Università degli Studi di Verona

Master Degree in Ingegneria e scienze informatiche

A Rule-Based Approach for Safe Information-Oriented Planning

Nereo Sorio

Supervisor

Prof. Alessandro Farinelli

Co-Supervisor

Giulio Mazzi

Ringraziamenti

Ringrazio il mio relatore Prof. Alessandro Farinelli e il mio correlatore Giulio Mazzi per l'aiuto e la disponibilità dimostratami.
Ringrazio la mia famiglia che mi ha sempre sostenuto.
Ringrazio Anna per aver condiviso insieme a me tutto il percorso durante questi lunghi anni.
Ringrazio Massimiliano senza il quale probabilmente non sarei qua.
Infine ringrazio gli amici di una vita, per aver reso questo viaggio più leggero e divertente.

Contents

Chapter 1

Introduction

Planning is a central research topic in Artificial Intelligence. It formalizes the problem of selecting which action, or sequence of actions, is the most beneficial for an autonomous agent. This is a challenging problem because these actions must maximize an objective (or goal) function. There are several frameworks that allow solving these problems, one of the most important is the *Markov Decision Process (MDP)* framework.

We can use MDPs to model domains with actions that are (partially) random. The outcome of an action is defined by the current state of the system, that is assumed to be fully observable. With this assumption, the agent always knows its current state. When the environment is only partially observable, the agent does not necessarily know the real current state. Thus, we use *Partially Observable Markov Decision Process (POMDP)* to handle with planning under uncertainty problems.

In this thesis, we consider a well known online approach to solve POMDPs: *Partially Observable Monte-Carlo Planning (POMCP)* [11].

One of the strength of this algorithm is that it does not require an explicit definition of the transition model, observation model, and reward, but it can be simulated using a black box simulator.

The uncertainty given by the partially observable environment, however, influences in an indirect way the reward. In POMCP all the reward is given by the true state, which is not known a priori, and therefore we have to approximate this information with the belief. The more precise the belief is, the less uncertainty there is, but this is not measured directly. However, the central focus of certain experimental domains is on reducing the uncertainty of the belief (e.g., robot patroling). Thus, in this thesis, we focus our attention on the ρ-POMDP extension, an extended framework that directly quantifies the influence of the belief. In this framework the reward function ρ depends on the belief state (belief-dependent), thus allowing to define not only *control oriented* objectives, but also *information-oriented* ones. In this perspective, a recent work proposes ρ-POMCP, which is an extension of POMCP that handles ρ-POMDPs problems [14]. ρ-POMCP performs well in practical domains, but as in standard POMCP, ρ-POMCP generad policies are hard to understand (i.e., it is not easy to understand the reason behind a choice of action). However, explainability in aritifical intelligence is a growing topic, and a crucial aspect of moder AI systems. For this reason, in this thesis we build and develop an integration of ρ-POMCP and a rule synthesis procedure, XPOMCP [6], that can build rule-based description of POMCP generated policies.

To achieve this goal, we implemented ρ-POMCP problem starting from POMCP, and we integrate this implementation with the capabilities of generating execution traces that can be used within XPOMCP.

Thanks to the integration of XPOMCP and ρ-POMCP, it is possible to build richer traces that collect more information. Specifically, in this thesis we describe how to generate *extended traces*. They exploit the fact that ρ-POMCP analyzes many different belief during a single run, and this allows us to collect a significantly higher number of belief-action pairs during each run.

We test ρ-POMCP and the rule generation algorithm in two experimental domains, namely *tiger* and *rocksample*. We show that XPOMCP can be used to build effective rules for the problems. We also show that in *rocksample*, a challenging problem, the usage of extended traces speed-ups the rule synthesis process significantly.

To summarize, in this thesis we present three contributions to the state-of-the-art:

- We integrate the ρ-POMCP algorithm with an explainability tool, *XPOMCP*, to build rules that describes the behavior of a policy.
- We use this integration to build *extended traces*, that collect more information during each run
- We evaluate the proposed approach in two standard POMDP domains, namely, *tiger* and *rocksample*

The thesis is structured as follow, in Chapter 2 we present theoretical background and definitions. Chapter 3 presents some of the important contribution of this thesis. In Chapter 4 we present EXplainable POMCP (XPOMCP) [6], which introduces a novel way of interpreting POMCP policies, and we extend its usage to the case of ρ-POMDPs. Finally, in Chapter 5 we show the experimental results of the proposed methodologies, namely the combination of trace generation for XPOMCP with ρ-POMCP, and the use of extended traces to speed up rule generation.

Chapter 2

Background

This chapter introduces the problem of planning under uncertainty. It provides theoretical background and definitions used in the thesis. In Figure 2.1 we presents a summary of the element in AI planning. Specifically, Section 2.2 presents Markov Decision Processes (MDPs) that is a framework used in a wide range of environments for solving planning and decision problems. In Section 2.3 we extend the concept of MDP in the Partially Observable environment with POMDPs. Section 2.4 presents an algorithm that solve in a efficienty way the POMDPs problem. Collecting information is important, so to integrate in the reward this thing we extends POMDPs to ρ-POMDPs in Section 2.5 and finally we provide an algorithm that solves this type of problem that is the ρ-POMCP in Section 2.6.

2.1 Planning Under Uncertainty

An *agent* is anything that can be viewed as perceiving its *environment* through *sensors* and acting upon that environment through *actuators* [10]. In certain domains, the agent must handle uncertainty, whether due to *partial observability*, *nondeterminism*, or a combination of the two. The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now. That is, the agent maintain some sort of *internal state* that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

An agent may never know for certain what state it is in or where it will end up after a sequence of actions. The aim is to design agents for environments where there may be incomplete or faulty information, where actions may not always have the same results and where there may be tradeoffs between the different possible outcomes of a plan. In this thesis, we define a *planning problem* as follow: Given a description of the current state of a system, a set of actions that can be performed on the system and a description of a goal set of states for the system, find a sequence of actions that can be performed to transform the system into one of the goal states $s_g \in S$. The *Probability* provides a way of summarizing the uncertainty that comes from our laziness and ignorance of the real world. In this way, we can quantify the uncertainty related to a certain information with probability statements that are made with respect to a knowledge state, not with respect to the real world.

Utility theory is interested in people's preferences or values and with assumptions about a person's preferences that enable them to be represented in numerically useful ways. Utility theory says that every state has a degree of usefulness, or utility, to an agent and that the agent will prefer states with higher utility (the utility of a state is relative to an agent) [3]. Then, we can think of *Decision theory* as the sum of *probability theory* and *utility theory.* The fundamental idea of decision theory is that an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all possible outcomes of the action. This is called the principle of *maximum expected utility* (MEU).

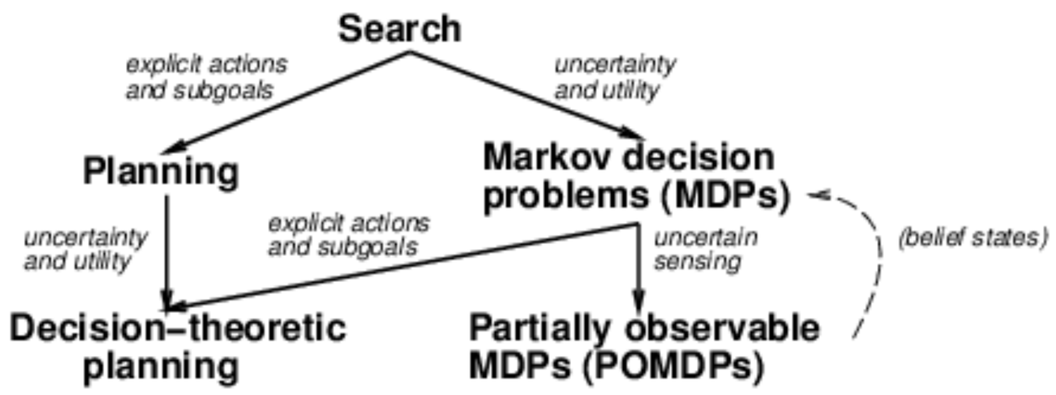

Search
explicit actions and subgoals
uncertainty and utility
Planning
Markov decision problems (MDPs)
uncertainty and utility
explicit actions and subgoals
uncertain sensing
(belief states)
Decision–theoretic planning
Partially observable MDPs (POMDPs)

Figure 2.1: Summary of the elements of AI planning.

2.2 Markov Decision Process (MDP)

A *Markov Decision Process*(MDPs) is a framework for planning and decision procedures. MDPs have been studied in several fields, including AI, economics, operations research and control theory. The success of Markovian approaches in areas such as speech recognition and the closely-related reinforcement learning techniques have encouraged work in planning using Markov Decision Processes.

As in classical planning, an MDP agent is a decision maker. The thing it interacts with, comprising everything outside the agent, is called the *environment*. The interaction between the agent and the environment is mutual, the agent selects actions and the environment responds to these actions by presenting new situations to the agent. This interaction is summarized in Figure 2.2. The environment also yields rewards $r \in R$. These are real number that the agent seeks to maximize over time through its choice of actions.

