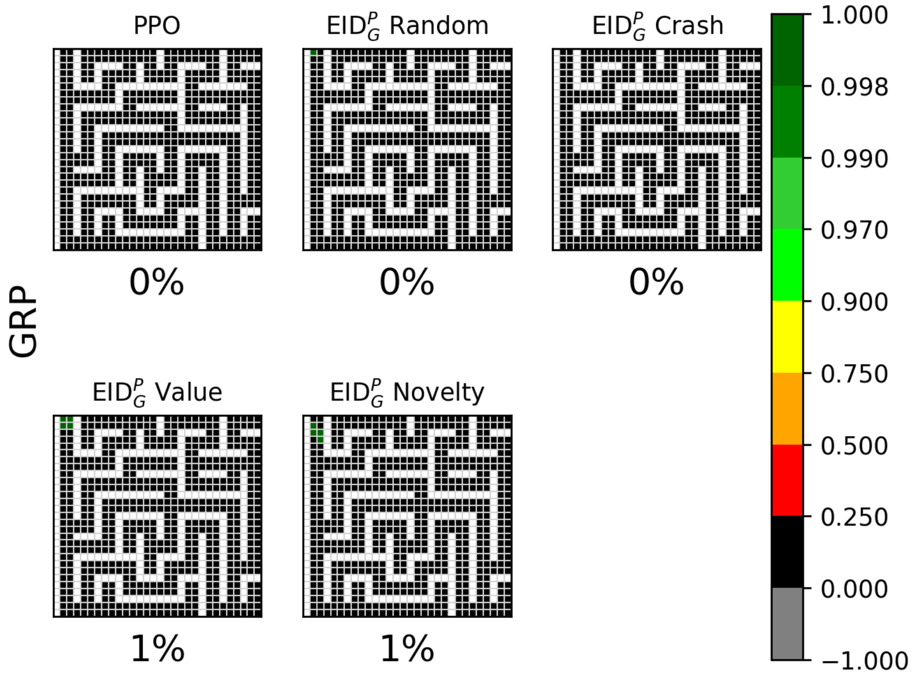


Figure 6.31 **Global GRP heatmaps of the best-performing EID$_G^P$ and PPO agents on Maze.**

In Figures 6.32, 6.33, 6.34, we can see the pseudo, considered states, and global GRP heatmaps on the Hansen-bigger map. The pseudo heatmaps show that the random and value strategy agents selected only a few pseudo initial states on the map's right side, whereas the crash and novelty strategy agents did not select any states from these areas. Looking at the considered states heatmaps, we see that only the random and value strategy agents selected pseudo initial states from the map's right side because only they could drive to these areas. Since they were the only ones to reach the goal line, the random and value strategy agents attained the highest global GRPs with 24% and 21%, respectively. However, we see in their global GRP heatmaps that they could only reach the goal from cells close to the goal line.

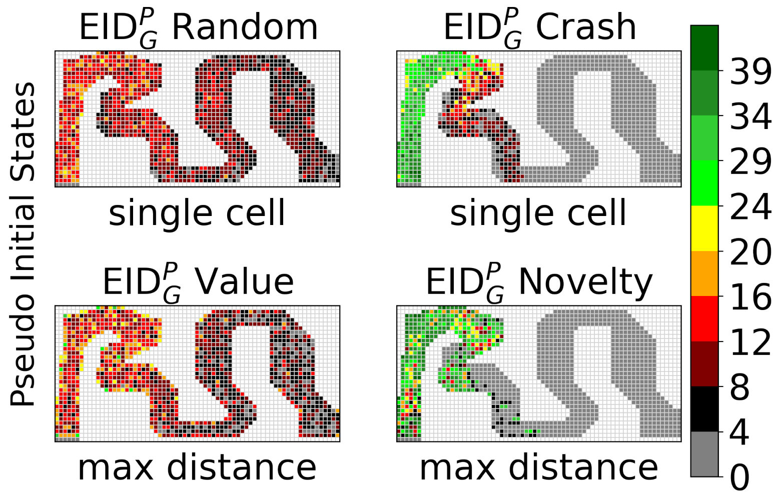


Figure 6.32 **Pseudo heatmaps of the best-performing EID$_G^P$ agents on Hansen-bigger.**

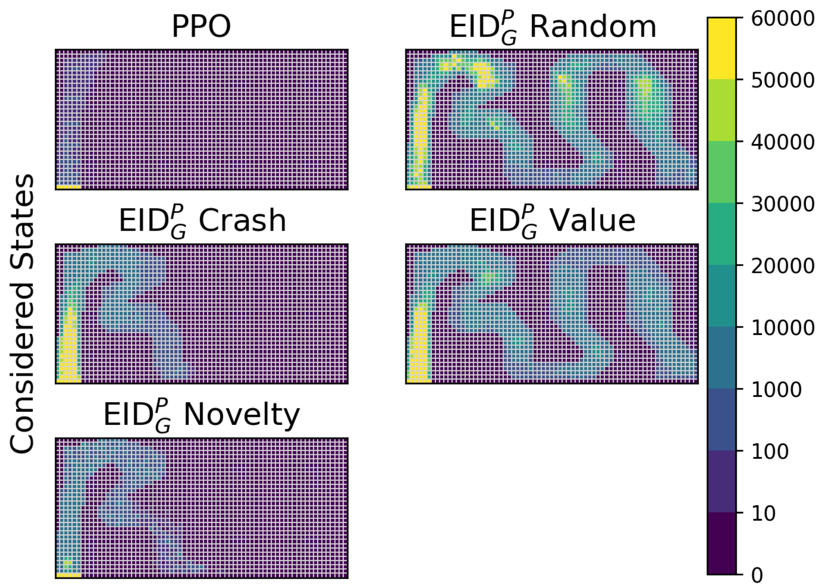


Figure 6.33 **Considered states heatmaps of the best-performing EID$_G^P$ and PPO agents on Hansen-bigger.**

