Master Thesis

Simulation Based Planning for Partially Observable Markov Decision Processes with Continuous Observation Spaces

Andreas ten Pas

Master Thesis DKE 09-16

Thesis submitted in partial fulfillment
of the requirements for the degree of Master of Science
of Artificial Intelligence at the Department of Knowledge
Engineering of the Maastricht University

Thesis Committee:

Dr. ir. Kurt Driessens
Michael Kaisers M.Sc.
Dr. Frans Oliehoek

Maastricht University
Faculty of Humanities and Sciences
Department of Knowledge Engineering
Master Artificial Intelligence

August 22, 2012

Abstract

Many problems in Artificial Intelligence and Reinforcement Learning assume that the environment of an agent is fully observable. Imagine, for instance, a robot that moves autonomously through a hallway by employing a number of actuators and that perceives its environment through a number of sensors. As long as the sensors provide reliable information about the state of the environment, the agent can base its decisions directly on the state of the environment. In many domains, however, sensors can only provide partial information about the state of the environment.

A mathematical model to deal with such domains is the partially observable Markov decision process (POMDP). Offline planning algorithms for POMDPs are known to arrive at optimal policies for POMDPs where the state space, the action space, the observation space and the planning horizon are all finite. However, these exact algorithms turn out to be computationally very expensive. As an alternative to offline planning, online planning algorithms try to overcome those computational costs. Monte Carlo Tree Search (MCTS), an online planning algorithm, has recently been successfully extended to POMDPs.

The observations recorded by sensors such as laser range finders and radars are typically continuous. Such systems can be modeled by POMDPs. This thesis deals with the problem of extending MCTS to POMDPs with continuous observations.

To tackle this problem, a novel online planning algorithm, called Continuous Observations Monte Carlo Tree Search (COMCTS), is proposed. The algorithm combines Monte Carlo Tree Search with an incremental regression tree learning technique that builds discretizations of the observation space and allows to incorporate the observations into the search tree.

Belief based Function Approximation using Incremental Regression tree techniques and Monte Carlo (BFAIRMC) is another novel online planning algorithm presented in this thesis. The algorithm combines Monte Carlo rollouts with an incremental regression tree learning technique that creates a discretization of the belief space and allows to base the agent's action selection directly on specific regions of the belief space.

For both novel algorithms, several strategies to avoid information loss in the case of a refinement of a discretization are discussed. These strategies evolve from loosing all information over loosing partial information to loosing no information.

An evaluation of the algorithms on a set of benchmark problems shows that COMCTS performs significantly better than a random algorithm but only slightly better than uniform Monte Carlo rollouts. BFAIRMC shows a clear improvement of over uniform Monte Carlo rollouts per sample but the algorithm turns out to be relatively slow in practice. The evaluation also indicates that the strategies to avoid information loss directly improve the performance of BFAIRMC per sample, but - surprisingly - do not improve the performance of COMCTS.

Preface

When I started thinking about the topic for my master's thesis, I was still focused on a topic related to game theory. During my considerations, I - kind of accidentally - ran into the topic of reusing knowledge in Tree Learning Search, a recent extension of Monte Carlo Tree Search to continuous actions. I had some experience with Monte Carlo Tree Search through a project during my master's study and got to like the ideas behind the algorithm is a lot. This planning was still during my semester abroad. So, when I came back, I contacted the inventors of this topic, but unfortunately, the topic and a similar topic had already been taken by two other students. That put some frustration into me and I had doubts that it was still possible to find some interesting and related topic. However, when I met with the inventors who later on became the supervisors for this thesis, they offered so many related ideas that my frustration was put aside.

For me, this experience clearly underlines that a master's thesis is not something that comes out of nowhere. Without the help of a group of people supporting me during the research, this thesis would not be where it is now.

First of all, I would like to express my gratitude to my three supervisors. Frans, thank you for your enthusiasm, support and constructive discussion during our many meetings. There have been times that my research did not proceed as well as I hoped or did not show the results that I expected, or I lacked motivation or inspiration. Every time one meeting with you was enough to get me back on track. I am looking forward to continuing this cooperation with you in the future. Kurt, thank you for all your good comments and suggestions. It is always a real pleasure to work with you. Michael, thank you for all your ideas and inspirations which established the basic idea for this thesis and lead to very fascinating algorithms in the end. I also want to thank you for all the informative - and often also entertaining - joint meetings that you initiated.