Definition 1 *A* Markov Decision Processes*(MDPs) can be defined as a tuple* $\langle S, A, T, R \rangle$*, where*

- S *is a finite set of states of the system (with an initial state* s_0*).*
- A *is a finite set of actions.*

- T *: $S \times A \rightarrow \Pi(S)$ is the transition model, it maps a state and an action to a probability distribution over the set of states with probability p(s' |s, a).*

- R*: $S{\times}A \rightarrow \mathbb{R}$ is the reward function, which defines the immediate reward achieved by selecting action $a{\in}A$ in state $s \in S$.*

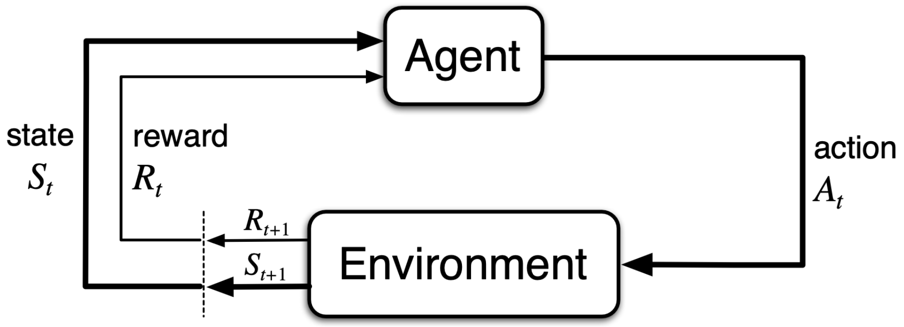


Figure 2.2: A summary of the interaction between an MDP agent and the environment (Reinforcement Learning with Exploration by Random Network Distillation, 2021)

To solve an MDP, we must find which action the agent should select given a state s. A solution of this kind is called a *policy.*

Definition 2 *A* policy*, $\pi : S \rightarrow A$ is a function that maps an action given the current state of the system.*

The goal of a policy is maximize the sum of the rewards achieved by the agent during its execution. In particular, an agent that acts indefinitely must maximize its discounted return, defined as:

Definition 3 *The infinite horizon discounted return is the expected sum of reward during the agent's life.* $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t * R_t]$

The parameter $\gamma \in [0, 1)$ specifies how much value to give to the reward at each step. The larger γ is, the higher is the impact of future rewards on the current decision.

Given a problem there can be multiple policies for it. For instance, we could simply take the first action that comes to mind, select an action at random, or run a heuristic.

There are different types of policies:

- *Non-stationary policy*: $\pi : S \times Z \rightarrow A$, is a mapping from the state space and time to the action space.
- *Stationary policy*: $\pi : S \rightarrow A$, is a mapping from the state space to the action space.

We say that the agent has a *complete policy* if it has an action for each possible state. We can measure the quality of a policy by adding up its rewards over time. The sum of the rewards obtained by the policy is called *discunted return.*

We consider two utility functions that combine the rewards achieved by a policy:

Definition 4 ***Additive utility function***
$U([s_0, s_1, s_2, ...]) = R(s_0) + R(s_1) + R(s_2) + ...$

Definition 5 ***Discounted utility function***
$U([s_0, s_1, s_2, ...]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...$
where γ *is the* discount factor

The quality of a policy is measured by the expected utility of the possible environment histories generated by that policy the one that yields the highest expected utility is called the *optimal policy.* A MDP is called *finite* if the state and action spaces are finite. The *value function* of a state s under a policy π, denoted $V^\pi(s)$, is the expected return when starting in state s and following π thereafter. For MDPs, we define V^π as

Definition 6 $V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s')V^\pi(s')$

A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $V^{\pi}(s) \geq V^{\pi^{'}}(s)$ for all $s \in S$.

There is always at least one policy that is better than or equal to all other policies. This is an optimal policy. Although there may be more than one, we denote all the optimal policies by π^*. The problem is to find an optimal policy given a Value Function V , defined as:

Definition 7 $\pi^*(s) = \underset{a}{\text{argmax}}[R(s,a) + \gamma \sum\limits_{s' \in S} T(s,a,s')V(s')]$

Policies that share the same state-value function, called the optimal state-value function, denoted V^*, and defined as

Definition 8 $V^*(s) = \max\limits_{\pi}[R(s,a) + \gamma \sum\limits_{s' \in s} T(s,a,s')V^*(s')]$

Similarly, we define the *action value* function (or *quality function*) as the value of taking action a in state s under a policy π, denoted $Q^{\pi}(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

Definition 9 $Q^{\pi}(s,a) = \sum\limits_{s'} p(s'|a,s)(R(s,a,s') + \gamma V^{\pi}(s'))$

Optimal policies also share the same *optimal action-value function*, denoted Q^*, also knows as *Bellman Optimality Equation* [10], and defined as:

Definition 10 $Q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \max\limits_{a'} Q^*(s',a')]$.

This formula can be used to solve MDPs, used for solving MDP, a popular algorithm that use the equation of Bellman is *Value Iteration* (presented in Algorithm 1).

Value Iteration computes the value for a given policy π, and makes the policy π greedy with respect to V^{π}. At each time step t, computes Q_t^a , which, returns the corresponding reward for taking action a in state s. Therefore, Q_s^a is the reward received at time t, summed with the future rewards the agent obtains for taking actions in the environment with the policy computed at step t-1.

The algorithm terminates after k iterations, or if $|V_t^\pi(s) - V_{t-1}^\pi(s)| < \epsilon$ for each state s. The policy is then built using the greedy policy with respect to the Value Function returned by Value Iteration. Value Iteration always converges to the optimal Value Function. Complexity is $\mathbf{O}(|S| \cdot |A|)$ per iteration, while the convergence rate is linear.

Algorithm 1 ValueIteration

```
 **Input:** MDP problem
 **Output:** Value function V
$V_1$(s) = 0 for all $s \in S$;
$t = 0$;
**repeat**
    $t = t + 1$;
    **for** $(s \in S)$ **do**
        **for** $(a \in A)$ **do**
            $Q_t^a(\mathrm{s}) = R(s,a) + \gamma \sum_{s' \in s} T(s,a,s') V_{t-1}(s')$;
        $V_t(\mathrm{s}) = \max_a Q_t^a(\mathrm{s})$;
**until** $|V_t(s) - V_{t-1}(s)| < \epsilon$
```

2.3 Partially Observable Markov Decision Processes (POMDPs)

The description of MDPs in Section 2.2 assumed that the environment was fully observable. with this assumption, the agent always knows its current state. When the environment is only partially observable, the agent does not necessarily know the real current state, thus it is not possible to use a policy $\pi(s)$ that only considers specific states. *Partially Observable Markov Decision Proces, (POMDP)* is an extension of MDP used to model partially observable environments.

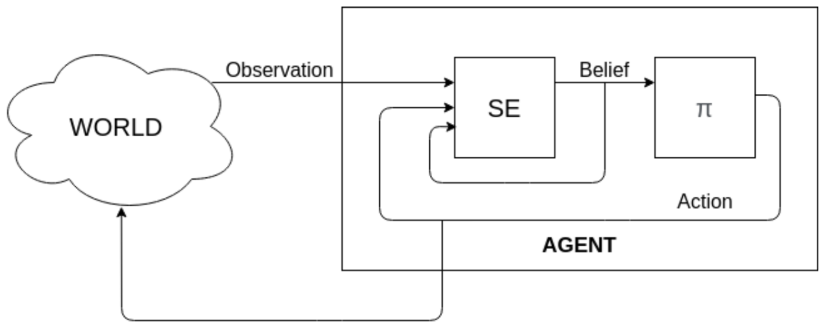


Figure 2.3: A time slice of the POMDP model

Definition 11 *A POMDP is defined as a tuple* $\langle S,A,O,T,\Omega,R,\gamma\rangle$ *where*

- S *is a finite set of states.*
- A *is a finite set of actions.*
- O *is a finite set of observations.*
- T $: S \times A \rightarrow \Pi(S)$ *is the transition model.*
- Ω $: S \times A \rightarrow \Pi(O)$ *is the observation function.*
- R $:S \times A$ *is the reward function.*

- $\gamma \in [0, 1)$ *is the discount factor that weighs the importance of current and future rewards.*

The observation function given a state and an action, returns the probability distribution of possible observations, while the reward function specifies the immediate reward for each state-action pair.

Note that *S, A, T, R,* γ is the tuple that defines an MDP, while *O,* Ω define a new set and a new function that allow to extend the MDP with uncertainty in POMDP.

There are three main problems. First, a POMDP may have a large number of states. Second, as the state is not fully observable, the agent must reason with beliefs, and the size of the belief space grows exponentially with the number of states. Finally, the number of action-observation histories that must be considered for POMDP planning grows exponentially with the planning horizon. A history $h_t = (a_0, z_0, a_1, z_1, ..., a_t, z_t)$ is a vector containing the path of all past observations and taken actions a belief node at depth t. The first two difficulties are usually referred to as the *"curse of dimensionality"*, and the *"curse of history"*respectively.

For these reasons, working with complete policies is not applicable in practice and computing an optimal solution for POMDPs with finite horizons is very hard (i.e., PSPACE-complete [8]). A key idea of this framework is to consider all possible configurations of the (partially unknown) states of the agent in the environment, and to assign to each of these states a probability value indicating the likelihood that the state is the true state. All these probabilities together form a probability distribution over states which is called **belief**, a rapresentation of the set of all possible world states that it might be in.

A policy $\pi : B \rightarrow A$ specifies which action $a \in A$ to select given a belief $b \in B$.

To find the optimal policy, reinforcement learning methods use the estimation of two functions:

- $V(s)$ called *value function* or also *V-function*, which indicates the expected return that can be achieved from state s.

- $Q(s,a)$ called *action-value function* or also *Q-function*, which indicates how much benefit it brings to the agent to perform the action a in the state s (evaluation function of an action performed in a state).

Our goal is to find an optimal policy π^* that maximize the discounted return. In principle, the optimal policy is not necessarily unique. These functions have the property of defining a partial ordering between policies, i.e.:

Definition 12 $\pi \geqslant \pi^{'} \Leftrightarrow V_{\pi}(s) \geqslant V_{\pi'}(s)$ *for all* $s \in S$

The optimal policy π^* will be such that $\forall s \in S$:

Definition 13 $V_{\pi*}(s) \geqslant V_{\pi}(s)\ \forall \pi \in \Pi$

2.4 Partially Observable Monte Carlo Planning (POMCP)

Calculating exact policies for POMDP is very difficult, we consider *Partially Observable Monte Carlo Planning (POMCP)* [11] that is a powerful online algorithm able to generate approximate policies for large Partially Observable Markov Decision Processes. The online nature of this method supports scalability by avoiding complete policy representation. It combines a Monte-Carlo update of the agent's belief state with a Monte-Carlo tree search from the current belief state. It has three properties:

- Monte-Carlo sampling is used to break *the curse of dimensionality* both during belief state updates and during planning.
- Only a black box simulator of the POMDP is required, rather than explicit probability distributions.
- The online nature of this method supports scalability by avoiding complete policy representation.