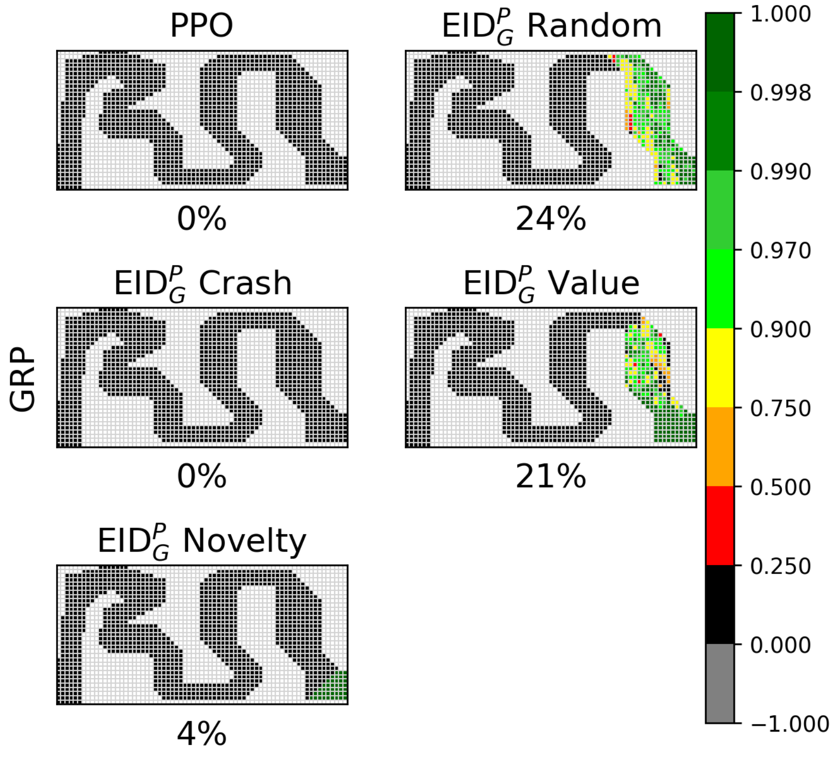


Figure 6.34 **Global GRP heatmaps of the best-performing EID$_G^P$ and PPO agents on Hansen-bigger.**

In section 3.3, we discussed that policy-gradient algorithms are prone to get stuck in local optima and often suffer from gradient estimates with high variance. These inherent problems and the fact that PPO cannot utilize a replay buffer may explain why the PPO-based EID_G^P agents learned so slowly in comparison to the DQN-based EID_G^P agents. Unfortunately, exactly determining which of these problems caused the poor training results requires further analysis beyond the scope of this thesis. However, the results on River-deadend are good examples of the agents getting stuck in local optima. Driving in a straight line from the starting point is a simple behavior that achieves a decent average return. To increase their returns, agents need to learn to drive in curves, which would at first lead to many crashes and a decreased average return. Therefore, the „drive-in-a-straight-line“ policy represents a local optimum in which all agents, except the value strategy agent, got stuck. This is further confirmed by the fact that these agents already achieve the same initial and global GRPs after 100.000 episodes, and their policies do not noticeably change during the remaining 300.000 episodes. Getting stuck in local optima may also explain the unusual patterns in the considered states heatmaps of the EID_G^P agents that utilized the random, crash, and value strategies on the Maze map. We observed in these heatmaps multiple small map regions on which the training was heavily focused but no clear pattern indicating that the agents found a path to the goal line. This could be caused by the agents learning to only drive in these regions, allowing them to avoid immediately crashing and thus large negative rewards. Attaining larger returns would require the agents to drive beyond these map regions, which would, at first, result in more crashes, leading to decreased returns.

It is unlikely that the underwhelming results of the PPO-based experiments are due to bad hyperparameter choices since we conducted a grid search over the most influential hyperparameters. Furthermore, additional experiments using an existing A2C implementation showed the same lack of performance. Therefore, it is likely that the number of training episodes was too small to compensate for PPO's comparatively slow learning speeds in the Racetrack task. For this reason, we did not expect PPO-based EID_R^P to perform better than PPO-based EID_G^P, so we did not conduct the corresponding experiments.

We saw in all considered states heatmaps that the agents exhibited a lack of exploration. It is reasonable to assume that given more training steps, the agents could gather the necessary experiences to learn better policies. In such a scenario, we can expect the EID_G^P agents to learn faster than the PPO agents since they explored the maps significantly more than the PPO agents.

6.4. Using the Single Cell Limit

The single cell limit imposes the least constraints on prospective pseudo initial states, meaning that their qualifications, evaluated by the corresponding selection strategies, are the main factor in deciding which candidates will be selected. Thus, the strategies behave the most characteristically when using this distance limit, making it insightful to analyze their selections. In the following, we will investigate the pseudo heatmaps of EID$_G^P$ agents that used the single cell distance limit since their behaviors are representative of the other DSMC ESR agents.