A special acknowledgment goes to my fellow students Lukas and Colin who worked on related topics for their master theses and with whom I had a lot of long and fruitful discussions. The exchange of knowledge and cooperation within our group has been on a much higher level than in any project group I had been part of during my studies.

I would also like to thank the readers of earlier versions of this thesis, Veronika and Henning.

Furthermore, I thank everyone who supported me and with whom I had very enjoyable times during my studies, especially my friends and family. Without you, I would not have been where I am now.

I hope that you will enjoy reading this thesis as much as I - nearly all of the time - enjoyed the process of creating it.

Contents

Chapter 1

Introduction

In many problems in Artificial Intelligence and Reinforcement Learning, the assumption is made that the environment of an agent is fully observable. Imagine, for instance, a robot that interacts with the outer world through a number of actuators and sensors. As long as the sensors provide reliable information about the state of the environment, the agent can base its decisions directly on the state of the environment. In many domains, however, sensors can only provide partial information about the state of the environment.

Partially observable Markov decision processes (POMDPs) are used to model such sequential decision-making problems in which the state of the world is not fully observable. They provide a mathematical framework which is capable of capturing uncertainty in both action effects and perceptual stimuli. In environments where the state is only partially observable, the agent can only estimate the world state by maintaining a probability distribution over the set of possible states.

The traditional approach to find the optimal policy in POMDPs is offline planning which includes value iteration and policy iteration algorithms. Offline planning algorithms are known to be able to determine the optimal policy in worlds where the state space, the action space, the observation space and the planning horizon are all finite. However, these exact approaches turn out to be computationally expensive.

Monte Carlo Tree Search (MCTS) is a search technique that combines stochastic simulations with tree search. Although MCTS has been mainly applied to the domain of game tree search in classical board games, it can in principle be applied to any domain that can be expressed in terms of state-action pairs. MCTS is unable to handle continuous action and state spaces. Recently developed approaches for quasi continuous games such as Poker perform an a priori discretization of the action or state space [11].

As an alternative to a priori discretization, Tree Learning Search (TLS) [46] is a recently developed algorithm that performs an online discretization of the action space. Based on ideas from data stream mining, TLS is a combination of MCTS and incremental regression tree induction which extends MCTS to continuous action spaces. It builds discretizations of the search space which are dependent on the game state and discovers high promising regions of the search space. TLS has been successfully applied to one-step [46] and multi-step [22, 39] optimization problems.

Recently, Monte Carlo Tree Search has been successfully extended to POMDPs with discrete observation spaces [40] by incorporating the observations into the search tree. However, POMDPs with continuous observations add another level of difficulty to the problem. A continuous space cannot directly be mapped into a tree because there are infinitely many possible observations in such a space. The incremental regression tree component of TLS has been successfully applied to discretize continuous action spaces and could, in theory, be used for any kind of continuous space including the (possibly) continuous observation space of a POMDP.

1.1 Problem Statement and Research Questions

The considerations above lead to the following problem statement.

Problem statement. *Is it possible to extend Monte Carlo Tree Search to Partially Observable Markov Decision Processes with continuous observations?*

This problem statement is refined in five research questions. First of all, there needs to be some algorithmic component that allows to discretize a continuous space while planning online. Tree Learning Search offers such a component for the action space. Theoretically, this component could be used for any kind of continuous space. In the ideal case, the discretizations help to improve the agent's performance. Considering POMDPs, it cannot be directly derived whether this component would provide such discretizations. This leads to the first research question.

Research question 1. *Can the incremental regression tree component of Tree Learning Search be used to discretize the observation space instead of the action space?*

Secondly, the belief space also forms a continuous space. This space represents any possible belief the agent can have about the world. As addressed in Section 2.2, the agent's action selection strategy can be based on its belief. Therefore, instead of discretizing the observation space, the regression tree component could directly discretize the belief space. This results in the second research question.

Research question 2. *Can the incremental regression tree component of Tree Learning Search be built on the belief space instead of the action space?*

Incremental regression tree learners constructs a tree that is built by collecting statistics and splitting nodes if there is sufficient evidence that a split would result in a different reward accumulated by the agent. This splitting causes the problem of information loss, and raises the third research question.

