Change of Plans!
Adaptive AlphaZero Planning Methods for Novel Test Environments

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Isidoro Tamassia
born in Rome, Italy

Sequential Decision Making Group
Department of Intelligent Systems
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Change of Plans!
Adaptive AlphaZero Planning Methods for Novel Test Environments

Author: Isidoro Tamassia
Student id: 6054765

Abstract

AlphaZero and its successors employ learned value and policy functions to enable more efficient and effective planning at deployment. A standard assumption is that the agent will be deployed in the same environment where these estimators were trained; changes to the environment would otherwise violate their expectations and could result in suboptimal decisions. In this work, we investigate how environment changes affect the usability of the learned estimators and develop criteria that can quickly detect and localize such changes. Moreover, we develop novel planning methods that leverage these principles as well as further modifications of standard Monte Carlo planning techniques. These methods demonstrate superior performance under several tested environment configurations. The main assumptions and limitations of our approaches are also discussed, providing a foundation for future research to broaden their applicability. The code is available on GitHub[1].

Thesis Committee:

University supervisor: Prof. Dr. Wendelin Böhmer, Sequential Decision Making, TU Delft
Committee Member: Prof. Dr. Anna Lukina, Algorithmics, TU Delft

[1]`https://github.com/TheEmotionalProgrammer/alphazero-vs-env-changes`

Preface

It feels bittersweet to reach the end of this adventure, made unforgettable by the exceptional people I have had the fortune to meet along the way.

I would first like to thank my thesis supervisor, Prof. Wendelin Böhmer, for providing invaluable guidance and feedback throughout the development of this project, as well as introducing me to the field of Deep Reinforcement Learning with his MSc course. I would also like to thank PhD student Max Weltevrede for reviewing earlier drafts of this manuscript and for the many fruitful discussions we had during our meetings.

My heartfelt thanks go to all the amazing friends I met in Delft, who made this experience so much more than just studying. I am especially grateful to Roberto for all the tips and support provided during my first year in Delft, as well as for being a great teammate during the many group projects we worked on together. I am also grateful to my partner, Sabrina, for listening to my daily complaints about pretty much anything, as well as suggesting interesting algorithmic ideas that were unfortunately too creative to be practically implemented in this thesis.

Finally, I would like to express my deepest gratitude to my family for all the remote support they have provided over the past two years, particularly to Angela and Alessandro, the best ”life supervisors” a son could ever hope for.

Isidoro Tamassia
Delft, the Netherlands
June 12, 2025

Contents

List of Figures

Chapter 1

Introduction

Over the last two decades, Reinforcement Learning (RL) has led to significant breakthroughs in several domains of application, ranging from dominating the landscape of chess-playing programs to enabling significant advancements in control and robotics. This was probably hardly imaginable in the past if looking back at the way it all started, with tabular reinforcement learning methods that, despite being theoretically sound, could never scale to real-world applications due to their inherently large or infinite state and action spaces. The advent of neural networks, however, completely changed the perspectives of both model-free methods, which teach an agent to act in an environment by directly interacting with it, and model-based methods, which rely on a given or learned model of the environment to plan ahead.

Two major contributions can be identified as the true turning points for model-free RL and model-based RL, respectively. In 2013, a team of research scientists from DeepMind (today's Google DeepMind) provided the first successful, end-to-end integration of a neural network architecture and a model-free reinforcement learning framework [21], which learned to play Atari games with a human-comparable performance. This is considered the actual start of the whole field of research now referred to as Deep Reinforcement Learning.

A few years later, in 2015, the same company released a computer program called AlphaGo [29], which achieved superhuman performance in the game of Go by combining supervised learning, reinforcement learning, Monte Carlo Tree Search [8], and game-specific heuristics. In a series of subsequent papers [31] [30], this was turned into a general, model-based RL algorithm called AlphaZero, able to reach superhuman performance in Chess, Shogi, and Go with the sole knowledge of the rules, i.e., a perfect simulator of the corresponding environment. AlphaZero trains a neural network that provides a prior policy and value function, which are used to guide planning at deployment. This was pushed even further in 2020 with the development of the MuZero algorithm [28], which learns both an internal representation of the environment and a dynamics model in that latent space, enabling planning without access to the true environment dynamics. This algorithm not only matched the extraordinary results of its predecessors in two-player games, but also achieved state-of-the-art results on the aforementioned Atari games suite, where model-based algorithms had previously struggled. As of today, an active branch of RL research focuses on improving AlphaZero and MuZero to address increasingly complex challenges, as well as reducing the substantial computational resources required to train the associated neural networks.

A potential issue is that AlphaZero-like algorithms have been mostly experimented on environments whose dynamics do not change over time. This implies that the neural network is still reliable at deployment if properly trained beforehand. In reality, many tasks that an agent is required to perform might change from time to time; this is often the case for robots, which can be trained in controlled or simulated environments to perform a variety of tasks, but once deployed in the real world usually face discrepancies, such as minor changes in layout, lighting, or object positioning, that were not present during training. For example, a household robot trained to navigate an apartment may suddenly encounter a chair that has been moved or a new piece of furniture, requiring it to adapt its behavior despite having never seen this configuration before. Similarly, a self-driving car trained on traffic patterns and road layouts in a specific area may encounter unexpected obstacles, such as a temporary construction site, a road closure, or a newly installed traffic signal. These changes can significantly impact the agent's ability to act safely and effectively if they are not adequately accounted for. Model-free RL can struggle in this scenario, but if we are given a model of the test environment, we may be able to check whether the predictions of the learned policy and value still hold and use this information to influence the agent's decisions.

This poses some questions: can previously learned policy and value functions still be used to guide a planning algorithm at deployment in a partially different environment? When is a change so disruptive that we would need to re-train the network? In principle, standard AlphaZero may compensate for such changes if we can simulate for a long time since it integrates the rewards observed while planning at deployment. However, in settings where our planning time per step is limited or rewards are sparse, this may not be sufficient for the agent to realize early enough that its priors are inaccurate. In other words, our self-driving car might come too close to an obstacle that was not present during training and realize it needs to steer when it is too late to change direction.

Our work tries to answer the following research questions:

1. *How can we detect changes to the environment at deployment when planning with learned estimators?*
2. *How can we leverage such detection principles to re-plan around these changes and subsequently overcome them?*
3. *What standard features of AlphaZero planning might be modified or improved to better plan and act in a potentially changed environment?*

We address question 1 with an analysis provided in chapter 3. We address questions 2 and 3 with the novel algorithms proposed in chapter 4 and conduct associated experiments reported in chapter 7.

1.1 Contributions

Our work includes several scientific contributions, summarized as follows:

- We provide an analysis of the problems raised by planning with learned estimators (value, policy) when deployed in a changed environment, with a particular focus on the AlphaZero algorithm.
- We develop novel analytical criteria aimed at detecting and localizing detrimental changes to the test environment when planning with learned policy and value functions.
- We develop a novel family of planning algorithms named AlphaZeroDetection (AZD), leveraging our detection criteria to localize problems far into the future and explore alternative action paths aimed at overcoming detrimental changes to the environment.
- We develop the Penalty-Driven Deep Planning algorithm (PDDP), which integrates the detection criteria directly into standard AlphaZero tree construction and penalizes changed paths by decreasing the corresponding value estimates.
- We identify several weaknesses of standard AlphaZero planning that can particularly damage performance in changed environments and propose simple modifications that dramatically boost it, even without relying on detection mechanisms. We implement these changes in the Extra-Deep Planning algorithm (EDP).
- We suggest several directions for future work, which could broaden the applicability of our methods by relaxing some of the main assumptions.
- We provide an open-source implementation of all the mentioned algorithms, integrated into a general-purpose AlphaZero framework.

1.2 Outline

The remainder of this manuscript is structured as follows:

- In chapter 2 we provide the necessary background for understanding the work presented in this thesis.
- In chapter 3 we provide an analysis of the problem of detecting changes to the environment. We present methods for identifying and localizing such changes, and investigate the error that we might commit depending on the employed assumptions.
- In chapter 4 we describe our novel planning methods.
- In chapter 5 we connect our work to existing literature, examining how our work aligns with and diverges from previous approaches.
- In chapter 6 we describe our training and test setups.

- In chapter 7 we present our main results and highlight the most prominent benefits and drawbacks of our novel algorithms.
- In chapter 8 we further discuss the outcomes of our analysis and experiments, as well as highlighting potential limitations.
- In chapter 9 we draw our conclusions and suggest some promising directions for future work.
- In Appendix A, we provide the implementation details of our training and evaluation frameworks, including the hyperparameters used.
- In Appendix B we provide results from complementary experiments, such as extensive evaluation and hyperparameter tuning of the employed baselines.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that focuses on how agents take actions in an environment to maximize cumulative rewards. It is fundamentally based on the concept of learning through interaction with an environment by receiving feedback in the form of rewards or penalties.

Mathematically, an RL environment is usually framed as a Markov Decision Process (MDP) [5], defined by the tuple:

$$(\mathcal{S}, \rho, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$$

where:

- $\mathcal{S}$ is the set of states the agent can occupy.
- ρ is the initial states distribution.
- $\mathcal{A}$ is the set of possible actions the agent can take.
- $\mathcal{P}(s'|s, a)$ is the transition probability, defining the likelihood of moving to state s' after taking action a in state s.
- $\mathcal{R}(s, a, s')$ is the reward distribution, which provides feedback to the agent depending on the action a taken in state s, and on the resulting next state s'.
- $\gamma \in (0, 1]$ is the discount factor, which determines the importance of future rewards.

The MDP formulation comprises several fundamental assumptions:

- **Discrete time**: MDPs assume that decision-making occurs at discrete time steps, indexed as $t = 0, 1, 2, \ldots$. This contrasts with continuous-time models, where transitions and decisions can happen at any instant.

- **Stationarity**: MDPs assume that the transition probability $\mathcal{P}(s'|s, a)$ and the reward function $\mathcal{R}(s, a, s')$ do not change over time.
- **Markov Property**: the future state distribution depends only on the current state and action, i.e. $\mathcal{P}(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) = \mathcal{P}(s_{t+1} \mid s_t, a_t)$.
- **Full observability**: the current agent observation corresponds exactly to the current state. This assumption can be relaxed by employing a modified version of an MDP, known as a Partially Observable Markov Decision Process (POMDP) [3]. In a POMDP, the agent no longer has access to the full state but instead receives observations that provide partial information about the true underlying state. This introduces an additional layer of complexity, as the agent must maintain a belief over possible states based on its observation history.

Moreover, we define the expected return, that is, the expected cumulative reward, obtained by starting from state s and acting under policy π as:

$$V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, \substack{r_t \sim \mathcal{R}(s_t, a_t, s_{t+1}) \\ s_{t+1} \sim \mathcal{P}(s_t, a_t) \\ a_t \sim \pi(s_t) \\ s_0 = s}\right]$$

This is commonly referred to as value function.

The goal of reinforcement learning is to find an optimal policy π^*, which maps states to actions in a way that maximizes the expected cumulative reward:

$$\pi^*(s) = \arg\max_\pi V_\pi(s),$$

We can further define the action-value function (or Q-function), which represents the expected return obtained by starting from state s, taking action a, and thereafter following policy π:

$$Q_\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, \substack{r_t \sim \mathcal{R}(s_t, a_t, s_{t+1}) \\ s_{t+1} \sim \mathcal{P}(s_t, a_t) \\ a_t \sim \pi(s_t) \\ s_0 = s,\, a_0 = a}\right]$$

The function $Q_\pi(s, a)$ quantifies the expected return for executing action a in state s and subsequently following policy π. It can also be expressed via the following relationship, known as one of the Bellman equations [4]:

$$Q_\pi(s, a) = \mathbb{E}\left[r_t + \gamma V_\pi(s_{t+1}) \,\middle|\, \substack{r_t \sim \mathcal{R}(s, a, s_{t+1}) \\ s_{t+1} \sim \mathcal{P}(s, a)}\right]$$

This allows us to treat RL problems as step-by-step optimization problems.

Note that this formulation enables the proper modeling of stochastic environments. The transition dynamics and reward function can be simplified under the assumption of a deterministic environment. Specifically:

- The transition probability function $\mathcal{P}(s'|s, a)$ becomes a deterministic mapping, meaning that for each state-action pair (s, a), there is a unique next state s'. This is often expressed as a deterministic transition function $f : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, where $s' = f(s, a)$.
- The reward distribution $\mathcal{R}(s', s, a)$ also becomes a deterministic function $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ where $r_i = R(s_i, a_i)$.

In the remainder of this thesis, we will primarily focus on deterministic environments, and therefore, the simplifications above apply.

Although a wide variety of RL algorithms exist, they can generally be classified into the following two categories:

2.1.1 Model-Free RL Methods

These methods do not assume knowledge of the transition dynamics $\mathcal{P}(s'|s, a)$ and let the agent learn by directly interacting with the environment. They are often divided into two subclasses of methods:

- **Value-based methods**: These methods learn a value function, such as Q-learning [36] and Deep Q-Networks (DQN) [21]. They then extract a policy from the learned value.
- **Policy-based methods**: These methods learn a policy directly, such as REINFORCE [37] and Actor-Critic methods [16].

One of the main advantages of model-free RL is that it avoids model bias and error, thereby improving the performance and robustness of the agent. By learning directly from experience, the agent can adapt to the environment without depending on a potentially flawed or incomplete model. Additionally, model-free RL is often simpler to implement since it focuses solely on learning a value function or policy without the need to design and update a model.

2.1.2 Model-Based RL Methods

Model-based approaches attempt to learn the transition dynamics or use a given transition model to plan ahead. Examples include Monte Carlo Tree Search [8] (given model), AlphaZero [30] (given model), and MuZero [28] (learned model).

The primary advantage of model-based RL methods is that they can simulate and evaluate potential actions without requiring direct interaction with the environment. This capability allows for learning effective policies with fewer real-world trials, often increasing sample efficiency. However, these methods can be computationally intensive due to the complexity of learning and maintaining accurate models, which may limit their applicability in certain scenarios.

Given the particular relevance of Monte Carlo Tree Search and AlphaZero in the context of this thesis, we provide further information in section 2.2 and section 2.3, respectively.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an online search algorithm used for decision-making in large or infinite state spaces. The term and original algorithm were coined by Rèmi Coulom [8] in 2006 as a way to apply Monte-Carlo methods to game tree search. In the same year, Kocsis and Szepesvári [15] first proposed the closely related UCT algorithm (Upper Confidence bounds for Trees), which applied multi-armed bandit theory to guide Monte-Carlo tree exploration. In the following years, MCTS gained extensive popularity as a game-playing algorithm, particularly for its remarkable results in the domain of Go [7].

Algorithm 1: MCTS Planning

```
Input: Environment model $env$, planning budget $N$, rollout budget $n$, discount
       factor $\gamma$, selection policy $\pi_{\text{sel}}$
Output: Root node $root$ with updated statistics
1  Initialize $root \leftarrow Node(env)$
2  while $root.visits < N$ do
3      $expNode, expAction \leftarrow$ Selection($root, \pi_{\text{sel}}$)
4      if $expNode.terminal$ then
5          Backup($expNode, \gamma$)
6      else
7          $child \leftarrow$ Expand($expNode, expAction$)
8          $child.value \leftarrow$ Rollout($child, n, \gamma$)
9          Backup($child, \gamma$)
10 return $root$
```

The key idea behind MCTS is to incrementally build a search tree by performing simulations and using the results to refine future decisions. In this context, a search tree is a tree where each node represents a unique action sequence started from the root, corresponding to the current state in the real environment. This formulation avoids inconsistencies that could arise if we directly mapped nodes to states, as different action sequences can lead to the same state, but the nodes will still be distinguished. Note that the available simulator might not perfectly reflect the real environment, i.e., it is not always assumed that we know the true transition and reward models; however, an extensive amount of literature does in practice make this assumption, as this is true, for instance, in games such as chess, go, and shogi, which have been the most common benchmark for MCTS-based algorithms. Moreover, standard MCTS can only be applied in discrete action spaces, which simplifies some of the considerations reported in the rest of this section. Nonetheless, modern approaches adapting MCTS to continuous action spaces exist in the literature (e.g., [39], [13], [19]).

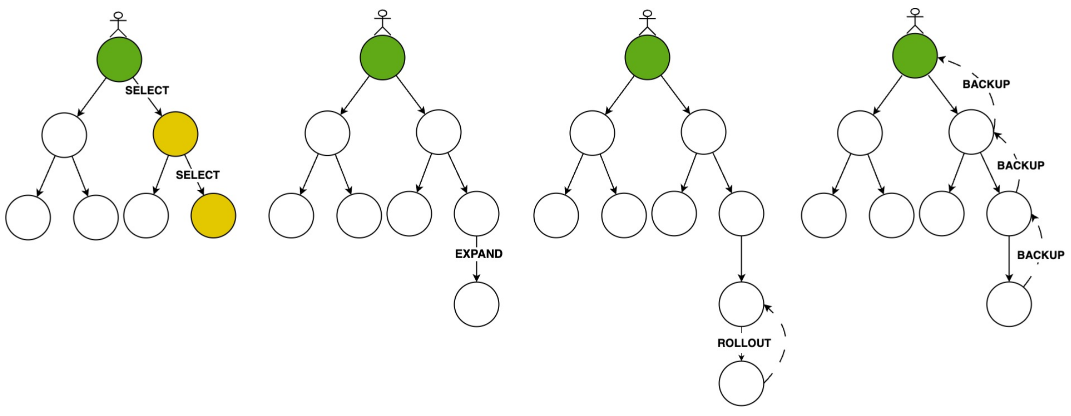


Figure 2.1: A single iteration of MCTS Planning, from left to right: the agent selects a path using its selection policy and reaching a not-fully-expanded node, expands a new child, estimates its value with a random rollout, and finally backpropagates the value through the path.

MCTS operates in four main phases, which are repeated for a fixed number of iterations, usually referred to as the planning budget, in the loop described in algorithm 1.

1. **Selection:** Starting from the root node, child nodes are recursively selected based on a selection policy until a node that has not been fully explored is reached. This phase is detailed in algorithm 2.
2. **Expansion:** If the selected node is not a terminal state, a child node is added to the tree, usually uniformly at random. This phase is detailed in algorithm 3.
3. **Simulation (Rollout):** A simulation is performed from the newly added node using a default policy (often random) until a terminal state is reached, in order to estimate its value. This phase is detailed in algorithm 4.
4. **Backpropagation:** The outcome of the simulation is propagated back through the tree to update the value estimates of the visited nodes along the selected trajectory. This phase is detailed in algorithm 5.

A sequential visualization of these phases is reported in Figure 2.1. As mentioned, the selection phase often employs the UCT formula to balance exploration and exploitation. This policy is based on the UCT score:

$$\mathrm{UCT}(x,a) = \bar{Q}(x \uplus a) + C\sqrt{\frac{\ln(N(x))}{N(x \uplus a)}}$$

where:

- x is a node of the tree.
- $x \uplus a$ is the node that we enter by taking action a from the state in node x.

Algorithm 2: Selection: Traverse the tree using a given selection policy.

```
Input: Root node $root$, selection policy $\pi_{\text{sel}}$
Output: Node to be expanded $expNode$, action to expand it $expAction$
1 $node \leftarrow root$
2 while not $node.terminal$ do
3     Select action $a$ using selection policy $\pi_{\text{sel}}$ on $node$
4     if $a \notin keys(node.children)$ then
5         break
6     $node \leftarrow node.children\{a\}$
7 return $node, a$
```

Algorithm 3: Expand: Expand the chosen node with a selected action.

```
Input: Node to be expanded $expNode$, action to expand it $expAction$
Output: Child node
1 $env \leftarrow expNode.env.copy()$
2 $env.step(expAction)$
3 $child \leftarrow Node(env)$
4 return $child$
```

- $N(x)$ is the total visit count of node x. Note that this corresponds to $1+\sum_{a\in\mathcal{A}} N(x\uplus a)$ since we always start planning from the root.
- $\bar{Q}(x)$ is the mean cumulative reward for the subtree rooted in node x, as computed in algorithm 5. In other words, this is the arithmetic average of the discounted returns of the $N(x)$ trajectories passing through x.
- C is an exploration constant that controls the balance between exploitation and exploration.

The UCT score is turned into the utilized selection policy as:

$$\pi'_{\text{UCT}}(x,a) = \frac{\delta(a \in \arg\max_{a'\in\mathcal{A}} \text{UCT}(x,a'))}{|\arg\max_{a'\in\mathcal{A}} \text{UCT}(x,a'))|}$$

Where:

$$\delta(z) = \begin{cases} 1 & \text{if } z \quad \text{is true} \\ 0 & \text{otherwise} \end{cases}$$

In practice, if there are no ties in the UCT scores, this policy is deterministic and sampling returns the single best action according to it. Otherwise, it corresponds to sampling uniformly at random among the multiple actions that maximize the score. In the remainder of this work, we will indicate such almost-deterministic policies, including fully deterministic

Algorithm 4: Rollout: Estimate the value of the given node with a random rollout starting from the corresponding state.

```
Input: Node to be evaluated $node$, rollout budget $n$, discount factor $\gamma$
Output: Estimate of the node's value $value$
if $node.terminal$ then
    return $0$
$value \leftarrow 0$
$env \leftarrow node.env.copy()$
for $i \leftarrow 0$ to $n$ do
    Sample an action $a$ uniformly at random.
    $s, r \leftarrow env.step(a)$
    $value \leftarrow value + \gamma^i r$
    if $s.terminal$ then
        break
return $value$
```

Algorithm 5: Backup: Traverse the tree in reverse order to backup the value of the expanded node and update mean Q value estimates.

```
Input: Previously evaluated leaf node $node$, discount factor $\gamma$
$R \leftarrow node.value$
while $node$ is not none do
    $R \leftarrow \gamma \cdot R + node.r$
    $node.Rsum \leftarrow node.Rsum + R$
    $node.visits \leftarrow node.visits + 1$
    $node.Q \leftarrow \frac{node.Rsum}{node.visits}$
    $node \leftarrow node.parent$
```

ones, with a dash ($'$) for notation convenience.

Note that if an action a is not expanded, then $N(x \uplus a) = 0$, and therefore the formula would imply UCT$(x, a) = \infty$ for all such actions. In practice, we can see this as applying the selection policy only to fully expanded nodes, while if any set of actions still needs to be expanded, we pick one of them uniformly at random. Note that we could not do otherwise, as the value estimate of a node needed to compute its UCT score is only known once that node is created and the corresponding rollout is performed.

After planning, a tree evaluation policy is used to decide which action to take in the real environment based on the statistics of the constructed tree. This policy is applied to the root node and is typically deterministic, as we aim to select the best possible action now that we must perform it in the real environment. A typical approach used in conjunction with UCT is to employ a visitation counts policy that simply picks the action whose corresponding child has been visited the most during planning, formally defined as follows:

$$\pi'_N(x,a) = \frac{\delta(a \in \arg\max_{a' \in \mathcal{A}} N(x \uplus a'))}{|\arg\max_{a' \in \mathcal{A}} N(x \uplus a'))|} \tag{2.1}$$

It is also possible to sample an action from a stochastic version of the policy:

$$\pi_N(x,a) = \frac{N(x \uplus a)}{N(x)} \tag{2.2}$$

2.3 AlphaZero

AlphaZero (AZ) is a framework that combines neural networks (NNs) with an MCTS variant (AZMCTS), which gained extensive popularity due to its success in mastering complex games like Chess, Shogi, and Go, without relying on human knowledge. Unlike its predecessor AlphaGo, which incorporated expert data and domain-specific heuristics, AlphaZero learned entirely from self-play, making it more generalizable. When applied to single-agent domains, the self-play mechanism reduces to simply sampling trajectories from the environment.

2.3.1 Neural Network Architecture

The core of AZ is a deep neural network $f_\theta : \mathcal{S} \to \Delta^{|\mathcal{A}|-1} \times \mathbb{R}$ parameterized by θ, which takes a state s as an input and outputs a policy distribution $\pi_\theta \in \Delta^{|\mathcal{A}|-1}$ [1] and a value $v_\theta \in \mathbb{R}$. For notation simplicity, we will use $v_\theta(s)$ to directly indicate the value-head NN output given input s, $\pi_\theta(s)$ for the policy-head NN output, and $\pi_\theta(s,a)$ to indicate the probability of taking action a according to the output distribution $\pi_\theta(s)$.

2.3.2 Planning

AZ employs AZMCTS (algorithm 6) to guide action selection and improve the policy. The first difference with standard MCTS is the way we estimate the value of leaf nodes. While MCTS employs policy rollouts, AZMCTS uses the learned values by calling the neural network.

The second difference is that at each simulation step, AZMCTS selects actions according to the PUCT formula, which incorporates the neural policy outputs as a prior policy:

$$\text{PUCT}(x,a) = \bar{Q}(x \uplus a) + C\,\pi_\theta(x,a)\,\frac{\sqrt{N(x)}}{1 + N(x \uplus a)}$$

[1]The notation $\Delta^{|\mathcal{A}|-1}$ here represents the $|\mathcal{A}|-$dimensional simplex, i.e., the set of vectors of dimensionality $|\mathcal{A}|$ whose elements sum up to one.

Algorithm 6: AZMCTS PLANNING

```
Input: Environment model $env$, planning budget $N$, rollout budget $n$, discount
       factor $\gamma$, selection policy $\pi_{\text{sel}}$, value function $v_\theta$
Output: Root node $root$ with updated statistics
1  Initialize $root \leftarrow Node(env)$
2  while $root.visits < N$ do
3      $expNode, expAction \leftarrow$ SELECTION$(root, \pi_{\text{sel}})$
4      $child \leftarrow$ EXPAND$(expNode, expAction)$
5      if child.terminal then
6          $child.value \leftarrow 0$
7      else
8          $child.value \leftarrow v_\theta(child.s)$
9      BACKUP$(child, \gamma)$
10 return $root$
```

Where $\pi_\theta(x, a) = \pi_\theta(s, a)$ with s being the state underlying node x. The corresponding selection policy can then be expressed in the same way as we did with UCT:

$$\pi'_{\text{PUCT}}(x, a) = \frac{\delta(a \in \arg\max_{a' \in \mathcal{A}} \text{PUCT}(x, a'))}{|\arg\max_{a' \in \mathcal{A}} \text{PUCT}(x, a'))|}$$

The search traverses the tree by selecting actions according to π'_{PUCT} until reaching a leaf node, at which point the neural network provides an estimated value. This value is then backpropagated to update $\bar{Q}$ exactly as in standard MCTS, i.e., as shown in algorithm 5.

It is common to incorporate Dirichlet noise $\eta \sim \text{Dir}(\alpha)$ into the prior probabilities during training to increase exploration. This is only applied at the root, allowing for more diverse trajectories to be sampled. The prior policy is then modified as follows:

$$\pi^\eta_\theta(x, a) = (1 - \epsilon)\pi_\theta(x, a) + \epsilon\, \eta,$$

where ϵ is a fixed mixing parameter. This modification is usually not applied at deployment, i.e., once the neural network parameters are fixed.

2.3.3 Training

The training loop consists of the following phases:

1. **Trajectory Collection.** Sample multiple trajectories from the environment using the evaluation policy π_{eval}. Typically, $\pi_{\text{eval}} = \pi_N$ in standard AlphaZero implementations, where π_N was previously defined in (2.1). At each step t, the planning tree rooted at the current state s_t is evaluated using AZMCTS, which returns a policy distribution

$\pi_t = \pi_{\text{eval}}(s_t)$. An action a_t is sampled from π_t and executed in the environment, resulting in a reward r_t and next state s_{t+1}.

Each trajectory $\mathcal{T}$ is stored as a tuple of sequences:

$$\mathcal{T} = \left(\{s_t\}_{t=0}^{l}, \{a_t\}_{t=0}^{l-1}, \{r_t\}_{t=0}^{l-1}, \{\pi_t\}_{t=0}^{l-1} \right)$$

where l is the final timestep of the trajectory. Each collected trajectory is then stored in a replay buffer.

2. **Network Update.** The neural network f_θ is trained via gradient descent on a minibatch of m trajectories $\mathcal{B} = (\mathcal{T}_0, ..., \mathcal{T}_m)$ sampled from the replay buffer. The objective is to minimize the AlphaZero loss:

$$L_{\text{AZ}}(\mathcal{B}) = \alpha \, L_V(\mathcal{B}) + \beta \, L_P(\mathcal{B})$$

where α, β are weighting hyperparameters, L_V is a value loss and L_P is a policy loss.

3. **Policy Evaluation.** At regular intervals, performance is evaluated using a deterministic version of the policy, specifically π'_N from (2.2) in the default AZ setting. This is done to reduce stochasticity and get a more objective understanding of how the agent is performing.

The value loss L_V is defined as the mean squared error between the predicted state values and the n-step targets, averaged over all the m trajectories in the minibatch:

$$L_V(\mathcal{B}) = \frac{1}{m} \sum_{j=1}^{m} \|\mathbf{v}_\theta(\mathcal{T}_j[s]) - \mathbf{y}_n(\mathcal{T}_j)\|^2$$

where:

- $\mathbf{v}_\theta(\mathcal{T}_j[s])$ is the vector of predicted values for each state s_t in $\mathcal{T}_j$, such that:

$$\mathbf{v}_\theta(\mathcal{T}_j[s])[t] = v_\theta(s_t)$$

- $\mathbf{y}_n(\mathcal{T}_j)$ is the vector of n-step value targets for the same states, computed as:

$$\mathbf{y}_n(\mathcal{T}_j)[t] = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n \cdot v_\theta(s_{t+n}) \cdot (1 - \text{term}(\mathcal{T}_j, t+n))$$

where $\text{term}(\mathcal{T}_j, t+n)$ is an indicator for whether the episode terminates before $t+n$:

$$\text{term}(\mathcal{T}_j, t+n) = \begin{cases} 1 & \text{if } t+n \geq l_j \\ 0 & \text{otherwise} \end{cases}$$

With l_j being the length of the j-th trajectory in the batch.

The policy loss L_P is the average cross-entropy between the policy distributions predicted by the neural network and the target distributions collected via AZMCTS:

$$L_P(\mathcal{B}) = \frac{1}{\sum_{j=1}^{m} l_j} \sum_{j=1}^{m} \sum_{t=0}^{l_j-1} \sum_{a\in\mathcal{A}} -\pi_t(a) \log \pi_\theta(s_t, a)$$

where $s_t = \mathcal{T}_j[s][t]$ and $\pi_t = \mathcal{T}_j[\pi][t]$.

In short, the network is trained to:

- Match its value predictions to n-step returns over each trajectory.
- Reproduce the improved policy distributions computed via planning.

2.4 Generic Tree Evaluation and Construction

Despite appearing somewhat arbitrary, the visitation counts evaluator used in AlphaZero has several desirable properties and theoretical guarantees. However, these are closely linked to the way the tree is constructed, i.e., the selection policy used. To better understand why this is the case and describe the impact of alternative evaluation and selection policies, we now generalize some of the steps taken in the default MCTS planning algorithm. This is more thoroughly detailed in [12], where the theoretical framework that we are going to summarize was originally developed.