In Figure 6.35, we can see the pseudo heatmaps of River-deadend. We observe that the random strategy agent selected states across the whole map, with a tendency for the outer areas. The agent drove through these areas in most episodes because traversing them is the quickest way of reaching a goal line from the start. Since this strategy selects states that the agent encounters with uniform probability, it collects more pseudo initial states in such frequently visited regions. The crash strategy agent's pseudo heatmap shows a distinctively different behavior of concentrating the pseudo initial states on the outer areas and, to a lesser extent, on the center areas. This indicates that mistakes leading to crashes usually occur when the agent is still on a direct path to a goal line and not when it has already deviated from such paths. We can observe a similar pattern in the value strategy agent's heatmap, which shows that this strategy selects states located on paths to goal states. In the novelty strategy agent's heatmap, we see that states close to the starting line have only been selected once, and in fact, these states were used as pseudo initial states only during the first evaluation stage. This is because these states were located in a frequently visited area and quickly became familiar to the agent. The remaining pseudo initial states were selected mainly from the center and right map areas. Comparing this to the considered states heatmap in Figure 6.38 shows that these were rarely visited areas, from which we can follow that random network distillation successfully identified novel states.

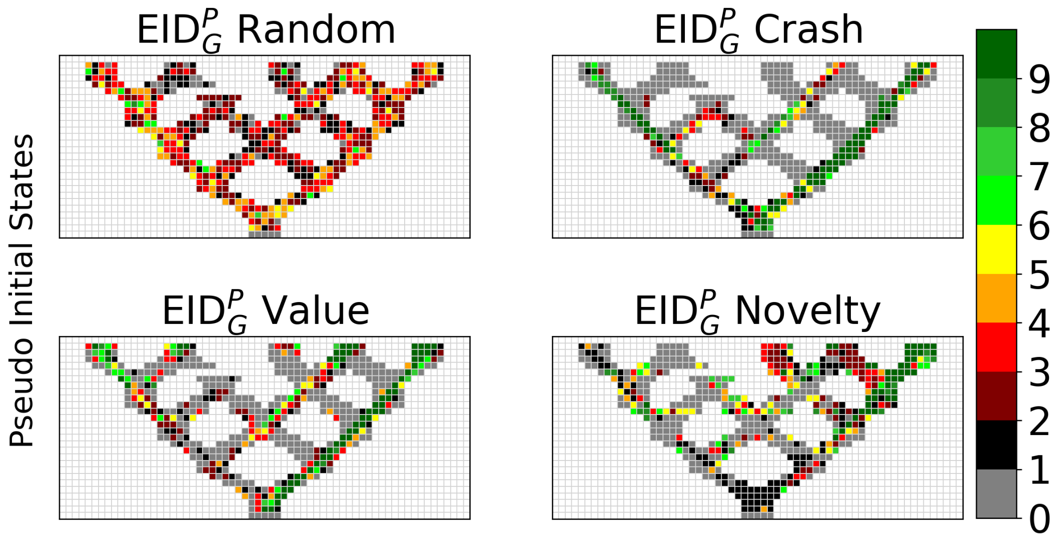


Figure 6.35 **Pseudo heatmaps of EID$_G^P$ agents that utilized the single cell distance limit on the River-deadend map.**

In Figure 6.36, we see that all strategies behaved similarly on the Maze map. It is clearly visible how the random strategy agent selected more states in the map's center areas, which were frequently visited. The crash strategy agent focused its pseudo initial state selection on only a few areas but did not reach the goal line, suggesting that the agent could not learn to avoid crashing after starting in the pseudo initial states. The value strategy agent's heatmap shows it focused on the shortest path to the goal while neglecting the rest of the map, whereas the novelty strategy selected pseudo initial states in an almost opposite way. This highlights how the value and novelty strategies focus on exploitation and exploration.

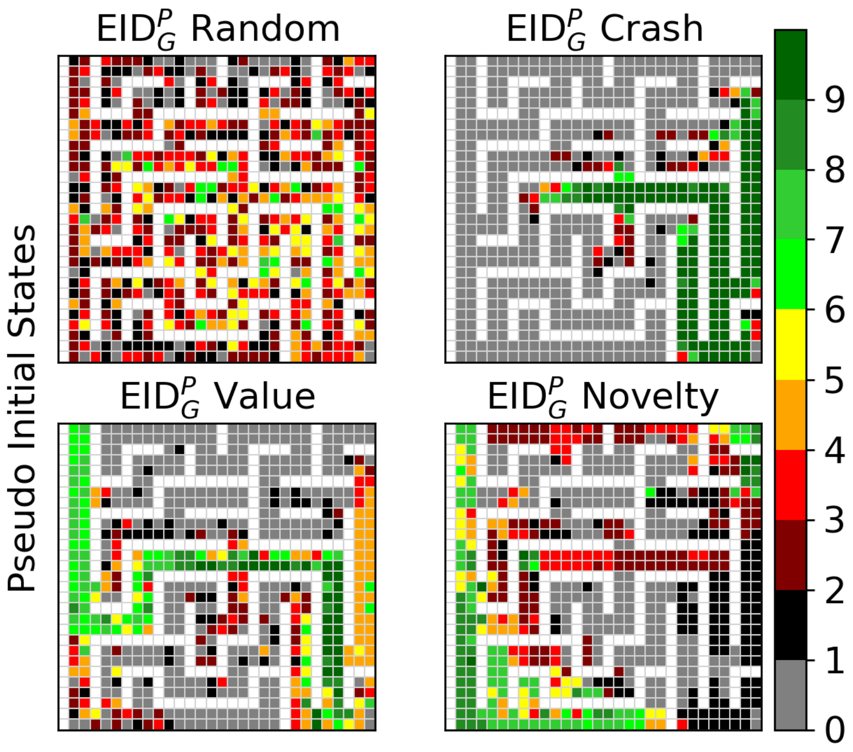


Figure 6.36 **Pseudo heatmaps of EID_G^P agents that utilized the single cell distance limit on the Maze map.**