Research question 3. *How can information loss be avoided in the techniques mentioned above?*

In order to analyze the memory and time consumption, the techniques mentioned above can be investigated from the perspective of theoretical computational complexity. The fourth research question is therefore as follows.

Research question 4. *What is the computational complexity of the techniques mentioned above?*

Finally, the techniques mentioned above need to be evaluated and compared to existing methods. This leads to the final research question.

Research question 5. *Which of the techniques mentioned above performs best on a set of benchmark problems?*

1.2 Contributions of the Thesis

This thesis develops two approximate algorithms that are based on stochastic simulations and incremental regression tree learning, and allows to derive near-optimal policies for an agent in POMDPs with continuous observation spaces. The first algorithm, called Continuous Observations Monte Carlo Tree Search, is an extension of Monte Carlo Tree Search to such POMDPs. The second algorithm, called Belief based Function Approximation using Incremental Regression tree techniques and Monte Carlo, combines an incremental regression tree learner with Monte Carlo rollouts. While the first algorithm builds multiple discretizations of the observation space, the second algorithm builds one discretization of the belief space.

For each algorithm, this thesis contributes by addressing a number of strategies that avoid information loss in case of a refinement of a discretization.

This thesis further contributes by investigating the theoretical running times and the space bounds of each novel algorithm, and by giving insights to their practical performance through an empirical evaluation on a set of benchmark problems.

1.3 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 introduces the topic of partial observable environments, and the mathematical model used to represent these environments, POMDPs. In addition, exact and approximate belief representations for POMDPs are discussed. From an elaboration on offline planning methods for POMDPs, the chapter then proceeds to online planning algorithms. A specific kind of online planning algorithms are Monte Carlo methods, including Monte Carlo Tree Search which builds the algorithmic foundation for the techniques proposed in this thesis. Theoretical bounds for the required depth of the search tree constructed by Monte Carlo Tree Search are given next. The chapter ends with a discussion of continuous observations and their importance in real world applications.

Chapter 3 presents a novel online planning algorithm that combines Monte Carlo Tree Search and incremental regression tree induction. This algorithm builds tree-based discretizations of the observation space by keeping statistical measures about the outcome of the simulations. The chapter continues by introducing strategies to avoid information loss in the case of a refinement of one of these discretizations. An analysis of the computational complexity of the algorithm completes the chapter.

Chapter 4 proposes another novel online planning algorithm that combines Monte Carlo rollouts and incremental regression induction. This algorithm creates exactly one tree-based discretization of the belief space by keeping statistical measures about the outcome of the simulations. The chapter proceeds by discussing strategies to avoid information loss in the case of a refinement of this discretization. The computational complexity of the algorithm is analyzed at the end of the chapter.

Chapter 5 provides an experimental evaluation of the algorithms proposed in the previous two chapters. A set of benchmark problems that is used to test the algorithms is introduced, and a number of additional methods is introduced to provide a broader analysis. Experiments are conducted that give insights into the behavior and the performance of the algorithms with respect to the set of benchmark problems.

Finally, Chapter 6 concludes this thesis by summarizing the results, and by answering the research questions posed in the previous section. Furthermore, it points to directions for future research.

Chapter 2

Online Planning in Partially Observable Environments

This chapter leads the reader from the presentation of partially observable environments over approximate methods that derive near optimal behavior for an agent through online planning to Monte Carlo methods. Section 2.1 introduces environments without complete observability. Partially observable Markov decision processes (POMDPs) constitute a mathematical model for such environments, and are discussed in Section 2.2.

There are two classes of algorithms that construct policies for POMDPs: offline and online planning algorithms. Both classes are addressed in Section 2.3. Monte Carlo methods, a family of online planning algorithms, range from simple rollout methods to tree search methods such as Monte Carlo Tree Search (MCTS), and are elaborated on in Section 2.4. The extensions of MCTS to POMDPs and to continuous action spaces are also discussed in that section.

Kearns et al. [21] derive theoretical bounds for the depth of the search tree build by a tree-based algorithm such as MCTS. These bounds ensure that the policies for the agent's behavior generated by the algorithm approach the optimal policy. The work of Kearns et al. is recapitulated in Section 2.5.

Section 2.6 discusses continuous observations and their relation to real world applications. In addition, it gives a brief overview of methods that have been applied to POMDPs with continuous observation spaces.