These properties enable POMCP to plan effectively in large POMDP instances.

Search algorithm builds online a search tree of histories. A *online approach*, tries to find a good local policy for the current belief state of the agent (i.e., it does not compute a complete representation of the policy, it only builds a fragment of the policy that can be used in the current belief). The advantage of such an approach is that it only needs to consider belief states that are reachable from the current belief state. This focuses computation on a small set of beliefs instead of *complete approaches* that return complete policies that can map each belief to a proper action. In some cases, online approaches may require a few extra execution steps (and online planning), since the policy is locally constructed and therefore not always optimal. However the policy construction time is often substantially shorter. Consequently, the overall time for the policy construction and execution is normally shorter for online approaches [9].

Each node of the search tree estimates the value of a history by Monte-Carlo simulation (see Figure 2.4). For each simulation, the start state is sampled from the current belief state, and state transitions and observations are sampled from a black box simulator.

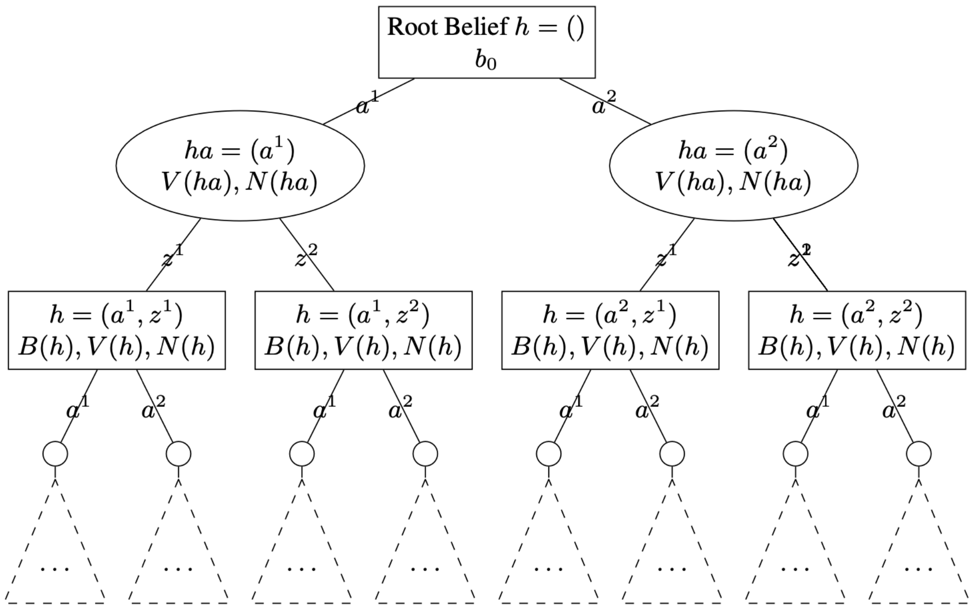


Figure 2.4: Example of a belief tree

POMCP consists of a *UCT search* that selects actions at each time-step; and a *particle filter* that updates the agent's belief state.

UCT uses the *Upper Confidence Bound*, UCB formula, to compute the value of a history followed by an action:

$$V^{+}(h, a) = V(h, a) + c\sqrt{\frac{\log N(h)}{N(h, a}} \tag{2.1}$$

In the second phase, simulation, a rollout policy is used (e.g random uniform policy). After simulation one new node, corresponding the the first history encountered during simulation is added to the tree. When the Backpropagation phase starts, the belief for each state needs to be updated. However, if the state space is large, it is too computationally expensive. One way to solve this issue is to use a particle filter to represent the probability distribution and so to update the belief. Partially Observable Monte Carlo Tree Search uses an unweighted particle filter, because it can be implemented using a black box simulator. The belief of state s, given history h_t, is the sum of $\hat{B}(s, h_t) = \frac{1}{K}\sum_{i=1}^{K} \delta_{sB_t^i}$, where $\delta_{ss'}$ is the kronecker delta function. At the start of the algorithm, K particles are sampled from the initial state distribution. After a real action a_t is executed, and a real observation o_t is observed, the particles are updated by Monte-Carlo simulation.

A state s is sampled from the current belief state $\hat{B}(s, h_t)$, by selecting a particle a_t random from B_t. This particle is passed into the black box simulator, to give a successor state s' and observation o', $(s', o', r) \sim G(s, a_t)$. If the sample observation matches the real observation, $o = o_t$, then a new particle s' is added to B_{t+1}. This process repeats until K particles have been added.

2.4.1 Monte Carlo Tree Search (MCTS)

The Monte Carlo methods learn value functions and optimal policies by sampling random states from its belief and by running episodes from these states using the black box simulator. This gives them advantages over Decision Process methods:

- They can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics.
- They can be used with simulation or sample models. For surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods.
- It is easy and efficient to focus Monte Carlo methods on a small subset of the states.

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At its base, MCTS is a rollout algorithm, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. MCTS has proved to be effective in a wide variety of competitive settings, including for single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation. MCTS is executed after encountering each new state to select the agent's action for that state; it is executed again to select the action for the next state, and so on. As in a rollout algorithm, each execution is an iterative process that simulates many trajectories starting from the current state and running to a terminal state (or until discounting makes any further reward negligible as a contribution to the return). The core idea of MCTS is to successively focus multiple simulations starting at the current state by extending the initial portions of trajectories that have received high evaluations from earlier simulations. MCTS does not have to retain approximate value functions or policies from one action selection to the next, though in many implementations it retains selected action values likely to be useful for its next execution. As in any Monte Carlo method, the value of a state–action pair is estimated as the average of the (simulated) returns from that pair. MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories. Any simulated trajectory will pass through the tree and then exit it at some leaf node. Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree something better is possible. For these states we have value estimates for of at least some of the actions, so we can pick among them using an informed policy, called the tree policy, that balances exploration and exploitation. In more detail, each iteration of a basic version of MCTS consists of the following four steps as illustrated in Figure 2.5.

1. *Selection.* Starting at the root node, a tree policy based on the action values attached to the edges of the tree traverses the tree to select a leaf node.

2. *Expansion.* On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.

3. *Simulation.* From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.

4. *Backup.* The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree.

MCTS continues executing these four steps, starting each time at the tree's root node, until no more time is left, or some other computational resource is exhausted. Then, finally, an action from the root node (which still represents the current state of the environment) is selected according to some mechanism that depends on the accumulated statistics in the tree. After the environment transitions to a new state, MCTS is run again and so on. [13]

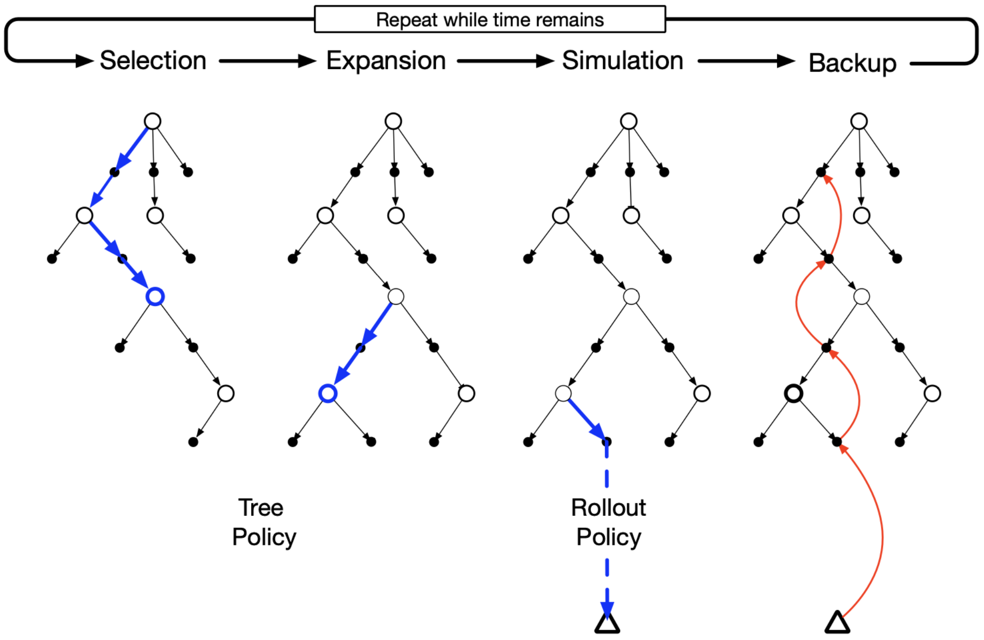


Figure 2.5: Show how Monte Carlo Tree Search work

2.5 ρ-POMDPs

POMDPs, described in Section 2.3, model sequential decision-making problems under uncertainty and partial observability. Unfortunately, some problems cannot be modeled with state-dependent reward functions, e.g., problems whose objective explicitly implies reducing the uncertainty on the state. ρ-POMDPs is an extension of POMDPs where the reward function ρ depends on the belief state (*belief-dependent*) [1]. This allows to define not only control oriented objectives, but also information-oriented ones.

All the domains that deal with partial observability have to overcome the issue of collecting information to achieve their goal. This problem is usually implicitly addressed in the resolution process, where acquiring information is only a means for optimizing an expected reward based on the system state. Some active sensing problems can be modeled this way (e.g., active classification), but not all of them. A special kind of problem arise when the performance criterion incorporates an explicit measure of the agent's knowledge about the system, which is based on the *beliefs* rather than states. We consider a way of defining rewards based on the acquired knowledge represented by belief states.

An important property used to solve normal POMDPs is the result that a belief-based value function is convex, because $R(s, a)$ is linear with respect to the belief, and the expectation, sum and max operators preserve this property. For ρ-POMDPs, this property also holds if the reward function $\rho(b, a)$ is convex. This approach is optimal if ρ and V_0 are convex functions over Δ, as shown in [1].