In the pseudo heatmaps of Hansen-bigger, shown in Figure 6.37, we see that the random strategy agent's heatmap depicts that significantly more pseudo initial states were selected on the map's left side, suggesting that the agent struggled to drive beyond this area. The crash strategy agent's selected states formed a pattern resembling a path from start to goal through the center of the racetrack. This further validates that fatal mistakes mostly happen when the agent is still on the correct path. Although less distinct, we see the same pattern of selected states with the value strategy agent. In contrast to the random strategy, the novelty strategy agent primarily focused on the right part of the map, which seems difficult to reach since Figure 6.38 shows that only a few states were considered from this area. Thus, the right side of Hansen-bigger contained the most novel states.

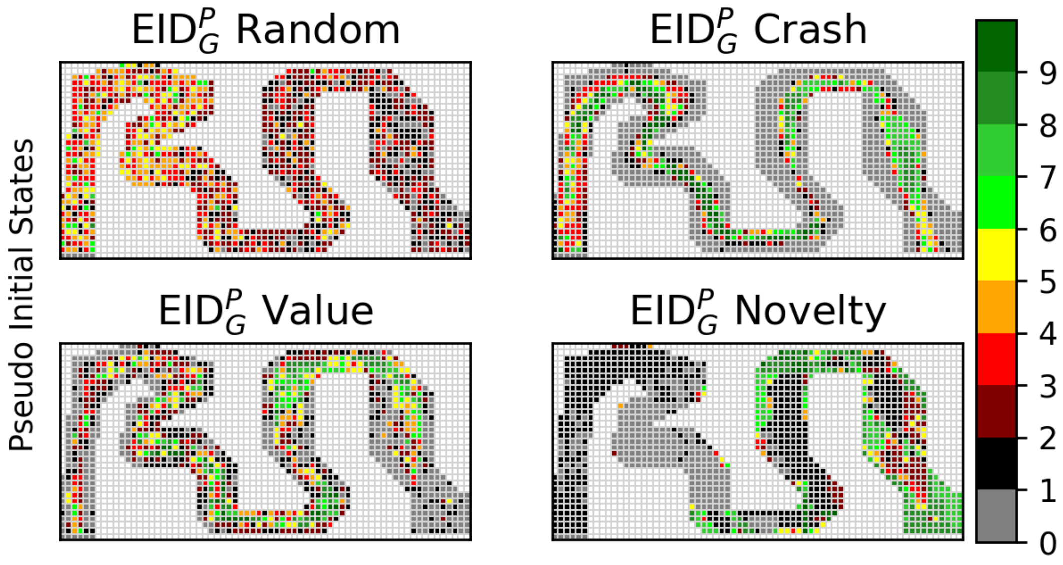


Figure 6.37 **Pseudo heatmaps of EID_G^P agents that utilized the single cell distance limit on the Hansen-bigger map.**

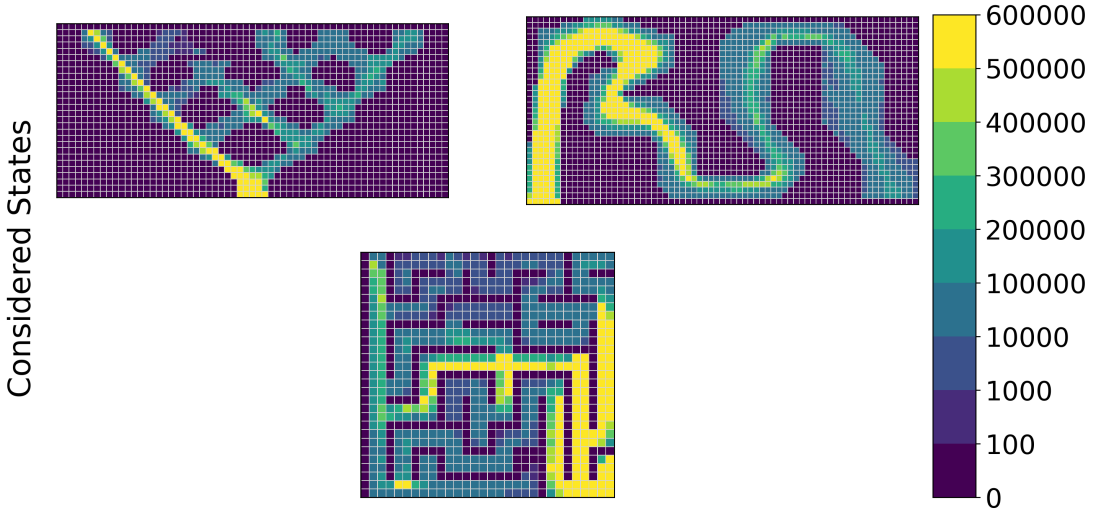


Figure 6.38 **Considered states heatmaps of the novelty strategy agent that utilized the single cell distance limit on all maps.**

6.5. Summary

Our experiments showed that the DQN-based EID^P and DQNPR-based EPR^P algorithms could achieve higher initial GRPs and higher initial returns with lower variances compared to the regular DQN and DQNPR algorithms. In most instances, we observed that the DSMC ESR agents explored the maps significantly more than the baseline agents, enabling them to perform better when starting in various map regions. Therefore, we followed that the DSMC ESR agents likely outperformed the baseline agents in the initial states because they had a smaller chance of crashing once they deviated from the direct paths between the starting and the goal line.