To start, we introduce the concept of a ”special” simulation action a_v, and subsequent extended action space $\mathcal{A}_v = \mathcal{A} \cup \{a_v\}$. This special action corresponds to performing the rollout in algorithm 4 to estimate a leaf's value, or to a neural network call in the AZ case. Clearly, we cannot take a_v in the real environment, but this formulation allows us to more easily express some of the definitions that will follow. Given a policy π defined over the action space $\mathcal{A}$, we then indicate the corresponding policy over the extended action space $\mathcal{A}_v$ as $\tilde{\pi}$. This can be easily converted back to a valid policy for choosing an action to step in the real environment as:

$$\pi(x, a) = \frac{\tilde{\pi}(x, a)}{1 - \tilde{\pi}(x, a_v)}$$

Now, we can define the value estimate of a node as:

$$\hat{V}_{\tilde{\pi}}(x) = \mathbb{E}[\hat{Q}_{\tilde{\pi}}(x \uplus a) \mid a \sim \tilde{\pi}(x)]$$

Where, assuming a deterministic environment:

$$\hat{Q}_{\tilde{\pi}}(x) = r(x) + \gamma \hat{V}_{\tilde{\pi}}(x)$$

Here, $r(x)$ denotes the reward obtained by entering node x through the corresponding parent action and is therefore set to zero for the root node. These equations are developed by adapting the Bellman equations to the tree setup, so that $\hat{Q}_{\tilde{\pi}}(s,a) := \hat{Q}_{\tilde{\pi}}(x \uplus a)$. Then, the planning Q-value estimate of a node x under policy $\tilde{\pi}$ can be expressed as a recursive formulation over the extended action space $\mathcal{A}_v$:

$$\hat{Q}_{\tilde{\pi}}(x) = r(x) + \gamma \sum_{a \in \mathcal{A}_v} \tilde{\pi}(x,a)\hat{Q}_{\tilde{\pi}}(x \uplus a) \tag{2.3}$$

As anticipated, $\hat{Q}_{\tilde{\pi}}(x \uplus a_v) = v(x)$ corresponds to the value estimated by the rollout in standard MCTS, or by the neural network in AZ, in which case $v(x) = v_\theta(x)$. On the other hand, $\tilde{\pi}(x \uplus a_v)$ shall be defined depending on the specific policy. In the case of $\tilde{\pi}_N$, for instance, this is set by definition to $\tilde{\pi}_N(x \uplus a_v) = \frac{1}{N(x)}$. If we substitute the generic evaluator $\tilde{\pi}$ with $\tilde{\pi}_N$, we obtain the described arithmetic value estimate as computed in algorithm 5:

$$\hat{Q}_{\tilde{\pi}_N}(x) = r(x) + \gamma \sum_{a \in \mathcal{A}_v} \tilde{\pi}_N(x,a)\hat{Q}_{\tilde{\pi}_N}(x \uplus a) =$$

$$r(x) + \gamma \frac{\hat{Q}_{\tilde{\pi}_N}(x \uplus a_v)}{N(x)} + \gamma \sum_{a \in \mathcal{A}} \frac{N(x \uplus a)}{N(x)} \hat{Q}_{\tilde{\pi}_N}(x \uplus a) = \bar{Q}(x)$$

This formulation reveals that in standard MCTS, planning is in practice performed by estimating Q-values according to the corresponding stochastic visitation counts policy $\tilde{\pi}_N$ defined over the extended action space. While this can be seen intuitively by comparing the reported equation with the algorithmic description of the backup, it is formally proven in [12]. Note that the reported equation also reflects the fact that the value estimate of a just expanded node x with no children, i.e., $N(x \uplus a) = 0 \quad \forall a \in \mathcal{A}$, depends entirely on the simulation value $\hat{Q}_{\tilde{\pi}_N}(x \uplus a_v)$.

Moreover, it is also proven in [12] that:

- Using π'_{UCT} or π'_{PUCT} as a selection policy and $\tilde{\pi}_N$ as an evaluation policy to estimate node Q-values let us converge to the optimal value for the root node with enough visitation counts, i.e., $\lim_{N(x)\to\infty} \hat{V}_{\tilde{\pi}_N}(x) = V^*(x)$.
- Using π'_{UCT} or π'_{PUCT} as a selection policy and under further assumptions, $\tilde{\pi}_N$ minimizes the variance of the value estimates.

In practice, we usually have a limited simulation budget, which may cause $\tilde{\pi}_N$ to underestimate the value of a node. Besides, we might want to construct the planning tree differently, something which will be particularly relevant in the rest of this thesis, and could therefore employ different tree evaluation policies that are not strictly tied to UCT/PUCT construction and the arithmetic mean value backup. This generic framework is therefore instrumental as it allows us to more easily employ new evaluation policies that are independent of the way Q-values are estimated, i.e., of which policy $\tilde{\pi}$ is used to estimate $\hat{Q}_{\tilde{\pi}}$. Moreover, the backup algorithm shown in algorithm 5 can be turned into a generic backup which is equivalent to the former if $\tilde{\pi} = \tilde{\pi}_N$. This is reported in algorithm 7.

Algorithm 7: GenericBackup: Traverse the tree in reverse order to backup the estimated value of *node* and update Q value estimates based on policy $\tilde{\pi}$.

```
Input: Previously evaluated leaf node $node$, discount factor $\gamma$, evaluation policy $\tilde{\pi}$.
1 while $node$ is not none do
2     $node.visits \leftarrow node.visits + 1$
3     $Probs \leftarrow \tilde{\pi}(node.s)$
4     $QValues \leftarrow node.value$ ++ [$child.Q$ for $child$ in $node.children$]
5     $weightedQValues \leftarrow$ dotProduct($QValues, Probs$)
6     $node.Q \leftarrow node.r + \gamma \cdot weightedQValues$
7     $node \leftarrow node.parent$
```

Some of the most relevant alternative evaluation policies relevant to our work are described below. Note that these can be used for evaluating a fully constructed tree to act in the real environment, as well as for estimating node Q-values during construction.

Q-evaluator

The Q-evaluator simply chooses the action with the associated highest Q-value estimate, or picks one at random among them if they are more than one:

$$\tilde{\pi}'_Q(x,a) = \frac{\delta(a \in \arg\max_{a' \in \mathcal{A}_v} \hat{Q}_{\tilde{\pi}}(x \uplus a'))}{|\arg\max_{a' \in \mathcal{A}_v} \hat{Q}_{\tilde{\pi}}(x \uplus a')|}$$

Or, for the stochastic version:

$$\tilde{\pi}_Q(x,a) = \frac{\hat{Q}_{\tilde{\pi}}(x \uplus a)}{\sum_{a' \in \mathcal{A}_v} \hat{Q}_{\tilde{\pi}}(x \uplus a')}$$

This is a greedy evaluator that has a low bias but can have a high variance.

Minimal Variance Constrained Evaluator

The Minimal Variance Constrained (MVC) Evaluator [12] aims at balancing between low variance evaluators like $\tilde{\pi}_N$, which might underestimate the optimal value, and low bias evaluators like $\tilde{\pi}_Q$, which are potentially more optimal but can yield a high variance. Entering into the details of how this evaluator is formally derived is outside of the scope of this section; however, we shall provide an intuition of its purpose and related assumptions, as it will be utilized both in some of our baseline and novel algorithms.

First, we can define the variance of the value estimates recursively in a way that is similar to how values are computed in (2.3):

$$\mathbb{V}[\hat{Q}_{\tilde{\pi}}(x)] = \gamma^2 \sum_{a \in \mathcal{A}_v} \tilde{\pi}(x,a)^2 \, \mathbb{V}[\hat{Q}_{\tilde{\pi}}(x \uplus a)]$$

By doing this, we are again assuming a deterministic environment, as well as the independence of the value estimates. If the latter did not hold, we would need to manage covariance structures, which can be difficult to compute in practice. Note that for leaf nodes, this simplifies to:

$$\mathbb{V}[\hat{Q}_{\tilde{\pi}}(x)] = \gamma^2 \, \mathbb{V}[v(x)]$$

Where $\mathbb{V}[v(x)]$ can be estimated in different ways depending on how the simulation values are computed in practice (random rollout, neural network, etc.).

The MVC evaluator returns the action probabilities according to the following formula:

$$\tilde{\pi}_{\text{MVC}}(x, a) = \frac{\mathbb{V}[\hat{Q}_{\tilde{\pi}}(x \uplus a)]^{-1} \exp(\beta \hat{Q}_{\tilde{\pi}}(x \uplus a))}{\sum_{a' \in \mathcal{A}_v} \mathbb{V}[\hat{Q}_{\tilde{\pi}}(x \uplus a')]^{-1} \exp(\beta \hat{Q}_{\tilde{\pi}}(x \uplus a'))} \tag{2.4}$$

Where β is a parameter controlling "how greedy" we want to be. It is easy to see that for $\beta \to \infty$, MVC converges to the Q-evaluator, while for $\beta \to 0$ it converges to selecting the action whose corresponding value estimate has the lowest variance. Note that thanks to the way we defined it, the variance can also be updated during the backup in a similar way to what is done for the values in algorithm 7.

Chapter 3

Detecting Environment Changes

In this chapter, we analyze the problem of detecting unexpected changes to the environment at deployment while planning with learned estimators, i.e., value and policy functions. In principle, significant changes that completely compromise our estimators would require retraining the agent from scratch. An example of such a scenario would be training an agent to navigate toward a fixed goal within a room. If the goal is relocated to an entirely different room, then the previously learned policy and value functions become ineffective, as the optimal strategy now requires following a totally different path. However, many real-world tasks might involve relatively minor changes to the environment, which only partially affect our estimates. Examples of this could be the ones mentioned in the introduction:

- A robot learned to navigate toward a goal, but some (new) obstacles are now present on the path.
- A road closure disrupts the usual route from home to the office, requiring our self-driving car to perform a short detour.

In such cases, while a previously learned policy may no longer be entirely accurate, it could still provide a partially correct directional guidance and regain full correctness once the problem is overcome. Likewise, the reliability of a learned value function may diminish as the agent approaches the obstruction, but it recovers once the agent is far away from it. Therefore, if we are able to detect the change early enough and localize it correctly, we can use this information to adjust our planning and subsequent actions accordingly.

Note that in theory, standard AZ planning (subsection 2.3.2) is already a way of adjusting the previously learned prior policy π_θ and value v_θ by incorporating additional information from the search tree. We could therefore ask ourselves whether this is enough to get around problems like the ones we just described. In general, planning trees constructed using standard UCT-based policies tend to be relatively shallow. This is because they include an exploration component designed to evaluate multiple potential actions at each node, preventing the search from prematurely committing to a single greedy path. Moreover, standard MCTS-based planning algorithms often discard the previously built tree after acting in the real environment, which limits the amount of information that we can gather with a fixed planning budget. These factors combined might imply that our tree only manages to reflect the changes in the associated estimates after the agent gets too close to them in the real environment. Nonetheless, options to directly modify this standard way of planning without employing explicit change detection will also be discussed in section 4.3.

In the remainder of this chapter, we will then only focus on finding a sound criterion that allows us to detect and localize those changes in the shortest time possible. Several options for integrating these principles into complete planning algorithms are proposed in section 4.1 (AlphaZeroDetection methods) and section 4.2 (Penalty-Driven Deep Planning).

A quick way to "have a look into the future" can be, for example, rolling out our prior policy π_θ by turning it into a deterministic policy and selecting the best action at each planning step. We can do it one step at a time, by just taking action $\pi'_\theta(s_i) := \arg\max_{a \in \mathcal{A}} \pi_\theta(s_i, a)$. From now on, we will refer to this operation as rolling out (or "following") π'_θ, i.e., the deterministic version of π_θ. Note that we are assuming no ties in the argmax operation. This will create an extremely sparse search tree where each node has only one child expanded. Intuitively, this deep tree allows us to look much further into the future than a commonly expanded one.

We now provide additional definitions that will help us during the next steps of our analysis:

- s_0: the current state in the (real) environment.
- s_n: the state that we would reach in the training environment (before the changes) after following π'_θ for n steps from s_0.
- s'_n: the state that we reach in the modified environment (after the changes) after following π'_θ for n steps from s_0. If no deviation has happened yet, then $s'_n = s_n$.
- τ_n: the sequence of states $s_0, s_1, ..., s_n$ that we would enter if we followed π'_θ in the training environment for n steps.
- τ'_n: the sequence of states $s_0, s'_1, ..., s'_n$ that we enter in the modified environment by following π'_θ for n steps.
- π^*: an optimal policy for the training environment. Note that there might be multiple optimal policies.
- V^*: the optimal value function for the training environment.

Note that if a model of the training environment is available, the solution to our problem is trivial since we can observe τ_n. Then, comparing τ_n and τ'_n immediately tells us if any deviation happened, and exactly where. If we trained our neural network with a model-based method, we may still retain such a model, unless it becomes inaccessible for some reason. One such case could be in sim-to-real robotics, where agents are trained in high-fidelity simulators that are not available at deployment. In this scenario, a planning model might then be obtained at deployment from real-time sensor data, onboard estimation of physical dynamics, or external mapping systems. Moreover, our work also addresses more generic situations where the policy and value functions could be learned, for instance, with a model-free method that did not require a planning model in the first place.

Excluding the availability of a model of the training environment, we then need an indicator that τ'_n no longer matches the (unknown) trajectory τ_n that we would have experienced had the environment not changed. In the remainder of this chapter, we show how the learned value function can be utilized for this purpose under two main assumptions:

- The model of the changed environment correctly predicts transitions and rewards.
- The environment (and therefore the model) is deterministic.

Future work ideas for extending our approach beyond these assumptions are discussed in section 9.1.

3.1 Value as a Change Indicator

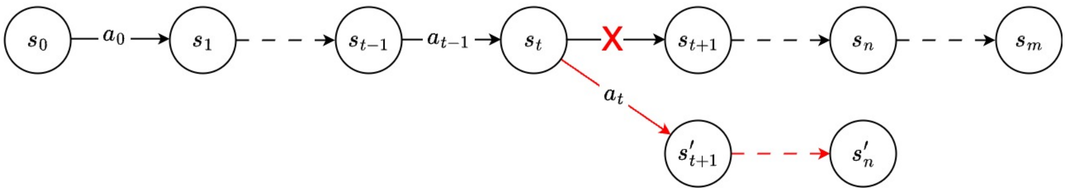


Figure 3.1: A representation of the agent following π'_θ for n steps in the training and test environments. The two trajectories τ_n and τ'_n first deviate after t steps.

Let $y'_n(s_0)$ be the n-step bootstrapped value estimate of the current state s_0 in the modified environment (i.e., after some changes happened) computed by rolling out π'_θ:

$$y'_n(s_0) = \sum_{i=0}^{n-1} \gamma^i r'_i + \gamma^n v_\theta(s'_n)$$

Where $r'_i = R(s'_i, \pi'_\theta(s'_i))$.

Then, it is easy to see that if $v_\theta = V_{\pi'_\theta}$, the following implication holds:

$$y'_n(s_0) \neq v_\theta(s_0) \implies \tau'_n \neq \tau_n, \quad \forall n > 0$$

In other words, if v_θ corresponds to the expected return of the deterministic policy π'_θ that we are rolling out, then a change in the corresponding n-step value estimate always implies that transitions have changed. This is true, for example, in the ideal case where $\pi_\theta = \pi^*$ and $v_\theta = V^*$, but should generally hold approximately even in cases where v_θ and π_θ are not optimal, since they are learned together. The problem is that in practice, the n-step value estimate may fluctuate as we roll out further due to the learned value, even if no change has occurred. This can result in the anticipated or delayed realization of this change, as well as reduced confidence about where the trajectory initially deviates from the expectation of the policy.

Let t be the last "safe" step, after which the two trajectories in the training and test environments diverge, as shown in Figure 3.1 or, formally:

$$t \in [0, n-1] \,|\, (\tau'_t = \tau_t) \wedge (s'_{t+1} \neq s_{t+1})$$

Since we are interested in detrimental changes to the environment rather than positive ones, we can consider the following criterion:

$$(\bigwedge_{i=0}^{n-1} \frac{y'_i(s_0)}{v_\theta(s_0)} \geq 1-\epsilon) \wedge (\frac{y'_n(s_0)}{v_\theta(s_0)} < 1-\epsilon) \implies t < n \tag{3.1}$$

In simple terms, this states that if our online value estimate becomes smaller than the prior value, then something must have changed along the trajectory. Setting a tolerance ϵ is important since we are working with neural networks and as anticipated, small fluctuations in the $y'_i(s_0)$ estimate could otherwise trigger our criterion when there is no actual problem in the environment. Note that this criterion only tells us that there is a problem along the trajectory, but it does not tell us exactly where.

In the most general case and without specific assumptions on the reward function, we can roughly express the estimate ratio as:

$$\frac{y'_n(s_0)}{v_\theta(s_0)} \approx \frac{y'_n(s_0)}{y_n(s_0)} = \frac{\sum_{i=0}^{t} \gamma^i r_i + \sum_{j=1}^{n-t} \gamma^{t+j}\, r'_{t+j} + \gamma^n v_\theta(s'_n)}{\sum_{i=0}^{t} \gamma^i r_i + \sum_{j=1}^{n-t} \gamma^{t+j}\, r_{t+j} + \gamma^n v_\theta(s_n)}$$

Where $y_n(s_0)$ corresponds to the n-step value estimate that we would have obtained by following π'_θ in the training environment, and is exactly equal to $v_\theta(s_0)$ if $v_\theta = V_{\pi'_\theta}$.

Note that the numerator is composed of:

- A first term $\sum_{i=0}^{t} \gamma^i r_i$ which is the discounted return obtained on the unchanged path until a certain step t is reached.
- A second term $\sum_{j=1}^{n-t} \gamma^{t+j}\, r'_{t+j}$ which is the discounted return obtained from the moment we first deviate from the trajectory that we would have followed in the training environment.
- A third bootstrapping term $\gamma^n v_\theta(s'_n)$, where s'_n is likely not the same state as s_n since we have deviated before.

Long deviations from the original trajectory τ_n ($n \gg t$) will likely bring the second and third terms of the numerator to be very dissimilar to the corresponding ones in the denominator. While this should make it easy to detect that something is off, it is challenging to analytically extract the moment t at which the deviation occurred, as we have to deal with a ratio of sums. However, after the very first step of deviation, i.e., when $n = t + 1$:

$$\frac{y'_n(s_0)}{v_\theta(s_0)} \approx \frac{y'_n(s_0)}{y_n(s_0)} = \frac{\sum_{i=0}^{t-1} \gamma^i r_i + \gamma^t r'_t + \gamma^{t+1} v_\theta(s'_{t+1})}{\sum_{i=0}^{t-1} \gamma^i r_i + \gamma^t r_t + \gamma^{t+1} v_\theta(s_{t+1})}$$

We can see how the difference between the two terms depends on the reward of the first different transition r'_t and on how different the state we enter is from the one we would have entered in the training environment, from the perspective of the value function. In usual situations, we can imagine s'_{t+1} would be fairly similar to s_{t+1}. In such a case, a big difference between r'_t and r_t should trigger our criterion very easily. That should also be the case in the opposite situation, where we get a fairly similar reward but end up in a state that is highly suboptimal compared to the one we would have entered in the training environment. If our tolerance parameter ϵ is properly tuned, we can then assume that nothing happened until the moment we detect a change, which coincides with the first problematic transition, i.e., we can set $t = n - 1$.

A potential issue with generic reward functions is that the ratio between the value estimates may not always be computable analytically. This can occur when the reward function produces both positive and negative values, resulting in both positive and negative value estimates. One way to address this problem could be to rescale the environment rewards to be always non-negative. If we know the minimum possible reward for a single step is $-c$, with c being a positive constant, we can then set:

$$r(s,a) = r(s,a) + c \quad \forall\, s \in \mathcal{S}, a \in \mathcal{A}$$

However, note that this creates an issue when bootstrapping the value of a terminal state, since it is usually set to zero, essentially assuming that upon reaching it, the agent enters an endless loop without accumulating further reward. Now, since we are introducing this reward trick, we need to adjust the value of terminal states accordingly as if we were looping in a state which yields a reward of c, i.e., for any value function V and any terminal state s_{term}:

$$V(s_{\text{term}}) = \sum_{i=0}^{\infty} c\gamma^i = \frac{c}{1-\gamma}$$

Where the last equation holds since the series is geometric. With this adjustment, the detection criteria can be applied without further modifications.

In the next section, we will present some relevant examples where the reward function and the nature of the changes allow us to analytically compute the moment at which the agent first deviates from the expected trajectory.

3.2 Detecting Obstacles

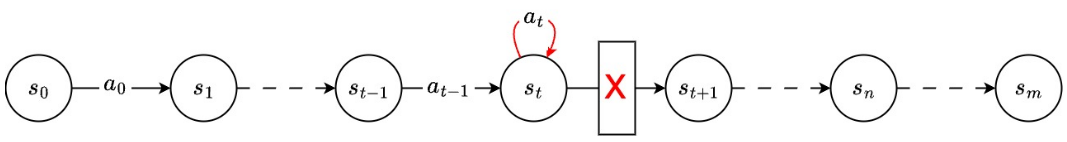


Figure 3.2: An obstacle prevents the agent from reaching state s_{t+1} and instead makes it loop in s_t forever.

We now restrict our analysis to cases where changes to the environment take the form of obstructions, such as obstacles along a navigation path. For now, let's further assume that trying to move towards an obstacle results in bumping into it, staying in the current state. This is shown in Figure 3.2, where the red loop represents the fact that $s_t = s_{t+i},\ \forall\ i \geq 0$.

Let $\delta_\theta(s) = V_{\pi'_\theta}(s) - v_\theta(s)$, i.e., how much our estimated value differs from the expected return of the policy we are following. If the only signal is a unitary reward obtained once reaching the goal state [1] and we assume following π'_θ would bring to it in m steps if there were no obstacles, then $V_{\pi'_\theta}(s_0) = \gamma^m$ and therefore $v_\theta(s_0) = \gamma^m + \delta_\theta(s_0)$. Moreover, if we roll out our prior and start bumping into an unexpected obstacle after t steps, we know that $y'_n(s_0) = \gamma^n v_\theta(s_t) = \gamma^n(\gamma^{m-t} + \delta_\theta(s_t))$. Therefore:

$$\frac{y'_n(s_0)}{v_\theta(s_0)} = \frac{\gamma^n(\gamma^{m-t} + \delta_\theta(s_t))}{\gamma^m + \delta_\theta(s_0)},\ \forall n > 0, t \leq n \tag{3.2}$$

Equation (3.1) states that we assume to be safe as long as $\frac{y'_n(s_0)}{v_\theta(s_0)} \leq 1 - \epsilon$, so we can plug (3.2) at the boundary of this condition and isolate t:

$$t = n + m - \frac{\ln[(1-\epsilon)\gamma^m + (1-\epsilon)\delta_\theta(s_0) - \gamma^n\delta_\theta(s_t)]}{\ln(\gamma)} \tag{3.3}$$

This expression is relatively complex, and cannot be computed in practice since we do not know m, $\delta_\theta(s_0)$, and $\delta_\theta(s_t)$. If we assume that the learned value function correctly reflects the expected return of the learned policy, then we can set $\delta_\theta(s) = 0,\ \forall s \in \mathcal{S}$, in which case:

$$\frac{y'_n(s_0)}{v_\theta(s_0)} = \gamma^{n-t},\ \forall n > 0, t \leq n \tag{3.4}$$

This way, (3.3) simplifies to:

$$t = n - \frac{\ln(1-\epsilon)}{\ln(\gamma)} \tag{3.5}$$

[1] We set it to 1 without loss of generality, as the analysis does not change if the reward is any other positive number.

Since this is generally a real number, we can set $t = \lfloor n - \frac{\ln(1-\epsilon)}{\ln(\gamma)} \rfloor$ as the last "safe" step along our trajectory. This criterion, which we name $\mathcal{C}$, is therefore fully expressed as:

$$\mathcal{C}(\tau'_n, \epsilon, \gamma) = (\bigwedge_{i=0}^{n-1} \frac{y'_i(s_0)}{v_\theta(s_0)} \geq 1 - \epsilon) \wedge (\frac{y'_n(s_0)}{v_\theta(s_0)} < 1 - \epsilon) \implies t = \lfloor n - \frac{\ln(1-\epsilon)}{\ln(\gamma)} \rfloor \quad (3.6)$$

Note that :

$$\lim_{\epsilon \to 0^+} \lfloor n - \frac{\ln(1-\epsilon)}{\ln(\gamma)} \rfloor = n - 1, \ \forall \gamma \in (0, 1) \quad (3.7)$$

This reflects the fact that if our value function perfectly matched the expected return of the policy, then we could set our tolerance arbitrarily small and we would always detect changes as soon as we encounter them, allowing us to set $t = n - 1$ without committing any error. In reality, v_θ might not exactly correspond to $V_{\pi'_\theta}$, and even by applying (3.6), we commit an error both in detecting and localizing the problem, which depends on $\delta_\theta(s_0)$ and $\delta_\theta(s_t)$.

3.2.1 Different Obstacle Interaction Outcome

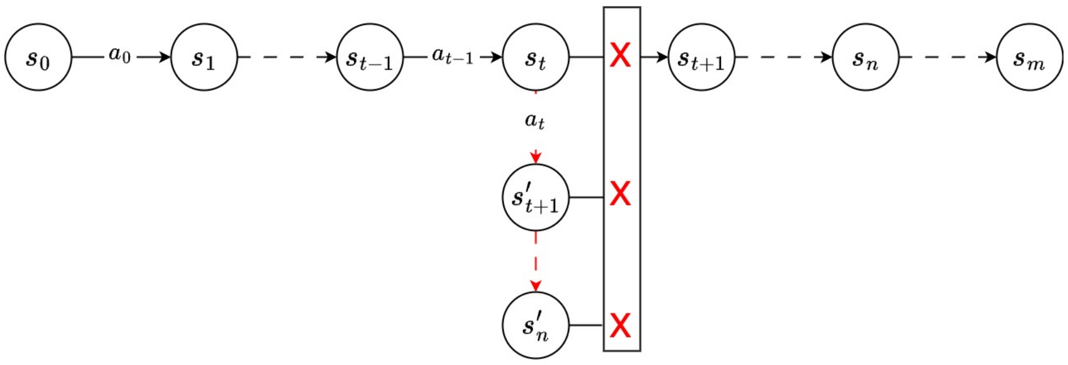


Figure 3.3: Negative deviation example: the agent keeps trying to move beyond the obstacle and gets one step farther away from the goal each time.

We now provide an intuition of what might happen when we apply our criterion in situations where the outcome of moving towards an obstacle is different. While anything could happen in theory, we can imagine three different reasonable situations:

- **Bumping**: the agent stays in the same state. We already analyzed this situation in detail and developed our obstacle localization method based on it.
- **Negative deviation**: when trying to pass through the obstacle, the agent deviates to a state that is farther away from the goal compared to the previous state.
- **Positive deviation**: when trying to pass through the obstacle, the agent deviates to a state that is closer to the goal compared to the previous state.

Say we get farther away from the goal of x steps after following the right path for t, i.e., $x = n - t$. This would be the case in the example situation in Figure 3.3, where we continually attempt to move beyond the obstacle, and each failure brings us one step farther away from the goal.

Then, it would take us $x + (m - t)$ steps to reach the goal from the position we reached. The bootstrapped value estimate of s_0 in the test environment would then be:

$$y'_n(s_0) = \gamma^n \gamma^{(m-t)+(n-t)} = \gamma^{m+2(n-t)}$$

The ratio of our estimates, assuming $\delta_\theta(s_0) = \delta_\theta(s_x) = 0$, would change as follows:

$$\frac{y'_n(s_0)}{v_\theta(s_0)} = \frac{\gamma^{m+2(n-t)}}{\gamma^m} = \gamma^{2(n-t)}$$

Therefore, the t estimate in (3.6) should also change if we plug this ratio into the boundary of the trigger condition:

$$\gamma^{2(n-t)} = 1 - \epsilon \implies t = n - \frac{\ln(1-\epsilon)}{2\ln(\gamma)}$$

This means that by using the previously developed (3.6) unchanged, we now commit a constant underestimation error of:

$$(n - \frac{\ln(1-\epsilon)}{\ln(\gamma)}) - (n - \frac{\ln(1-\epsilon)}{2\ln(\gamma)}) = -\frac{\ln(1-\epsilon)}{2\ln(\gamma)}$$

In other words, this means that the agent will believe the deviation started earlier than it actually did. In most cases, this is preferable to an overestimation error, especially if we want to ensure that any countermeasure is taken before reaching the deviation in the real environment. Note that this error fades away as $\epsilon \to 0$, meaning that if our value function v_θ is reliable enough, we can set a small tolerance ϵ and its impact will be minimal.

Conversely, deviating in the positive deviation case would bring us closer to the goal. This is exemplified in Figure 3.4. In this case, the criterion will not even be triggered unless there are significant flaws in the value function. While this prevents detecting the change, the states that the agent enters when deviating are still equally optimal from a value perspective, and it therefore makes sense that the criterion is not triggered.

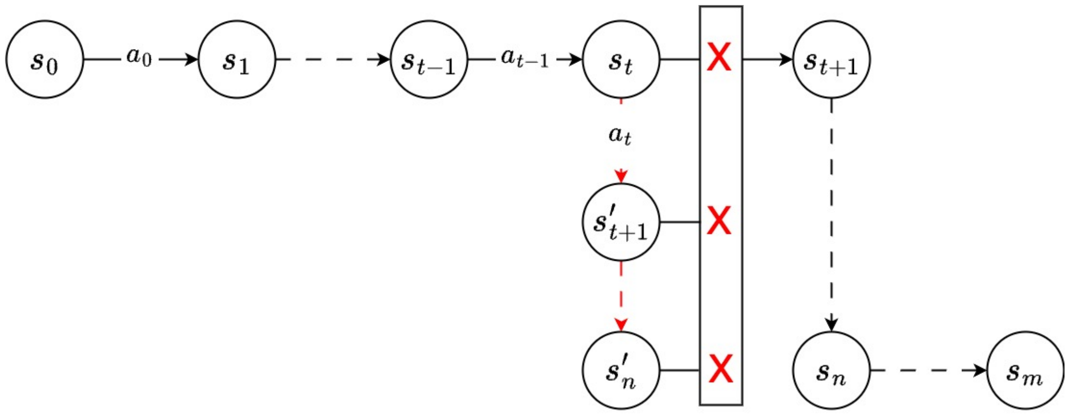


Figure 3.4: Positive deviation example: the agent keeps trying to move beyond the obstacle and gets one step closer to the goal each time.

While these two additional situations do not prove that our criterion would always work as expected in more complex cases, they provide an intuition that in standard situations and with acceptably good estimators, it should still function with sufficient reliability.

3.3 Addressing Value Underestimation with an Overestimation

Can we use something better than $v_\theta(s_0)$ as an estimate of $V_{\pi'_\theta}(s_0)$ for our criterion? Let's first analyze two possible scenarios:

- $v_\theta(s_0)$ is an **overestimation** of $V_{\pi'_\theta}(s_0)$, i.e. $\delta_\theta(s_0) < 0$: our criterion might be wrongly triggered as we roll-out further and $y'_i(s_0)$ becomes a more precise (and therefore lower) estimate. To counter this, we can increase our tolerance ϵ.
- $v_\theta(s_0)$ is an **underestimation** of $V_{\pi'_\theta}(s_0)$, i.e. $\delta_\theta(s_0) > 0$: $\frac{y'_i(s_0)}{v_\theta(s_0)}$ might grow $\gg 1$ as we roll-out further since $y'_i(s_0)$ will likely increase. When we start bumping into the obstacle, $y'_i(s_0)$ will start to decrease due to the γ discounting, but it might take much more time before the ratio goes below $1 - \epsilon$. This would result in delayed detection.

How can we deal with $v_\theta(s_0)$ underestimation? Note that when $\frac{y'_i(s_0)}{v_\theta(s_0)} < 1$, we might have detected a problem, depending on our tolerance ϵ. Conversely, while $\frac{y'_i(s_0)}{v_\theta(s_0)} \geq 1$, we are always assuming no detrimental change has happened along τ'_n, i.e. $\tau'_n = \tau_n$, irrespective of the tolerance. Note that this always holds at step 0 since $\frac{y'_0(s_0)}{v_\theta(s_0)} = 1$ by definition.