2.6 ρ-POMCP

POMCP cannot be used to solve ρ-POMDPs. POMCP generates trajectories that follow a single sequence of states and samples associated rewards. It collects visited states in belief nodes, which are not sufficient to correctly estimate the belief and thus belief-dependent rewards.

At high level, ρ-POMCP is similar to POMCP algorithm [14]. During the selection step, trajectories are generated by sampling states and observations using the generative model G (i.e., the black box). When a belief node h is visited, it collects the state that led to that node to generate an estimate of the true belief state $b(h)$. As shown in Figure 2.6, applying POMCP directly by adding only the current state of the trajectory to the belief node, as proposed in the original ρ-POMCP algorithm, may result in poorly estimated immediate rewards during the initial step of the algorithm, causing the MCTS algorithm to inefficiently spend time on branches with overestimated rewards and slowing down the exploration of branches with underestimated rewards. To add more states at each visit to a belief node we consider in ρ-POMCP(β) , which uses the particle filter explicitly. Thus, when selecting the action a from the trajectory state s_t and sampling the observation z_{t+1}, at each transition the algorithm generates a set β_{t+1} of $|\beta|$ states, called a *small bag of particles*, and uses G to make the $|\beta|$ samples consistent with the observation z_{t+1}. It then adds this small bag β_{t+1} to $B(haz)$, the cumulative bag for the history haz, stored in the corresponding belief node: $B(haz) \leftarrow B(haz) \cup \beta_{t+1}$. Note that ρ-POMCP(0) consists in adding, at each visited belief node, only the state of the trajectory that led to this node, and thus corresponds to the initial ρ-POMCP algorithm.

In the simulation step, belief estimates are required at least for estimating immediate rewards and possibly for decision making. Thus, the sequence of small bag β must be maintained. Finally, during the back-propagation step, the cumulative bag $B(h)$ of belief node h is used to estimate the true belief state $b(h)$ by normalizing the weights and computing $\rho(B(h))$.

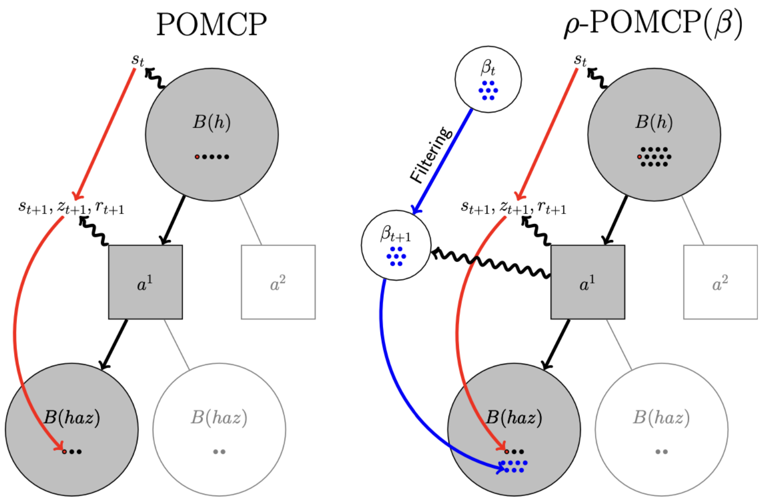


Figure 2.6: Difference between POMCP and ρ-POMCP(β) (Monte Carlo Information-Oriented Planning)

Chapter 3

Methods

In this chapter we present some of the important contribution of this thesis. In Section 3.1 we present how POMCP was extended to handle ρ-POMDP problems. To extend the rule-based framework we introduce the concept of bag of beliefs. Through the use of bags, we can improve trace generation by also collecting beliefs in future states that are not explored. In Section 3.2 we show how to creating extended traces to generate more precise rules using the XPOMCP algorithm.

3.1 Extending POMCP to Handle ρ-POMDPs

An extension of POMCP that handles ρ-POMDPs is presented in [14]. We present a description of the algorithm in Section 2.6. An implementation of ρ-POMCP, written in java, was presented alongside the original paper. In this thesis, we adapt this implementation to the original implementation of POMCP, presented in [11] and written in c++, to integrate ρ-POMCP with the rule-generation algorithm presented in [6].

One of the main contribution is the introduction of the `BAG` class. It extends the concept of belief to handle hypothetical beliefs in future nodes of the Monte Carlo tree. We decribe how to initialize and update these bags, and we describe

how the algorithm that handle the MCTS has to be extended to handle this new data structure. Finally, we modified the rollout mechanism used to build a rough estimation of a state that was never visited before. I made an open source implementation available at
<`https://github.com/Nereo1996/rhopomcp-with-trace-generation`>.

3.1.1 Bag

The *Bag* class can be considered as an extension of the concept of *belief state*. In the original implementation of POMCP, the belief is a set of particles, each one of them represent a possible state of the system. In this class, a very useful information related to each particle is added which is the concept of *weight*, which allows to provide an estimate that specific particle corresponds to the true state.

Giving a weight to the particles has a central role in the algorithm as many functions are based on this. Specifically, it is use to including updating with importance sampling that would be very expensive to implement otherwise. To do this a *map* is used which maps states to weights. Some functions are provided, but in particular the one that allows adding a particle to the *cumulative bag of particles* and the one that allows adding an entire bag . They are used a lot, for creating new bags by adding particles to them, or for adding whole bags to a cumulative bag allowing to better manage the concept of weight. The weights are not stored normalized in $B(h)$, so new weighted particles can be added without introducing bias.

A function for visualization is also provided.

3.1.2 Bag Initialization

The Algorithm 2 show the procedure to generate an initial bag.

Algorithm 2 generateInitialBag

```
Input: State $s$, bag $initialBelief$.
Output: The generated bag.
BAG $b$;
b.AddSample(s);
for 0 to $n$ do
    STATE sampleState = initialBelief.CreateSample();
    b.AddSample(sampleState);
```

The initial bag is created by adding the initial state and another n particles created from the *generative model*, where n is passed from the command line by the user with the *bagsize* parameter. In this case, as the initial bag is populated, the weight of each particles is set to 1 by default.

3.1.3 How to Generate Bags in the Other Steps

To generate a bag given the current cumulative bag, as explained in Section 2.6, the *CreateBag_ beta* fuction is used. The Algorithm 3 explains that function. The goal of this function is to build future beliefs to estimate the reward, to do so, we must evolve the belief respecting actions and observations.

Algorithm 3 CreateBag_beta

```
Input: State *previous*, action *action*, observation *obs*, bag *bag*, state *next*.
Output: The generated bag.
BAG generatedBag;
for i=0 to $n$ do
    STATE state= bag.CreateSample();
    Simulator.Step(state, action, obs_step);
    prob= Simulator.ProbObs(obs_step, previous_s, action, state_s);
    if prob $>$ 0 then
        generatedBag.AddSample(Simulator, state, prob);
probability = Simulator.ProbObs(obs, previous, action, next);
generatedBag.AddSample(next, prob);
```

After performing action *action* and receiving observation *obs*, a new small bag β_{t+1} is generated from the previous bag β_t by using the *generative model* and following these steps:

1. Sample *state* from the previous bag.

2. Sample a state by using the generative model. This is implemented through the step function that, passing the state by reference, updates it internally.

3. Store this particle s in the new bag β_{t+1} with an associated weight of $p(o_{t+1}|s, a_t, s')$ corresponding to the probability of having generated observation o_{t+1} by executing the action a_t from state s to state s'.

4. Repeat from step 1 n times, where n is the size of a small bag of particles.

Finally we add to the new bag the true action and observation performed.

3.1.4 Search in ρ-POMCP

Search in ρ-POMCP is a method that allows to find the best action in the current belief state. It is implemented via two main recursive functions: *SimulateV_rho* show in Algorithm 4 and *SimulateQ_rho* show in Algorithm 5.

Algorithm 4 SimulateV_rho

```
  Input: State state, VNODE vnode, bag beta.
  Output: totalReward.
  a = GreedyUCB(vnode);
  PeakTreeDepth = TreeDepth;
  if TreeDepth $\geq$ MaxDepth then
      return 0;
  if TreeDepth $\geq$ 1 then
      AddSample(vnode, state);
  QNODE qnode = vnode$\rightarrow$Child(action);
  totalReward = SimulateQ_rho(state, qnode, a, beta, vnode$\rightarrow$Beliefs());
```

SimulateV_rho is the function that samples an action from all possible actions available in the current state. This choice is made using UCT, as explained in Section 2.4 . This heuristich guarantees a balance between exploration of new actions and exploitation of already available information on the expected return of a certain action.

A big difference from the original function is insert to the cumulative bag of the node the state in all levels after the root (in POMCP algorithm they were only added to the first level below the root). Once the action is sampled, we call the SimulateQ function (as presented in Algorithm 5), which is responsible of sampling the observations.

Algorithm 5 SimulateQ_rho

```
Input: State $state$, QNODE $qnode$, action action, bag $beta$, bag $prevbeta$.
Output: totalReward.
term = Simulator.Step(state, action, observation, immediateReward);
History.Add(action, observation);
VNODE vnode = qnode.Child(observation);
if !vnode & !term & qnode.Value.GetCount() $\geq$ Params.ExpandCount then
    vnode = ExpandNode(state);
    BAG bprime = CreateBag_beta(prev_state, action, obs, beta, state);
if TreeDepth $\geq 1$ then
    prevbeta.AddSample(Simulator, beta);
if !term then
    TreeDepth++;
    if vnode then
        delayedR = SimulateV_rho(state, vnode, bprime);
    else
        delayedR = Rho_Rollout(state, bprime);
    TreeDepth–;
immediateR = Simulator.Rho_reward(prevbeta, action);
totalReward = immediateR + Simulator.GetDiscount() $\cdot$ delayedR;
```

Specifically, to sample the observation, we use the step function. With the action sampled in SimulateV_rho and with the observation just selected, we go one step down the tree to the next node. If this node has not already been visited and is not a terminal state, it is instantiated (with the function *ExpandNode*). After that, a bag β_{t+1} is created from the previous cumulative bag. On all layers below the root, the bag β is added to the cumulative bag of the previous node($B(h)$). If the next state is non-terminal and the maximum tree expansion has not yet been reached, the SimulateV_rho function is called to simulate the next step. Otherwise, the Rho_Rollout function is called (which will be discussed in Section 3.1.5).