On the River-deadend map, our methods achieved only small improvements because it is easy for agents to reach a goal line on this map without driving through various map areas. Thus, the increased exploration had a smaller benefit for the DSMC ESR agents. We achieved more extensive performance improvements on the Maze map since the agents were much more likely to deviate from the direct path to the goal line. Therefore, the DSMC ESR agents benefited significantly from the increased exploration. This need for exploration also explains why the best-performing EID^P_G, EPR^P_G, and EPR^P_R agents utilized the novelty strategy. The Hansen-bigger map posed a significant challenge to the DQNPR and DQN agents since they could not reach the goal line. On the contrary, all DSMC ESR agents performed very well since restarting episodes in states between the starting and goal line allowed them to reach the goal line.

We generally saw no clear pattern indicating which strategies work best in which scenarios, except for the novelty strategy on the Maze map. However, most of the best-performing DSMC ESR agents utilized distance limits larger than the single cell limit, which enabled them to select pseudo initial states across all map areas. This suggests that on the Racetrack benchmark, the DSMC ESR algorithms are not sensitive to the selected pseudo initial state selection strategy when given a sufficiently large distance limit.

We were unable to attain satisfactory performances in the PPO-based experiments, which can be attributed to PPO learning significantly slower than DQN and DQNPR in the Racetrack task. However, we observed that the DSMC ESR agents explored the maps significantly more than the PPO agents, indicating that given more training time, the DSMC ESR agents would attain suitable performance faster than the PPO agents.

Lastly, we examined the locations of the pseudo initial states selected by DSMC ESR agents that utilized the single cell limit. This showed us that the qualification measures could significantly impact the pseudo initial state selection. We observed that the random strategy covered all map areas with a tendency for frequently visited areas. The crash and value strategies selected states that were located on the direct path between the starting and goal lines. This suggests that fatal mistakes and significant drops in state-values occur when agents deviate from these paths. The novelty strategy selected states almost oppositely since its pseudo initial states were concentrated in the most infrequently visited map areas.

7. Related Work

Collecting promising states and restarting episodes in them is a paradigm that has been proven to be successful in DRL by the *Go-Explore* algorithms [9], which achieved state-of-the-art performance on several hard-exploration Atari benchmarks. The authors hypothesize that two main factors hinder an agent's exploration and, thus, its performance: Detachment occurs when an agent stops exploring promising state space regions too early, which may cause it to forget how to return to them at later time steps. Derailment occurs when agents fail to return to promising states because exploration techniques, such as ϵ-greedy policies or entropy bonuses, induce randomness in their decision-making. Go-Explore addresses these issues by strictly separating the returning to states from exploring the state space. Starting with the initial states, Go-Explore iteratively builds an archive of visited states suitable for exploration. Returning to an archived state is then achieved by resetting the environment to that state, allowing the agent to begin exploring. Thus, promising states will not be forgotten and can always be returned to, which avoids detachment and derailment.

In Figure 7.1, we see a schematic of the basic Go-Explore algorithm. Each exploration phase begins by sampling an archived state with a probability inversely proportional to how often similar states have been visited in previous exploration phases. Afterward, the algorithm returns to the selected state by resetting the environment, and the agent begins exploring by taking random actions until the episode is finished. To update the archive with the encountered states, Go-Explore maps each state to a lower-dimensional cell representation such that only one state of each cell can be stored in the archive. This is necessary because non-trivial tasks have gigantic state spaces, making storing all states visited throughout the training intractable. If the archive entry of a cell already stores a state, the new state replaces the old state only if it was encountered in a better-performing trajectory than the old state. If no archive entry exists for a visited state's cell, a new entry is added, and the corresponding state is stored. The exploration phase then repeats for several iterations, after which the robustification phase begins. In this phase, the best-performing trajectories encountered during the exploration phase are used to update the policy. To ensure the robustness of the policy, the authors add stochasticity to the environment and train the agent on the best-performing trajectories T using the backward algorithm [27]. This algorithm trains the agent to reach each trajectory's goal state $s_{|T|} \in T$ from every other state $s_i \in T$. Starting at $s_{|T|-1}$, the agent is trained to reach $s_{|T|}$ from s_i until it achieves suitable performance. This process repeats for s_{i-1} until the agent can reliably reach $s_{|T|}$ from s_0. Once robustification is completed, the next exploration phases begin.