In principle, we could therefore change our current s_0 value estimate for the training environment from $v_\theta(s_0)$ to $y'_i(s_0)$ if we are confident that no deviation has been experienced yet, i.e., if $\frac{y'_i(s_0)}{v_\theta(s_0)} \geq 1$. This can be done iteratively while adding nodes to our trajectory, every time $y'_i(s_0)$ increases compared to the previous value of the denominator. This should help to combat the value underestimation of $v_\theta(s_0)$ with a (potential) overestimation. The described iterative process corresponds to selecting the maximum i-step value estimate along the trajectory τ'_n rolled out so far, i.e.:

$$y_n^{\max}(s_0) \; = \; \max\{\, y'_i(s_0) \;\mid\; i \leq n\}$$

Our final criterion $\mathcal{C}_{\text{max}}$ becomes:

$$\mathcal{C}_{\text{max}}(\tau'_n, \epsilon, \gamma) = (\bigwedge_{i=0}^{n-1} \frac{y'_i(s_0)}{y_i^{\max}(s_0)} \geq 1-\epsilon) \wedge (\frac{y'_n(s_0)}{y_n^{\max}(s_0)} < 1-\epsilon) \implies t = \lfloor n - \frac{\ln(1-\epsilon)}{\ln(\gamma)} \rfloor \tag{3.8}$$

If the approximation errors of the learned value function with respect to the expected value of our policy are more similar for closer states, i.e., $i \approx j \implies \delta_\theta(s_i) \approx \delta_\theta(s_j)$, then the new criterion should also provide a better estimate of t. This is because the index which maximizes $y_n^{\max}(s_0)$, i.e. $j = \arg\max_i\{\, y'_i(s_0) \;\mid\; i \leq n\}$, is closer to n than 0 is, and therefore $|\delta_\theta(s_j) - \delta_\theta(s_n)| < |\delta_\theta(s_0) - \delta_\theta(s_n)|$. Such a hypothesis will be confirmed empirically by the experiment reported in section 7.1.

Chapter 4

Planning in a Changed Environment

In this chapter, we focus on principles and methods that can be leveraged to plan and subsequently act in an environment whose estimators, i.e., learned policy and value functions, are potentially unreliable due to some unknown environment modification. Note that the environment is assumed to have changed compared to the original training environment, but not to change further while the agent is acting; i.e., the changed test environment remains stationary.

All the methods presented assume, as in standard AZ planning, that we possess a learned value function v_θ. Some of them also require a learned policy π_θ; if we performed AZ training as described in subsection 2.3.3, these two come by default, but we want to further stress the fact that, in principle, they could also be learned by different, model-free methods and then used at deployment given a model of the test environment. Moreover, we still assume that our planning model correctly represents the underlying environment, as we did in our analysis in chapter 3.

To ensure a fair comparison with our baselines in terms of computational cost of the planning, we develop anytime algorithms where the maximum number of nodes that can be evaluated at each step is fixed and independent of whether these are used to detect a change, subsequently plan, or check when the change is cleared. Importantly, in all our proposed algorithms, as well as the baselines, both neural network calls and transition model invocations occur only when a new node is created and evaluated. Therefore, by counting the number of node evaluations, we can fairly compare computational costs regardless of whether the bottleneck arises from neural network evaluations or transition model computations.

4.1 AlphaZeroDetection

The first new family of algorithms that we introduce is named AlphaZeroDetection (AZD). The general idea is straightforward: we want to follow a given policy, which we assume was trained sufficiently well in the training environment, and deviate from it when we detect a disruption along the path. While such disruption could in principle be caused by any kind of change to the environment, we will informally refer to it as an obstacle as this makes it easier to visualize and explain related examples. If such an obstacle is detected, we want to plan as much as possible to find a path ”around” it. Finally, once the obstacle is cleared, we can resume following our prior policy until a new obstacle is detected... and so on.

In our setting, the policy that we want to follow is the learned prior policy π_θ. We can then turn it into a deterministic policy by always selecting the action with the highest probability according to it, and look into the future by rolling it out. This creates a sequence of nodes which we will simply refer to as a trajectory. Then, the way we can detect obstacles is exactly as described in chapter 3, specifically using one of the described criteria $\mathcal{C}$ (3.6) or $\mathcal{C}_{\max}$ (3.8). Once we have found the node after which the obstacle is encountered, we may want to retain all the nodes before it for the subsequent planning phase. However, the shape or extent of the obstacle is, in principle, not known. This means we don’t know if it was just a small obstacle we can simply move around, or a closed door that now requires a long detour starting from the beginning of the trajectory:

1. How do we expand the trajectory into a planning tree?

2. How do we identify nodes located beyond the obstacle?

Note that standard AZ planning described in algorithm 6 always plans from the root node and discards the previous planning tree after stepping into the environment. In our case, however, we may not be able to find a path around the obstacle right after detecting it (due to the limited planning budget). We could then keep expanding the previously constructed tree even in subsequent environment steps, until we finally find a path that can be safely followed in the real environment. Before discussing ways to expand our rolled-out trajectory in detail, let’s first focus on the second question, assuming we already know how to construct the tree. This will make the advantages and disadvantages of the two expansion methods we are going to describe easier to understand.

4.1.1 Clearing the Obstacle with a Value Search

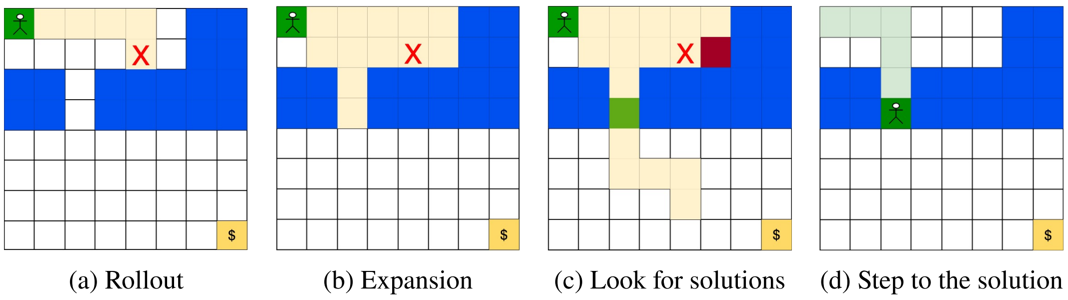
When constructing our planning tree after a successful change detection, our goal is to find a node from which we could start relying on our prior policy again, i.e., we do not have to worry about the detected obstacle from then on. If such a node is identified, the idea is to simply stop planning and follow the path of nodes that lead from the root to it in the real environment. To do so, we devised a method which we call Value Search (VS). The idea is that to start relying on the prior policy again, we should find a node with the following properties:

- The node’s value is at least as high as the root value, i.e., the value of the current state of the environment.

- If we checked again our criterion by rolling out a trajectory from the node, we would not detect an obstacle anymore.

The second is perhaps the most intuitive; since our way of detecting obstacles involves rolling out a trajectory guided by the prior, we now consider an obstacle cleared if the same criterion indicates that we are finally safe when taking the same number of steps in the direction of the prior from the found node. The problem is that only checking this would very easily fool us into going ”backwards”. We might have found a node far away from the obstacle and the current state, from which our fixed-length rollout does not detect an obstacle, simply because we went back a few steps. As a result, we will face the same problem again once we move forward. Instead, by checking that the node's value is higher than the root's, we ensure that such a node is ”ahead” of the agent's current position, and we therefore avoid fooling our detection criterion by just going backwards.

There is, however, a major issue in practice. While planning, we will encounter several nodes whose value is higher than the root value. Every time, we should then roll out a detection trajectory from there, which results in a certain number of node evaluations. To be fair with respect to the baselines, we should then deduct that number from the available planning budget. This could potentially cause us to run out of budget extremely quickly, which might severely deteriorate the algorithm's performance, as we will be able to find and check fewer potential solution nodes. To avoid this, we instead decide to only check the nodes whose value is higher than the last safe node of the trajectory (i.e., the one before the obstacle). Since this should be a higher value than the root, we will then check fewer candidates and maintain part of the planning budget for expanding other nodes. A visual example of the workings of the Value Search mechanism is shown in Figure 4.1.



(a) Rollout (b) Expansion (c) Look for solutions (d) Step to the solution

Figure 4.1: Visual example of the application of the Value Search mechanism. The light-yellow cells correspond to the states that the agent sees during planning. Initially, the agent rolls out the prior and detects an obstacle by bumping into the wall. The tree is later expanded, and we look for states beyond the obstacle. The red state has a larger value than the problematic state (red cross), but rolling out the trajectory from it results in another bump. Conversely, the green state has a value as high as the problematic state, and we do not detect any obstacle by checking again our criterion starting from it. The agent can then reach that state by following the corresponding trajectory in the expanded tree.

We can now present two AZD algorithms that share the principles presented so far but differ in the way they roll out and expand the future trajectory.

4.1.2 The MiniTrees Algorithm

The MiniTrees algorithm rolls out a fixed-length trajectory of n nodes to perform change detection, as described in algorithm 8. Note that we make use of the $\mathcal{C}_{\text{max}}$ detection criterion in the pseudocode description, but the simpler $\mathcal{C}$ could also be used. If no obstacle is detected along the trajectory, we follow it for one step, i.e., we act deterministically according to π'_θ without further planning. Recall we use π'_θ to define the deterministic policy extracted from π_θ by just following its argmax action. Conversely, if the detection is triggered, we want to expand the nodes in a way that can address both the "obstacle situation" (we should expand the last nodes) and the "door situation" (we should expand the first nodes).

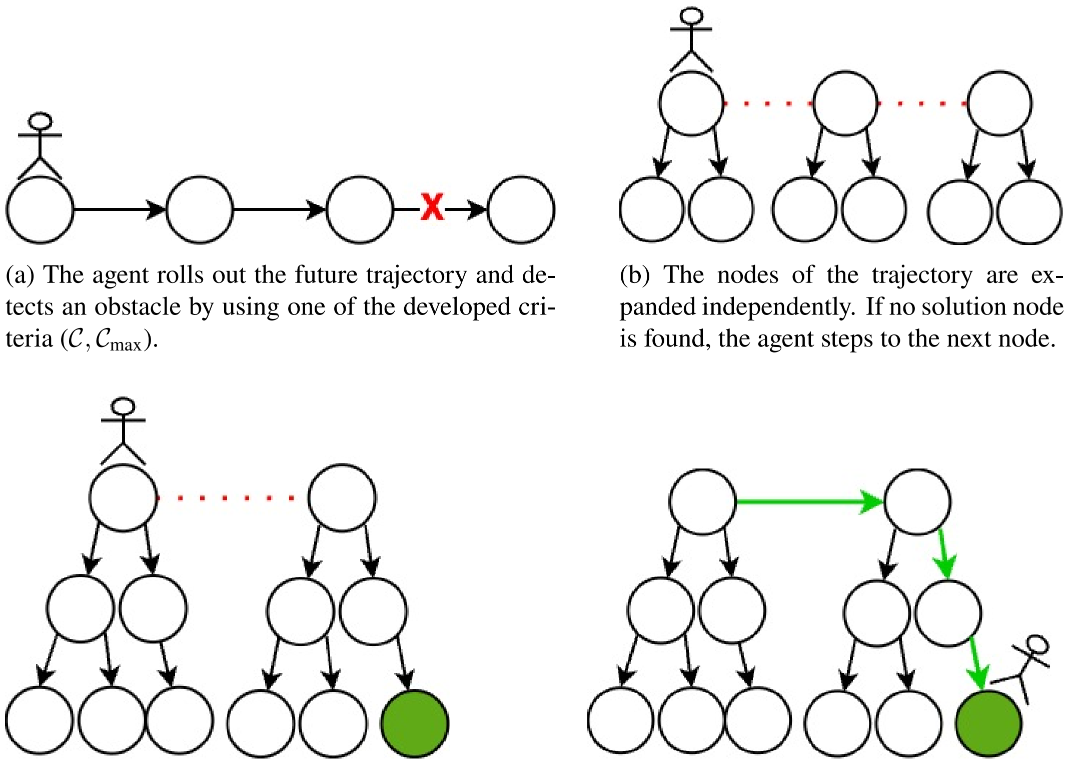
Algorithm 8: DetectionRollout: Detect changes using the $\mathcal{C}_{\text{max}}$ criterion.

Input: Root node $root$, discount factor γ, policy policy π_θ, prior value v_θ, rollout budget n, tolerance ϵ

Output: Trajectory τ containing nodes and actions, change index $changeStep$, number of node evaluations $numEval$

```
1  Initialize $node \leftarrow root$, $\tau \leftarrow []$, $changeStep \leftarrow None$, $R \leftarrow 0$, $numEval \leftarrow 0$
2  $node.value \leftarrow v_\theta(node.s)$
3  $y_{\text{max}} \leftarrow node.value$
4  for $i = 0$ to $n$ do
5      $a \leftarrow \arg\max_{a'} \pi_\theta(node.s, a')$
6      $\tau.append([node, a])$
7      $node \leftarrow$ Expand$(node, a)$
8      $R \leftarrow R + \gamma^i \cdot node.r$
9      $node.value \leftarrow v_\theta(node.s)$ if not $node.terminal$ else 0
10     $numEval \leftarrow numEval + 1$
11     $y \leftarrow R + \gamma^{i+1} \cdot node.value$
12     if $y > y_{max}$ then
13         $y_{\text{max}} \leftarrow y$
14     if $\frac{y}{y_{max}} < 1 - \epsilon$ then
15         $t \leftarrow \max(\lfloor i + 1 - \frac{\ln(1-\epsilon)}{\ln(\gamma)} \rfloor, 0)$
16         $changeStep \leftarrow \min(t + 1, i + 1)$
17         return $\tau, changeStep, numEval$
18     if $node.terminal$ then
19         break
20 return $\tau, None, numEval$
```

We opt for the simplest approach balancing these two scenarios by uniformly expanding the nodes. Given a total planning budget N, the number of nodes $n' \leq n$ that we rolled out at this step before detecting the obstacle, and the index t which is the last safe step along the trajectory, we split our trajectory into completely independent "mini" trees and plan with standard AZMCTS by each of them for $\lfloor \frac{N-n'}{t+1} \rfloor$, starting from the first one [1].



(a) The agent rolls out the future trajectory and detects an obstacle by using one of the developed criteria $(\mathcal{C}, \mathcal{C}_{\max})$.

(b) The nodes of the trajectory are expanded independently. If no solution node is found, the agent steps to the next node.

(c) The remaining nodes of the trajectory are further expanded. The green node indicates that we found a VS solution node.

(d) The agent follows the path that leads to that node. From there on, it will trust the prior again until a new obstacle is detected.

Figure 4.2: MiniTrees detection, planning, and Value Search visualized.

The per-tree budget is computed to be fair with respect to standard AZ, ensuring that the total number of node evaluations remains the same. This means we also need to take into account the ones used for the Value Search. Each time we find a node whose value is greater than that of the last node of the trajectory, we perform an n-step policy rollout to check whether we have cleared the obstacle. If we have not, then we must subtract n from the remaining total budget for the current step. This implies that in practice, the individual planning budget per node of the trajectory will not always be the same, and we might end up exploring less from the last nodes of it. However, if no solution is found immediately, we will then step into the next node and grow the remaining mini trees further. The closer we get to the end of the trajectory, the more we will expand these trees since we have to share the budget among fewer roots. An example of this is provided in Figure 4.2, while the pseudocode detailing MiniTrees expansion and Value Search is reported in algorithm 9.

Moreover, this algorithm allows for employing the standard visitation counts + PUCT/UCT

[1]Note that the trajectory is $t + 1$ nodes long as indexing starts at zero.

framework without losing its properties discussed in section 2.4, since each mini-tree is expanded independently of the others in the usual AZ way.

Algorithm 9: ExpandMiniTrees: Expand nodes independently and perform VS.

```
Input: Trajectory $\tau$ containing nodes and actions, sel. policy $\pi_{\text{sel}}$, planning budget
       $N$, disc. factor $\gamma$, rollout budget $n$, prior policy $\pi_\theta$, prior val. $v_\theta$, tolerance $\epsilon$
Output: Solution sequence of actions $path$ to be undertaken by the agent (if any)
1  $netPlanning \leftarrow N$
2  for $i = 0$ to $|\tau|$ do
3      $miniroot \leftarrow \tau[i].node$
4      $miniroot.parent \leftarrow None$
5      $visits \leftarrow miniroot.visits$
6      while (miniroot.visits - visits $< N$ // $|\tau|$) $\wedge$ (netPlanning $< N$) do
7          TrackSelect: Runs standard selection and keeps track of the traversed
8          action sequence $actions$.
9          $expNode, expAction, actions \leftarrow$ TrackSelect$(miniroot, \pi_{\text{sel}})$
10         if $expNode.terminal$ then
11             Backup$(expNode, \gamma)$
12         else
13             $child \leftarrow$ Expand$(expNode, expAction)$
14             $child.value \leftarrow v_\theta(evalNode.s)$
15             Perform Value Search:
16             if $(child.value \geq \tau[-1].node.value) \wedge (netPlanning > N - n)$
                 then
17                 Roll-out the prior from $child$ to check if there is still a problem.
18                 $problem, numEval \leftarrow$ DetectionRollout$(child, \gamma, \pi_\theta, v_\theta, n, \epsilon)$
19                 if not problem then
20                     Concatenate the action sequence leading to $miniroot$ with the
21                     one leading from $miniroot$ to the newly expanded node.
22                     $path \leftarrow \tau[: i].actions$ ++ $actions$
23                     return $path$
24             Backup$(child, \gamma)$
25 If no solution is found, simply return None to indicate this.
26 return $None$
```

The pseudocode description of the overall algorithm orchestrating change detection and minitrees expansion is provided in algorithm 10.

The MiniTrees algorithm is conceptually simple, and while being a good start, it may underperform in certain scenarios. For instance, the fact that we roll out a fixed-length trajectory might be a problem in situations where the obstacle needs to be spotted from very long distances in order to find a solution. While we could increase the length of the rollouts, we would then need to uniformly expand a trajectory consisting of many nodes, which would result in the individual budget per node becoming very small. If we can increase our overall available budget, this is not a problem. Nonetheless, we would like to know whether an alternative way of expanding the trajectory that mitigates the issue on its own exists. Finally, note that MiniTrees does not expand the rolled-out trajectory further when no obstacle is detected. While this makes it very fast until we spot an obstacle, it might be possible to plan ”in advance” to increase the chances of finding a solution once an obstacle is detected.

Algorithm 10: MiniTrees Algorithm

```
Input: Environment model $env$, prior policy $\pi_\theta$, prior value $v_\theta$, rollout budget $n$,
       planning budget $N$, discount factor $\gamma$, tolerance $\epsilon$, selection policy $\pi_{\text{sel}}$,
       rolled out trajectory $\tau$ containing nodes and actions, detected $changeStep$
Output: Sequence of actions bypassing the detected change (if any) or single prior
        policy action.
1  if changeStep is not None then
2      Change previously detected, we can keep expanding the same trajectory
3      $\tau \leftarrow \tau[1:]$
4      $solutionPath \leftarrow$ ExpandMiniTrees$(\tau, \pi_{\text{sel}}, N, \gamma, n, \pi_\theta, v_\theta, \epsilon)$
5      if solutionPath is not None then
6          We can directly return the whole sequence of actions that the agent will
7          follow to bypass the change.
8          return $solutionPath$
9      else
10         No solution found, save $\tau$ to keep expanding it at the next step.
11         return $\arg\max_a \pi_\theta(root.s, a)$
12 $root \leftarrow Node(env)$
13 $\tau$, $changeStep$, $numEval \leftarrow$ DetectionRollout$(root, \gamma, \pi_\theta, v_\theta, n, \epsilon)$
14 if changeStep is None then
15     No change detected, we can trust the prior.
16     return $\arg\max_a \pi_\theta(root.s, a)$
17 else
18     We expand the non-changed part of the rolled-out trajectory.
19     $N \leftarrow N - numEval$
20     $solutionPath \leftarrow$ ExpandMiniTrees$(\tau[:changeStep], \pi_{\text{sel}}, N, \gamma, n, \pi_\theta, v_\theta, \epsilon)$
21     if solutionPath is not None then
22         return $solutionPath$
23     else
24         No solution found, save $\tau$ to keep expanding it at the next step.
25         return $\arg\max_a \pi_\theta(root.s, a)$
```

4.1.3 The MegaTree Algorithm

The MegaTree algorithm generates an increasingly long trajectory of nodes. We start by rolling out an n-step trajectory, as done in MiniTrees. If no obstacle is detected, we follow this trajectory for one step in the environment and expand it by adding additional n nodes. This is repeated until the detection is triggered. If this is not the case, it is easy to see that the trajectory ahead of the agent will be $n + (t - 1)(n - 1)$ nodes long after t steps taken in the environment. As it can become challenging to sufficiently expand the nodes of such a long trajectory independently, the MiniTrees approach would not be effective in this case.

We can then switch back to a more standard expansion method starting from the root of the trajectory. The problem is that we have already constructed part of the tree in a non-standard manner, which could influence the usual subsequent evaluation by visitation counts in unpredictable ways, as well as breaking the nice properties of the visitation counts evaluator under the original AZ construction. A solution to this is to employ one of the alternative construction decoupled evaluation policies shown in section 2.4 and estimate the node Q-values accordingly while planning (as shown in algorithm 7). For example, we could employ the MVC evaluation policy (2.4), whose parameter β allows us to adjust how greedy we want to be.

Moreover, we now continually expand our trajectory step by step, even when no obstacles are detected. Since all the nodes of the trajectory are now connected during planning (unlike in MiniTrees), this will grow a large ”mega” tree, which will hopefully explore a variety of potential solution nodes. Like in MiniTrees, we do this while maintaining the total number of node evaluations at each step at most the available planning budget N.

The main drawback of the MegaTree approach is that the practical implementation of the Value Search mechanism is less intuitive. In MiniTrees, it is sufficient to evaluate whether a node is a solution only at the time of its initial expansion. In contrast, we now must also check previously expanded nodes during selection, as they may have been created before an obstacle was detected. Additionally, since MegaTrees can identify obstacles much farther ahead than n steps, it may make sense to also conduct deeper detection rollouts when checking potential Value Search solution nodes. However, this would quickly consume a significant portion of the planning budget.

Due to the relatively lengthy implementation of this algorithm, which involves rewriting many of the steps already described for MiniTrees, we do not provide a full pseudocode description but instead list below the main practical differences:

- DetectionRollout (algorithm 8) has to be adjusted to append new nodes to the previously rolled out trajectory at each step, instead of rolling out from scratch.
- The nodes of the trajectory are not separated from each other, and the tree is expanded with a standard Selection procedure (algorithm 2) starting from the root, even when no obstacle is detected during the rollout.

- The Value Search has to also be checked on previously expanded nodes when they are selected during planning.
- Nodes should be evaluated with a construction decoupled evaluation policy (e.g., $\tilde{\pi}_{\text{MVC}}$). Then, GENERICBACKUP (algorithm 7) can be used instead of BACKUP to update the estimates.

Finally, we explored alternative versions of MegaTree that incorporated the detection rollout as a means to penalize value estimates, or their associated variance, along the agent's trajectory, thereby encouraging deviation without relying on the Value Search method. However, these variants performed poorly in early experiments, largely due to the limited information provided by a single-trajectory detection. Penalizing a node can push the agent to deviate immediately from the trajectory which brings to it, but this information is quickly lost afterwards, as the penalized node is no longer part of the planning tree and thus no longer contributes to value (or variance) estimation. Nonetheless, the idea of using the detection criteria as a way to adjust the planning estimates through penalties would make much more sense if we were able to dynamically detect multiple disruptions at each step without constraining the agent to follow a fixed trajectory while we do so. This is indeed the focus of the next section and proposed algorithm.

4.2 Penalty-Driven Deep Planning

As discussed, AZD methods allow us to detect an environment change far into the future while deterministically following the learned prior policy π_θ. Additionally, the Value Search mechanism provides a conceptually simple way of finding alternative paths that get around the change. However, the effectiveness of these methods is inherently dependent on how good π_θ is. If the prior is inaccurate with respect to the training environment, our agent would subsequently follow a suboptimal strategy even when no change has occurred yet. This can be problematic in complex environments where obtaining accurate prior estimates is hard.

Moreover, we always localize at most one problematic state each time we roll out the prior, but this does not give us any further information about, for example, the shape or size of this obstacle. If our agent is close to a wall, we might need to mark several paths that lead to it as problematic, rather than just a single one. This would provide us with more information, indicating whether the obstacle is a minor issue that can be easily overcome without a significant deviation from the rolled-out trajectory, or if a substantial change in direction is required.

Ideally, we would like to balance between the ability to detect changes far into the future and more standard planning that considers multiple paths, in a cleaner, single-phase algorithm. Following this intuition, we outline the steps taken to develop a change-aware planning method that leverages the detection mechanism without requiring a separate detection rollout.

4.2.1 Reusing the Previous Planning Tree

One main issue with standard MCTS, which we mentioned earlier in chapter 3, is that the previously built planning tree is discarded after the agent acts in the environment. This highly constrains the maximum depth that our planning trees can reach with limited simulation budgets, even when planning greedily. Let x_t be the root node of the planning tree constructed at step t, and a_t be the action that we are going to undertake in the environment after planning. In a fully deterministic setting, we can set $x_{t+1} = x_t \uplus a_t$ and resume planning from there, since we are sure that the corresponding state will match the one reached in the real environment. A more generic approach, however, is to directly check whose child of the previous root node (if any) holds the same state as in the environment and resume planning from there. Note that these two approaches are not always equivalent, as there might be multiple children of the root that correspond to the current state, and in the second approach, we should then decide which one to step into. Since our main goal is building increasingly deep trees, it then makes sense to choose the child whose corresponding subtree is the deepest, where the depth of a node's subtree is defined recursively as the "height" of the node:

$$x.height = 1 + \max_{x' \in x.children} x'.height$$

The recursion stops at leaf nodes whose height is set to 0 by definition. The tree re-usage mechanism is exemplified in Figure 4.3.

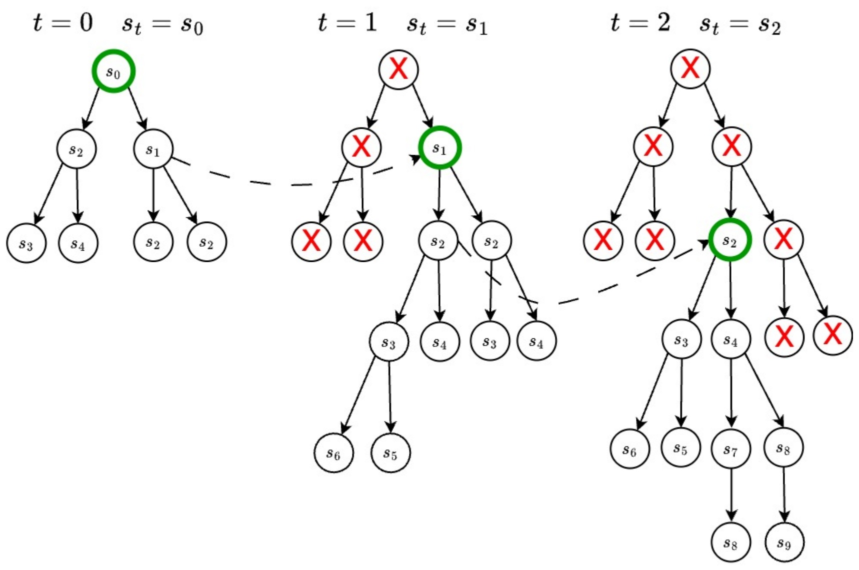


Figure 4.3: Example of the tree re-usage mechanism on a simple binary tree. The green node in each tree corresponds to the current state s_t in the real environment at step t. After step 1, we can reuse the right subtree of the previous root node as the corresponding child is the only one whose state is s_1. After step 2, we could reuse both children of the previous root, but we choose the left one as the corresponding subtree is deeper.

It is easy to see that this mechanism allows us to reuse many more nodes if the previous planning tree was deep rather than wide. As an example, the trees shown in Figure 4.4 are built by always expanding 1, 2, and 3 children per node, respectively. They are all complete trees of N nodes where each parent has a branching factor of exactly b children; then, each subtree of the root has exactly $\frac{N-1}{b}$ nodes.

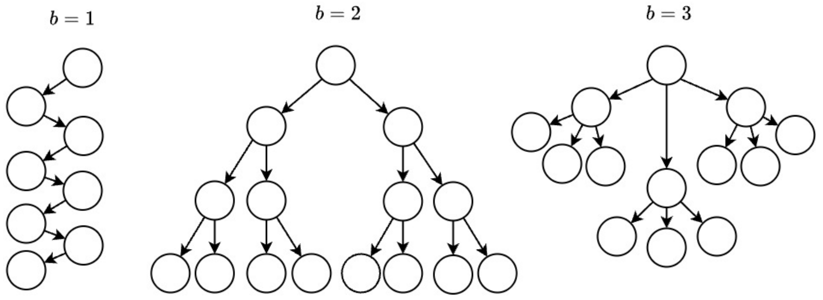


Figure 4.4: Three different trees with branching factor $b = 1$, $b = 2$, $b = 3$, respectively.

4.2.2 Integrated Change Detection

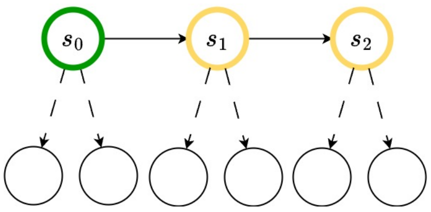
We propose to integrate the detection mechanism into the standard planning by dynamically checking our criterion on the trajectories followed by the selection policy π_{sel}. To better understand what we mean by this, it is important to recall that at each planning iteration, MCTS traverses the tree until reaching a node which is not fully expanded; then, it creates a child of this node by expanding an action according to an expansion policy, which is usually a uniform one. We should therefore apply our detection criterion to the selection trajectory before performing the expansion. It is important not to include the expanded node in the detection process, as this would result in computing our bootstrapped root value estimation on a trajectory built by two different policies. An example of the application of the $\mathcal{C}_{\text{max}}$ criterion during selection is shown in Figure 4.5. Two main challenges affect this approach:

1. In the formalization of our criteria, we assumed that the value estimates produced by v_θ should approximately correspond to what the followed policy π expects, i.e., $v_\theta \approx V_\pi$. This makes sense when following π'_θ in AZD methods, but is generally not the case for standard selection policies, especially since they incorporate an exploration term which does not depend on the value estimates.

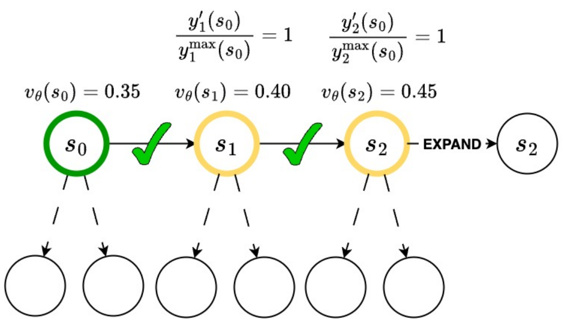
2. π_{sel} is usually non-stationary. For example, both the exploration term and the Q-value estimates used in π'_{UCT} and π'_{PUCT} to select an action dynamically change while we are planning[2].

The first problem can be addressed by using a completely greedy selection policy that simply selects the action to undertake from node x as $a_* = \arg\max_{a \in \mathcal{A}} \hat{Q}_{\tilde{\pi}}(x, a)$, where $\tilde{\pi}$ is

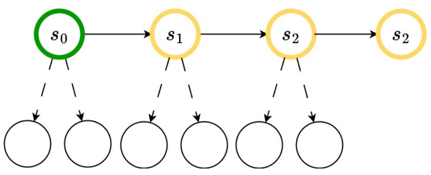
[2]Note that the exploration constant C is fixed, but the whole exploration term also depends on the visitation counts which evolve over time.