Finally, when the two recursive functions have finished simulating, during backpropagation the *ImmediateReward* and consequently the *DelayedReward* are calculated.

3.1.5 ρ-Rollout Algorithm

Rho_Rollout is a method for evaluating a state. A state *s* is evaluated by average in the discounted return obtained by performing N simulations from state *s*. Algorithm 6 is used once the simulation leaves the scope of the search tree for estimate the leaf nodes.

Algorithm 6 Rho_Rollout

```
  Input: State state, bag beta.
  Output: totalReward.
totalReward = 0.0;
discount = 1.0;
term = false;
for (i=0; i+TreeDepth < Params.MaxDepth & !term; i++) do
    STATE prev_state = Simulator.Copy(state);
    act = Simulator.SelectRandom(state, History, Status);
    term = Simulator.Step(state, act, obs, reward);
    History.Add(act, obs);
    BAG bprime = CreateBag_beta(prev_state, act, obs, beta, state);
    reward = Simulator.Rho_reward(beta, act);
    totalReward += reward * discount;
    discount *= Simulator.GetDiscount();
    beta.Move(betaprime, Simulator);
```

After initializing the variables, the core of the algorithm is the for loop where it cycles until it reaches the maximum depth of the search tree or the simulation reaches a terminal state. In this cycle an action is sampled using the *SelectRandom* method. It uses domain knowledge to select actions stochastically during rollouts. Through the method Steps a simulation step is performed. Then a bag β_{t+1} is created, which will not be added like in the previous Section 3.1.4 to the cumulative bag, but it will be used only as a temporary cumulative bag for this simulation. The function compute the reward, and it calculate the average reward over the total number of simulations N.

3.2 Generation of Extended Traces

Another contribution of this thesis is the implementation of a method for generating extended traces. A *class XES* is used to generate XML traces in the c++ implementation. We base this extension on Extensible Event Stream (XES), a standard that standardize a language to transport, store, and exchange event data (e.g., for process mining).

For generating traces with a depth that POMCP could reach we use this algorithm:

```
Simulator.log_beliefs(Root->Beliefs());
Simulator.log_action(action);
Simulator.log_observation(observation);
Simulator.log_reward(reward);
XES::logger().start_list("Valuable actions");
for a=0 to NumberofAction:
    Simulator.log_action(a);
    XES::logger().add_attribute({"Q Value", Root->Child(a).GetValue()});
```

With this procedure you can generate a trace where you show:

- Belief of the node Root.
- Action taken.

- Perceived observation.
- Reward obtained.
- List of values of all possible actions on which the best action was chosen.

Using the ρ-POMCP algorithm, however, we get more information and and we use them to extend the traces. Due to the fact that the bags are move down in the search tree even in levels lower than the first one, it is possible to generate traces even for deeper levels. The procedure that was used for this purpose is shown below:

```
for a=0 to NumberofAction:
    for o=0 to NumberofObservation:
        if NextNode s' (node after performing action a and get observation o)
                Simulator.log_beliefs(s'->Beliefs(), a, o);
                Simulator.log_action(action);
                Simulator.log_observation(observation);
                Simulator.log_reward(reward);
                XES::logger().start_list("Valuable actions");
                for ac=0 to NumberofAction:
                    Simulator.log_action(ac);
                    XES::logger().add_attribute({"Q Value", s'->Child(ac).GetValue()});
                XES::logger().end_list();
```

With this procedure, an extended trace can be generated where the following is shown in addition to the previous information; For each node_haz under the root node:

- Belief of the next node.
- Action that will be taken.
- Observation that will be perceive.
- Reward that will be obtain.

- List of values of all possible actions starting from the next node.

Chapter 4

Rule Synthesis for ρ-POMCP

Explainability [4] is becoming a key feature of artificial intelligence systems since in several contexts humans need to understand the reason behind specific decisions taken by an autonomous agent. The presence of erroneous behaviors in methodologies (due, for instance, to the wrong setup of internal parameters) may have a strong impact on autonomous cyber-physical and robotic systems that interact with humans, and detecting these errors in automatically generated policies is very hard in practice. For this reason, improving policy explanability is fundamental. POMCP's online nature of avoiding a complete policy representation leads to a lack of interpretability of its decisions. Thus, POMCP acts as a black-box. In this thesis, we use EXplainable POMCP (XPOMCP) [6], which introduces a novel way of interpreting POMCP policies, and we extend its usage to the case of ρ-POMDPs. It generates a compact logic-based representation of the system's behavior called a *rule* as shown in Figure 4.1 (image taken from [6]). The rule provides a human-readable local representation of the policy behavior that incorporates the prior knowledge specified by the expert. Rules are generated by combining logical formulas with traces generated by a POMCP agent. In particular, XPOMCP is able to find unexpected behaviors, i.e., anomalies, which are decisions made by the policy that violate the high-level intuition provided by the expert with the rule template. In Section 4.1, we intro-

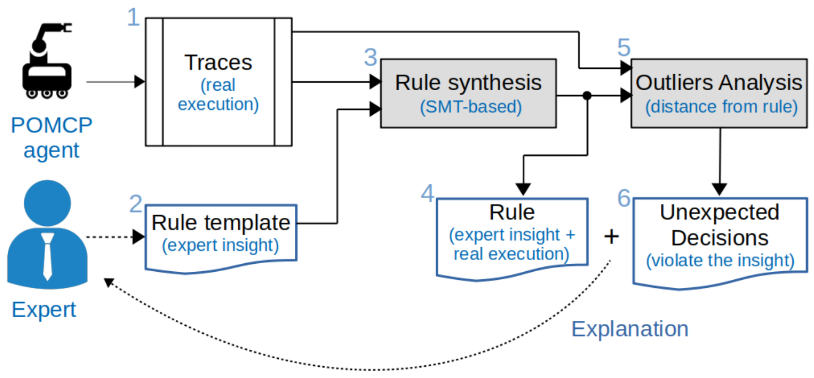


Figure 4.1: XPOMCP overview.

duce *rule templates* that can be used to explain the expected relation between beliefs and actions. While in Section 4.2 we define the algorithm that implement rule synthesis. Finally, in Section 4.3 we show how to extend XPOMCP to handle ρ-POMDPs.

4.1 Rule Template Creation

A Rule template is a logical formula with free variables. It defines a relationship between some properties of the belief (e.g., the probability to be in a specific state) and an action.

A rule template represents the question the expert wants to investigate. It is a set of first-order logic formulas without quantifiers explaining some properties of the policy, and has the following form:

$$r_1\texttt{:}\quad \texttt{select } a_1 \texttt{ when } \texttt{(}\bigvee_{i^1} subformula_{i^1}\texttt{);} \tag{4.1}$$

$$r_n\texttt{:}\quad \texttt{select } a_n \texttt{ when } \texttt{(}\bigvee_{i^n} subformula_{i^n}\texttt{);} \tag{4.2}$$

$$\texttt{[where } \bigwedge_j \ (requirements_j)\texttt{;]} \tag{4.3}$$

$$\tag{4.4}$$

where $r_1, ..., r_n$ are action rule, a is an action choosen among all the actions in the action space, and *subformula* represents some other logic formula connected in logic OR. The template is then combined with the execution traces and encoded as a *MAX-SMT* problem. MAX-SMT problems have two kinds of constraint: hard and soft constraints. Hard means that the clauses must be satisfied. These are used to define prior knowledge on the domain which is used by the rule synthesis algorithm (explained in Section 4.2) to compute optimal parameter values. [where $\bigwedge_j$ $(requirements_j)$] in Definition 4.4 is an *hard constraint*.

Soft means that the clauses can be satisfied, and we are interestend in the solution that satisfies the highest number of soft clauses. [r_1: select a_1 when $(\bigvee_{i^1} subformula_{i^1})$] in Definition 4.4 is an *soft constraint* and is used to explain which action the agent should take when the conditions expressed in logic formula are met. A model of the MAX-SMT problem hence satisfies all the hard clauses and as many soft clauses as possible, and it is unsatisfiable only when hard clauses are unsatisfiable. A subformula is defined as $\bigwedge_k p_s \approx \overline{x}_k$, where p_s is the probability of state s, symbol $\approx$ represents one arithmetic connector choosen among the set of possible connectors $<, >, \geq, \leq$, and $\overline{x}_k$ is a free variable that is automatically instantiated by the SMT solver analyzing the traces (when the problem is satisfiable).

4.2 Rule Synthesis

When the rule template is ready, hence the expert is satisfied with question she/he wants to pose, the next step is assigning a value to the free variables to generate a rule that explains as many of the decisions taken by the policy as possible. The process of finding an optimal assignment for the free variables in the formula is called *rule synthesis.* In Algorithm 7 we show how rule synthesis works.