2.1 Partially Observable Environments

Consider an agent that perceives its environment through a number of sensors and tries to reach an objective which may need several sequential actions to accomplish. As long as the sensors provide the complete state of the environment, such sequential decision problems can theoretically be solved by a number of reinforcement learning and planning algorithms [43]. In many domains, however, sensors do not receive complete information about the state of the environment. The reasons for non-reliable sensors are numerous, and include range limits, noise or electronic failure.

Domains where the environment cannot be completely observed by the agent range from industrial over scientific applications to such diverse areas as military,

business and social applications [6]. Some of the most well-known applications are autonomous robots and games such as Poker. In addition, machine maintenance, structural inspection, medical diagnosis, marketing and questionnaire design [6] form a collection of example areas that emphasize the diversity of partially observable domains.

How can agents make optimal decisions when the environment is not fully observable and the outcome of actions is not completely certain? For fully observable environments, the optimal behavior is a mapping from states to actions. If the state cannot be completely determined, the agent needs to use the observation signal that it receives from the environment to influence its behavior. This setting is illustrated in Figure 2.1. Formally, the agent interacts

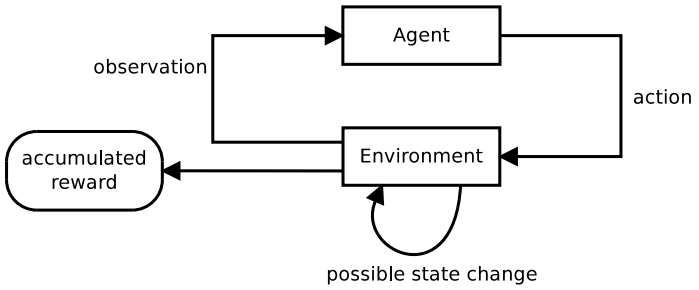


Figure 2.1: The relation between the agent and the environment.

with the environment in a sequence of discrete time steps, $t = 0, 1, \dots$. At each time step t, the agent selects an action a^t and receives an observation o^t. Each action might change the state of the environment. Over multiple time steps, the agent accumulates rewards given by the environment. This agent-environment interaction can be modeled in terms of a partially observable Markov decision process which is introduced in the next section.

2.2 Partially Observable Markov Decision Processes

Markov Decision Processes (MDPs) provide a model for sequential decision-making problems in which the state of the world is completely observable. An MDP can be defined as a 4-tuple (S, A, r, T) where

- $S = \{s_1, s_2, \dots\}$ is a set of possible world states,
- $A = \{a_1, a_2, \dots\}$ is a set of possible actions,
- $r(s, a)$ is a reward function, and
- $T(s, a, s') = \Pr(s'|s, a)$ is a transition model.

An MDP has the Markov property, named after the Russian statistician Andrei Markov [26]: the conditional probability distribution of any future state depends only on the present state and not on the past states of the system. This property

is expressed by the following equation.

$$P(s^{t+1} = j | s^t = i) = P\{s^{t+1} = j | s^t = i, s^{t-1} = i^{t-1}, \ldots, s^1 = i^1, s^0 = i^0\} \tag{2.1}$$

where $i, j \in S$.

When the state of the world is not fully observable, the problem can be modeled by a Partially Observable Markov Decision Process (POMDP). This mathematical framework is capable of capturing uncertainty in both action effects and perceptual stimuli. A POMDP can be defined as a 6-tuple (S, A, r, T, O, Ω) where

- S, A, r, T are the same as for MDPs,
- $O = \{o_1, o_2, \ldots\}$ is a set of possible observations, and
- $\Omega(a, s, o) = \Pr(o|a, s')$ is an observation model that specifies the probability of perceiving observation o in state s' after performing action a.

In environments where the state is not directly observable, the agent needs to derive its behavior from the complete history of actions and observations up to the current time step t, defined as follows.

$$h^t = \{a^0, o^1, a^1, o^2, \ldots, a^{t-1}, o^t\} \tag{2.2}$$

Instead of explicitly storing the complete history which is typically very memory intensive, it is possible that the agent maintains a probability distribution over the state space S, representing the likelihood the agent beliefs to be in each state. This probability distribution is called belief state and is given in the following equation.

$$b(s) = \Pr(s^t = s | h^t = h) \tag{2.3}$$

Since $b(s)$ is a probability, the following axioms of probability are required to hold.

$$b(s) \in [0, 1] \ \forall s \in S, \text{ and} \tag{2.4}$$

$$\sum\nolimits_{s \in S} b(s) = 1. \tag{2.5}$$

The belief state has the Markov property, making it a sufficient statistic for the initial belief state and the past history of the agent [41]. Additional data about the past actions and observations of the agent cannot provide any further information about the current state of the world, given the current belief state.