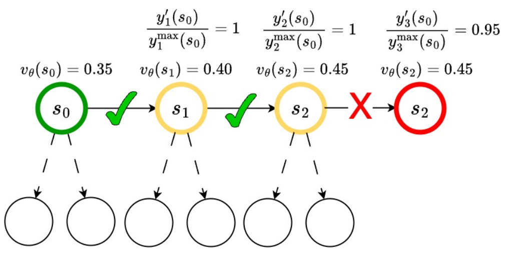
(a) The selection traverses the tree until a not-fully expanded node is reached.



(b) The criterion is checked on the traversed trajectory. As $\mathcal{C}_{\max}(\tau_2, 0.01, 0.95) = \perp$, no change is detected.



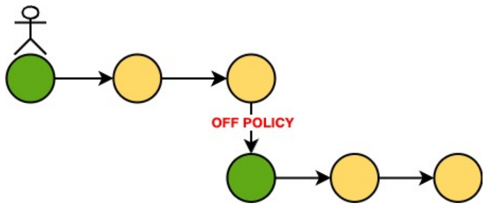
(c) The selection traverses the tree again until a not-fully expanded node is reached.


$\frac{y'_1(s_0)}{y_1^{\max}(s_0)} = 1$
$\frac{y'_2(s_0)}{y_2^{\max}(s_0)} = 1$
$\frac{y'_3(s_0)}{y_3^{\max}(s_0)} = 0.95$
$v_\theta(s_0) = 0.35$
$v_\theta(s_1) = 0.40$
$v_\theta(s_2) = 0.45$
$v_\theta(s_2) = 0.45$
s_0
s_1
s_2
s_2

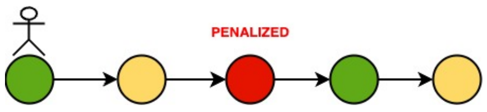
(d) This time, $\mathcal{C}_{\max}(\tau_3, 0.01, 0.95) = \top$ and we extract $t = 2$ as the last safe step. Therefore, the last node of the trajectory can be marked as problematic (red).

Figure 4.5: Example of integrated change detection during planning, with $\gamma = 0.95$ and $\epsilon = 0.01$. The green circled node represents the root of the planning tree. The yellow circled nodes are traversed by the selection policy.

the evaluation policy that we use to estimate Q-values as described in chapter 2. The second problem can be addressed by "interrupting" the criterion any time we select an action that is suboptimal according to π'_θ. Strictly speaking, this would involve checking at every selection step whether or not $\arg\max_{a \in \mathcal{A}} \hat{Q}_{\tilde{\pi}}(x, a) = \arg\max_{a \in \mathcal{A}} \pi_\theta(x, a)$. Especially in environments with large action spaces, however, this is probably too strict of a constraint since there might be several optimal or close-to-optimal actions, and it is therefore unlikely that the selection policy will match the choice of π'_θ for many consecutive steps. Alternatively, we can check whether the policy logit $\pi_\theta(x, a_*)$ exceeds a fixed threshold ζ, indicating that the selected action is sufficiently aligned with π_θ, allowing us to continue applying our criterion. If we instead follow an off-policy action with respect to the prior policy, we should stop detection. We can, however, resume it from the next node; if no other off-policy action is taken from there on, that is a valid trajectory on which to apply our criterion. An example of this is visualized in Figure 4.6a.



(a) The selection follows an off-policy action w.r.t. the prior. We restart detection from the next node.



(b) The selection traverses an already penalized node. We restart detection from the next node.

Figure 4.6: Special cases in detection. The agent sits on the root node, which is the node for which we estimate the n-step value for detection, unless one of these two special cases occurs.

4.2.3 Penalizing Detected Changes

Since the detection process is now continuously applied during tree construction, we may detect numerous problematic transitions along several planning trajectories. Having identified them, we might now leverage such information in several ways. A simple option is to directly penalize the value estimate of nodes located immediately after the problematic transitions, as the red circled node in Figure 4.5d. We do so by modifying their prior value estimate from $v_\theta(x)$ to $v_\theta^p(x) = v_\theta(x) - p$. The parameter p can be tuned depending on the nature of the changes that we assume could show up and the scale of the value estimates. Although simple, applying the penalties in this way has a profound impact on subsequent planning. To understand why, recall that given the evaluation policy $\tilde{\pi}$ that we are employing (e.g., $\tilde{\pi}_N$ or $\tilde{\pi}_{\text{MVC}}$), the node Q values are updated during the backup in the following way:

$$\hat{Q}_{\tilde{\pi}}(x) = r(x) + \gamma\, \tilde{\pi}(x, a_v)\, v_\theta(x) + \gamma \sum_{a \in \mathcal{A}} \tilde{\pi}(x, a) \hat{Q}_{\tilde{\pi}}(x \uplus a)$$

Where we now substitute $v_\theta(x)$ with $v_\theta^p(x)$ if x is marked as problematic. This means that even if a node has been penalized, the overall Q value estimate can still increase if its children have a high value. We believe this to be a suitable property since it avoids fully removing exploration pressure from the problematic nodes and underlying subtrees, while simultaneously encouraging the agent to explore alternatives. If the agent's planning focus is on a portion of the environment that presents several detected changes, then multiple nodes will be affected by the penalties. This should cause the agent to gradually shift its planning focus to other promising regions of the state space, also considering actions with low associated prior estimates.

Once a node is penalized, we no longer want to consider it in subsequent detections, as its value has decreased and no longer reflects the expected value of the prior policy. Similarly to the case where an off-policy action is followed, we can still apply the detection to the rest of the trajectory as shown in Figure 4.6b.

The pseudocode detailing the integration of the change detection and penalization within selection is reported in algorithm 11.

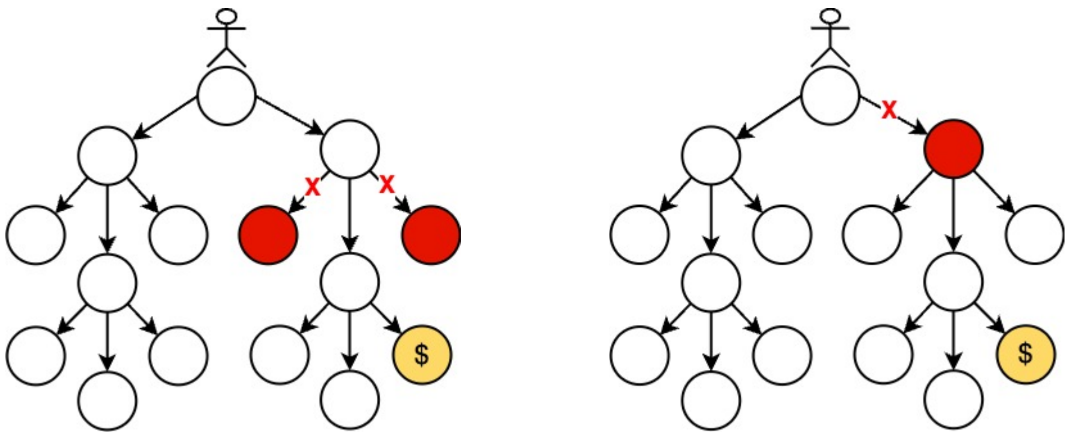
Algorithm 11: PDDPSelection: Traverses the tree using a selection policy while applying change detection and penalization.

```
Input: Root node $root$, discount factor $\gamma$ , selection policy $\pi_{\text{sel}}$, prior policy $\pi_\theta$,
       tolerance $\epsilon$, prior threshold $\zeta$, value penalty $p$
Output: Node to expand $expNode$, action $expAction$

1  Initialize $node \leftarrow root$, $\tau \leftarrow [root]$, $i \leftarrow 0$, $R \leftarrow 0$, $y_{\max} \leftarrow node.value$
2  while not node.terminal do
3      Select action $a$ using selection policy $\pi_{\text{sel}}$ on $node$
4      if $a \notin keys(node.children)$ then
5          break
6      $node \leftarrow node.children\{a\}$
7      $R \leftarrow R + \gamma^i \cdot node.r$
8      $\tau$.append($node$)
9      $i \leftarrow i + 1$
10     if $(\pi_\theta(node.parent.s, a) < \zeta) \vee$ node.parent.penalized then
11         We restart detection from the current node:
12         $i \leftarrow 0$, $\tau \leftarrow [node]$, $R \leftarrow 0$, $y_{\max} \leftarrow node.value$
13     else
14         $y \leftarrow R + \gamma^i \cdot node.value$
15         if $y > y_{max}$ then
16             $y_{\max} \leftarrow y$
17 if $(\frac{y}{y_{max}} < 1 - \epsilon) \wedge (\neg node.penalized)$ then
18     $t \leftarrow \max(\left\lfloor i - \frac{\ln(1-\epsilon)}{\ln(\gamma)} \right\rfloor, 0)$
19     $changeStep \leftarrow \min(t + 1, i)$
20     $changeNode \leftarrow \tau[changeStep]$
21     $changeNode.penalized \leftarrow True$
22     $changeNode.value \leftarrow changeNode.value - p$
23 return $node, a$
```

4.2.4 Pseudo-Deterministic Evaluation

The last piece of the puzzle that we want to introduce in our algorithm is the adoption of a special evaluation policy, which changes the way we estimate nodes' Q values during planning. We will first define the policy and then explain the reasons behind this particular choice. The special pseudo-deterministic policy $\mathring{\pi}$ can be defined starting from a generic



(a) Penalties are ignored as they are not on the path to the goal.

(b) The penalty is on the path to the goal, and will influence the agent's decision.

Figure 4.7: Two examples of the way our pseudo-deterministic evaluation policy "sees" the penalized nodes.

evaluation policy $\tilde{\pi}$ over the extended action space $\mathcal{A}_v$:

$$\dot{\pi}_{\tilde{\pi}}(x,a) = \begin{cases} \frac{N(x)-1}{N(x)} & \text{if } a = \arg\max_{a \in \mathcal{A}} \tilde{\pi}(x,a) \\ \frac{1}{N(x)} & \text{if } a = a_v \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

The rationale behind this policy is simpler than its relatively uncommon definition; in fact, $\dot{\pi}$ turns any stochastic evaluation policy $\tilde{\pi}$ into a pseudo-deterministic policy whose only non-zero logits are the best real action ($\in \mathcal{A}$) according to $\tilde{\pi}$ and the special action a_v, which is assigned a probability of $\frac{1}{N(x)}$ that fades away as node visits increase [3]. Using such a policy to estimate node Q-values with (2.3) results in the following recursion:

$$\hat{Q}_{\dot{\pi}}(x) = r(x) + \gamma \frac{v_\theta(x)}{N(x)} + \gamma \frac{N(x)-1}{N(x)} \hat{Q}_{\dot{\pi}}(x \uplus a_*) \tag{4.2}$$

where $a_* = \arg\max_{a \in \mathcal{A}} \tilde{\pi}(x,a)$. Intuitively, this recursive formulation evaluates a node solely based on its most promising underlying path, rather than a weighted average of the entire subtree, while maintaining a prior value term that fades away as the number of visits increases. There are two main reasons for doing this:

- This is a very greedy approach which should further encourage deep planning, allowing us to detect changes farther away.

- In certain situations, this approach can cope better with the agent's goal as well as our way of penalizing problematic nodes.

Algorithm 12: PDDP Algorithm

```
Input: Environment model $env$, previous root node $root$, prior policy $\pi_\theta$, value
       network $v_\theta$, selection policy $\pi_{\text{sel}}$, evaluation policy $\dot{\pi}$, planning budget $N$,
       discount factor $\gamma$, tolerance $\epsilon$, prior threshold $\zeta$, value penalty $p$
Output: Root node $root$ with updated statistics
1  if $root$ is None then
2      $root \leftarrow Node(env)$
3  else
4      Select the child of $root$ with the deepest subtree among the ones whose
5      state corresponds to $env.s$. If there is no such node, start planning from scratch.
6  $visits \leftarrow root.visits$
7  while $root.visits - visits < N$ do
8      $expNode, expAction \leftarrow$ PDDPSelection(root, $\gamma$, $\pi_{\text{sel}}$, $\pi_\theta$, $\epsilon$, $\zeta$, $p$)
9      if $expNode.terminal$ then
10         GenericBackup($expNode$, $\gamma$, $\dot{\pi}$)
11     else
12         $child \leftarrow$Expand($expNode, expAction$)
13         if $child.terminal$ then
14             $child.value \leftarrow 0$
15         else
16             $child.value \leftarrow v_\theta(expNode.s)$
17         GenericBackup($child$, $\gamma$, $\dot{\pi}$)
18 return $root$
```

We motivate the latter with a simple example. Figure 4.7a shows a situation where following the right child of the root node can eventually bring the agent to the goal (represented with a dollar). However, two out of the three actions that can be undertaken from it triggered our detection criterion during planning, which therefore penalizes the associated values (red nodes). Our standard way of estimating nodes' Q-values (according to a stochastic evaluation policy) might therefore doom the right child because of its many penalized children, preferring the left one, which, however, does not retain any path to the goal underneath. Instead, the new method for evaluating Q-values would ignore the penalties since it would only consider the non-penalized sibling and therefore correctly lead the agent to the goal. Note that this situation is different from the one visualized in Figure 4.7b, where the penalty is instead applied directly to the right child of the root, which is now a problematic node. In this case, the new value estimation method would not ignore the penalty, which is reflected in the prior term of our formula. Depending on the scale of the penalty, this could prevent the agent from stepping into it, which now makes more sense since the detected

[3]Note that the probability $\tilde{\pi}(x, a_v)$ for any node x is sometimes (as in [12]) assumed to be equal to $\frac{1}{N}$ regardless of the utilized policy $\tilde{\pi}$, so our definition does not constitute an exception in that sense.

change is right on the path to the goal, and it might be better to look for an alternative solution.

It is important to underline that (4.2) is not equivalent to what we would obtain by simply plugging a standard deterministic evaluation policy $\tilde{\pi}'$ in the original recursive formulation (2.3), in which case our estimate would become:

$$\hat{Q}_{\tilde{\pi}'}(x) = r(x) + \gamma\, \hat{Q}_{\tilde{\pi}'}(x \uplus a_*)$$

where now a_* is selected from the whole extended action space $\mathcal{A}_v$, i.e., $a_* = \arg\max_{a \in \mathcal{A}_v} \tilde{\pi}(x, a)$. The reason we do not prefer this option is that, in most cases, the prior term would be ignored by the argmax, and we would completely lose the influence of the penalties. Instead, (4.2) guarantees that the prior term only fades away as we keep expanding the corresponding subtree.

This completes our description of the Penalty-Driven Deep Planning algorithm (PDDP). The overall algorithm pseudocode is reported in algorithm 12.

4.3 Is Depth All You Need?

The algorithms presented in section 4.1 (AZD algorithms) and section 4.2 (PDDP) have two major commonalities:

1. They both try to detect unexpected changes to the environment.
2. They both try to build deep planning trees.

For several reasons that we already discussed, it should now be clear that the first cannot be truly effective without the second. This will be further demonstrated by the experiments presented in chapter 7. However, we might wonder whether standard planning methods could be adjusted to construct so deep trees that they can update the online value estimates more quickly without the need for explicit change detection mechanisms. For example, imagine our agent keeps planning towards a wall without even trying to look elsewhere, repeatedly bumping into it. This will adjust the corresponding value estimates along the unsuccessful trajectories much faster than if we tried to look in different directions. To achieve extremely far-sighted planning without the need to increase our planning budget, which is a trivial but not always feasible solution, we propose combining three modifications to standard AZ/MCTS planning.

4.3.1 Greedy Selection

In standard AlphaZero planning, we typically aim to strike a balance between very high values of C, which lead to essentially uniform planning, and very low values of C, which plan optimistically with respect to the learned value function, potentially an issue if such a value function is not accurate enough. At first thought, we could hypothesize that a more pessimistic approach, exploring more actions per node, might be a better approach if the learned value function is damaged by some change. In the end, this makes sense if the planning budget still allows us to conduct a sufficiently deep search that ”sees” the changes early enough. However, when the budget is limited, we might not even encounter such changes until we are very close to them, which could be too late to act accordingly. If that is the case, building a very deep tree ($C \approx 0$) might allow us to detect that something is off in our prior estimates much earlier. That should subsequently be reflected in the planning value estimates, which should slowly turn the planning focus in a different direction. Still, if the budget is insufficient, we might not be able to explore anything beyond the actions that are best according to the training.

4.3.2 Reusing the Previous Planning Tree

We again propose reusing the previous planning tree, exactly as we did for PDDP in subsection 4.2.1. As mentioned, one of the reasons it is difficult to build deep search trees in MCTS is that we usually immediately discard the previous planning tree after stepping into the environment. There are some key reasons why this is the most common practice in the literature:

- **Model inaccuracy**: If our transition model is inaccurate, we might accumulate planning errors and end up modeling highly inaccurate transitions.
- **Memory and time constraints**: Building an incrementally deeper tree requires recursively traversing and backing up from much longer trajectories, which might be undesirable if there are specific time constraints. It also requires more memory to maintain the increasing number of nodes.

While we acknowledge these potential limitations, we still believe that allowing our agent to reuse part of the previous planning tree can significantly reduce the amount of planning budget needed to properly plan in modified environments, particularly when combined with a greedy selection policy. Moreover, specific memory and/or recursion limits could simply be addressed by setting a maximum number of consecutive tree reuses, after which we can drop the previously constructed tree and start planning from scratch.

4.3.3 Blocking Loops

In certain environments, especially single-agent games, it is common to fall into loops during planning. This can happen, for example, if our agent is stuck in a corner and keeps bumping into the wall. We define a loop as the repetition of the same state along a single trace from the root to a leaf; moreover, we refer to the second, redundant occurrence as the node ”closing” the loop. Intuitively, planning from a repeated state is rarely useful, as we could simply take the same steps from the equivalent appearance of the state in a node earlier in the trace.

Moerland et al. [22] proposed an alternative version of MCTS where they incorporated the concept of variance of a node and modified the selection policy so that nodes with higher

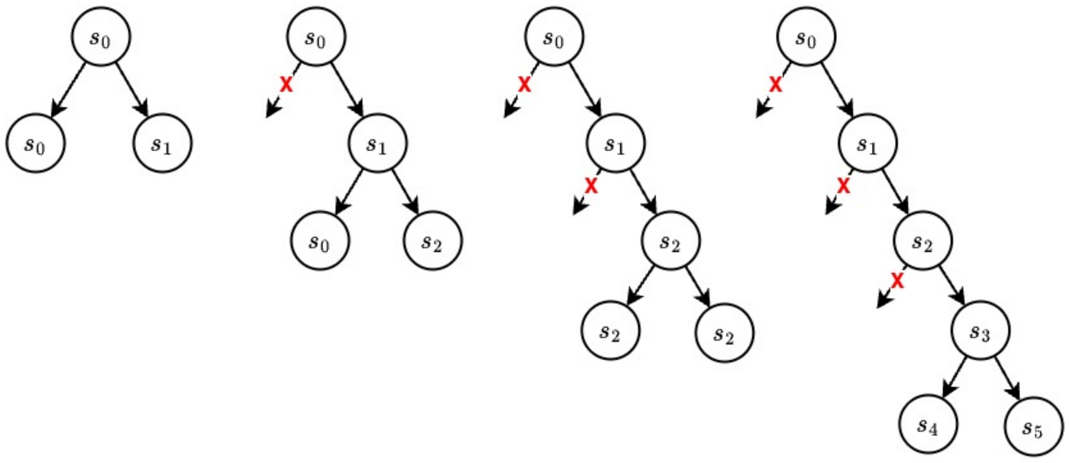


Figure 4.8: Example of loop blocking mechanism. From left to right, we expand two nodes at a time and block exploration through actions that led to a loop.

variance are more frequently explored [4]. Then, the variance of nodes closing a loop is set to zero as soon as they are created, immediately stopping the exploration pressure from being wasted through them. Another option for implementing the same principle is to directly remove the edge corresponding to the parent action of the node that closes the loop, thereby forcing the agent to no longer choose that action from the parent node. This can be easily implemented, for instance, via action masking and subsequent renormalization of the policy.

A visual example of the loop blocking mechanism is provided in Figure 4.8. After blocking the loop, we might still backup its estimated value normally, but we decide to backup a value of zero instead. Note that a more theoretically principled way of estimating the value of a repeating state would be transferring information from its first occurrence to the repeated one. However, this can complicate the implementation, and backing up a value of zero instead can decrease the Q estimate of nodes that lead to a large number of loops, which is usually a desirable property.

In environments with continuous state spaces or very large discrete spaces, exact loops are rare. In those cases, we could instead check whether $||s - s'||_2 \leq \eta, \;\; \eta \in \mathbb{R}$ for any pair of nodes s, s' along the planning trajectory. Note that setting $\eta = 0$ corresponds to only blocking exact loops.

By combining greedy construction, previous tree reuse, and the loop blocking mechanism, we create the Extra-Deep Planning algorithm (EDP). The algorithm is detailed in algorithm 14 (main) and algorithm 13 (selection with nodes tracking).

[4]Note that this is a different way of characterizing the variance from what we described in chapter 2.

Algorithm 13: EDPSelection: Tracks nodes during selection for loop blocking.

```
Input: Root node $root$, selection policy $\pi_{\text{sel}}$
Output: Node to be expanded $expNode$, action to expand it $expAction$, set of
        traversed states $visited$
1  Initialize $node \leftarrow root$, $visited \leftarrow Set(root.s)$
2  while not $node.terminal$ do
3      Select action $a$ among non-masked actions using $\pi_{\text{sel}}$ on $node$
4      if $a \notin keys(node.children)$ then
5          break
6      $node \leftarrow node.children\{a\}$
7      $visited.add(node.s)$
8  return $node, a, visited$
```

Algorithm 14: EDP Algorithm

```
Input: Environment model $env$, previous root node $root$, value network $v_\theta$,
       planning budget $N$, discount factor $\gamma$, selection policy $\pi_{\text{sel}}$, eval. pol. $\tilde{\pi}$
Output: Root node $root$ with updated statistics
1  if $root$ is None then
2      $root \leftarrow Node(env)$
3  else
4      Select the child of $root$ with the deepest subtree among the ones whose state
5      corresponds to $env.s$. If there is no such node, start planning from scratch.
6  $visits \leftarrow root.visits$
7  while $root.visits - visits < N$ do
8      $expNode, expAction, visitedStates \leftarrow$ EDPSelection(root, $\pi_{\text{sel}}$)
9      if $expNode.terminal$ then
10         GenericBackup($expNode$, $\gamma$, $\tilde{\pi}$)
11     else
12         $child \leftarrow$Expand($expNode, expAction$)
13         if $child.s \in visitedStates$ then
14             $child.parent.mask[expAction] \leftarrow 0$
15             $child.value \leftarrow 0$
16         else if $child.terminal$ then
17             $child.value \leftarrow 0$
18         else
19             $child.value \leftarrow v_\theta(expNode.s)$
20         GenericBackup($child$, $\gamma$, $\tilde{\pi}$)
21 return $root$
```

Chapter 5

Related Work

5.1 Planning with Imperfect Estimators

As already discussed throughout this work, AlphaZero algorithms have traditionally been applied to unmodified environments at deployment. As such, the specific challenges explored in this thesis have, to the best of our knowledge, received relatively limited attention in the existing literature. However, Min and Motani [20] proposed a simple benchmark environment named Brick Tic-Tac-Toe, inspired by the Tic-Tac-Toe game, for evaluating AlphaZero's performance on novel test configurations. Their results demonstrate how training AlphaZero on a set of diverse training configurations can help it adapt to novel test configurations. However, their setup differs from ours, as brick tic-tac-toe is a two-player game, whereas we experimented with single-agent environments. Moreover, the paper does not propose new approaches to address the problem of generic environment changes, except for varying the training game configurations during self-play. Lan et al. [17] evaluate the robustness of AlphaZero agents to adversarial perturbations in the state space during deployment, specifically for the game of Go. However, their definition of perturbations is tailored to the structure of Go, and no alternative modification to the AZ framework or novel algorithm is proposed to address them. Pettet et al. [24] tackle the challenge of combining online search with a previously learned, partially outdated policy to operate in non-stationary environments. They introduce Policy-Augmented MCTS (PA-MCTS), in which the agent selects actions based on a convex combination of a previously learned Q-value function and online estimates from standard MCTS. The key distinction between their approach, our methods, and standard AlphaZero lies in how prior training information is used: PA-MCTS incorporates it outside the planning tree, whereas AlphaZero and our approaches integrate it within the planning process itself.

5.2 Modified MCTS Planning

While our work proposes several novel modifications of AlphaZero/MCTS planning, such as the Value Search mechanism used in AZD methods (subsection 4.1.1) or the detection-based value penalties applied in PDDP (subsection 4.2.2, subsection 4.2.3), part of our contribution also consists in combining and integrating into our framework existing approaches that had only been separately applied before. As mentioned, the idea of blocking loops during planning, leveraged by our EDP algorithm, was first proposed in [22] to enhance their MCTS-T algorithm. However, this was implemented differently by integrating it into their variance estimation method and testing it only on purely online MCTS (without prior estimators). Other MCTS-related works have also explored combining their novel contributions with the tree reuse mechanism described in subsection 4.2.1. For instance, Lathrop et al. [9] propose applying tree reuse as an additional enhancement to their Subgoal-MCTS algorithm. However, the core of their framework, as well as its main objectives, profoundly differ from ours: they aim to improve MCTS efficiency in complex environments by integrating automatically discovered macro-actions that lead to subgoals. Another related example is Model Predictive Trees [18], which incorporates tree reuse in a framework that can analyze model mismatch over time, for instance, due to evolving system dynamics, and decide whether to partially reuse past information or fully reset the tree. It may be possible to integrate their contribution into our framework to make it robust to cases where the model does not fully match the underlying environment dynamics, something which we have not experimented with in this thesis. It would also possible to view MCTS trees as directed (acyclic) graphs and apply graph techniques to reuse information across different branches, such as transposition tables [25]. Finally, our newly proposed algorithms try to build deep planning trees by mostly being greedy during selection; other fundamentally different methods have been proposed in the literature to achieve more generic MCTS-based deep exploration, such as Epistemic-MCTS (EMCTS) [23], which propagates the epistemic uncertainty of a learned model and/or value function throughout the search. Note, however, that there is ”no free lunch” [38], which in the MCTS context means that most of the modifications to the general algorithm will introduce a bias that might benefit certain applications and be detrimental to others [32].

5.3 Planning with an Imperfect Model

While the focus of our work is on cases where the learned value/policy is imperfect but the model of the environment correctly predicts the agent transitions, a parallel branch of research tries to address the case where the model itself is not fully correct. An early work in this direction is [1], where the authors propose a hybrid approach that updates a previously learned approximate model of the environment through real-world evaluation and correction. Robotics researchers have developed recent advancements in this direction; for instance, CMAX [34] adapts its planning strategy online to address inaccuracies in the model without requiring explicit modifications to the model. It does so by biasing the planner away from regions whose dynamics are inaccurately modeled. A later evolution, named CMAX++ [35], integrates real-world experience through model-free learning into the standard CMAX planning algorithm, resulting in a hybrid approach that improves upon the baselines in several simulated robotic tasks. MCTS-based approaches to the problem of inaccurate simulators also exist, one example being robust-MCTS (rMCTS) [26], which plans to optimize a worst-case scenario of the imprecision of the model.

Chapter 6

Experimental Setup

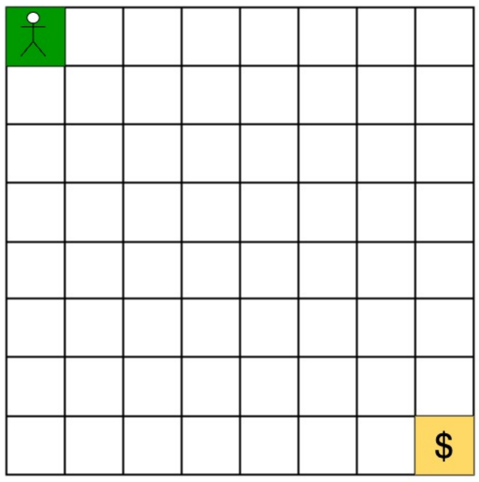
6.1 Training

As our work focuses on modifying the behavior of an agent at test time, we train our policy and value functions using a standard AZ agent and following the training procedure described in subsection 2.3.3. Specific implementation details and utilized hyperparameters are detailed in section A.1. Tree construction and evaluation are carried out by using the standard PUCT and visitation counts policies, respectively. For each of the training environments described below, we train our AZ agent 10 times with different seeds. This is crucial, as the remaining policy and value errors after training will differ for each seed. The learned value and policy functions for the grid world environments that we are going to introduce can be easily visualized; therefore, we display their average statistics in the appendix (subsection A.1.5). Both our detection criteria and planning methods should be robust to these error variations in order to be reliable. Note that it may be possible to further tune the training parameters to obtain nearly perfect prior estimates in these toy environments. Nonetheless, we believe this would not benefit the analysis and subsequent conclusions of this thesis, since learned value and policy functions in more complex environments are usually far from being perfect, even when trained until convergence. By running evaluation experiments with multiple imperfect estimators, we can make more general considerations that do not require the assumption of a perfectly trained agent.

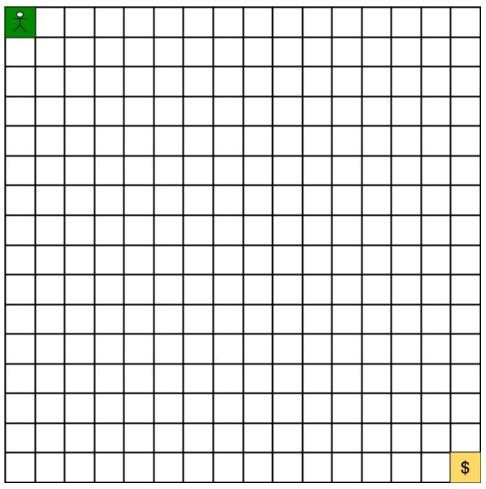
6.1.1 Environments

Empty Grid Worlds

As our first training environments, we employ two grid worlds of size 8×8 and 16×16, respectively. In both cases, the agent has to navigate from the start position, located in the upper-left corner, to the goal position, situated in the lower-right corner. The action space is discrete and consists of the 4 actions $\{\text{Left}, \text{Down}, \text{Right}, \text{Up}\}$. The goal state is the only terminal state, and trying to move towards the boundaries of the environment results in staying in the same state. The two configurations are represented in Figure 6.1. Although simple, these training environments can be modified with arbitrary obstacles at test time,



(a) 8×8 Empty Grid World

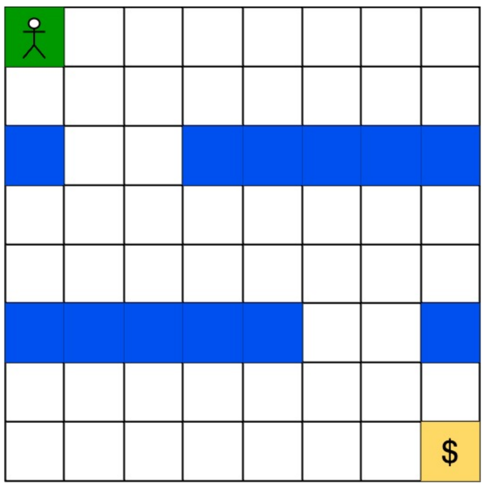


(b) 16×16 Empty Grid World

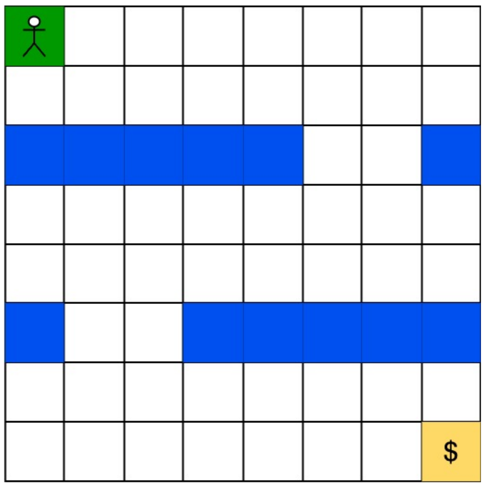
Figure 6.1: Empty Grid World Environments. The green cell represents the initial state. The yellow cell represents the goal state.

providing an intuitive and explainable setting where the optimal value function is based on a Manhattan distance. The reasons for testing on two configurations of largely different sizes will be evident later, but in essence, we aim to determine to what extent the performance of our algorithms is influenced by the scale of the configuration and the obstacles that will be added.