Algorithm 7 RuleSynthesis

```
    Input: a trace generated by POMCP $ex$ a rule template $r$
    Output: an instantiation of $r$
 1: $solver \leftarrow$ probability constraints for thresholds in $r$;
 2: for each $action\ rule$ $r_a$ with $a \in A$ do
 3:     for each step $t$ in $ex$ do
 4:         build new dummy literal $l_{a,t}$ ;
 5:         $cost \leftarrow cost \cup l_{a,t}$;
 6:         compute $p_0^t ... p_n^t$ from $t.particles$;
 7:         $r_{a,t} \leftarrow$ instantiate rule $r_a$ using $p_0^t ... p_n^t$;
 8:         if $t.action \neq a$ then
 9:             $r_{a,t} \leftarrow \neg(r_{a,t})$;
10:         $solver.add(l_{a,t} \vee r_{a,t})$;
11: $solver.minimize(cost)$;
12: $goodness \leftarrow 1-$ $distance\_to\_observed\_boundary$;
13: $model \leftarrow solver.maximize(goodness)$;
    return: $model$
```

First the solver is initialized, and all the hard constraints are added. Then, the first for loop iterates over the actions analyzed in the rule template. Infact a rule template can contain many actions to analyze, and therefore, many rules, one rule template per action. While, the second for loop iterates over the steps of the trace. For each step of the trace:

1. Build a dummy literal to satisfay artificially the unexplainable step, thus we can converge to a solution even if the rule cannot explain everything.
2. Add to the cost function the dummy literal for counting the number of fake assignments. We need to use a cost function to tell the solver that the goal is satisfying as many steps as possible using the free variables of the rule template, not using dummy literals.
3. Computes the belief values for the states from the particles in the trace.
4. Assign the belief for that step to the rule template, substituting the probability in the rule template with actual values computed from the trace.
5. Check that if the step action is different than the action which we are iterating over, the rule should not be satisfied (i.e., we want an if and only if relation between beliefs and actions), so the rule template is negated.
6. Literal and the rule is added to the solver connected by an OR, so that if the rule is unsatisfiable, the literal is set to true, to make step satisfied.

After the loops ends, the solver is called to minimize the cost function, which tries to satisfy as many rules as possible. The goodness of the model is defined as one minus the distance to the observed boundary, that we want to maximize to build a rule that is as close as possible to the beliefs observed in the trace. And finally, the solver is called to maximize, the goodness. Finally, the algorithm return a model that contains a proper assignement for all the free variables of the rule template.

4.3 Extending XPOMCP to Handle $\rho-$POMDPs

XPOMCP lends itself particularly well to handling ρ-POMDPs because it works with beliefs, and so the description it builds, based on beliefs, can be adapted directly. In Section 3.2, we explain how to build a trace from a bag in the ρ-POMCP implementation. In addition, thanks to the fact that in ρ-POMCP

we have many more beliefs because we can extend them to the next levels, we can extend the trace and improve the description. Original POMCP simulates one particle at a time, thus belief only exists in the root, greatly limiting the creation of traces. By extending the concept of belief to the concept of Bag in ρ-POMCP, particles are carried forward in the tree in order to make future estimates. With this additional information, it is possible to create more extensive traces. Specifically, it is possible to visit, and consequently keep track of action/observation pairs that do not occur (cause they were not selected), look at the ranking of the actions, and associate this hypothetical belief with the highest ranking action. This process can be repeated at each step, thus we could track the belief estimate and rank of each subsequent action.
Finally, ρ-POMDP traces can be used directly because they already have the belief-action pair. In POMCP problems XPOMCP works well because the reward is indirectly influenced by the belief but in ρ-POMCP this problem works even better because the belief directly influences the reward, so the explanation provided by the rules is even more consistent and direct.

Chapter 5

Evaluation

In this section, we show the experimental results of the proposed methodologies, namely the combination of trace generation for XPOMCP with ρ-POMCP, and the use of extended traces to speed up rule generation. In Section 5.1, we describe the two application domains considered, tiger and rocksample . In Section 5.2, we discuss the results in tiger, a well-known problem that can be solved in an optimal way, to test how well the rule generation system works. Finally in Section 5.3, we discuss the rocksample problem. In this, we show that rules can help to understand the operation of the generated policy, we also show that thanks to extended traces it is possible to improve rule synthesis significantly.

5.1 Application Domain

5.1.1 Tiger

Tiger is a standard problem in POMDP planning [5]. In this problem, the agent is placed in front of two closed doors. Behind one door there is a treasure, while the other hides a tiger. The goal of the agent is to find the treasure while avoiding the tiger. This problem can be easily formalized with a POMDP, that contains two states, the first state where the tiger is on the right while the second state where the tiger is on the left. The agent has three possible actions: *Open_ Right*, *Open_ Left* and *Listen* and it has three observations available: *Roar_ Right*, *Roar_ Left* and *Obs_ None*. Opening one of the two doors (*Open_ Right* or *Open_ Left*) yields a *positive reward* if the door hides a treasure (in our implementation, reward=+10) otherwise it yields a *negative reward* if the agent finds the tiger (in our implementation, reward=-100). These two actions always return a "none" observation. With the listen action (*Listen*), the agent waits and listen for a roar, to estimate what is the real position of the tiger. This action yields *negative reward* (i.e., reward=-1), which is that, while being negative, is significantly lower than the penalty of finding the tiger. The agent's perception however is not 100% accurate. Specifically, there is a 15% chance of hearing a roar comming from the wrong door. Thus, the agent does not have absolute certainty of where the tiger is. Through this action the agent will receive a *Roar_ Right* or *Roar_ Left* observation based on where they heard the roar. the episode ends when the agent decides to open a door and receives a reward. It is possible to solve this POMDP in an optimal way, computing an optimal policy that always does the best thing.

5.1.2 ρ-Tiger

ρ-Tiger is an extension of Tiger that reinterprent the model as a ρ-POMDP problem. Thus, the reward achieved by a policy it is not only related to the real state of the system, but also to the belief of the agent. To extend it, two functions are used: *ρ_ reward* and *ObservationProbability*.

The Rho_Reward function, which extends the original reward function, allows to estimate the expected return of a certain belief. This is implemented by the problem adding up, for every particle of the bag, the probability to end in that specific particle (i.e., the weight of that specific particle) multiplied to the reward that would be received if the simulation were to evolve in that specific state. If the estimate of the the belief is good we will find ourselves in the situa-

tion in which the negative rewards due to not optimal actions will be multiplied for low probabilities, while to the contrary, positive reward will be multiplied for higher probabilities, impacting more on the final reward. Thus, we can add an information-oriented goal to the problem.
ProbObs function, allows to compute the probability of receiving a specific observation given the current state, a possible next state and the action selected by the agent. Specifically, if the action selected by the agent is a *Open_ Right* or *Open_ Left* we return a probability of 0.5, while in the case the action is *Listen* we return a probability of 0.15 or 0.85 depending on whether the observation taken was correct or not. This function is used to update the bag of particle creating a new particle bag to estimate the probability of evolving in that specific case.

5.1.3 Rocksample

The *rocksample* problem was originally presented by Smith and Simmons (2004) [12]. In this domain, an agent must explore the environment, sample valuable rocks, and exit. The environment can be viewed as an $n \times n$ grid that contains m rocks, where $n, m \in \mathbb{N}$. The rocks may or may not be valuable, but this is not known in advance by the agent, however, the position of the rock is fixed an known a priori. The initial belief is a uniform distribution among the possible states of the problem, in this problem then we have 2^8 possible states (every possible combination of rocks). The agent has 4 actions for movement (move on the north, on the south, on the east and on the west), one action to sample a rock(*E_ SAMPLE*), and n actions (one for each rock) to check if the specific rock is valuable or not (e.g., "check 1" for measuring the value of the first rock). There are three observations available: the "none" observation, the "good" observation and the "bad" observation. The four movement actions are deterministic and simulate the movement of the agent within the environment. They always return an observation always "none". Doing this type of action does not generate any reward, but increasing the number of actions performed by the agent in the simulation will gradually decrease the final discounted reward while a discount factor γ lower than one is used. The E_SAMPLE action is used to sample the rock at the current location of the agent and yields observation always OBS_NONE. This action will simulate the sampling of a rock (removing it from the available rocks) and will generate a positive reward (i.e., reward=+10) if the rock is valuable or a negative reward (usually reward=-10) if the rock is not valuable. Each E_CHECK action of the rock returns a noisy observation for rock i. if the agent thinks the rock is valuable return a OBS_GOOD observation, OBS_BAD instead. The accuracy of the sensor is measured by the distance between the agent and the rock it would like to check and is given by the formula:

$$efficiency = (1 + 2^{\frac{-d}{h}}) \cdot 0.5 \tag{5.1}$$

Where d is the *Euclidean distance* and h is the *half efficiency distance* The check action does not generate reward, but is used to improve the agent's knowledge of the world. The agent's goal is to dig as many valuable rocks as possible before exiting the environment to the east side.

5.1.4 ρ-Rocksample With Parameterization

We implemented an extension of the Rocksample problem to ρ-POMDP. This extension is similar to the one proposed in Section 5.1.2, adapted to the specific problem.

In addition, we extended this problem so that we can change some parameters from the command line (in particular rewards), in order to facilitate experiments launched afterwards. We created a struct that allows to modify:

- Movement rewards.
- Rocks rewards.
- Half efficiency distance.

- Reward range.

The parameters of reward movement (*rNorth*, *rSouth*, *rWest*, *rEast*) are used to set the rewards for exiting the environment in the four directions. We used *rValuable* and *rNotValuable* to set the rewards of the rocks in case they are valuable or not while *rAlreadySampled* indicates the reward in case the agent samples a rock he has already dug before.
HalfEfficiencyDistance is used to modify measure how fast the quality of a check decreases while the distance between the rock and the agent increases implemented by the equations described in equation 5.1.
RewardRange is used to modify the accuracy of the agent. These parameters allow to test the Rocksample domain under different circumstances.

In particular it was necessary to intervene on the functions *GenerateLegal* and *GeneratePreferred*. These two functions allow, during the simulation, to select smarter actions in order to guide the agents towards more effective behaviours. These functions can be invoked from command line through the *TreeKnowledge* parameter, that can be set with three possible configurations:

- *Pure*: Is the default value. This does not introduces any bias on the action, thus the algorithm must asses the quality of each action individually. This could lead to poor performances, expecially if there are many actions that are known to be suboptimal in a given belief.

- *Legal*: At this level, the algorithm considers only legal (i.e., not prohibited) actions in the current belief. Actions that cannot be performed (e.g., digging a rock in a location that does not contain rocks) are blocked, but no sorting is introduced on other actions. This can greatly reduce the search space, but is not always sufficient to achieve acceptable performance.