In environments that do not allow resetting to an arbitrary state, we can use policy-based Go-Explore, which enables returning to states, in step 2 of the exploration phase, by using a policy that is conditioned on guiding the agent towards them. Once the agent

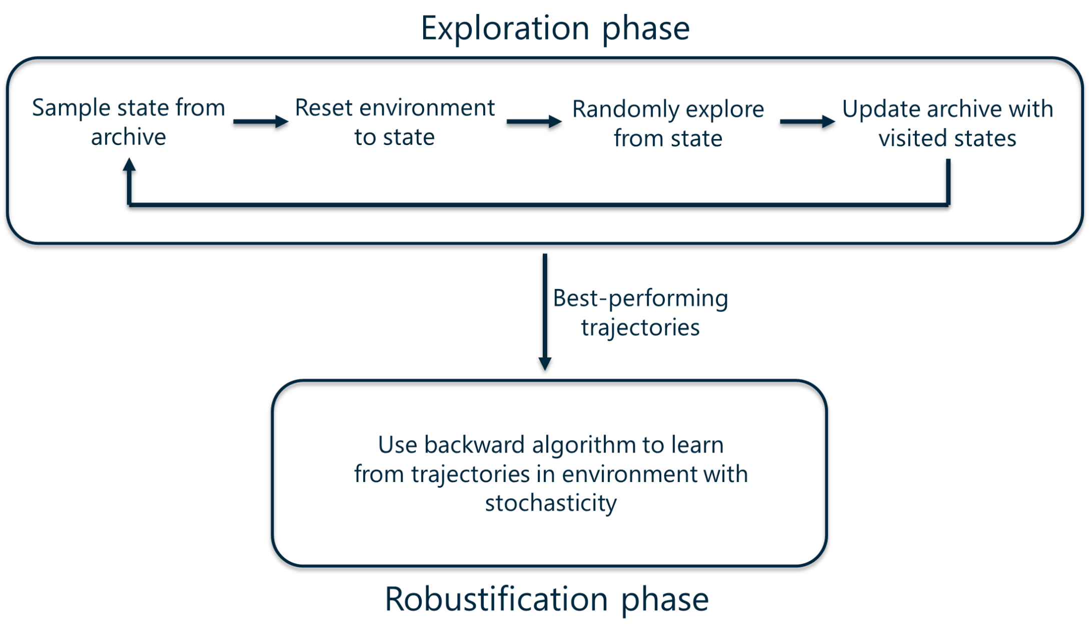

Exploration phase
Sample state from archive
Reset environment to state
Randomly explore from state
Update archive with visited states
Best-performing trajectories
Use backward algorithm to learn from trajectories in environment with stochasticity
Robustification phase

Figure 7.1 **Diagram of the Go-Explore algorithm.**

reaches a state with the same cell representation as the desired state, the exploration begins by conditioning the policy on reaching a state with a new cell representation. The authors find that this policy-based exploration is superior to taking random actions since, in their experiments, the agents discover significantly more states with unique cell representations than the regular Go-Explore agents. The policy-based exploration also allows using the gathered transitions to update the policy using a DRL algorithm, in their case PPO, eliminating the need for a computationally expensive robustification phase.

Although we developed DSMC ESR to increase safe behavior and not to increase exploration, it shares many similarities with Go-Explore, making it interesting to compare both methods. DSMC ESR starts episodes in previously encountered states like the regular Go-Explore algorithm but then explores from its starting point according to its policy, similar to policy-based Go-Explore. However, DSMC ESR does not explore using a conditioned policy. This means our method's agents do not explicitly try to find new state space regions when episodes are restarted but focus on reaching a goal state. Since DSMC ESR collects states from visited state space regions, we can still achieve significant exploration by utilizing large distance limits, such that the pseudo initial states are distributed across all visited regions, and by using the novelty strategy, meaning we restart episodes in unexplored regions.

Instead of building an archive of all visited states like Go-Explore, DSMC ESR computes a set of promising states between each evaluation stage in which later episodes will restart. Thus, DSMC ESR may suffer from detachment, i.e., forgetting how to return to promising states, since only the most recent pseudo initial states are stored. However, we mitigate this problem by reusing pseudo initial states from previous iterations if the agent could only discover fewer than B new candidates or if old pseudo initial states were stored among the B most qualified new candidates. Go-Explore avoids excessive memory requirements by archiving only one state for each cell representation. DSMC ESR achieves this by clearing the set of stored candidates C after every evaluation stage and by limiting the number of pseudo initial states to B.

Whereas Go-Explore restarts episodes in the least frequently visited archived states, the DSMC ESR algorithm EID^P selects episodes' initial states according to how poor the agent's behavior is in the corresponding initial state regions. We discussed that agents tend to perform badly in unexplored state space regions, making it likely that EID^P also restarts episodes in rarely visited states.

8. Conclusion

In this thesis, we developed an extension of the DSMC ES algorithms, called DSMC ESR, to make their performance independent of the given task's set of initial states.

We showed that using the DSMC ES algorithms is only sensible if the given task's initial states $\mathcal{I}$ cover a significant portion of the state space. Otherwise, the algorithms will fail to focus the training on the state space regions where deficient behaviors occur.

We addressed the shortcomings of DSMC ES by developing the DSMC ESR algorithms. To do so, we relied on the concept of restarting episodes in previously encountered states, which we called pseudo initial states. This allowed us to focus the training on state space regions where the agent behaved unsafely. We made the utilization of pseudo initial states proportional to the agent's performance in the initial states $\mathcal{I}$, so the training was only focused on pseudo initial states to the extent that it benefited the performance in $\mathcal{I}$. Furthermore, we introduced four pseudo initial state selection strategies, which gave us additional control over which state space regions the training was focused on. The random strategy selected states randomly, providing the agent with a diverse set of pseudo initial states. The crash strategy selected states where the agent made fatal decisions such that it learned to avoid them. The value strategy selected states where the agent deviated from paths of high return so that it learned to stay on them. The novelty strategy selected the most novel states to increase the agent's exploration.