Mazes



(a) 8×8 MAZE_LR

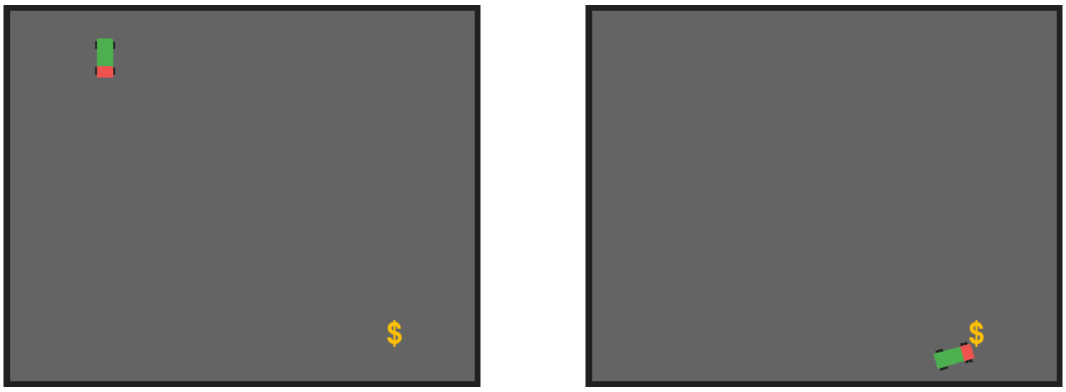


(b) 8×8 MAZE_RL

Figure 6.2: Maze Grid World Environments. The blue cells represent obstacles that the agent cannot surpass.

We further trained our agents on two additional grid worlds where we placed obstacles to form different small mazes. These configurations are depicted in Figure 6.2. Note that Maze Right-Left (MAZE_RL) is more challenging to learn than the specular Maze Left-Right (MAZE_LR), as it requires following a longer and more complex trajectory to reach the goal. The interesting aspect of these configurations is that we can move the two "doors" to different positions and observe how much the agents can adapt to such changes. They also let us evaluate our algorithms in a setting where the learned value function is relatively more complex compared to the empty grid training.

The CarGoal Environment



(a) Possible starting position. (b) Car reaching the goal.

Figure 6.3: CarGoal environment visualized. The goal position is indicated with a dollar.

While our main quantitative experiments focus on the previously described grid worlds, we also include additional results using a customized CarGoal environment [1], visualized in Figure 6.3. In this training environment, the car starts from a random position and heading angle and has to reach the goal indicated with a dollar. We consider this an important extension of our evaluation, as CarGoal introduces a higher degree of complexity compared to the grid worlds due to three key features:

- Continuous state space.
- Larger action space.
- Car dynamics.

The environment models a single-track vehicle (bicycle model), where the state of the car is represented as a quadruple:

$$\mathbf{s} = [x, y, \theta, v]$$

[1]Implementation largely based on the repository: https://github.com/KexianShen/parking-env.git

Here, x and y are the spatial coordinates, θ is the heading angle, and v is the linear velocity.

The control input is a pair:

$$\mathbf{u} = [a, \delta]$$

where a is the acceleration and δ is the steering angle.

To ensure compatibility with the AZ framework, which requires a discrete action space, both a and δ are sampled from predefined discrete sets. Each possible combination of a and δ is then mapped to a unique discrete action. To balance approximation accuracy with action space size, we restrict the choices to $a \in \{-1, 1\}$ and $\delta \in \{-\frac{\pi}{4}, -\frac{\pi}{8}, 0, \frac{\pi}{8}, \frac{\pi}{4}\}$, resulting in 10 total combinations.

Time is discretized with a fixed step size $\Delta t = 0.25$, and the system evolves according to the following kinematic equations:

$$\begin{aligned} v_{t+1} &= \text{clip}(v_t + a \cdot \Delta t, -v_{\text{max}}, v_{\text{max}}) \\ \beta &= \tan^{-1}\left(\frac{1}{2}\tan\delta\right) \\ x_{t+1} &= x_t + v_{t+1} \cdot \cos(\theta_t + \beta) \cdot \Delta t \\ y_{t+1} &= y_t + v_{t+1} \cdot \sin(\theta_t + \beta) \cdot \Delta t \\ \theta_{t+1} &= \theta_t + \frac{v_{t+1}}{L/2} \cdot \sin(\beta) \cdot \Delta t \end{aligned}$$

Where:

- β is the slip angle approximation resulting from the steering input.
- $L = 4.8$ is the car length.
- The maximum velocity is set to $v_{\text{max}} = 4.0$. Starting from $v_0 = 0$ and with the chosen Δt, it therefore takes 16 consecutive positive accelerations to reach maximum velocity.

We remark that, as in all the grid world environments, training was repeated for 10 different seeds.

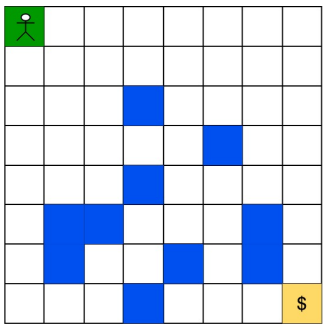
More details on the implementation of the CarGoal environment can be found in section A.1.4.

6.2 Evaluation

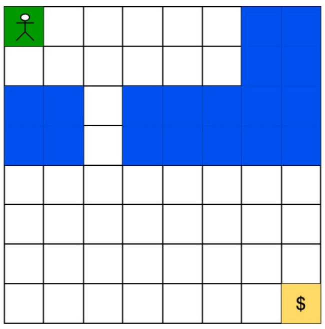
6.2.1 Test Configurations

Empty Grid Worlds $\rightarrow$ Added Obstacles

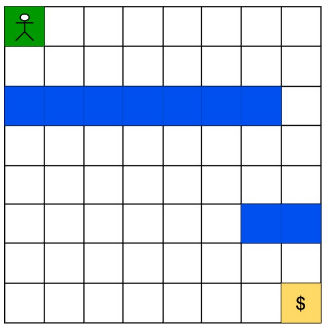
In these test configurations, we modify the 8×8 and 16×16 empty grid words as shown in Figure 6.4 and Figure 6.5, respectively.



(a) SPARSE 8x8

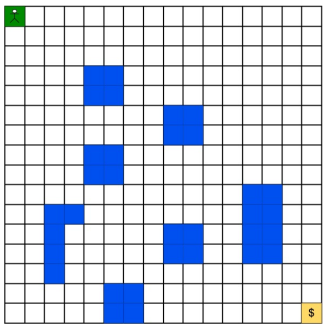


(b) NARROW 8x8

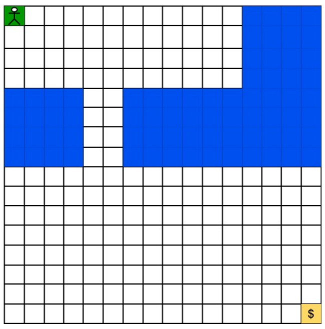


(c) SLALOM 8x8

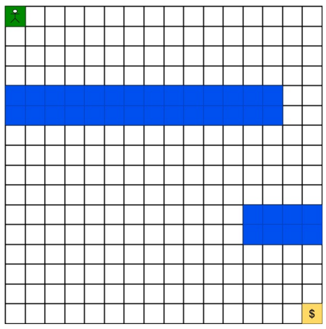
Figure 6.4: 8×8 test challenges for agents trained on Figure 6.1a



(a) SPARSE 16x16



(b) NARROW 16x16



(c) SLALOM 16x16

Figure 6.5: 16×16 test challenges for agents trained on Figure 6.1b

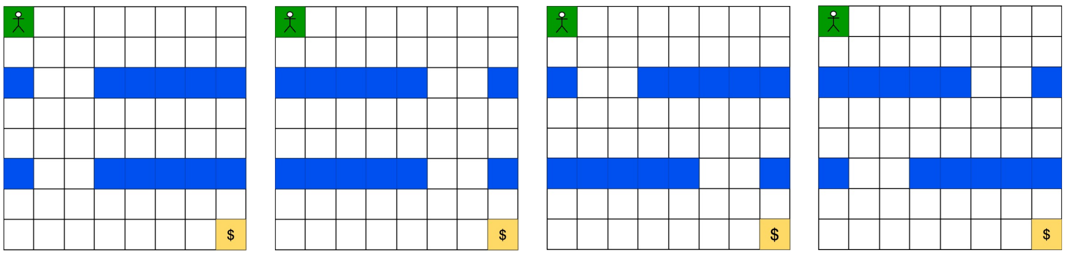
The rationale behind the choice of the obstacles' placement is reflected in the name of each of them and aims to test the agent's ability to deal with different challenges:

- The SPARSE configurations aim at testing the agent's ability to get around relatively small, sparse obstacles. In these configurations, a large portion of the paths that were optimal in the training environment remain optimal and can potentially be followed by the agent. Therefore, we consider this to be the easiest test setting among the three, both in the 8×8 and 16×16 versions.

- The NARROW configurations challenge the agent to find a (narrow) free path through the placed wall. Missing such an entrance brings the agent to a "dead end" which is hard to escape, especially in the 16×16 version. Clearly, the placement and size of the entrance, as well as the depth of the dead end, can have an impact on the agent's performance. According to early experimentation, the visualized setting appeared to pose a significant challenge for baseline AZ; therefore, we stick to it for our main experiments. Note that, while considerably fewer than in SPARSE, some of the optimal paths for training are still optimal in the NARROW test configurations.

- The SLALOM configurations require the agent to perform a ”slalom” in order to reach the goal. This can only be achieved by taking a certain number of actions in a direction that is highly suboptimal in the training environment. Therefore, we consider this the hardest configuration among the three, specifically designed to test whether our algorithms can perform sufficiently well in situations where the change to the environment is significant and cannot be avoided by following any previously effective strategy.

Mazes → Modified Mazes

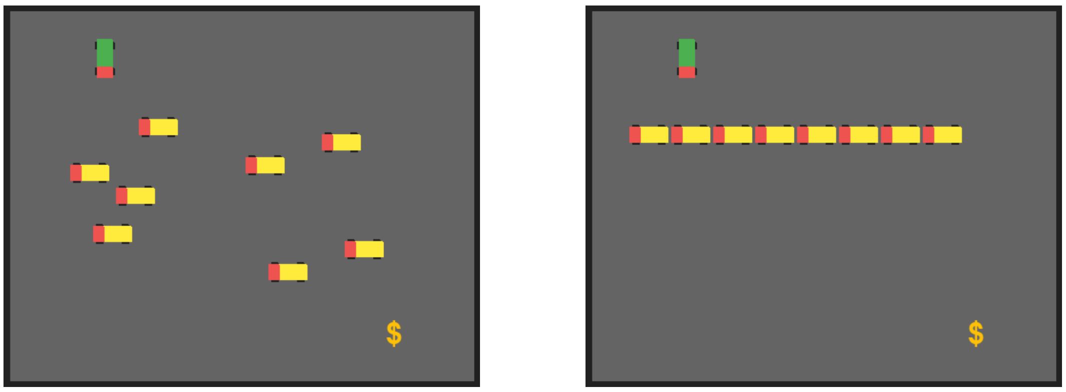


(a) MAZE_LL 8x8 (b) MAZE_RR 8x8 (c) MAZE_LR 8x8 (d) MAZE_RL 8x8

Figure 6.6: MAZE test challenges for agents trained on Figure 6.2a and Figure 6.2b

As anticipated, we test our agents on the mazes by moving the position of the entrances as shown in Figure 6.6. Clearly, moving both entrances (MAZE_LR → MAZE_RL or MAZE_RL → MAZE_LR) represents a harder challenge than moving only one.

CarGoal → Added Obstacles



(a) SPARSE obstacles configuration. (b) HORIZONTAL obstacles configuration.

Figure 6.7: CarGoal test configurations. Obstacles are represented by the parked (static) yellow cars.

We can conduct an evaluation on the CarGoal environment by adding new obstacles, as done with the grid world environments. We choose two different obstacle configurations, named SPARSE and HORIZONTAL, both of which are reported in Figure 6.7. Note that the starting position of the agent is now fixed to the upper-left corner as in the figures (while it was randomized during training). The rationale behind the choice of these two configurations is similar to the one behind some of the grid world ones; SPARSE should represent a challenge because of the many obstacles, which are, however, easy to get around, while HORIZONTAL requires the car to follow a more specific path to avoid bumping into the long queue of cars in front of it.

Importantly, the agent bumping into an obstacle (or the wall) results in its velocity immediately being set to zero. While we do not set a negative reward for bumping, which would likely make the test configurations relatively easier, the fact that the observed velocity drops may have a strong effect on the NN value estimates.

The reason why we consider these CarGoal experiments interesting is that the car, contrarily to the grid world agent, cannot avoid an obstacle by changing direction immediately in front of it. It must instead realize that it is about to hit it well in advance and progressively adjust both its velocity and steering angle.

6.2.2 Agents

Baselines

We evaluate the performance of our methods against the following four baseline agents:

- Standard AlphaZero (**AZ+PUCT**): This is the baseline agent also used during training, combining the visitation counts evaluation with PUCT for node selection.
- AlphaZero without prior policy (**AZ+UCT**): This variant removes the use of the prior policy network and selects nodes using standard UCT. We consider this an equally relevant baseline because, depending on how much the test configuration diverges from the training environment, avoiding reliance on the prior policy might already lead to improved results.
- AlphaZero with MVC evaluation (**MVC+PUCT**): Some of our proposed algorithms use the MVC policy for evaluating the search tree and estimating node Q-values. To fairly assess their components, we include a baseline AZ agent using MVC in the same way.
- AlphaZero with MVC evaluation and without prior policy (**MVC+UCT**): As done for standard AZ, we include a baseline that uses MVC for tree evaluation but relies on UCT for node selection.

Note that the performance of the standard AZ agents is affected by the choice of the exploration parameter C used during tree construction. This was tuned separately for each of the training configurations, as reported in detail in section B.1. Similarly, the MVC agents rely on the β parameter that controls how greedy we want our evaluation policy to be. This was also tuned for each training configuration in section B.2. In the main results that follow, we therefore compare our novel agents with the best baseline performances that we achieved through tuning.

Another potential baseline could have been standard MCTS, i.e., using random rollouts for node evaluation. However, we chose to exclude it because, in sparse-reward settings like ours, its performance heavily depends on the length of the rollouts, and its computational cost is not directly comparable to our approach or the other baselines, as it does not involve a neural network. Additionally, early qualitative experiments not reported in this thesis showed that even with large planning and rollout budgets, MCTS remained highly suboptimal across most test configurations.

Novel Methods

We test the performance of the following four agents:

- The **MINITREES** agent, which plans and acts according to the MiniTrees algorithm presented in subsection 4.1.2.
- The **MEGATREE** agent, based on the MegaTree algorithm described in subsection 4.1.3.
- The **PDDP** agent, employing the Penalty-Driven Deep Planning algorithm presented in section 4.2.
- The **EDP** agent, which builds extra-deep planning trees using the techniques detailed in section 4.3.

Chapter 7

Results

7.1 Evaluation of the Detection Criteria

We run an independent experiment to evaluate the performance of the detection criteria $\mathcal{C}$ (3.6) and $\mathcal{C}_{\text{max}}$ (3.8). This is useful for validating our hypothesis that $\mathcal{C}_{\text{max}}$ might provide more accurate detection under certain assumptions, as discussed in section 3.3. Moreover, the outcome will indicate us which criterion to employ in the main experiments presented in the next section.

Intuitively, a good detection algorithm would notify the agent of an issue as soon as possible and precisely localize the change. We have shown that the $\mathcal{C}$ works flawlessly in the simple bumping obstacle setting if we are following the deterministic policy π'_θ and $v_\theta = V_{\pi'_\theta}$. Therefore, we can compare what would happen in that ideal situation with the performance of (3.5) and (3.7) in a realistic learning setting. To do so, we employ the 16×16 empty grid environment previously shown in Figure 6.1b and corresponding 10 policies and value functions that we learned for the different seeds. Note that we ensured, for all seeds, that following the prior policy deterministically is an optimal solution in the training environment. This makes it easy to compute the exact expected return of the policy $V_{\pi'_\theta}$ for the ideal setting, i.e., the optimal value function $V^*(s) = \gamma^{d(s,s_m)}$, where $d(s, s_m)$ represents the Manhattan distance between s and the goal state s_m.

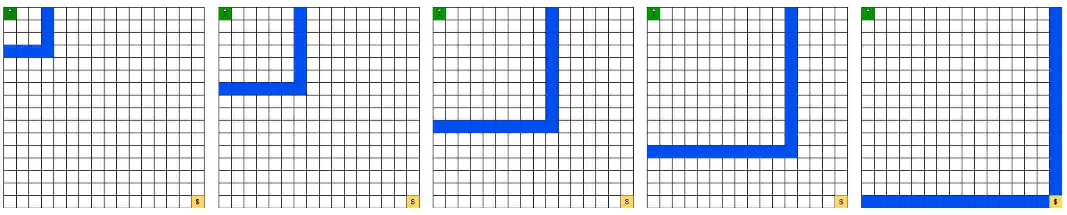


Figure 7.1: Test configurations D3,D6,D9,D12,D15 for detection experiment, where Dn indicates a Manhattan distance of n steps between the initial state (0, 0) and the obstacle.

We then create 5 different test configurations, which are shown in Figure 7.1. Note that there is no way the agent can get past the obstacle in any of the configurations, but this is not a problem, as we only want to test the detection criterion. We also included the training configuration (no obstacles) labeled as ”Empty”, to ensure the detection is not wrongly triggered when no obstacle is present. We test our 10 agents on each of these configurations by rolling out the policy in the following way: before every step in the real environment, we roll out the prior for n steps and check our criterion for $i = 1, 2, ..., n$. Then, we step into the real environment by following such a policy and add n nodes to the rolled-out trajectory. We now check our criterion for $i = n + 1, n + 2, ..., 2n$. We keep repeating this process until we detect a problem. We choose $\gamma = 0.95$ as it is the same value used to train the agents on this environment, and $\epsilon = 0.05$, which we observed to be enough to avoid false positive detections with these value estimates. Moreover, we set $n = 4$ since it provides a good balance between situations where the obstacle is immediately visible to the agent and situations where the agent must take a few steps into the real environment before noticing anything. This allows us to more properly evaluate detection and localization accuracies.

We define the following metrics:

$$\begin{aligned}\text{Accuracy Error} &= |t_{v_\theta} - t_{V^*}|\\ \text{Sensitivity Error} &= \tau_{v_\theta} - \tau_{V^*}\end{aligned}$$

Where:

- t_{V^*} is the future timestep along the rolled out trajectory at which our criterion localizes the obstacle to be, using V^* as the value function. This corresponds exactly to the number of steps it takes our agent to bump into the obstacle, starting from the initial position and following π'_θ.
- t_{v_θ} is the future timestep along the rolled out trajectory at which our criterion localizes the obstacle to be using v_θ, starting from the initial position and following π'_θ.
- τ_{V^*} is the timestep of the real environment at which our criterion using V^* first realizes that there is an obstacle, regardless of where it is, starting from the initial position and following π'_θ.
- τ_{v_θ} is the timestep of the real environment at which our criterion realizes that there is an obstacle using v_θ, regardless of where the obstacle is, starting from the initial position and following π'_θ.

The results reported in Figure 7.2 show how applying $\mathcal{C}_{\text{max}}$ can greatly improve the agent's ability to detect and localize changes accurately, particularly when detecting far away from the current position. The fact that the standard criterion performance degrades significantly when performing long-distance detection is clearly related to the significant underestimation error of the value functions (see Figure A.3). More specifically, the values are severely underestimated near the starting state, while being much more accurate near the goal. Therefore, performing long n-step value estimates will increase the gap with the 0-step estimate used as a comparison in $\mathcal{C}$. Conversely, $\mathcal{C}_{\text{max}}$ compares closer estimates and therefore closer estimation errors, drastically reducing their impact. Given the significant accuracy and sensitivity benefits that $\mathcal{C}_{\max}$ showed in this experiment, we set it as the utilized criterion for the rest of our experiments.

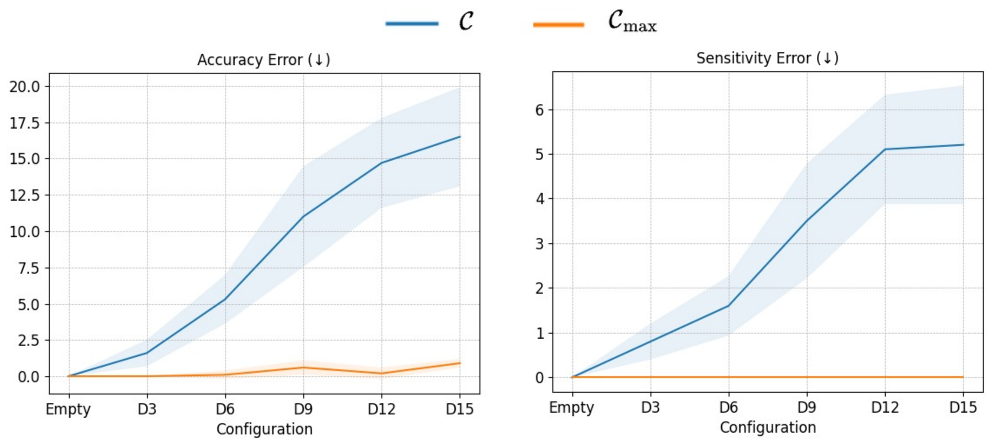


Figure 7.2: Accuracy and sensitivity errors of $\mathcal{C}$ and $\mathcal{C}_{\max}$ detection criteria.

7.2 Main Evaluation of the Planning Algorithms

As anticipated, we present our main quantitative results focusing on the grid world configurations described in subsection 6.2.1.

For each tested configuration and algorithm, we show averaged results over 10×10 seeds, i.e., the 10 trained models and 10 evaluation seeds. Changing the training seed modifies both the utilized prior policy and value, whereas changing the evaluation seed affects certain stochastic operations, such as the random UCT expansion order or the resolution of ties in policy argmax computations. The visualized standard error is computed over the training seeds.

We employ the discounted return, R_γ, as the primary metric. In our case, the reward function only returns a positive reward of 1 when reaching the goal, thus, this can simply be defined as:

$$R_\gamma = \begin{cases} \gamma^t & \text{if goal is reached} \\ 0 & \text{otherwise} \end{cases}$$

where t is the number of steps it took the agent to reach the goal. This retains more information than the undiscounted return, which would only indicate whether the agent has reached the goal within the available maximum number of steps.

7.2.1 AZD Agents Main Results

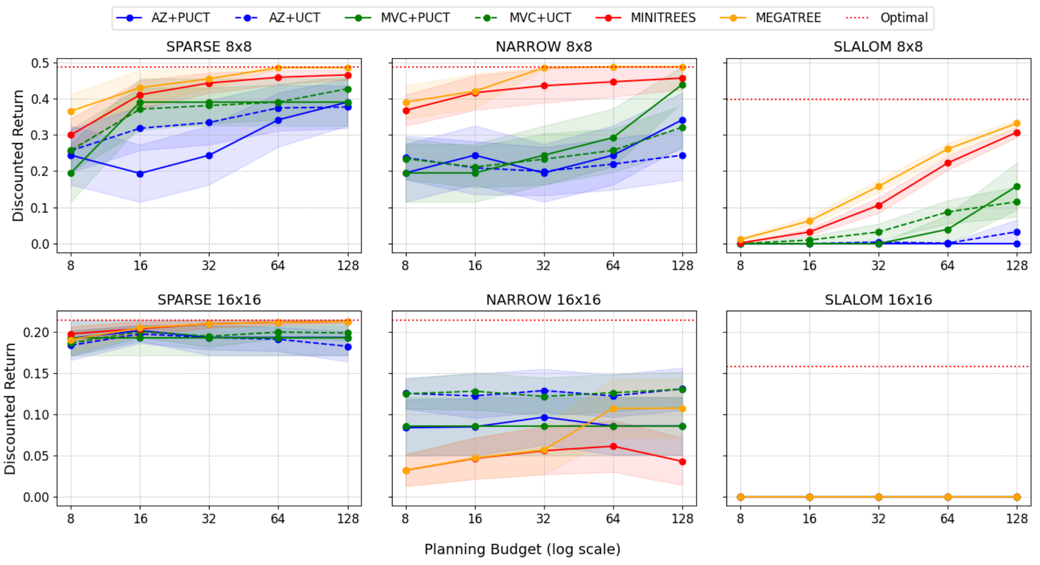


Figure 7.3: MINITREES and MEGATREE results on 8×8 and 16×16 grid world test configurations.

The results of the AZD agents MINITREES and MEGATREE on the 8×8 and 16×16 configurations are reported in Figure 7.3. As a first insight, we can see that both the algorithms outperform the baselines on all the 8×8 test configurations. Note that the agent will almost never (and in the case of SLALOM, exactly never) reach the goal by simply following the prior. Since the only way for both the AZD agents to deviate from it is by finding a solution through the Value Search mechanism, these results demonstrate its effectiveness in cases where such a solution is relatively close to the detected obstacle (since the environment is relatively small). However, the outcome is significantly different when testing on the equivalent 16×16 configurations. While results are as good on the SPARSE 16×16 challenge, performance dramatically decreases on NARROW 16×16 and is always 0 on SLALOM 16×16, indicating that the agent struggles to get past the significantly larger obstacles present in these configurations.

7.2.2 PDDP Agent Main Results

The results of the PDDP agent on the 8×8 and 16×16 configurations are reported in Figure 7.4. Like AZD agents, PDDP also obtains convincing performance on 8×8 configurations, outperforming all the baselines. Unlike AZD agents, PDDP also achieves strong performance on the challenging NARROW 16×16 configuration, significantly outperforming the baselines and reaching the optimal mean discounted return with a sufficiently high planning budget.

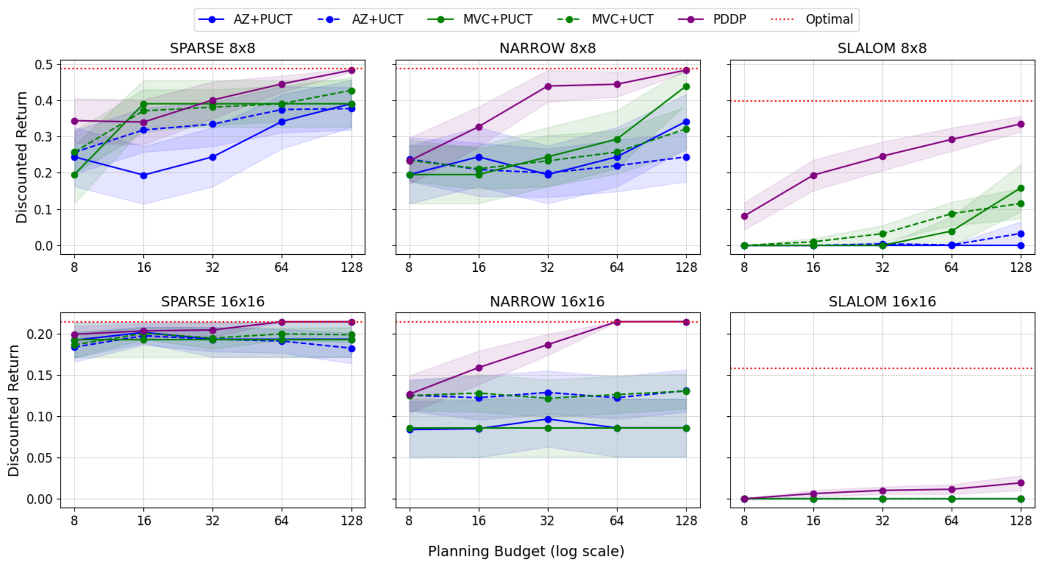


Figure 7.4: PDDP results on 8×8 and 16×16 grid world test configurations.

Recall that compared to the baselines, the PDDP algorithm introduces several novel components, specifically:

- Reusing part of the previous planning tree.
- Detection-based penalization during construction.
- A special, pseudo-deterministic evaluation policy for estimating nodes Q values.

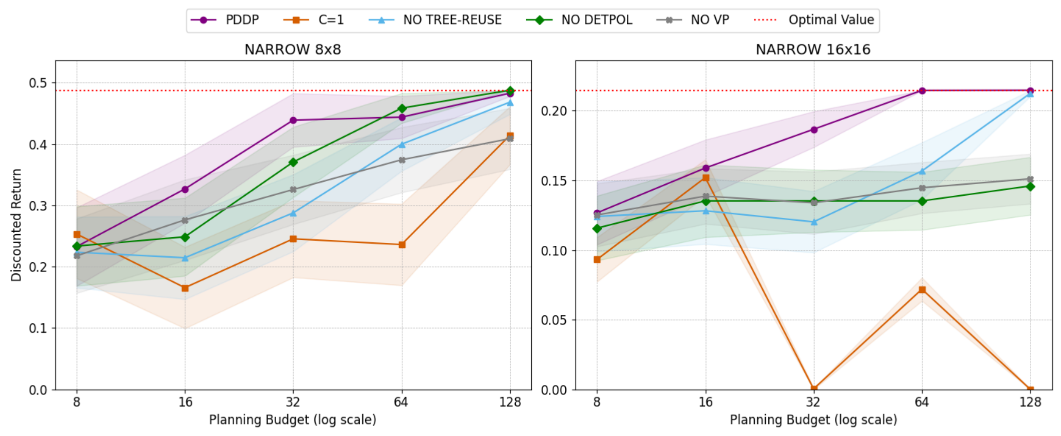
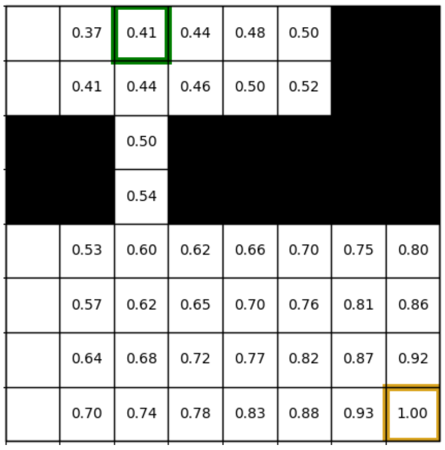


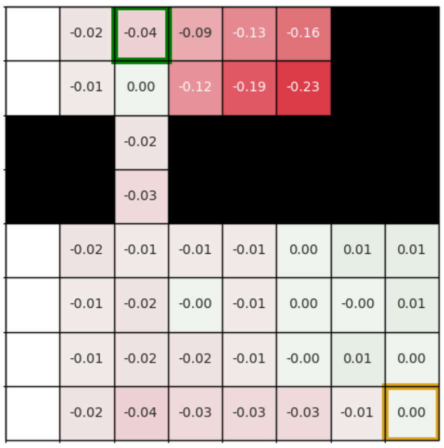
Figure 7.5: Ablation study of PDDP features on NARROW test configurations.