- *Smart*: At this level the algorithm not only filters legal actions, but it also introduces a bias for actions that are known to be effective in a certain belief. Specifically it sets the initial value for the expected return inside the tree, thus it rewards certain types of behaviors. This leads to an increase in performance, because the algorithm requires less particle to explore well known action, and it can focus its search toward unknown (but legal) actions. In the GeneratePreferred function then we had to intervene taking into account that there could be other directions in which the agent can go out to get non negative rewards. To be noted however that the initial value must be well tuned, otherwise it can be detrimental to the search (because it leads to overestimating some actions).

5.2 Results on Extended Tiger

To test the rule generation capabilities of the proposed approach, we consider the popular POMDP problem tiger, extended to consider information-oriented objective (i.e., ρ-tiger). This problem is not computationally hard. In fact, it possible to compute an exact solution for it using incremental pruning [2]. We build rules to describe the policy, and we compare it to the exact solution. The two solutions are the same. In this context, the use of extended traces does not provide any benefit, because the problem is easy to solve using standard traces. However, this is an important experiment to evaluate the rule synthesis capabilities of the methodology.

To further test the rule synthesis procedure, we analyzed $\rho-$Tiger with different values of reward range(c). The reward range is an important parameter of POMCP, becuase it regulates the balance between exploration and exploitation. In table 5.1 we see that higher Reward Range value not always corresponds to a better discounted/undiscounted return. This is because is a simple problem the benefit of exploration (achieved by using an higher value of c) is not as influential as other domain. However, a low value of c (i.e., 20) results in very low performance for the system.

For the results, we have used the following rule:

c	**Discounted return**	**Undiscounted Return**
20	-8.53	-8.74
50	2.02	2.95
80	4.22	5.75
110	3.24	4.56

Table 5.1: Discounted and Undiscounted return for $\rho-$Tiger.

```
declare-var $x_1, x_2, x_3, x_4$ $prob$;
declare-rule
    action $Listen$ $\iff$ $(p(right) \leq x_1 \wedge p(left) \leq x_2)$;
    action $Open_R$ $\iff$ $p(right) \geq x_3$;
    action $Open_L$ $\iff$ $p(left) \geq x_4$;
where $(x_1 = x_2) \wedge (x_3 = x_4) \wedge (x_3 > 0.9)$;
```

That describes when to listen (x_1) and when to open (x_3).

Table 5.2 shows the results obtained.

c	x_1	x_3	n_E	**total**
20	0.762881	0.918019	53	214
50	0.825119	0.958646	32	475
80	0.95794	0.962695	1	525
110	0.955195	0.955373	0	514

Table 5.2: result for ρ-Tiger

We note that as c increases, accuracy increases. The difference between Reward Range 80 and 110 is practically non-existent, 80 has only one error in 100 runs (which is correctly identified by the rule). The case with 20 has less belief, because the policy tends to open doors very quickly without listening (so many runs end sooner). We can notice that with $c = 80$ we generate a low number of error (only one, in our case) but a slightly higher undiscounted return. This happens because in some rare instances the policies decided to take an action that is, in general, too risky. However, in our example the agent was lucky during this risky action, thus it achieved an higher reward. The results were generated using the extended traces, but there was no substantive difference from non-extended traces.

5.3 Results on Extended Rocksample

We test the rule synthesi procedure also in $\rho-$POMCP, a more challenging problem. As a first experiment, we analyzed $\rho-$rocksample with different values of reward range(c). The reward range is an important parameter of POMCP, because it regulates the balance between exploration and exploitation. Since it is not easy to compute an optimal value for it, the parameter must be hand-tuned, an error-prone procedure that can lead to faulty decisions. As shown in the table 5.3, we see that a higher Reward Range value corresponds to a better discounted/undiscounted return in rocksample problem. Changing the value of c in ρ-rocskample has an higher impact than in $\rho-$tiger.

c	**Discounted return**	**Undiscounted Return**
10	31.75	56.00
20	38.74	57.00
50	58.91	92.00
100	61.91	98.00

Table 5.3: Discounted and Undiscounted return for $\rho-$Rocksample.

We analyzed the ρ-rocksample to figure out what level of confidence is needed before sampling a rock. We used the following formula:

`declare-var` u *prob*;

`declare-rule`

`action` *sample*

$$\iff \bigvee_{r \in \{rocks\}} (pos = r.pos \wedge \neg \texttt{collected}(r, step) \wedge p(r.valuable) \geq u);$$

Where u is a free variable that must be instantiated by XPOMCP by parsing traces.

We generated rules to explain how rho-POMCP changes as c varies in table 5.4.

c	u	n_E	**Total**
10	0.611	71	1290
20	0.617	93	1216
50	0.67	93	1253
100	0.701	82	1381

Table 5.4: Rule generation for $\rho-$Rocksample.

Table 5.4 shown the results of the rule generation for $\rho-$Rocksample. The c column represent the values of the reward ranges used. The threshold u is the confidence level that is needed before sampling a rock (e.g., in the first row, we need to be at least 61% sure that the rock is valuable before digging it). The number of errors are the beliefs in which the POMCP fails to meet this rule. Finally, the last column is the total number of beliefs considered.

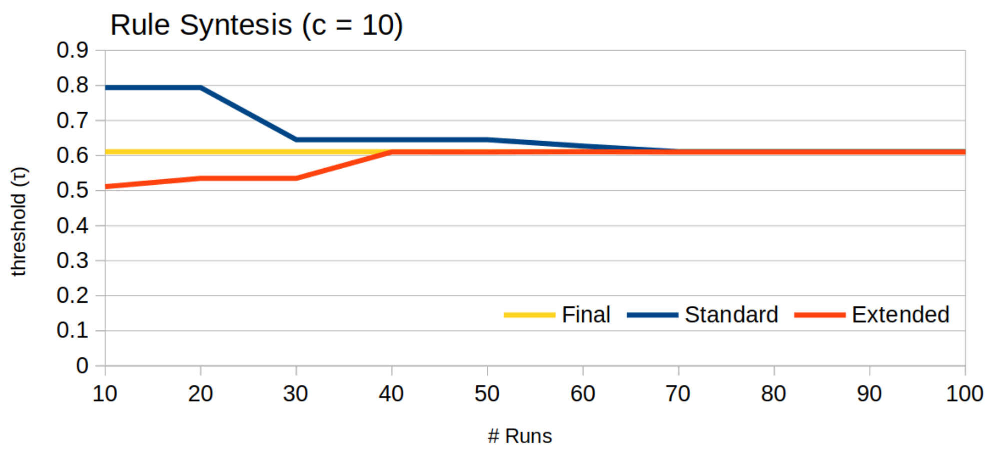
From this table we can see that as c increases, the rock only sample when its confidence of finding a valuable rock is higher. It can also be seen that the number of errors, i.e., actions in which the POMCP digs with lower confidence, is independent of the value of c. This depends on the fact that $\rho-$POMCP cannot always compute the optimal strategy even when using a high number of particles for simulation. To solve this problem, one could use alternative strategies, such as rule-based shielding [7].
With the use of the bags, it is possible to improve trace generation by also collecting beliefs in future states that are not explored. To do this we consider future states that have at least 150 particles in their bags, to ensure that these extra beliefs are adeguatelly explored, which leads to a trace extension of about 20%. These addictional information significantly improves rule generation.

5.3.1 Performance Evaluation

In this section we show an example of a generated extended trace of the algorithm. Here we compared rule generation with and without the extension, as the number of traces analyzed varies. There are four tables, one for each value $c \in \{ 10, 20, 50, 100\}$.

	Standard Traces			Extended Traces		
Run	τ	n_E	**total**	τ	n_E	**total**
10	0.794	3	103	0.511224	10	133
20	0.794	12	231	0.53493	22	302
30	0.645	19	351	0.53493	30	450
40	0.645	26	489	0.610296	42	629
50	0.645	30	615	0.610	50	794
60	0.627	40	776	0.611	65	999
70	0.611	46	930	0.611	77	1212
80	0.611	55	1055	0.611	88	1371
90	0.611	61	1172	0.611	97	1523
100	0.611	71	1290	0.611	110	1670

Figure 5.1: Rule generation difference between Standard Traces and Extended Traces for $\rho-$Rocksample with c=10.

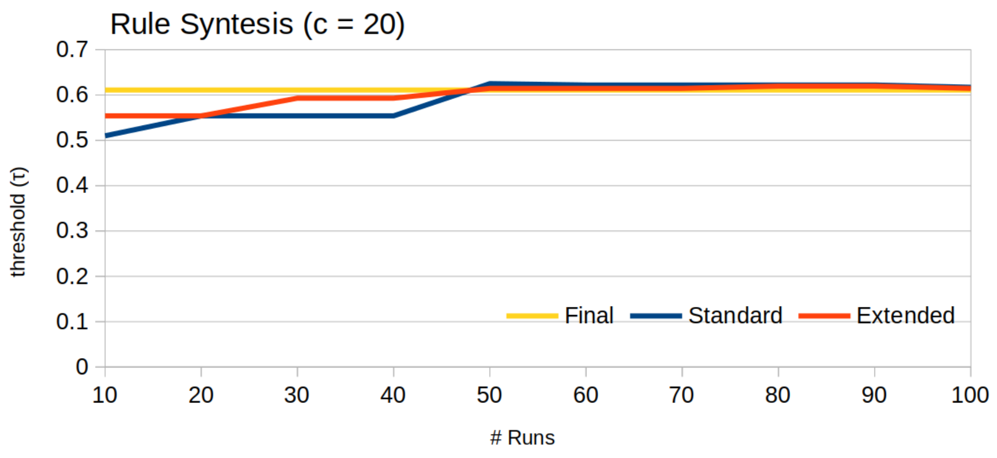


In figure 5.1 with RewardRange set to 10 the confidence level difference between standard and extended traces is zero when we use an high number of traces (i.e., 70 or more). However, when the number is limited the extended trace generate results that are closer to the final value. This is particularly relevant when we consider only 10 runs in the traces. Here, the distance between the real value (i.e., 0.611) and the value generated using standard traces (i.e., 0.794) is nearly double the distance between the real value and the value generated using the extended trace (i.e., 0.511). The extended trace converges considering only 40 runs, while the standard trace requires 70 runs. Note that even if the threshold is equal, the number of errors where the rule does not apply is higher in the extended traces. This is due to the fact that with extended traces more beliefs are considered than with respect to normal traces.