Lastly, we evaluated the results of training DSMC ESR agents on the popular Racetrack benchmark. We found that the EID_G^P / EPR_G^P algorithms can achieve higher GRPs in the initial states $\mathcal{I}$ and that the EID_R^P / EPR_R^P algorithms can achieve higher expected returns with lower variances in $\mathcal{I}$ compared to DQN and DQNPR. We followed that these performance increases were likely due to the DSMC ESR agents being able to drive in many more map areas than the DQN and DQNPR agents. Furthermore, we found that the performance of the DSMC ESR algorithms was not sensitive with regard to the utilized pseudo initial state section strategy, given that we enforced sufficient distance between the pseudo initial states. Using PPO as the basis of EID_G^P showed how PPO learned significantly slower than DQN and DQNPR on the Racetrack benchmark. However, our methods could significantly increase exploration compared to the regular PPO algorithm. Additionally, we compared in which map areas the pseudo initial state selection strategies selected pseudo initial states to gain insights into how they could affect an DSMC ESR agent's training.

8.1. Future Work

Since we developed DSMC ESR intending to make it applicable to any task, the logical next step is to test it on other benchmarks and investigate whether it can achieve the same performance improvements as in Racetrack.

We used in our work a fixed minimum distance limit to ensure that the pseudo initial states cover a sufficient portion of the state space. However, gradually decreasing the distance limit throughout the training might be advantageous. This means that at the beginning of the training, we would focus on pseudo initial states that cover large parts of the state space, and after sufficient exploration, we would focus on only the most qualified pseudo initial states.

Another possibility for future work is the development of further pseudo initial state selection strategies. A promising approach would be to select states according to the policy's uncertainty, which quantifies how uncertain the agent is about which action to take in a given state.

One limitation of the DSMC ESR algorithms is that in tasks with high-dimensional state representations, like RGB images, we might encounter memory issues when storing visited states as candidates $c \in C$, despite clearing C after every evaluation stage. Thus, a valuable addition to DSMC ESR might be to store candidates using a cell-based archive as in Go-Explore [9].

Bibliography

[1] Joshua Achiam and Shankar Sastry. Surprise-based intrinsic motivation for deep reinforcement learning, 2017.

[2] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J. Santamaría, Mohammed A. Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1):53, Mar 2021.

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, nov 2017.

[4] Christel Baier, Maria Christakis, Timo P. Gros, David Groß, Stefan Gumhold, Holger Hermanns, Jörg Hoffmann, and Michaela Klauck. Lab conditions for research on explainable automated decisions. In Fredrik Heintz, Michela Milano, and Barry O'Sullivan, editors, *Trustworthy AI - Integrating Learning, Optimization and Reasoning*, pages 83–90, Cham, 2021. Springer International Publishing.

[5] Josh Beitelspacher, Jason Fager, Greg Henriques, and Amy Mcgovern. Policy gradient vs. value function approximation: A reinforcement learning shootout. 03 2006.

[6] Richard Bellman. Dynamic programming. *Princeton University Press*, 1957.

[7] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation, 2018.

[8] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de las Casas, Craig Donner, Leslie Fritz, Cristian Galperti, Andrea Huber, James Keeling, Maria Tsimpoukelli, Jackie Kay, Antoine Merle, Jean-Marc Moret, Seb Noury, Federico Pesamosca, David Pfau, Olivier Sauter, Cristian Sommariva, Stefano Coda, Basil Duval, Ambrogio Fasoli, Pushmeet Kohli, Koray Kavukcuoglu, Demis Hassabis, and Martin Riedmiller. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, Feb 2022.

[9] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021.

[10] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-

Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, Oct 2022.

[11] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[12] Martin Gardner. Mathematical games. *Scientific American*, 229:104–109, 1973.

[13] Timo P. Gros. Tracking the race: Analyzing racetrack agents trained with imitation learning and deep reinforcement learning. Master's thesis, Saarland University, 2021.

[14] Timo P. Gros, David Groß, Stefan Gumhold, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Tracevis: Towards visualization for deep statistical model checking. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*, pages 27–46, Cham, 2021. Springer International Publishing.

[15] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep statistical model checking. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 96–114, Cham, 2020. Springer International Publishing.

[16] Timo P. Gros, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. Dsmc evaluation stages: Fostering robust and safe behavior in deep reinforcement learning. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems*, pages 197–216, Cham, 2021. Springer International Publishing.

[17] Timo P. Gros, Daniel Höller, Jörg Hoffmann, and Verena Wolf. Tracking the race between deep reinforcement learning and imitation learning - extended version. *CoRR*, abs/2008.00766, 2020.

[18] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 73–84, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[20] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Px00E9;rez. Deep reinforcement learning for

autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2022.

[21] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015.

[22] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[24] Georg Ostrovski, Marc G. Bellemare, Aaron van den Oord, and Remi Munos. Count-based exploration with neural density models, 2017.

[25] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A generalist agent, 2022.

[26] Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 2007.

[27] Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration, 2018.

[28] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

[29] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, dec 2020.

[30] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

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

[33] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[34] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[35] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, Jun 2015.