Given the strong results on NARROW 16×16 that only this algorithm is able to achieve, we are interested in determining which of the described components has the most significant influence on performance. To do so, we run an ablation experiment on both the NARROW 8×8 and NARROW 16×16 configurations, so that we can also see whether the importance of any of them is particularly relevant when changing the configuration size. Note that we also include $C = 1$ as a modification, since being greedy during selection is one of the key ingredients needed to build deep trees that can localize problems in the far-off future.

The results of this ablation are shown in Figure 7.5. The plots show how all the main components of the algorithm contribute to the overall performance of the PDDP algorithm in both cases. However, it is interesting to see how performance can degrade much more when removing components in the 16×16 case; as expected, increasing a lot the value of C leads to shallow construction and therefore strongly damages the recycling of the previous tree (fewer nodes to reuse) and the value penalty mechanism (selection is more off-policy w.r.t. the value network). Nonetheless, deep trees alone seem insufficient, as removing the detection and penalization mechanism also results in significantly lower performance.



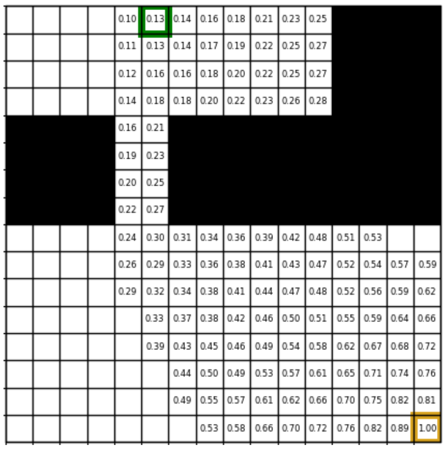
(a) Planning values without penalization.



(b) Value difference when applying penalties.

Figure 7.6: Effect of applying PDDP penalties on NARROW 8×8. The agent is positioned in (0,2) (green border state). Values are averaged across 10×10 training/evaluation seeds.

While the ablation suggests that penalties effectively work as intended, we take one step forward and create visualizations showing how the planning value estimates are affected by using such a mechanism. Figure 7.6a shows the average planning values of each observed state in the NARROW 8×8 configuration, computed across 10×10 training/evaluation seeds after constructing a planning tree from a selected state (green border), without applying value penalties, and with a budget of 128 node expansions. Figure 7.6b shows how these values change when applying the value penalties. The rest of the (hyper)parameters are the same that we used for the main PDDP results on this configuration, but note that, as we are constructing these trees starting from the specified state, we are also not leveraging the tree-reuse mechanism. As we can see from the images, penalizing detected changes results in considerably lower average estimates for the states situated to the right of the agent. This is desirable since it is never a good move to step to the right from the current position, and the value of the states leading through the narrow entrance is, on average, significantly higher. Conversely, planning without value penalties results in not correctly adjusting the values, which remain strongly overestimated for the states to the right of the agent. This will likely lead the agent down a dead-end path. Note that planning towards the narrow entrance also results in some bumps which subsequently decrease the associated values; however, as we can see, this penalization is considerably weaker and gets out-weighted by what the agent is able to see at the other end of the entrance.


(a) Planning values without penalization.

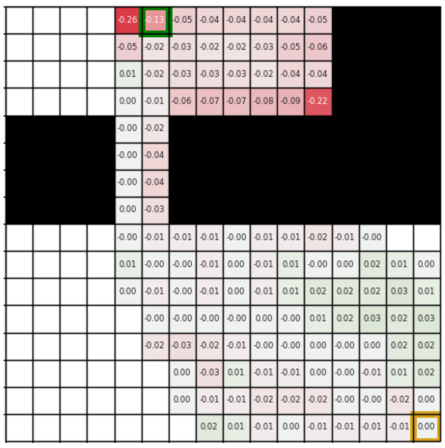

(b) Value difference when applying penalties.

Figure 7.7: Effect of applying PDDP penalties on NARROW 16×16. The agent is positioned in (0,5) (green border state). Values are averaged across 10 × 10 training/evaluation seeds.

The same experiment is repeated on the corresponding 16 × 16 configuration and shown in Figure 7.7. Note that in this case, some states are clearly penalized incorrectly, such as the one to the left of our agent, whose average planning value decreases by 0.26. However, this does not constitute a problem in practice since it is never a good action to go left in this configuration. As in the 8 × 8 case, values of states near the wall(s) of the dead-end path are penalized more, which discourages the agent from moving right and increases the chances that it will choose to go towards the narrow entrance.

Despite the compelling results on these configurations, it could be argued that such a greedy approach might struggle when the learned value function is more complex than the one learned on the empty grid environments. Therefore, we evaluate the performance of PDDP on the MAZE configurations, for which the learned value functions show great variability depending on the training seed, as it can be observed by directly inspecting the visualizations in Figure A.4 and Figure A.5. The results of the PDDP agent on the maze configurations are reported in Figure 7.8. Overall, we can see that all agents struggle more when trained on MAZE_LR and then tested on its variations (first row of plots in the figure) than when trained on MAZE_RL (second row of plots in the figure). This intuitively makes sense since MAZE_RL is a more challenging configuration in terms of the number of steps the agent must take to optimally solve the goal, as well as the different actions it needs to perform.

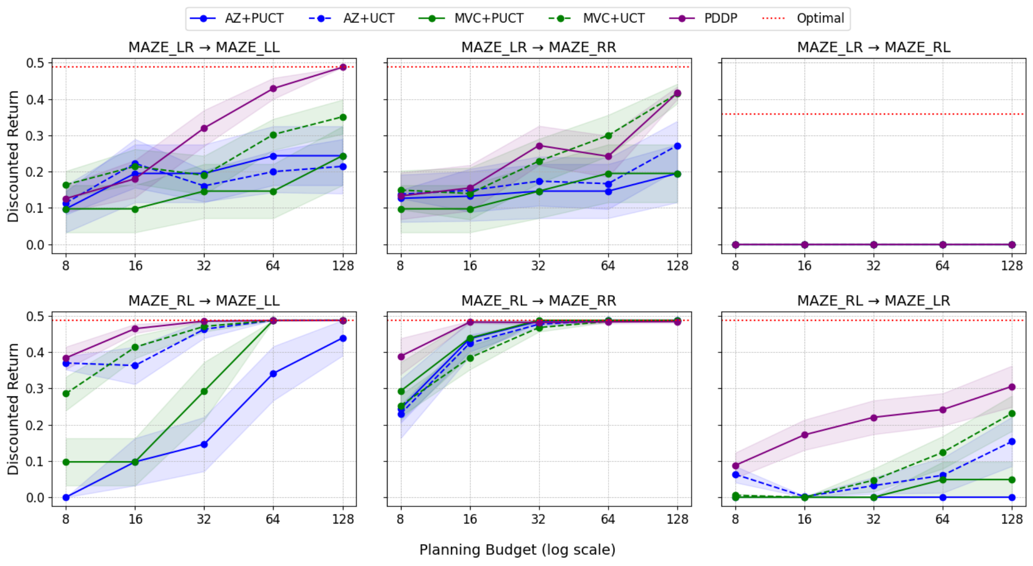


Figure 7.8: PDDP results on MAZE test configurations. The label MAZE_X $\rightarrow$ MAZE_Y on top of each plot indicates training on the MAZE_X configuration and testing on the MAZE_Y one.

Thus, moving the entrances at deployment does, in a way, make it easier to solve, despite damaging the estimators. PDDP's performance is, anyway, convincing, as the algorithm surpasses or matches the best baseline results on all the train/test combinations.

7.2.3 Evaluation with Different Obstacle Deviation

The results obtained by the algorithms making use of the detection mechanism might be correlated with the nature of the environment changes. Specifically, the analytical extraction of the change step, detailed in section 3.2, partially relies on the assumption that bumping into the obstacle causes the agent to remain in the same state. If we instead crash and receive a negative reward, detection should be even easier; however, we are interested in evaluating other non-trivial situations where the agent's reaction to moving into an obstacle differs. Some examples of this were analyzed in subsection 3.2.1, and we would like to validate our hypothesis that different yet reasonable deviation directions after moving towards an obstacle should not particularly damage the application of our $\mathcal{C}_{\max}$ criterion. To do so, we repeat the experiments run so far on the 8×8 and 16×16 configurations (mazes excluded) in the following setups:

- Clockwise (CW): trying to move towards an obstacle results in a deviation in clockwise direction, i.e., according to the following map: $\{(\uparrow:\rightarrow), (\rightarrow:\downarrow), (\downarrow:\leftarrow), (\leftarrow:\uparrow)\}$.
- Counter-Clockwise (CCW): trying to move towards an obstacle results in a deviation in counter-clockwise direction, i.e., according to the following map: $\{(\uparrow:\leftarrow), (\rightarrow:\uparrow), (\downarrow:\rightarrow), (\leftarrow:\downarrow)\}$.

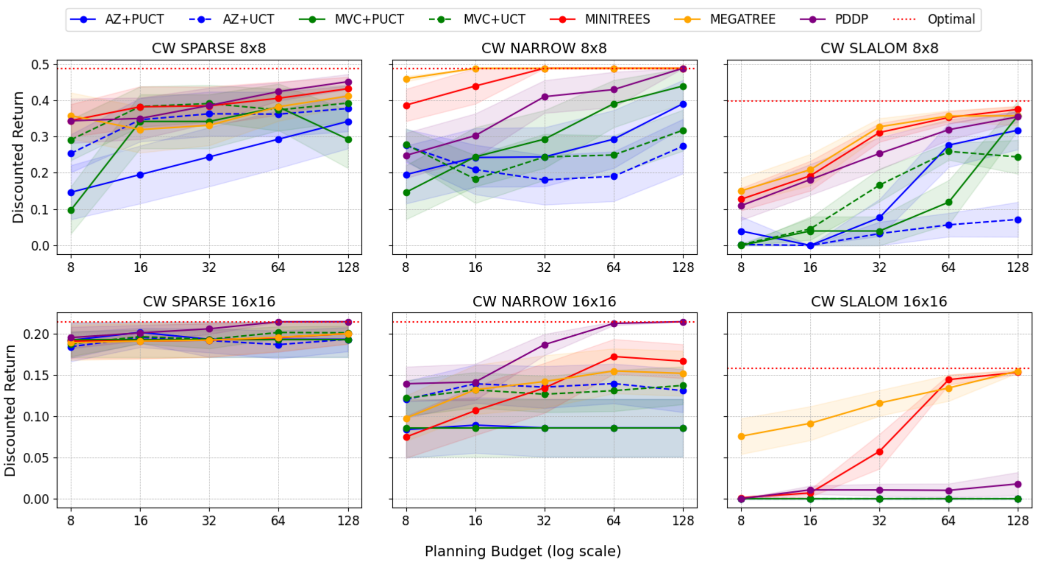


Figure 7.9: Results of detection-based algorithms with clockwise (CW) obstacle deviations on 8×8 and 16×16 grid world test configurations, compared with baselines.

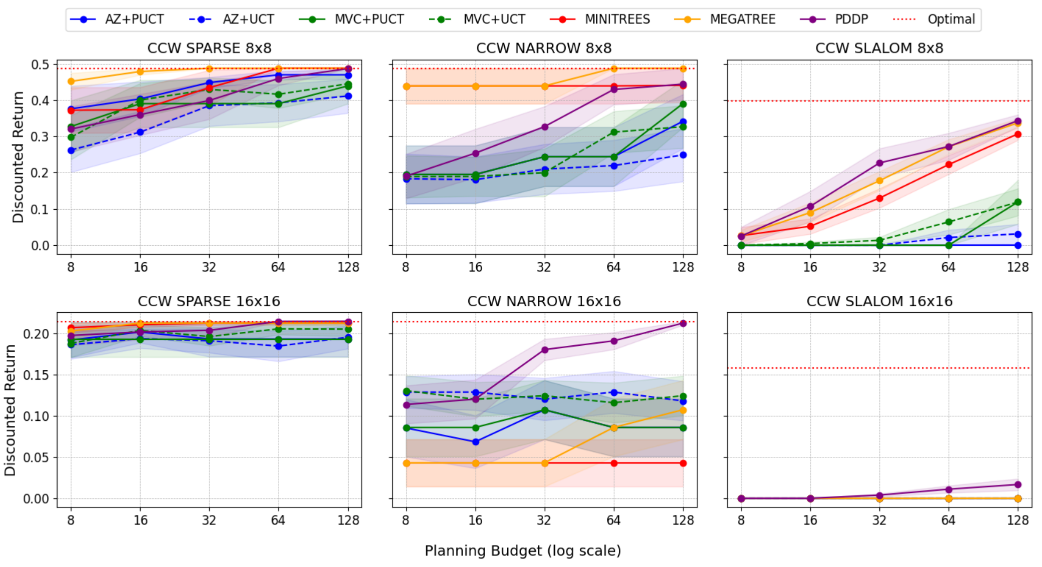


Figure 7.10: Results of detection-based algorithms with counter-clockwise (CCW) obstacle deviations on 8×8 and 16×16 grid world test configurations, compared with baselines.

In specific cases where the agent cannot move in the deviation direction (e.g., due to another obstacle), it remains in the same position as before. This can happen, for instance, if the agent gets stuck in a corner. For all the algorithms, we keep the hyperparameters fixed at the previously utilized values for the main experiments.

Figure 7.9 shows the results of the baselines, the AZD agents (MINITREES, MEGATREE), and PDDP on 8×8 and 16×16 test configurations with CW obstacle deviations. Overall, the relative comparisons between the agents stay close to the main experiments. Note that AZD algorithms perform slightly worse than before on SPARSE 8×8 but achieve considerably better results on NARROW 16×16 and SLALOM 16×16 configurations. This may be motivated by the fact that, in both configurations, a clockwise deviation causes downward movement into the dead end to shift the agent leftward, which can help escaping it. This also explains why the baselines performed better on the corresponding NARROW 8×8 and SLALOM 8×8 configurations compared to their performance in the main results.

Figure 7.10 shows the corresponding plots with counter-clockwise obstacle deviations. In this case, the performance improvement observed for AZD algorithms on 16×16 configurations is lost, as the new dynamics no longer help in overcoming the obstacles in NARROW and SLALOM. Nonetheless, the overall relative performance differences between the algorithms remain similar to those in the main results, confirming that our detection-based algorithms can handle more diverse dynamic changes than the original one we experimented with.

7.2.4 EDP Agent Main Results

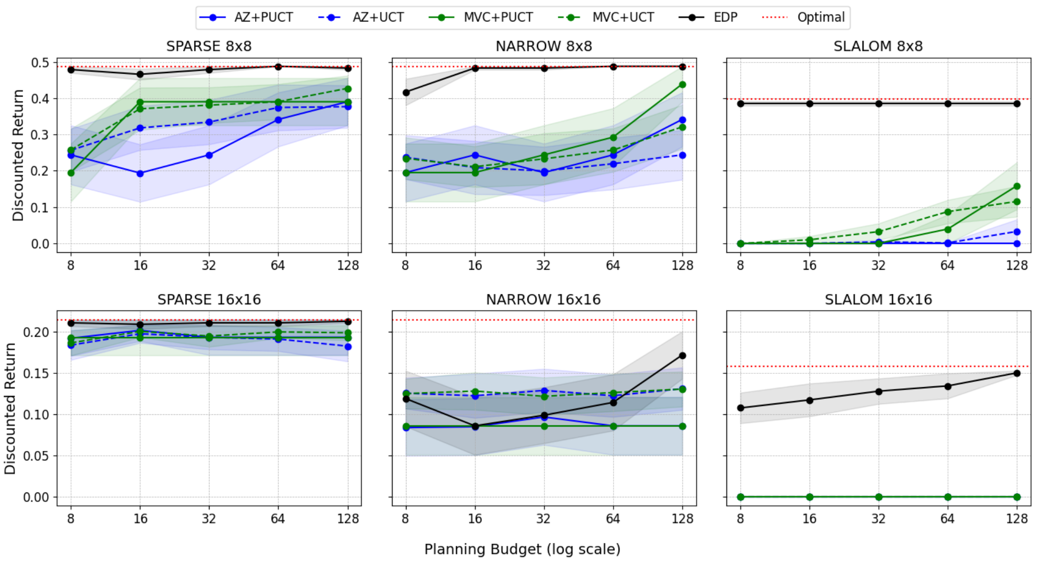


Figure 7.11: EDP results on 8×8 and 16×16 grid world test configurations.

The results of the EDP agent on the 8×8 and 16×16 configurations are reported in Figure 7.11. Overall, EDP achieves excellent results, except for failing to outperform the baselines on the NARROW 16×16 configuration, where performance only begins to rise with the highest employed planning budget. However, what is perhaps most surprising are the almost optimal results for SLALOM configurations, even with very limited planning budgets, for both 8×8 and 16×16 configuration sizes.

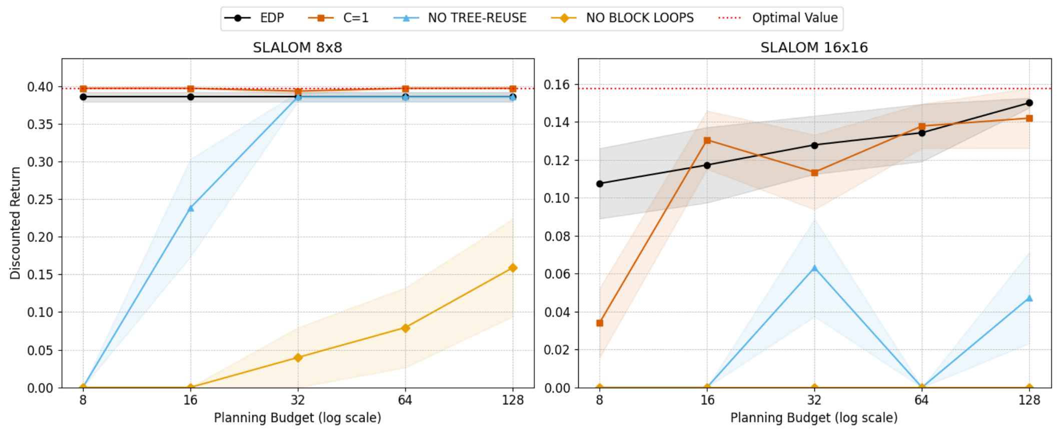


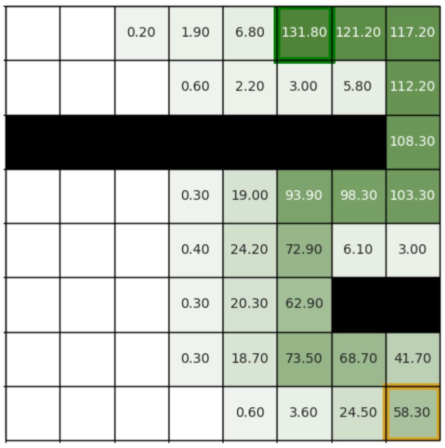
Figure 7.12: Ablation study of EDP features on SLALOM test configurations.

Similarly to what we did for PDDP, we then conduct a simple ablation experiment to understand which component of the EDP algorithm contributes the most to these results. We consider three main ingredients as the core of EDP:

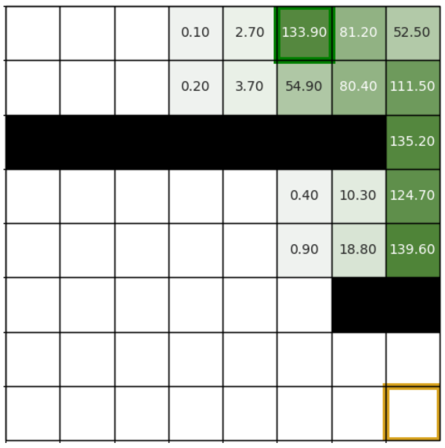
- Greedy selection ($C \approx 0$).
- Reusing part of the previous planning tree.
- Blocking loops.

The results of this ablation are plotted in Figure 7.12. Each of the lines represents the EDP algorithm without one of the features mentioned above, except for the black line, which represents the complete algorithm. The results of this experiment clearly show that blocking the loops is by far the most impactful feature of EDP. In fact, removing it results in an almost complete drop in performance even in the 8×8 case. Not recycling the previous step's planning trees also leads to a performance decrease, particularly in the 16×16 case, which makes sense as considerably more planning is needed in that case to clear the obstacle. What is perhaps surprising is that increasing the C exploration constant does not significantly damage the results, which even become slightly better in the 8×8 case.

To better understand what makes blocking the loops so effective on these configurations, we propose similar visualizations to the ones shown for PDDP, but this time we plot the mean visitation count of each state across 10×10 training/evaluation seeds after building a planning tree from a selected state, with and without blocking the loops.



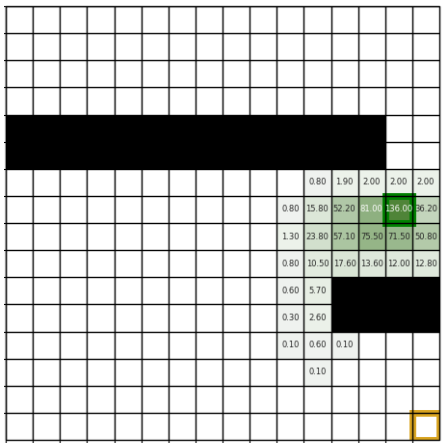
(a) Block loops on.



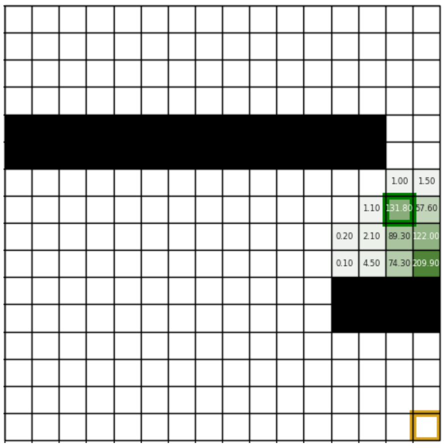
(b) Block loops off.

Figure 7.13: Comparison of EDP planning with and without blocking loops on SLALOM 8 × 8. The agent is positioned in (0,5) (green border state).

For the SLALOM 8 × 8 configuration, this is reported in Figure 7.13. We can see how the agent blocking the loops can directly observe the goal state during planning, and the dark green color persisting along such a path confirms that most of the planning budget is successfully directed there. Conversely, the agent that does not block the loops continues to select roughly the same path leading to the corner state (4,7). Acting from there will produce a lot of loops, which is why performance changes so much if we block them.



(a) Block loops on.



(b) Block loops off.

Figure 7.14: Comparison of EDP planning with and without blocking loops on SLALOM 16 × 16. The agent is positioned in (7,14) (green border state).

An equivalent visualization for the SLALOM 16×16 configuration is reported in Figure 7.14. In this case, the agent blocking the loops is unable to directly observe the goal state during planning, which is not surprising, given that the utilized planning budget remains the same as before, while the distance between the agent and the goal is significantly larger. Nonetheless, it still manages to observe many more states around the obstacle without focusing on the corner. Note that, as shown by the ablation, this alone is not sufficient to consistently solve the configuration, but it is when combined with tree recycling, which further increases the size of the constructed trees over time.

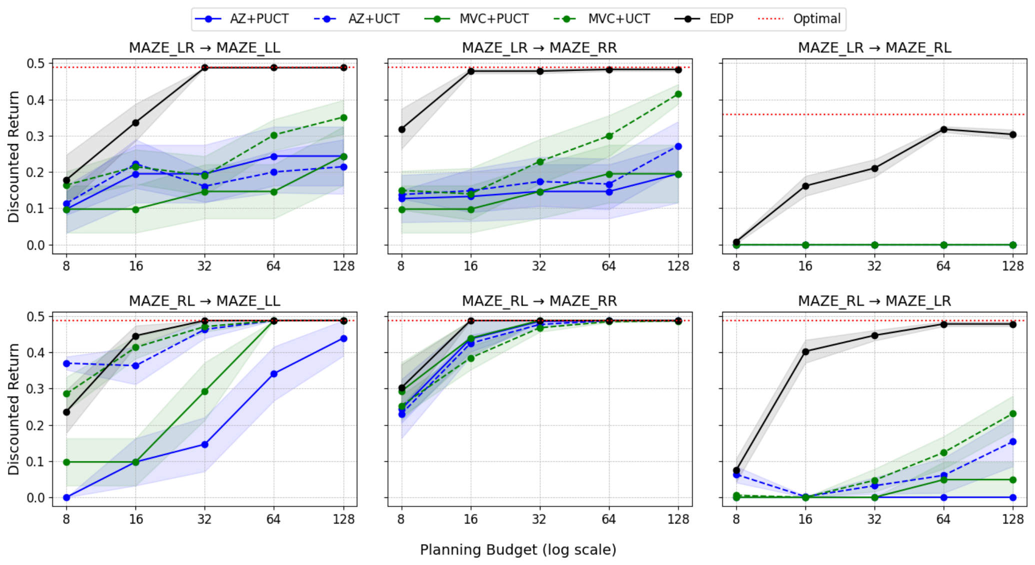


Figure 7.15: EDP results on MAZE test configurations. The label MAZE_X $\rightarrow$ MAZE_Y on top of each plot indicates training on the MAZE_X configuration and testing on the MAZE_Y one.

Finally, we evaluate EDP on the maze configurations, with results reported in Figure 7.15. The performances achieved by the EDP agent are beyond our expectations, as it not only largely outperforms the baselines when changing the position of one entrance, but also obtains by far the best results so far on the challenging MAZE_LR $\rightarrow$ MAZE_RL and MAZE_RL $\rightarrow$ MAZE_LR combinations. Note that this is the only agent tested so far to achieve non-zero performance on MAZE_LR $\rightarrow$ MAZE_RL, and instead almost reaching optimality with high planning budgets. We believe that such a significant performance improvement over the baselines highlights the generally inefficient planning standard AZ carries out, making its performance tightly constrained by how accurate the learned priors are. We will discuss how much these results can be expected to generalize to other environments in chapter 8.

7.2.5 Summary Plots

To facilitate the comparison of all our novel algorithms, we hereby report a few summary plots of the main results. We avoid reiterating the baseline results since we have already thoroughly demonstrated their suboptimal performance in comparison to our algorithms.

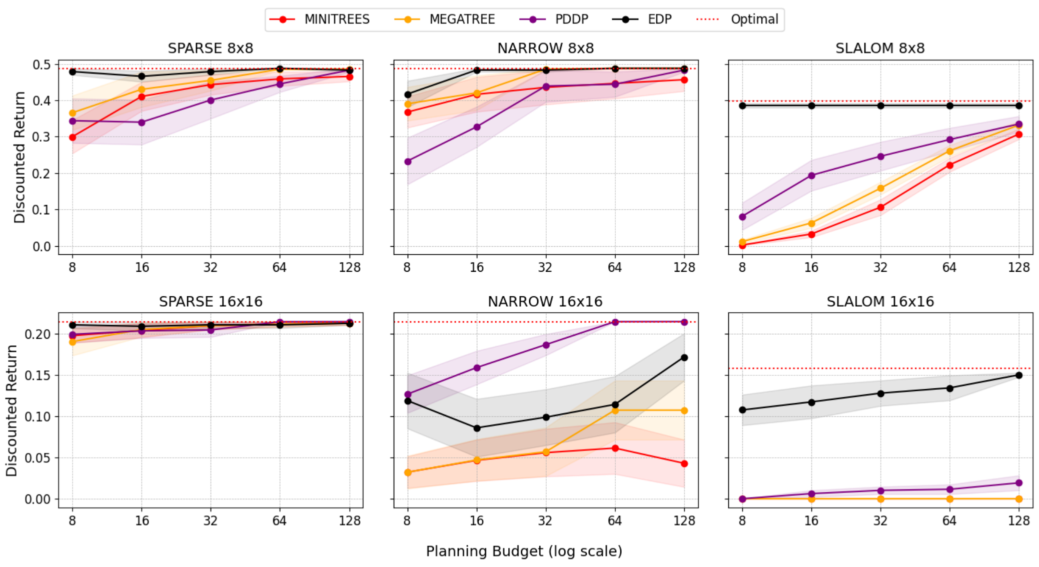


Figure 7.16: Performance of all novel algorithms on 8×8 and 16×16 configurations.

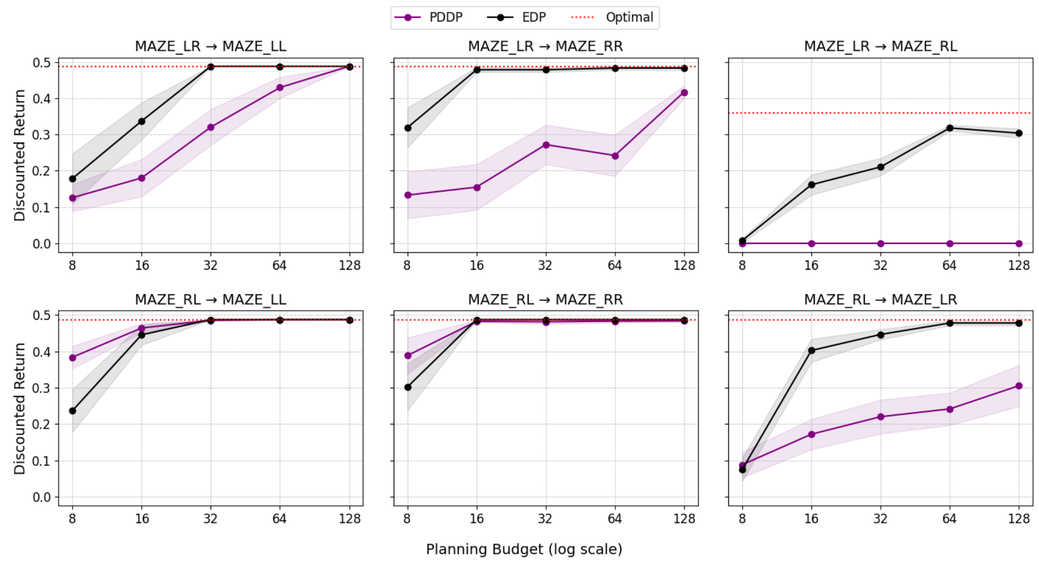


Figure 7.17: PDDP and EDP performances compared on MAZE configurations. The label MAZE_X $\rightarrow$ MAZE_Y on top of each plot indicates training on the MAZE_X configuration and testing on the MAZE_Y one.

Figure 7.16 shows the performance of all the novel algorithms on the main 8×8 and 16×16 grid world experiments. Overall, all the algorithms demonstrate strong performance on 8×8 configurations, whereas PDDP and EDP are the only ones that reach optimality on the challenging NARROW 16×16 and SLALOM 16×16 configurations, respectively.

Figure 7.17 compares PDDP and EDP performance on the MAZE configurations. While both algorithms outperform the baselines as previously shown, these plots make it easier to see how EDP achieves significantly higher returns in most configurations. We can therefore conclude that EDP is our overall best-performing algorithm on the grid world experiments.

7.3 CarGoal Evaluation

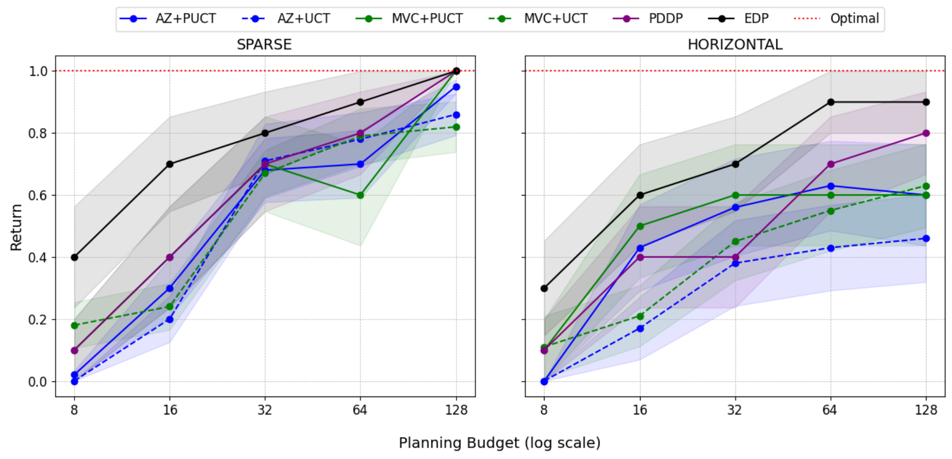


Figure 7.18: Return on CarGoal test configurations.