In figure 5.2 it can be note that the results obtained are almost equivalent, if not slightly better than the results with $c = 10$.

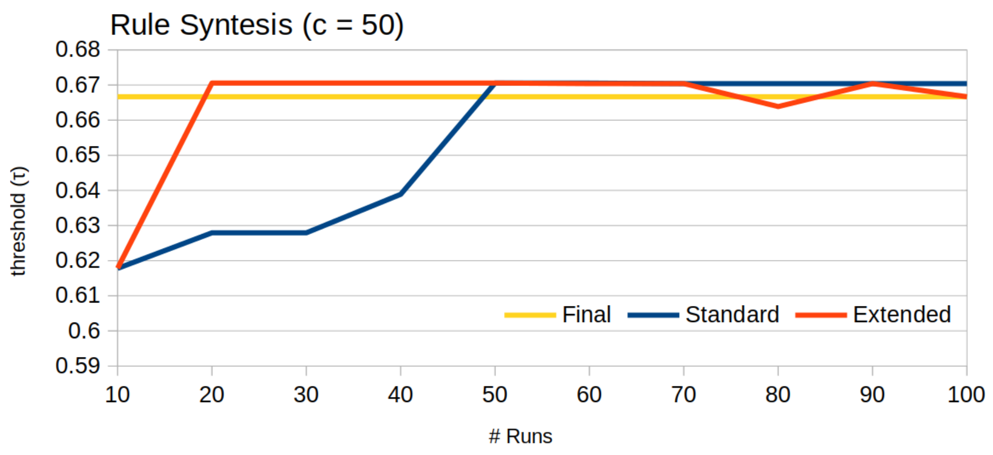
	Standard Traces			**Extended Traces**		
Run	τ	n_E	**total**	τ	n_E	**total**
10	0.51	5	102	0.554	8	117
20	0.554	9	212	0.554	21	248
30	0.554	18	322	0.593	33	376
40	0.554	34	467	0.593	56	565
50	0.625	45	595	0.614582	70	717
60	0.622	54	767	0.614582	83	977
70	0.622	65	874	0.614582	99	1103
80	0.622	73	1002	0.619699	113	1262
90	0.622	83	1103	0.61969	129	1373
100	0.617	93	1216	0.61969	148	1507

Figure 5.2: Rule generation difference between Standard Traces and Extended Traces for $\rho-$Rocksample with c=20.



	Standard Traces			Extended Traces		
Run	τ	n_E	**total**	τ	n_E	**total**
10	0.6178	6	102	0.617835	9	118
20	0.627931	16	221	0.670556	22	249
30	0.627931	24	323	0.670556	33	359
40	0.638889	35	466	0.670556	48	527
50	0.670556	45	581	0.670556	60	648
60	0.670556	55	710	0.670397	78	796
70	0.670396	64	851	0.670397	95	964
80	0.670396	69	948	0.663865	104	1076
90	0.670396	83	1109	0.670397	126	1258
100	0.670396	93	1253	0.666668	137	1415

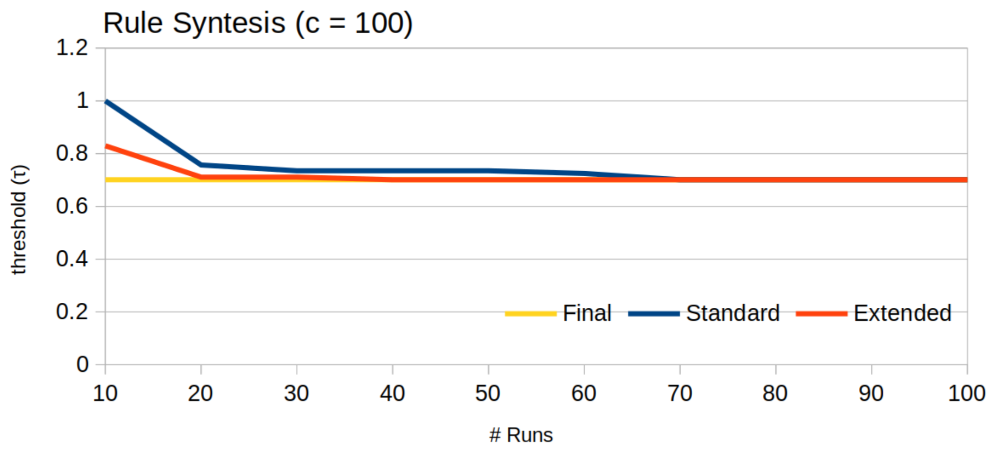
Figure 5.3: Rule generation difference between Standard Traces and Extended Traces for $\rho-$Rocksample with c=50.



In figure 5.3 with RewardRange set to 50 the confidence level is increased in respect to previous figures. By increasing the threshold, the number of errors where the rule brokes is lower than the results with c = 10 and c = 20. This is due to the fact that increasing the threshold consequently decreases the probability of making errors.

	Standard Traces			**Extended Traces**		
Run	τ	n_E	**total**	τ	n_E	**total**
10	1	10	122	0.829703	15	140
20	0.757	14	256	0.711	27	291
30	0.735	21	399	0.711	39	450
40	0.735	26	528	0.701	49	594
50	0.735	34	631	0.701	61	709
60	0.725	43	782	0.701	81	884
70	0.701	51	915	0.701	95	1038
80	0.701	61	1059	0.701	106	1197
90	0.701	72	1203	0.701	120	1352
100	0.701	82	1381	0.701	133	1546

Figure 5.4: Rule generation difference between Standard Traces and Extended Traces for $\rho-$Rocksample with c=100.



In figure 5.4 with RewardRange set to 100 the confidence level is the highest of all reward range proved and consequently the n_e errors are the lowest.

In Figures 5.1 to 5.4 we see how threshold and number of errors varies as the number of runs considered varies. The figures show that the extension leads the rules to converge using fewer traces, thus making the rule synthesis process faster and more efficient. It can be seen that the number of errors increases, in percentage, when considering extended runs. This is due to the fact that not all of the added data is optimal. However, the rule generation process is robust to these types of errors, we can see this from the u values that tend to converge to the final value first, regardless of the errors.

Chapter 6

Conclusion

In this thesis, we presented an integration between the $\rho - POMCP$ algorithm and an explainability procedure, XPOMCP. This integration combines the strenght of the two algorithms, and it leads to the generation of extended traces, capable of collecting a greater number of significant belief-action pairs.

To do this we extended two standard POMDP benchamarking domain, i.e., rocksample and tiger, creating a ρ-POMDP version for each of them.

To extend the rule-based framework we introduce the concept of bag of beliefs. Thanks to the use of bags, it is possible to improve the generation of traces by also collecting beliefs in future states that are not explored. Finally we create extended trace to generate more precise rule thanks to XPOMCP algorithm. We have shown that this extension is particularly relevant in complex problems, such as rocksample, which contain thousands of possible states. Thanks to the new method you can build rules that explain the behavior of the policy using fewer runs than POMCP.

Rule generation can be useful to better understand ρ-POMCP generated policies, and it is possible to use the properties to improve the process of trace generation.

This work paves the way for several possible research direction. One immediate extension could be to apply the extended framework to other domains such as Velocity Regulation or patroling problem, that are highly influenced by information-oriented objective. A more long term extension is to define rules that specify temporal properties that must be satisfied for an extended period of time. Finally, an interesting direction is to implement extended traces in classical POMCP, a popular and widely used algorithm. This is a challenging extension, due to the difference in the usage of the beliefs in the two algorithms. However, this could greately improve rule generation procedures like XPOMCP.

Bibliography

[1] Mauricio Araya-López et al. "A POMDP Extension with Belief-dependent Rewards". In: *NIPS*. Curran Associates, Inc., 2010, pp. 64–72.

[2] Anthony R. Cassandra, Michael L. Littman, and Nevin Lianwen Zhang. "Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes". In: *UAI*. Morgan Kaufmann, 1997, pp. 54–61.

[3] Peter C Fishburn. "Utility theory". In: *Management science* 14.5 (1968), pp. 335–378.

[4] Wai-Tat Fu et al., eds. *Proceedings of the 24th International Conference on Intelligent User Interfaces, IUI 2019, Marina del Ray, CA, USA, March 17-20, 2019*. ACM, 2019.

[5] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. "Planning and Acting in Partially Observable Stochastic Domains". In: *Artif. Intell.* 101.1-2 (1998), pp. 99–134.

[6] Giulio Mazzi, Alberto Castellini, and Alessandro Farinelli. "Identification of Unexpected Decisions in Partially Observable Monte-Carlo Planning: A Rule-Based Approach". In: *AAMAS*. ACM, 2021, pp. 889–897.

[7] Giulio Mazzi, Alberto Castellini, and Alessandro Farinelli. "Rule-based Shielding for Partially Observable Monte-Carlo Planning". In: *ICAPS*. AAAI Press, 2021, pp. 243–251.

[8] Christos H Papadimitriou and John N Tsitsiklis. “The complexity of Markov decision processes”. In: *Mathematics of operations research* 12.3 (1987), pp. 441–450.

[9] Stéphane Ross et al. “Online planning algorithms for POMDPs”. In: *Journal of Artificial Intelligence Research* 32 (2008), pp. 663–704.

[10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[11] David Silver and Joel Veness. “Monte-Carlo planning in large POMDPs”. In: *Advances in neural information processing systems* 23 (2010).

[12] Trey Smith and Reid Simmons. “Heuristic search value iteration for POMDPs”. In: *arXiv preprint arXiv:1207.4166* (2012).

[13] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[14] Vincent Thomas, Gérémy Hutin, and Olivier Buffet. “Monte Carlo Information-Oriented Planning”. In: *ECAI*. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 2378–2385.