Appendices

A. Appendix

A.1. Hyperparameters

Hyperparameters that are used in multiple algorithms but only have one table entry have the same value in all instances.

Parameter	Description	Value

DQN:

E	Number of episodes	100.000
T	Maximum episode length	100
γ	Discount factor	0.99
K	Q-network update frequency	4
	Batch size	512
τ	Soft update coefficient	0.001
ϵ_{start}	Initial exploration coefficient of ϵ-greedy policy	1
ϵ_{decay}	Decay factor of ϵ in each episode	0.999
ϵ_{end}	Value of ϵ at the end of the training	0.05
$\|D\|$	Size of replay buffer	10^8
α_{Adam}	Learning rate of Adam optimizer	$8 \cdot 10^{-4}$
s	Seeds used for each agent type	$0, 1, 2, 3, 4$

DQNPR:

α	Prioritization coefficient for sampling priorities	1
ϵ_p	Minimum priority	10^{-6}

PPO:

E	Number of episodes	400.000
K	Batch size	4
M	Number of epochs	4
ϵ_{clip}	Clipping range	0.2
α_{Actor}	Learning rate of actor	$5 \cdot 10^{-4}$, $8 \cdot 10^{-4}$
α_{Critic}	Learning rate of critic	10^{-3}
c_2	Entropy weight	10^{-3}, $4 \cdot 10^{-3}$
λ	Bias-variance trade-off parameter in GAE	0.9

EID$_G^p$/ EPR$_G^p$:

W	Number of pre-training episodes	10.000
L	Original initial state region evaluation frequency	100
U	Pseudo initial state region evaluation frequency	10.000
ψ_{min}	Lower pseudo initial state region priority bound	0.2, 0.4 (PPO)
ψ_{max}	Upper pseudo initial state region priority bound	0.2
B	Maximum size of P_j	$\frac{\lvert\text{free map cells}\rvert}{4}$
ϵ_p	Minimum priority	0.2
ϵ_{err}	Error in DSMC's evaluation	0.05
κ	Probability that DSMC's error is at most ϵ_{err}	0.05
n	Crash strategy n	5

EID$_R^p$/ EPR$_R^p$:

L	Original initial state region evaluation frequency	500
ϵ_{err}	Error in DSMC's evaluation	4
s	Seeds used for each agent type	$0, 1, 2$

A.2. Configurations of the Best-performing Agents

Strategy	Distance limit	Map

EID$^{P}_{G}$:

Random	Max distance	River-deadend
Crash	Regions 3 × 3	
Value	Single cell	
Novelty	Regions 3 × 3	
Random	Single cell	Maze
Crash	Regions 3 × 3	
Value	Max distance	
Novelty	Max distance	
Random	Regions 2 × 2	Hansen-bigger
Crash	Single cell	
Value	Regions 3 × 3	
Novelty	Regions 3 × 3	

EPR$^{P}_{G}$:

Random	Max distance	River-deadend
Crash	Single cell	
Value	Single cell	
Novelty	Max distance	
Random	Max distance	Maze
Crash	Regions 2 × 2	
Value	Max distance	
Novelty	Max distance	
Random	Regions 2 × 2	Hansen-bigger
Crash	Single cell	
Value	Max distance	
Novelty	Max distance	

Strategy	Distance limit	Map

EID$_R^P$:

Random	Regions 3×3	River-deadend
Crash	Single cell	
Value	Max distance	
Novelty	Regions 2×2	
Random	Regions 2×2	Maze
Crash	Max distance	
Value	Regions 3×3	
Novelty	Regions 3×3	
Random	Single cell	Hansen-bigger
Crash	Regions 2×2	
Value	Regions 3×3	
Novelty	Regions 3×3	

EPR$_R^P$:

Random	Regions 3×3	River-deadend
Crash	Max distance	
Value	Regions 3×3	
Novelty	Regions 3×3	
Random	Regions 3×3	Maze
Crash	Regions 2×2	
Value	Regions 2×2	
Novelty	Regions 2×2	
Random	Regions 3×3	Hansen-bigger
Crash	Max distance	
Value	Regions 2×2	
Novelty	Max distance	

Strategy	Distance limit	Map

EID$_G^P$ with PPO as basis:

Random	Single cell	River-deadend
Crash	Single cell	
Value	Regions 2×2	
Novelty	Regions 2×2	
Random	Regions 2×2	Maze
Crash	Regions 2×2	
Value	Regions 3×3	
Novelty	Regions 2×2	
Random	Single cell	Hansen-bigger
Crash	Single cell	
Value	Max distance	
Novelty	Max distance	

Strategy/Baseline	Learning rate	Entropy coefficient	Map
PPO	$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	River-deadend
Random	$8 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	
Crash	$8 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	
Value	$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	
Novelty	$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	
PPO	$5 \cdot 10^{-4}$	$4 \cdot 10^{-3}$	Maze
Random	$5 \cdot 10^{-4}$	$4 \cdot 10^{-3}$	
Crash	$5 \cdot 10^{-4}$	$4 \cdot 10^{-3}$	
Value	$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	
Novelty	$8 \cdot 10^{-4}$	$4 \cdot 10^{-3}$	
PPO	$5 \cdot 10^{-4}$	$4 \cdot 10^{-3}$	Hansen-bigger
Random	$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	
Crash	$8 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	
Value	$8 \cdot 10^{-4}$	$4 \cdot 10^{-3}$	
Novelty	$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	