To conclude this chapter, we show a relatively brief evaluation of our best-performing algorithms, PDDP and EDP, conducted on the CarGoal test configurations shown in Figure 6.7. We do this to give a glimpse of the potential performance of our algorithms in more complex environments. Note that in this case, we consider the undiscounted return as the main metric, since the discounted return is no longer an intuitive performance indicator, and it is difficult to compute the corresponding optimal value for this environment. Moreover, two adjustments to our algorithms were found necessary:

- For PDDP, we do not apply the pseudo-deterministic policy modification described in subsection 4.2.4. We make this change based on early experimentation, which revealed that this extremely greedy approach was particularly detrimental in the CarGoal environment, where the action space is relatively larger. As such, we might not want to risk over-committing to a single greedy path during planning.
- For EDP, we apply the already mentioned generalized definition of loops necessary in continuous state spaces, i.e., we consider s' a repetition of s if $\|s - s'\|_2 \leq \eta$, where η is a constant loop threshold.

Figure 7.18 shows the results of this evaluation. As the main highlight, we can see how EDP significantly outperforms all the baselines on both configurations. This is perhaps surprising considering that the definition of a loop in continuous state spaces is less trivial, and wrongly tuning the threshold η could block potentially non-redundant planning paths. Nonetheless, our results suggest that this can still be a powerful feature if we manage to find the correct balance.

On the other hand, PDDP shows a more modest improvement over the baselines, particularly in the SPARSE experiment, where all algorithms besides EDP perform similarly. In the HORIZONTAL experiment, PDDP reaches a higher return than all the baselines for large planning budgets, but still significantly less than EDP. The most plausible explanation for this lies in the fact that, as anticipated, bumping into an obstacle causes the car's velocity to drop to zero. This makes standard AZ planning already see this as a detrimental move, which makes applying PDDP penalties less effective than what we experienced in the grid world experiments, where standard planning could not update the values quickly enough to reflect the changes. Conversely, EDP generally makes planning more efficient by loop blocking.

We do not exclude that better tuning of the several PDDP hyperparameters could bring its performance closer to that of EDP, which only relies on the loop threshold η and is therefore easier to tune. This experiment, however, confirms that this is an inherent limitation of PDDP, which would likely benefit from a method to dynamically estimate the value penalty to be applied to each penalized node.

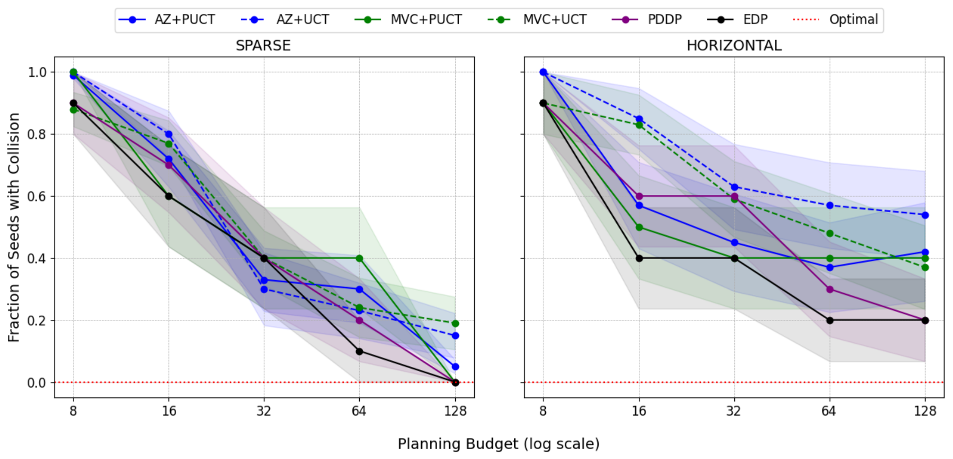


Figure 7.19: Fraction of seeds reporting at least one collision on CarGoal test configurations.

Finally, it might be argued that the improvements in return shown in Figure 7.18 would be more meaningful if they also corresponded to a reduction in collisions. After all, in a real-world setting, a driver who reaches the goal while repeatedly bumping into obstacles along the way would not be considered very professional. To address this, we report the fraction of seeds that experienced at least one collision in the previous experiment, as shown in Figure 7.19. This plot reveals that the collision metric is inversely correlated with the return: when the car collides, it rarely manages to get unstuck and reach the goal. In the SPARSE case, EDP's advantage in avoiding collisions is somewhat smaller than its advantage in return, indicating that there are instances where the algorithm succeeds in reaching the goal despite some collisions. In the HORIZONTAL case, however, EDP outperforms the baselines in both return and collision avoidance, showing a more consistent ability to avoid obstacles and follow a clear path to the goal.

Chapter 8

Discussion

We now wish to further discuss the outcomes of our analysis and experiments, as well as highlight the potential limitations of our framework.

This work started with the simple idea that a model of the test environment would have enabled us to test whether a learned policy was still reliable at deployment, despite relevant changes to the environment. Following this idea, we devised a detection method enabling the agent to localize changes far into the future and conducted an in-depth analysis in chapter 3. The related experiment presented in section 7.1 shows how the standard criterion $\mathcal{C}$ presents important sensitivity and precision issues, even when applied to the relatively simple and local type of change that an obstacle constitutes. This is likely due to the underestimation of the learned value functions (Figure A.3) which is roughly proportional to the distance of the state from the goal; as such, the values that we are bootstrapping when computing n-step estimates get increasingly high as we roll out further, and it then takes a lot of time (and bumps) for the agent to decrease such estimate and realize it encountered an obstacle. The error that we experienced is indeed also roughly proportional to the distance of the agent from the obstacle. In contrast, $\mathcal{C}_{\max}$ shows a close-to-zero error because taking the max value estimate along the path as the comparison value combats the underestimation with an overestimation.

It could be argued that the advantage of using this method in this situation may be offset by its disadvantage in the opposite situation, i.e., cases where values close to the agent's position are overestimated. However, it is easy to see how in that case, $\mathcal{C}_{\max}$ would converge to $\mathcal{C}$ as the maximum value estimate along the path would be the one of the root. We therefore ultimately see $\mathcal{C}_{\max}$ as a generally more robust criterion.

One important point that still needs to be discussed is what would happen if the state representation were different. In theory, our detection methods do not utilize explicit information regarding the type of observation; instead, they solely rely on the value and policy estimates that the neural network outputs. In practice, the ability to observe different features of the state may affect detection; for instance, if we used images instead of coordinates to represent the agent's position, new obstacles would be reflected in the observation, which would be out of distribution and subsequently impact the NN outputs.

We can then imagine two possible situations:

- The agent was trained on some obstacle configurations and is now challenged with a new one. The neural network might generalize to the new observations. If so, we will likely follow a generally correct direction, and our criterion should still be triggered if the agent ends up bumping into an obstacle anyway, correctly indicating a change.
- The agent was trained with no obstacles. The new "concept" of an obstacle reflected in all the test-time image observations could result in globally wrong policy and value predictions. Nonetheless, if the two estimators are learned together, as in our case, we can expect them to be similarly affected by the changes; i.e., our assumption that $v_\theta \approx V_{\pi'_\theta}$ should still hold, despite v_θ and π_θ being now both strongly suboptimal. If this is not the case and the two estimators are somehow differently affected, then the detection might be triggered even without encountering a change.

We can therefore imagine that in most cases, the detection should work regardless of whether the changes are reflected in the NN inputs, though its practical impact may be reduced. An example of this is what we experienced when testing PDDP on the CarGoal test configurations (section 7.3), where bumping into an obstacle causes the car's velocity to drop to zero. We believe that the reduced impact of PDDP penalties over baseline performances in those experiments is due to the neural network already identifying post-collision states as significantly worse, unlike in the main grid world experiments, where the observation after bumping stayed exactly the same, and the baselines could not easily account for it.

The impact of incorporating the detection mechanism during planning has been extensively validated in section 7.2. We now wish to discuss under which assumptions we can expect such successful outcomes to generalize, especially when moving to more complex environments. The performance of AZD methods is inherently constrained by the quality of the learned prior policy, as we follow it blindly until a problem is detected. Usually, the main advantage of retaining a model of the environment is that we can adjust an imperfect policy; AZD methods address this when environment discrepancies cause such imperfections. If, however, the neural network was underfitted in the first place, then these methods are bound to perform suboptimally even if the environment stays the same. On the other hand, PDDP integrates the detection into a more general planning algorithm and should therefore better cope with situations where the estimators were originally suboptimal. Moreover, the parameter controlling how much we want to penalize the detected changes gives us more flexibility; if we know what kind of changes might affect the environment (e.g., obstacles, lava blocks, slippery floor), then we can adjust such parameter to make the agent more or less pessimistic w.r.t. them. This introduces a bias in our planning algorithm, which might be detrimental if the changes end up being different than what we expected. Recall that PDDP also relies on another proprietary hyperparameter, which controls how strict we want to be when checking whether the actions chosen by the selection policy are on-policy w.r.t. the learned prior. The presence of several parameters that have a concrete impact on performance constitutes the main limitation of the algorithm.

Finally, the EDP algorithm proposes a more general approach that does not rely on detection principles and instead aims to plan in the most efficient way possible with the available planning budget. While the individual components of the algorithm are not new to the MCTS literature (as discussed in section 5.2), we showed that their combination can significantly improve the agent's performance. It is important to point out that our grid world configurations naturally bring to a lot of exact loops during planning, being the state space discrete and relatively small; nonetheless, the performed CarGoal experiments demonstrated how the generalized definition of a loop, which considers a similarity metric between states, can still improve performance in large or continuous state spaces if the threshold parameter η is properly tuned.

Note that there might be cases where repeatedly entering a certain sequence of states leads to a net positive/negative reward, in which case blocking the loop and assigning it a value of zero might not be desirable. Moerland et al. [22] suggest that few real-world tasks require an agent to repeatedly travel a loop, and in practice, such situations are typically domain artifacts. While this makes sense when thinking of most deterministic, single-player games like the environments both our and their algorithms were tested on, we could likely craft some examples where blocking loops could be a suboptimal choice even under these assumptions, such as navigation tasks requiring return trips or repeated laps, or resource-gathering settings where revisiting the same locations is necessary for collecting more resources. Whether these situations result in actual loops depends on the state representation that the agent can observe.

Most importantly, blocking loops can be detrimental in stochastic environments, even if simple. For example, in the popular FrozenLake grid world environment, the agent has a $1/3$ probability of moving in the intended direction when acting and a $2/3$ probability of slipping in a perpendicular direction because of the frozen floor. In such a situation, we might enter a loop by bumping into the wall even when taking an optimal action. Therefore, blocking it and assigning it a value of zero would strongly damage the planning process. This can happen from the very beginning of the episode since the agent starts in the $(0, 0)$ corner and can bump into the wall if slipping left when trying to go down, or slipping up when trying to go right.

These considerations highlight that while loop blocking can prevent inefficient planning in common scenarios, its indiscriminate application risks hiding valuable, even optimal, behavior. It should therefore be applied with careful consideration of the environment's dynamics and task-specific requirements.

Chapter 9

Conclusions and Future Work

In this chapter, we draw our conclusions and propose some future directions for expanding our work. To do so, we now come back to our original research questions:

1. *How can we detect changes to the environment at deployment when planning with learned estimators?*

   The analytical criteria developed in chapter 3 allow the agent to quickly detect that some detrimental change has happened along the rolled out trajectory without the use of external knowledge or a model of the original environment. In some cases, it is also possible to determine the exact location where the deviation began along the trajectory; this depends on the nature of the changes, as well as the reward and transition functions governing the environment. We analyzed some of these cases and provided insights on the related errors and needed approximations. Finally, we discussed the applicability of our results and considerations in chapter 8.

2. *How can we leverage such detection principles to re-plan around these changes and subsequently overcome them?*

   We proposed two classes of algorithms that exploit the detection criteria in fundamentally different ways. AZD methods (section 4.1) look far into the future by rolling out a prior policy that can detect long-distance deviations. Then, the trajectory can be expanded into a planning tree, which lets us search for alternative paths through the novel Value Search mechanism. Within this category, we showcased two algorithms, MiniTrees and MegaTree, which mostly differ in the way the trajectory is expanded. We went on to describe the PDDP algorithm (section 4.2), which directly integrates the detection check along the trajectories followed by the selection policy while building the planning tree, and penalizes each deviation by manually reducing the value of nodes marked as potentially problematic. We tested both approaches with multiple experiments and related ablations in section 7.2 and section 7.3, and discussed their limitations in chapter 8.

3. *What standard features of AlphaZero planning might be modified or improved to better plan and act in a potentially changed environment?*

   Our baseline experiments show how the shallow planning typical of AZ can be too slow in updating the value estimates necessary to realize that something has changed in the environment. Based on this observation, we developed the EDP algorithm, which modifies several features of AZ to make more intelligent use of the available planning budget, even without modifying the core of the original algorithm. This resulted in largely improved results, showing how standard AZ's inability to account for wrong prior estimates largely depends on an overall inefficient use of the planning budget. We found out that EDP particularly benefits from blocking state loops during planning, allowing the agent to explore many more options and avoid getting stuck in corners or other types of dead ends, and from partially reusing the previous planning tree instead of completely discarding it after each step. We also discussed which kind of assumptions and subsequent biases are introduced by these planning modifications in chapter 8.

Overall, our methods are demonstrated to largely improve performance under certain assumptions. Our future work section is therefore focused on what steps should be followed to relax such assumptions.

9.1 Future Work

Our work opens the way to several future improvements and extensions of the proposed framework. We hereby suggest some of the most promising.

9.1.1 Extension to Stochastic and Partially Observable Environments

Our analysis and proposed solutions assume that the environment (and associated model) is deterministic and that states are fully observable. This means the agent is certain of the current state it is occupying. This assumption is inherited from the standard AlphaZero framework, and relaxing it can impact our algorithms in different ways. All the proposed algorithms keep part of the previous planning tree instead of completely discarding it. They do so by ”stepping” into the node that corresponds to the current state of the environment and only retaining its subtree, as the rest of the tree is not useful anymore if we are certain that we are planning from the correct current state. If this is not the case, one possible enhancement would be storing some of these planning trees, which could come in handy if we realized some steps later that the environment state is different from what we believed. It would also be possible to directly modify the way the planning trees are constructed to take stochasticity into account, for instance, by following the approach of Stochastic MuZero [2] but with a given simulator, instead of learning a model as done in MuZero methods.

9.1.2 Dealing with Imperfect Models

While our algorithms are specifically tailored to deal with estimators that are inaccurate due to environment changes, we are, on the other hand, assuming an accurate transition and reward model of the test environment. This is a strong assumption that can limit the applicability of our methods, as well as the employed baselines, to real-world tasks where models are usually incomplete and/or inaccurate. Unfortunately, planning with an imperfect model is a hard challenge, and many recent approaches rely on fundamentally different frameworks that are not necessarily MCTS-based, as some of the ones described in section 5.3. If we have some knowledge of the degree of error of our transition model, then it might be possible to integrate MCTS-based techniques that deal with model uncertainty, such as robust MCTS [26] (rMCTS) or Epistemic-MCTS [23] (E-MCTS), also mentioned in chapter 5. By combining a method that addresses model uncertainty with our algorithms, we may be able to tackle more realistic challenges.

9.1.3 Addressing Non-Stationary Environments

It would be interesting to test how our algorithms perform in non-stationary environments. For instance, we could have similar configurations to the ones we experimented with, where the obstacles dynamically move during the execution of an episode. This could represent realistic situations, such as a car driving through traffic and therefore having to deal with various moving objects that must be detected early on. While the detection itself should not be impacted, we could not fully reuse the gathered information during the next step because the changes would be different. Therefore, the methods should be adapted to either only exploit detection information gathered at the current step or to account for errors arising from non-stationarity. Mechanisms such as recycling the previous planning tree would also need modifications in order to be applied to these scenarios.

Bibliography

[1] Pieter Abbeel, Morgan Quigley, and Andrew Y. Ng. Using inaccurate models in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 1–8, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933832. doi: 10.1145/1143844.1143845. URL `https://doi.org/10.1145/1143844.1143845`.

[2] Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K Hubert, and David Silver. Planning in stochastic environments with a learned model. In *International Conference on Learning Representations*, 2021.

[3] K. J. Åström. Optimal control of markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205, 1965.

[4] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[5] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. `https://arxiv.org/abs/1606.01540`, 2016. arXiv preprint arXiv:1606.01540.

[7] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 2012. doi: 10.1109/TCIAIG.2012.2186810.

[8] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. volume 4630, 05 2006. ISBN 978-3-540-75537-1. doi: 10.1007/978-3-540-75538-8_7.

[9] Thomas Gabor, Jan Peter, Thomy Phan, Christian Meyer, and Claudia Linnhoff-Popien. Subgoal-based temporal abstraction in monte-carlo tree search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5562–5568. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/772. URL `https://doi.org/10.24963/ijcai.2019/772`.

[10] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 315–323, 2011. URL `http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf`.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. URL `https://arxiv.org/abs/1502.03167`.

[12] Albin Jaldevik. General tree evaluation for alphazero. Master's thesis, TU Delft, 2024.

[13] Beomjoon Kim, Kyungjae Lee, Sungbin Lim, Leslie Kaelbling, and Tomas Lozano-Perez. Monte carlo tree search in continuous spaces using voronoi optimistic optimization with regret bounds. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):9916–9924, Apr. 2020. doi: 10.1609/aaai.v34i06.6546. URL `https://ojs.aaai.org/index.php/AAAI/article/view/6546`.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL `https://arxiv.org/abs/1412.6980`.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. volume 2006, pages 282–293, 09 2006. ISBN 978-3-540-45375-8. doi: 10.1007/11871842_29.

[16] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL `https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf`.

[17] Li-Cheng Lan, Huan Zhang, Ti-Rong Wu, Meng-Yu Tsai, I-Chen Wu, and Cho-Jui Hsieh. Are alphazero-like agents robust to adversarial perturbations?, 2022. URL `https://arxiv.org/abs/2211.03769`.

[18] John Lathrop, Benjamin Rivi'ere, Jedidiah Alindogan, and Soon-Jo Chung. Model predictive trees: Sample-efficient receding horizon planning with reusable tree search, 2024. URL `https://arxiv.org/abs/2411.15651`.

[19] Jongmin Lee, Wonseok Jeon, Geon-Hyeong Kim, and Kee-Eung Kim. Monte-carlo tree search in continuous action spaces with value gradients. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34:4561–4568, 04 2020. doi: 10.1609/aaai.v34i04.5885.

[20] John Tan Chong Min and Mehul Motani. Brick tic-tac-toe: Exploring the generalizability of alphazero to novel test environments, 2022. URL `https://arxiv.org/abs/2207.05991`.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL `https://arxiv.org/abs/1312.5602`.

[22] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Monte carlo tree search for asymmetric trees, 2018. URL `https://arxiv.org/abs/1805.09218`.

[23] Yaniv Oren, Villiam Vadocz, Matthijs T. J. Spaan, and Wendelin Böhmer. Epistemic monte carlo tree search, 2025. URL `https://arxiv.org/abs/2210.13455`.

[24] Ava Pettet, Yunuo Zhang, Baiting Luo, Kyle Wray, Hendrik Baier, Aron Laszka, Abhishek Dubey, and Ayan Mukhopadhyay. Decision making in non-stationary environments with policy-augmented search, 2024. URL `https://arxiv.org/abs/2401.03197`.

[25] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie Bruin. Exploiting graph properties of game trees. 09 1998.

[26] Maxim Rostov and Michael Kaisers. Robust online planning with imperfect models. 2021.

[27] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0.

[28] 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, December 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-03051-4. URL `http://dx.doi.org/10.1038/s41586-020-03051-4`.

[29] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, L. Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016. URL `https://api.semanticscholar.org/CorpusID:515925`.

[30] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel,

Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL `https://arxiv.org/abs/1712.01815`.

[31] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature 550*, 2017. URL `https://doi.org/10.1038/nature24270`.

[32] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023. doi: 10.1007/s10462-022-10228-y. URL `https://doi.org/10.1007/s10462-022-10228-y`.

[33] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goul ao, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium. `https://zenodo.org/record/8127025`, March 2023. Zenodo.

[34] Anirudh Vemula, Yash Oza, J. Andrew Bagnell, and Maxim Likhachev. Planning and execution using inaccurate models with provable guarantees, 2020. URL `https://arxiv.org/abs/2003.04394`.

[35] Anirudh Vemula, J. Andrew Bagnell, and Maxim Likhachev. Cmax++ : Leveraging experience in planning and execution using inaccurate models. In *Proceedings of 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 6147 – 6155, February 2021.

[36] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3): 279–292, 1992. doi: 10.1007/BF00992698. URL `https://doi.org/10.1007/BF00992698`.

[37] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992. doi: 10.1007/BF00992696. URL `https://doi.org/10.1007/BF00992696`.

[38] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893.

[39] Timothy Yee, Viliam Lisy, and Michael Bowling. Monte carlo tree search in continuous action spaces with execution uncertainty. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 690–696. AAAI Press, 2016. ISBN 9781577357704.

Appendix A

Implementation Details

A.1 Training Details and Hyperparameters

In this section, we detail our training setting, as well as the hyperparameters used, whose full list is reported in Table A.1 for the grid world trainings and in Table A.2 for the CarGoal training.

Category	**Parameter**	**E-8x8**	**E-16x16**	**MZ-LR-8x8**	**MZ-RL-8x8**
Environment	`max_ep_len`	100	200	200	200
	`disc_factor`	0.95	0.95	0.95	0.95
Training	`iterations`	50	60	100	150
	`learning_epochs`	4	4	4	4
	`sample_size`	6	6	6	6
	`buffer_size`	90	90	90	90
	`batch_size`	22	22	22	22
	`learning_rate`	0.001	0.003	0.001	0.001
	`optimizer`	Adam	Adam	Adam	Adam
Loss	`value_weight`	0.7	0.7	0.7	0.7
	`policy_weight`	0.3	0.3	0.3	0.3
	`n_steps`	2	2	2	2
Neural Network	`hidden_size`	64	64	64	64
	`hidden_num`	2	2	2	2
	`activation`	ReLU	ReLU	ReLU	ReLU
	`batch_norm`	No	No	No	No
Planning	`planning_budget`	64	128	64	64
	`c`	1	1	0.5	0.5
	`dir_eps`	0.4	0.4	0.4	0.4
	`dir_alpha`	2.5	2.5	2.5	2.5

Table A.1: Training parameters for the different grid world environments.

Since our novel agents are specifically designed for being deployed once learning is over,

Category	**Parameter**	**CarGoal**
Environment	`max_ep_len`	200
	`disc_factor`	0.975
Training	`iterations`	1000
	`learning_epochs`	4
	`sample_size`	12
	`buffer_size`	180
	`batch_size`	45
	`learning_rate`	0.0001
	`optimizer`	Adam
Loss	`value_weight`	100
	`policy_weight`	0.01
	`n_steps`	1
Neural Network	`hidden_size`	64
	`hidden_num`	3
	`activation`	ReLU
	`batch_norm`	Yes
Planning	`planning_budget`	128
	`c`	1
	`dir_eps`	0.4
	`dir_alpha`	2.5

Table A.2: Training parameters for the CarGoal environment.

the NN training is carried out following the original AlphaZero framework and by employing one of its several available implementations, specifically the one used as a baseline in [12]. Consequently, many of the implementation details reported here closely follow those described in their work.

Each agent is trained by alternating three phases, and we refer to the completion of these three phases for a single time as completing one of the multiple `iterations` :

1. **Sampling** of episodes
2. **Learning**
3. **Evaluation** (periodical)

During the sampling phase, the agent collects a number of `sample_size` episodes sampled by planning and acting in the environment using the current estimators. In this phase, we want to somewhat explore the environment and as such, we always act by sampling from the (stochastic) evaluation policy after planning. Dirichlet noise is also applied to the prior policy at the root as described in subsection 2.3.2. Each episode ends when we reach a terminal state or we hit the `max_ep_len`. The neural network parameters are not updated during this phase, i.e., we do not let any gradient flow. Then, these sampled episodes are added to a replay buffer of maximum capacity `buffer_size`.

During the learning phase, we repeat a learning loop for a number of `learning_epochs`. For every epoch, we extract `batch_size` episodes from the replay buffer uniformly at random and use them to compute the value and policy losses following a modified version of the process described in subsection 2.3.3. The details of the losses computation are described in subsection A.1.1. The total AZ loss is then computed as their weighted sum with weights `value_weight` and `policy_weight` respectively. Based on that, the selected `optimizer` updates the neural network's parameters. Note that we selected the Adam optimizer [14] in all our trainings. The pace at which the parameters are updated depends on the `learning_rate`.

Every `eval_period` iterations, an evaluation phase is also run. The goal of this phase is to evaluate the current evaluation policy as a deterministic policy, i.e., by always taking the argmax action. Note that the neural network is not updated during this phase and that Dirichlet noise is also not applied. We therefore run an arbitrary number of evaluation episodes of maximum length `max_ep_len` and compute the averaged metrics of interest, such as the mean discounted and undiscounted return. Based on the outcomes, we may decide to prematurely stop training if performance is already (close to) optimal.

A.1.1 Loss Computation

Compared to standard AlphaZero implementations, both the policy and value losses are normalized over the number of unique states in the utilized training framework. This adjustment helps prevent the total loss from being overly influenced by repeated occurrences of the same state, especially in situations where an agent gets stuck in one place, resulting in most of the sampled trajectory consisting of that single state [12]. In practice, this is done in the following way:

- First, count how many times each unique state appears in the batch.
- Then, for each step, divide the corresponding loss term by the count of how often that state appears.
- Finally, normalize these terms so that the loss scale is preserved.

Regarding hyperparameters, the `n_steps` parameter sets the value of n used for the bootstrapped n-step value loss. The `disc_factor` parameter indicates the chosen γ for discounting.

A.1.2 Neural Network Architecture

Since our input state embeddings are simple vectors for both grid world and CarGoal environments, our neural network architecture is a standard Multi-Layer Perceptron (MLP) [27]. We also need the network to have separate output heads for the value and policy, respectively, where the policy output needs to be preceded by a softmax operation. We chose to retain the standard architecture configuration used in [12], as it provided effective

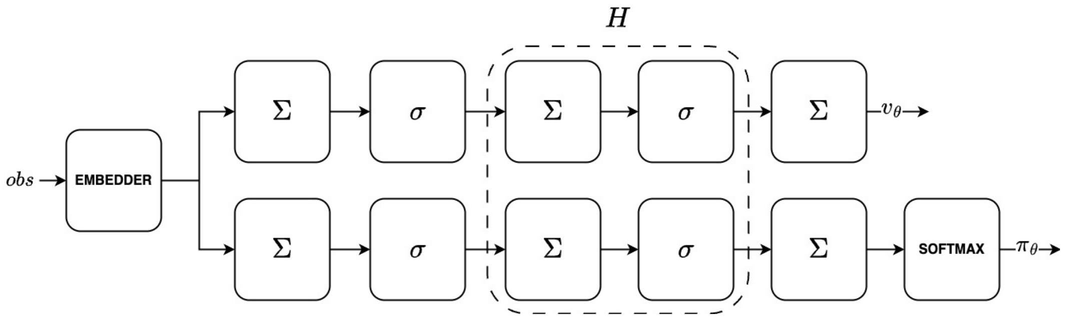


Figure A.1: MLP architecture with a variable number of hidden layers H. The Σ block represents a sum and the σ block represents an activation function (e.g., ReLU).

and quick training performance during early experimentation. A diagram visualizing the architecture is reported in Figure A.1. The number of utilized hidden layers H is reported in our hyperparameters table as `hidden_num`, while the number of neurons per layer is indicated by the `hidden_size` parameter. The utilized `activation` function is always ReLU [10]. Sometimes, it can be useful to apply batch normalization [11] to stabilize training; whether it is used for a specific environment is indicated by the parameter `batch_norm`. This can be implemented by adding additional batch normalization layers before each activation block of the H hidden layers.

A.1.3 Planning Parameters

The `planning_budget` indicates the number of node expansions that AZ can perform during planning before it must choose an action to undertake in the real environment. The `c` parameter determines the amount of exploration that our selection policy commits. During training, we also add Dirichlet noise to the prior policy, controlled by the parameters `dir_eps` and `dir_alpha`, to encourage more exploration at the root.

A.1.4 Environments Implementation

Grid Worlds

Our Grid World environments have been implemented as custom modifications of the popular FrozenLake Gym environment, and as such, they are compatible with the usual Gym/Gymnasium framework [6] [33]. In FrozenLake, the obstacles are holes that terminate the current episode if the agent moves towards them; therefore, we modified them to obtain the standard obstacle behavior that we wanted for our experiments. The standard Gym observation format for this environment is a discrete value d, from which we can extract the 2D coordinate as:

$$(x, y) = \left(\left\lfloor \frac{d}{\text{ncols}} \right\rfloor, d \bmod \text{ncols} \right)$$

Where ncols is the number of columns of the configuration. Note that the number of rows is the same since all the employed configurations are square. We employ this 2D coordinate vector as an input to our neural network.

CarGoal

As anticipated, we created CarGoal by modifying an existing environment created by Kexian Shen and available through this repository. The main modifications that we applied are the following:

- The original environment allows the agent to directly control the car's velocity, without modeling acceleration. We replaced this with acceleration control to better reflect realistic vehicle dynamics.
- Although the original environment supports various action space types (continuous, discrete, multi-continuous, and multi-discrete), its discrete setting does not allow backward motion, since velocity is kept constant and only the steering angle is controlled in that case. We addressed this by implementing the previously discussed discrete action space that enables both forward and backward movement via positive or negative acceleration.
- The default timestep in the original environment is $\Delta t = 0.4$. We reduce it to 0.25 to make velocity and heading angle changes less abrupt.

Since our default state representation $\mathbf{s} = [x, y, \theta, v]$ is already in a suitable form to be passed to the neural network, we directly use it as our state embedding. The framework also allows for training using RGB image observations instead. While we did not experiment with this setting, it would be an interesting future experiment to see how this approach differently affects the results of our evaluation.

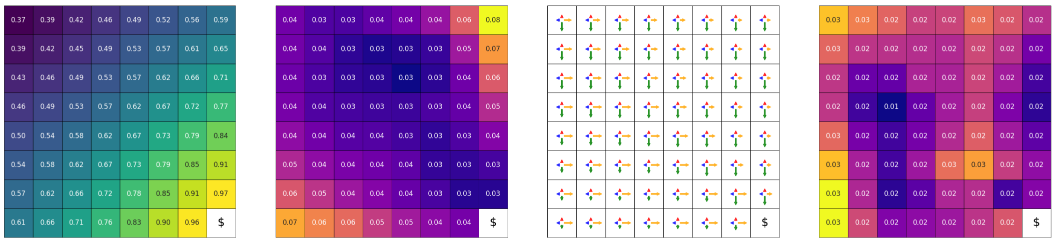
A.1.5 Learned Values and Policies

An advantage of simple grid world environments is that we can easily visualize the learned estimators. We can then represent the averaged statistics of the learned value and policy functions. For the values, we visualize the average per-state value across the 10 seeds, along with the corresponding standard deviation (SD). For the policies, we visualize the policy logits by showing corresponding weighted-length arrows indicating which direction(s) the agent is more inclined to follow across the seeds. Similarly, we report the SD of the policy logits by showing the average per-action SD in each state. All these visualizations are created for each grid world training environment:

- Figure A.2 shows average statistics of the learned value and policy functions for the 8x8 empty grid world environment.
- Figure A.3 shows average statistics of the learned value and policy functions for the 16x16 empty grid world environment.

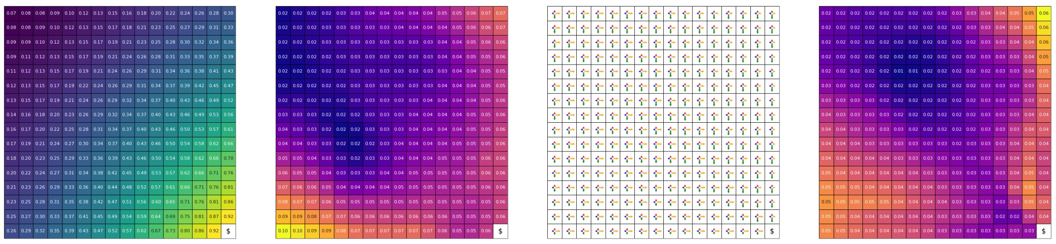
- Figure A.4 shows average statistics of the learned value and policy functions for the MAZE_LR environment.

- Figure A.5 shows average statistics of the learned value and policy functions for the MAZE_RL environment.



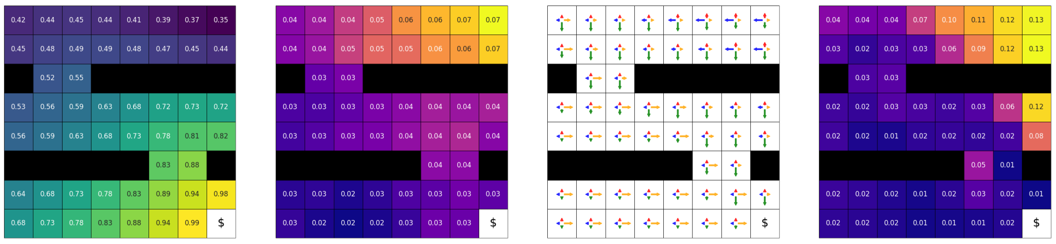
(a) Mean values (b) Values SD (c) Mean policy logits (d) Logits SD

Figure A.2: Mean and SD of NN estimates over 10 seeds on 8×8 empty grid environment.



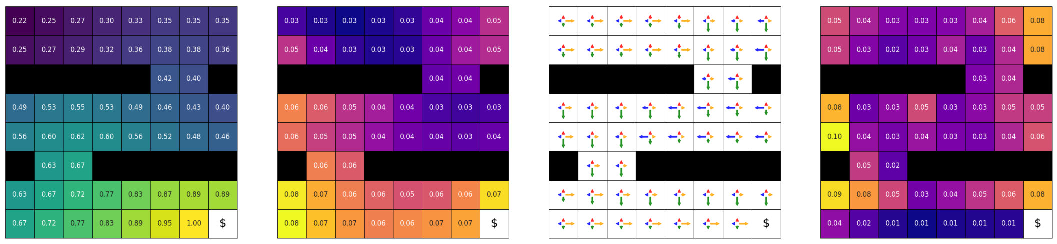
(a) Mean values (b) Values SD (c) Mean policy logits (d) Logits SD

Figure A.3: Mean and SD of NN estimates over 10 seeds on 16×16 empty grid environment.



(a) Mean values (b) Values SD (c) Mean policy logits (d) Logits SD

Figure A.4: Mean and SD of NN estimates over 10 seeds on the MAZE_LR environment.



(a) Mean values (b) Values SD (c) Mean policy logits (d) Logits SD

Figure A.5: Mean and SD of NN estimates over 10 seeds on the MAZE_RL environment.

A.2 Evaluation Details and Hyperparameters

In this section, we detail the evaluation settings of novel agents and baselines on all the test configurations we experimented with, as well as corresponding hyperparameters whose full list is reported in Table A.3 for the grid world evaluation and in Table A.4 for the CarGoal evaluation. The only parameters not reported in the tables are the ones unrelated to the specific algorithm, i.e., the discount factor and the maximum number of steps before truncating the episode. These are set to `disc_factor` $= 0.95$, `max_ep_len` $= 100$ for all the grid world environments, and `disc_factor` $= 0.975$, `max_ep_len` $= 200$ for the CarGoal environment.

In the tables, the `eval_policy` parameter represents the evaluation policy used to pick an action after constructing the tree, as well as the one used to estimate node Q-values, as discussed in section 2.4. The `sel_policy` parameter indicates the utilized selection policy during tree construction along with the corresponding `c` exploration constant. Note that choosing PUCT has an influence even if `c` is zero, since it also modifies the order of expansion of unexpanded nodes according to the prior policy (instead of uniformly at random). For the algorithms using MVC evaluation, we also indicate the chosen value for the β parameter, indicated as `beta`.

All the other reported parameters are specific to one or more of the novel algorithms employed. The `det_tol` parameter represents the ϵ tolerance used during detection in all three detection-based algorithms, while `det_roll_budget` indicates the n nodes rolled out for detection by MINITREES and MEGATREE. The `pol_tol` parameter indicates the ζ threshold over which we consider an action off-policy w.r.t. the prior in PDDP detection. The `val_pen` parameter indicates how much we penalize the value of a problematic node after detection and corresponds to the parameter p described in subsection 4.2.3.

Finally, `loop_th` corresponds to the η threshold of EDP's loop blocking mechanism mentioned in subsection 4.3.3. Recall that setting it to zero corresponds to only blocking exact loops, which is the standard behavior for discrete state spaces, like in the grid world case.

Algorithm	Parameter	E-8x8	E-16x16	MZ-LR-8x8	MZ-RL-8x8
MINITREES	`eval_policy`	visit	visit	/	/
	`sel_policy`	PUCT	PUCT	/	/
	`c`	0.1	0.1	/	/
	`det_tol`	0.08	0.05	/	/
	`det_roll_budget`	4	4	/	/
MEGATREE	`eval_policy`	MVC	MVC	/	/
	`sel_policy`	PUCT	PUCT	/	/
	`c`	0.1	0.1	/	/
	`beta`	10	1	/	/
	`det_tol`	0.08	0.05	/	/
	`det_roll_budget`	4	4	/	/
PDDP	`eval_policy`	MVC	MVC	MVC	MVC
	`sel_policy`	UCT	UCT	UCT	UCT
	`c`	0	0	0.1	0.1
	`beta`	10	1	100	10
	`det_tol`	0.05	0.05	0.08	0.05
	`pol_tol`	0.1	0.1	0.1	0.1
	`val_pen`	1	1	0.1	0.3
EDP	`eval_policy`	MVC	MVC	MVC	MVC
	`sel_policy`	PUCT	PUCT	PUCT	PUCT
	`c`	0	0	0	0
	`beta`	10	10	10	10
	`loop_th`	0	0	0	0

Table A.3: Utilized planning hyperparameters for grid world experiments. The configurations (e.g., E-8x8) indicate the training environment; the same parameters were used for all the corresponding experiments on the different test configurations.

Algorithm	Parameter	CarGoal
PDDP	`eval_policy`	MVC
	`sel_policy`	PUCT
	`c`	0.1
	`beta`	10
	`det_tol`	0.025
	`pol_tol`	0.05
	`val_pen`	0.2
EDP	`eval_policy`	MVC
	`sel_policy`	PUCT
	`c`	0.1
	`beta`	1
	`loop_th`	0.03

Table A.4: Utilized planning hyperparameters for CarGoal experiments. The same hyperparameters were used for all the corresponding experiments on the different test configurations.

Appendix B

Complementary Experiments

B.1 Influence of the Exploration Constant on AZ+PUCT and AZ+UCT Baselines

The standard AlphaZero planning only requires tuning the C hyperparameter at deployment, since Dirichlet noise is no longer applied, and neural networks are not updated. Note that the value for C chosen during training does not necessarily represent the optimal choice at deployment. A higher value might be more useful during training, where exploration is strictly necessary, while a lower value at deployment can be a better choice if the prior estimates are reliable. As this is not necessarily true in our case due to the changed test configurations, we want to tune the parameter in order to compare our algorithms to the best possible baseline we can achieve. We do this for every environment that we employ and for both AZ+PUCT and AZ+UCT baselines:

- AZ+PUCT and AZ+UCT on 8×8 grid world test configurations (Figure B.1).
- AZ+PUCT and AZ+UCT on 16×16 grid world test configurations (Figure B.2).
- AZ+PUCT and AZ+UCT on MAZE test configurations (MAZE_LR training) (Figure B.3).
- AZ+PUCT and AZ+UCT on MAZE test configurations (MAZE_RL training) (Figure B.4).
- AZ+PUCT and AZ+UCT on CarGoal test configurations (Figure B.5).

Overall, the plots reveal that, despite the changes, a relatively low value of C (≤ 0.5) remains the best-performing choice for most configurations. Increasing its value leads the agent to highly suboptimal performances even in relatively simple test configurations, especially for low planning budgets. This is likely because, with insufficient visits, a high C value slows down the decay of the exploration term in planning. As a result, the planning distribution remains close to uniform, and using visitation counts to guide actions ultimately resembles near-random behavior.

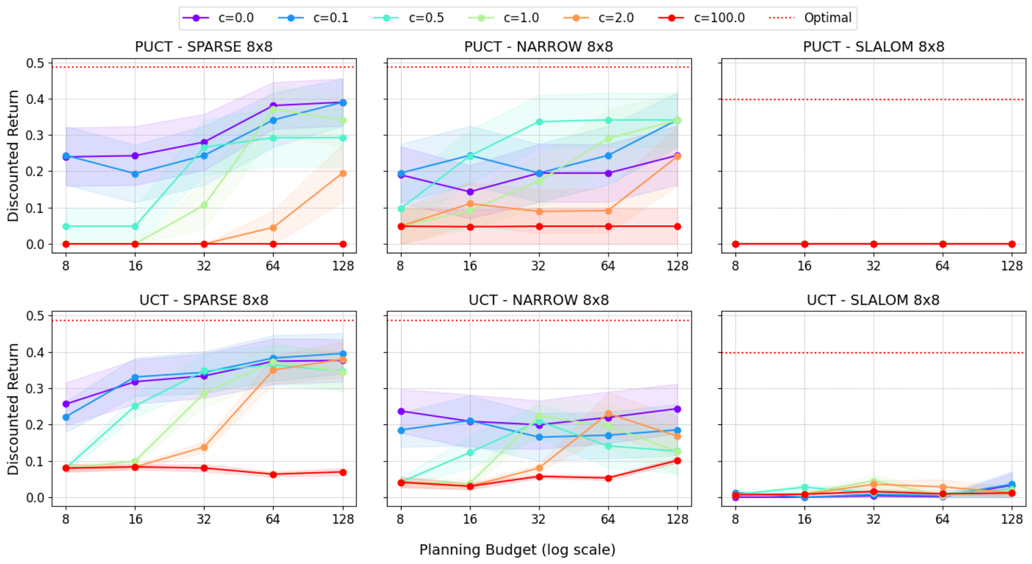


Figure B.1: Influence of the C parameter on AlphaZero baselines in 8×8 grid world test configurations.

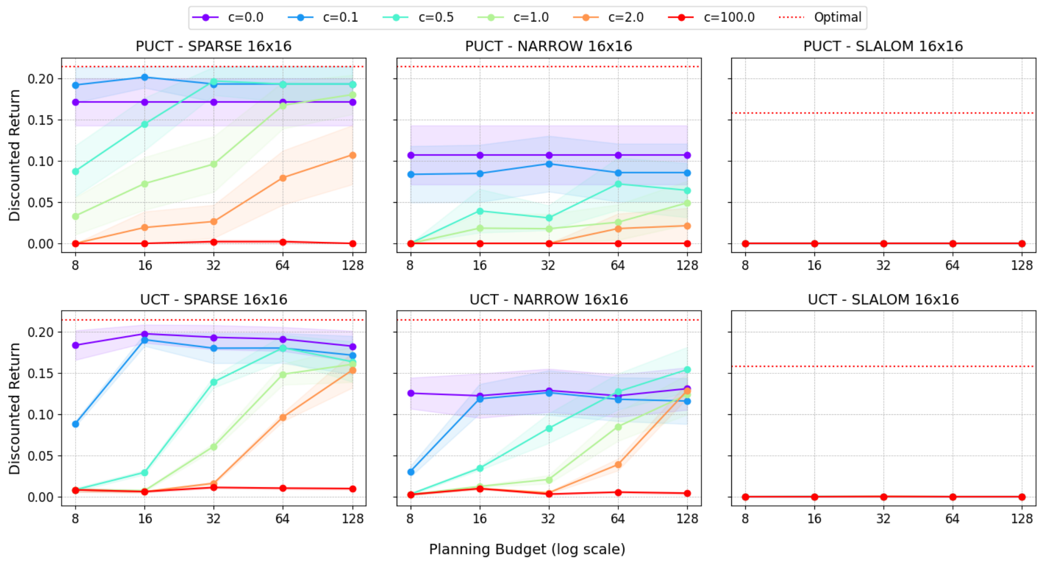


Figure B.2: Influence of the C parameter on AlphaZero baselines in 16×16 grid world test configurations.

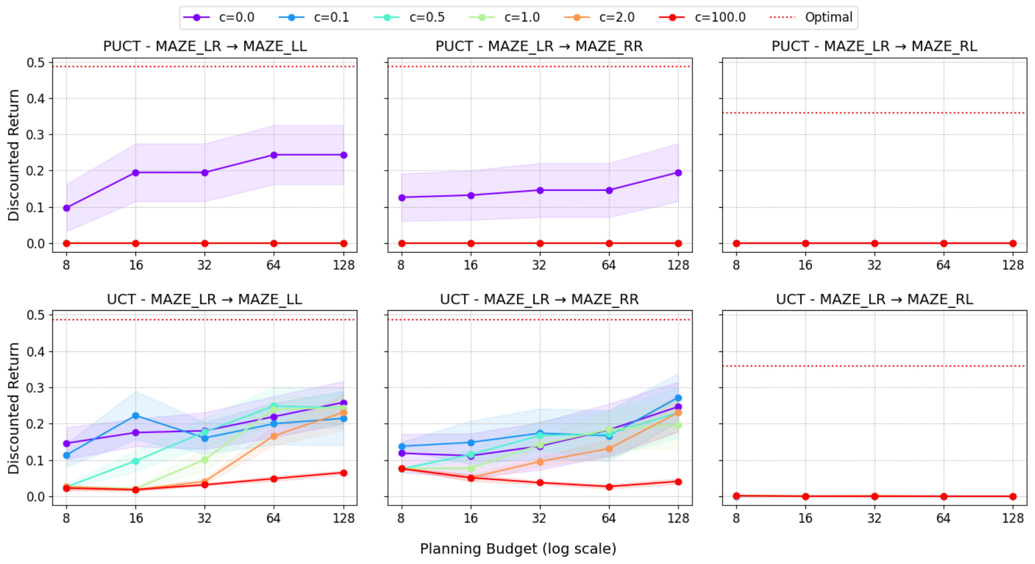


Figure B.3: Influence of the C parameter on AlphaZero baselines in MAZE test configurations with MAZE_LR training. The label MAZE_X $\rightarrow$ MAZE_Y on top of each plot indicates training on the MAZE_X configuration and testing on the MAZE_Y one.

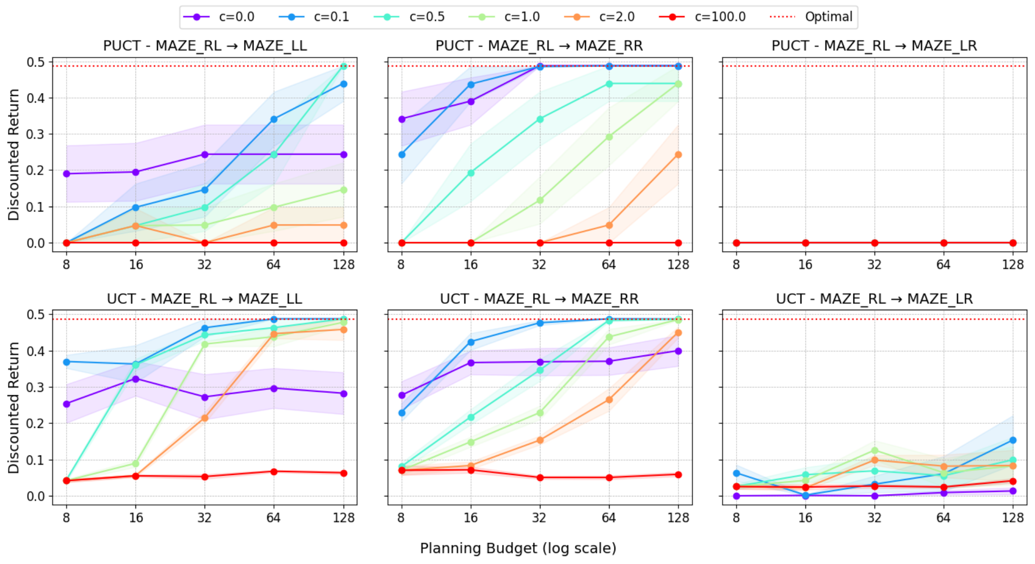


Figure B.4: Influence of the C parameter on AlphaZero baselines in MAZE test configurations with MAZE_RL training. The label MAZE_X $\rightarrow$ MAZE_Y on top of each plot indicates training on the MAZE_X configuration and testing on the MAZE_Y one.

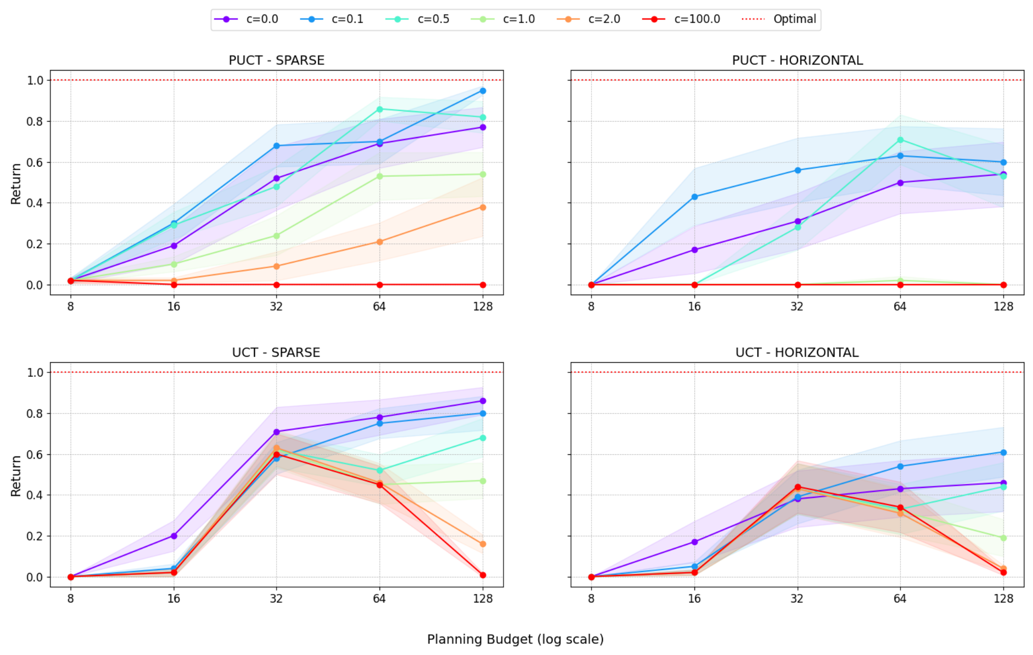


Figure B.5: Influence of the C parameter on AlphaZero baselines in CarGoal test configurations.

B.2 Influence of the Beta Parameter on MVC+PUCT and MVC+UCT Baselines

The MVC agents are more challenging to tune, as their performance depends on both the same C parameter used in standard AZ and the β parameter, which determines the importance we want to give to the variance of the value estimates. As mentioned in chapter 2, high values of β make the agent more greedy and optimistic in both acting and estimating nodes' Q values during planning. To perform the tuning, we select two values of C that showed the overall best performance in the standard AZ tuning, $(0, 0.1)$. We then test their combinations with $\beta = 0, 10, 100$ on all the employed environments, both with and without a prior policy network. The results are reported in the following plots:

- MVC+PUCT and MVC+UCT on 8×8 grid world test configurations (Figure B.6).
- MVC+PUCT and MVC+UCT on 16×16 grid world test configurations (Figure B.7).
- MVC+PUCT and MVC+UCT on MAZE test configurations (MAZE_LR training) (Figure B.8).
- MVC+PUCT and MVC+UCT on MAZE test configurations (MAZE_RL training) (Figure B.9).
- MVC+PUCT and MVC+UCT on CarGoal training configurations (Figure B.10).

Overall, we observe that varying the hyperparameters has a relatively less significant influence on the results compared to what we saw when tuning C on AZ agents. In general, using MVC evaluation alone is insufficient to overcome the challenges of the most complex test configurations, such as the SLALOM test configurations or the MAZE_LR $\rightarrow$ MAZE_RL and MAZE_RL $\rightarrow$ MAZE_LR challenges, where performance remains approximately zero regardless of the utilized parameters.

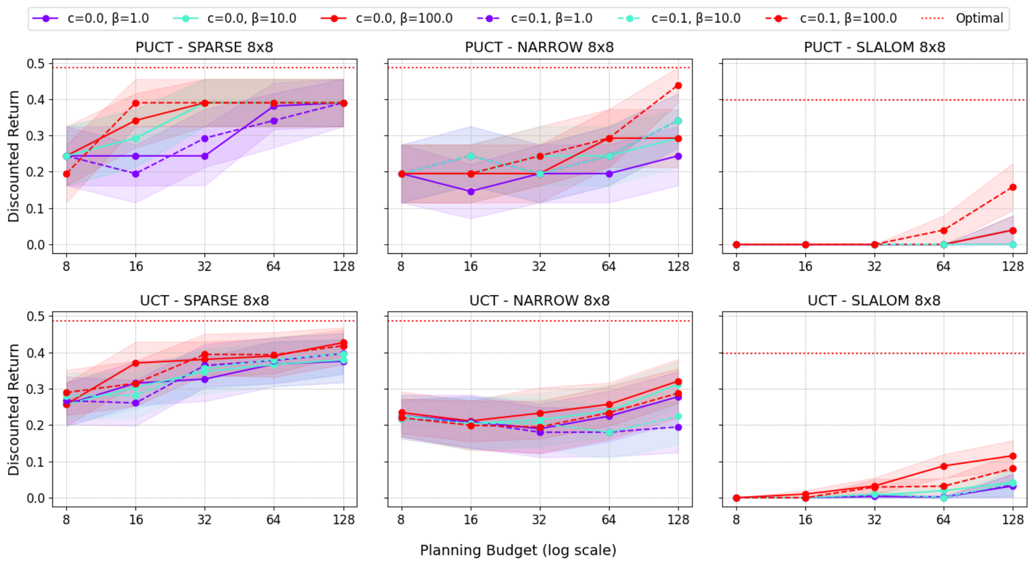


Figure B.6: Influence of β and C on MVC baselines in 8×8 grid world test configurations.

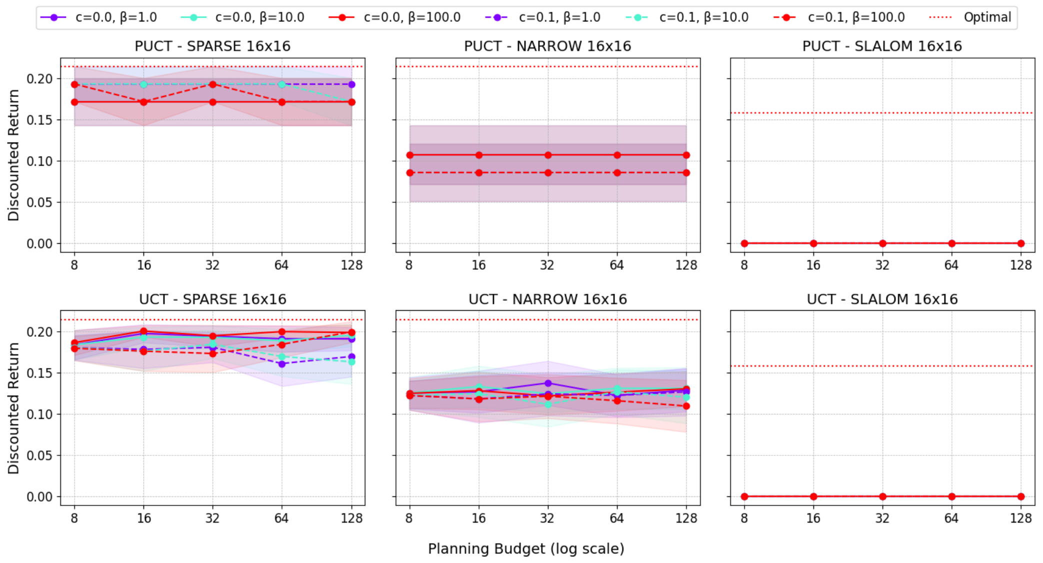


Figure B.7: Influence of β and C on MVC baselines in 16×16 grid world test configurations.

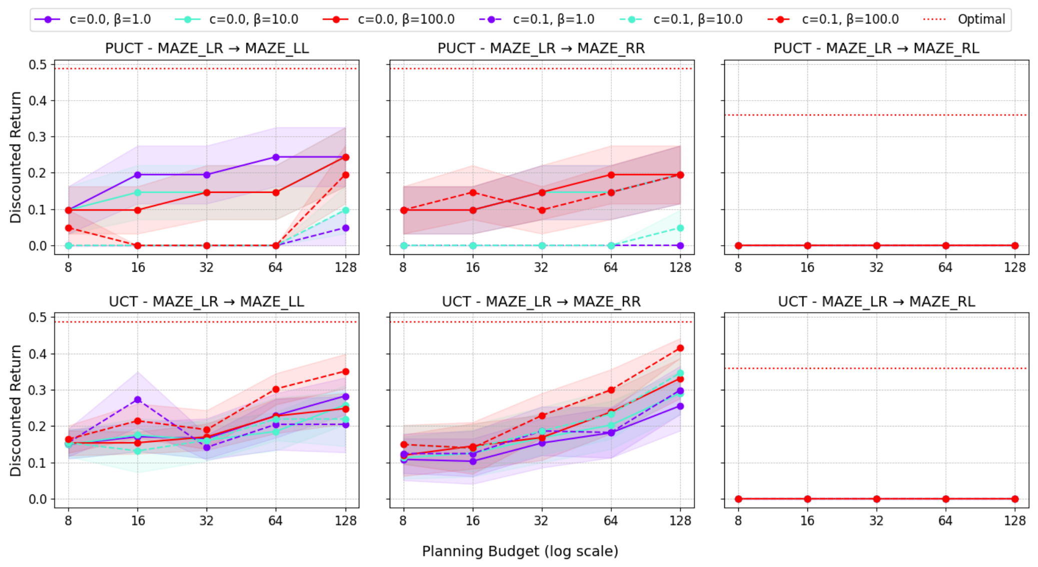


Figure B.8: Influence of β and C on MVC baselines in MAZE test configurations with MAZE_LR training. The label MAZE_X $\rightarrow$ MAZE_Y on top of each plot indicates training on the MAZE_X configuration and testing on the MAZE_Y one.

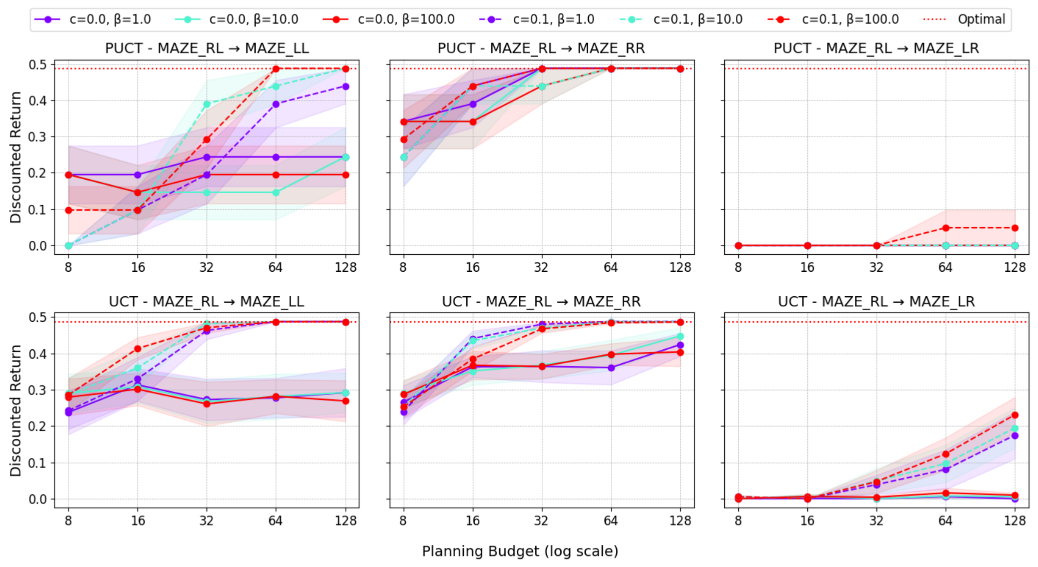


Figure B.9: Influence of the β and C on MVC baselines in MAZE test configurations with MAZE_RL training. The label MAZE_X $\rightarrow$ MAZE_Y on top of each plot indicates training on the MAZE_X configuration and testing on the MAZE_Y one.

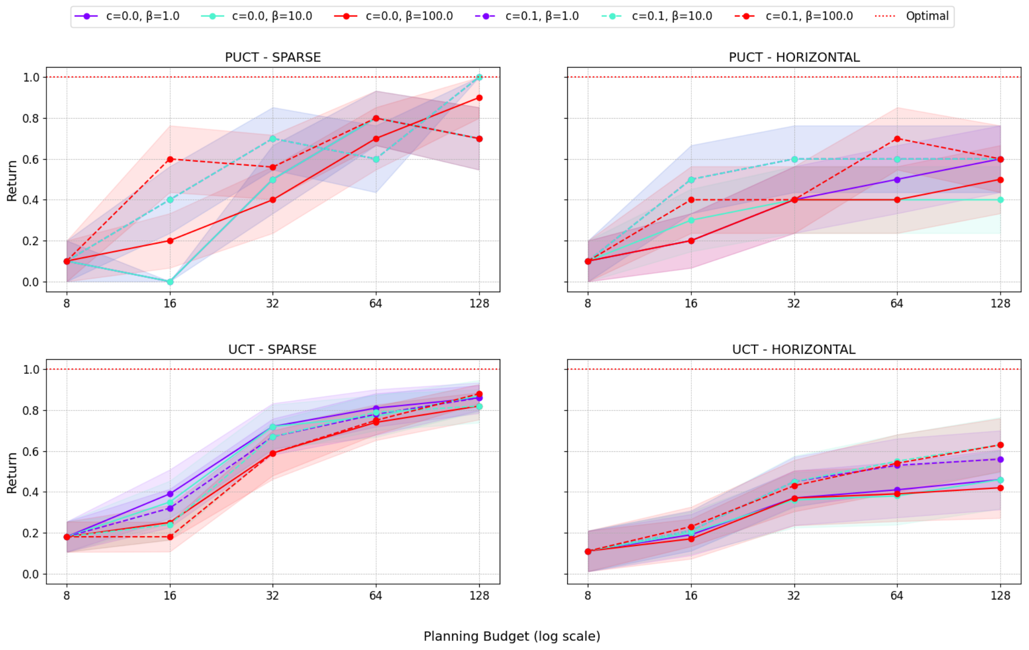


Figure B.10: Influence of β and C on MVC baselines in CarGoal test configurations.