The belief state can be represented in two different ways. The traditional approach is an exact representation that is based on Bayesian abduction. Given the current belief state $b(s)$, an action a executed and an observation o perceived by the agent, the next belief state $b(s')$ can then be computed according to Bayes theorem [12] as

$$b'(s') = \alpha \Omega(a, s', o) \sum_{\forall s \in S} T(s, a, s') b(s) \tag{2.6}$$

where α is a normalization factor that assures that the sum over all belief states is 1 and $\Omega(a, s', o)$ is the probability of perceiving observation o in state s' after performing action a.

The size of the belief space is illustrated best by an example. Consider a 2×2 grid world with its 4 discrete states. The belief is a 4-dimensional

probability mass function and to update it the agent needs to consider the transition probabilities of all possible states, the belief over these states and the probability of the current observation in each of these states. While the computations necessary to update the belief state can still be performed in a relatively small amount of time for problems with a small state space, computing the update is not feasible for problems with thousands or millions of states. In addition, it might not be possible to represent the transition or observation probabilities in a compact form.

As an alternative to the exact representation, the belief state can be approximated by model estimation techniques such as Kalman or particle filters which break these two computational barriers. Kalman filters represent the belief state by a multivariate normal distribution, while particle filters represent the belief state as a set of particles that correspond to possible states of the POMDP. The details of these two estimation techniques are given in Appendix A.

The belief state is initialized according to a specific probability distribution, $b^{t=0} = b^0$, which represents the agent's initial degree of uncertainty about the environment's actual state. The choice of the distribution b^0 depends on the domain. If the agent does not know at all what the true state is, the initial belief is usually uniform over the state space S. In many robot localization problems [44], the belief can be centered around the robot's actual location, and a Gaussian provides a more suitable choice for the initial probability distribution.

The agent's behavior at any time is defined by a policy $\pi(b)$ which maps from belief states to a probability distribution over actions, and is given by the following equation.

$$\pi(b) = \Pr(a^{t+1} = a | b^t = b) \tag{2.7}$$

The agent's goal is to maximize expected return, also called discounted future reward, and calculated as

$$R^t = \sum_{k=t}^{\infty} \gamma^{k-t} r(s^k, a^k) \tag{2.8}$$

where $0 \leq \gamma < 1$ is a discount factor that specifies the importance of future rewards and gives a finite bound to the infinite sum.

2.3 Solving POMDPs

The expected value achieved by following a policy $\pi(b, a)$ from a specific belief state b is defined in terms of the value function, which is given as

$$Q^\pi(b, a) = \sum_{s \in S} b(s) r(s, a) + \gamma \sum_{o \in O} \Pr(o|b, a) Q^\pi(b_a^o, a) \tag{2.9}$$

where the first term of the equation specifies the expected immediate reward of action a in belief state b and the second term specifies the summed value of all successor belief states. Here, b_a^o is the belief state reached by performing action a and perceiving observation o. The second term of Equation 2.9 is weighted by the probability of observing o in the successor belief states.

The maximum value function that can be reached by any policy is the optimal value function. For each POMDP, there is at least one optimal policy π^*

that achieves the optimal value function and fulfills the agent's goal of maximizing the return R.

The optimal policy can be derived from the optimal value function Q^*. This function can be computed by the following Bellman equation [2]:

$$\pi^*(b) = \arg\max_{a \in A} [Q^*(b, a)] \tag{2.10}$$

Equation 2.10 expresses that the optimal policy for a belief state b is the action a for which the value function $Q^*(b, a)$ is the maximum over all optimal value functions for that belief state. A class of algorithms that is based on the Bellman equation and that exactly solves POMDPs is presented in the next subsection.

2.3.1 Offline Planning

From the Bellman equation, it can be seen that it is very hard to directly solve POMDPs. In fully observable environments, algorithms based on dynamic programming [2] can be used to solve the Bellman equation as long as the state and action spaces are not too large. For POMDPs, this is not feasible because the belief space is a continuous and usually high dimensional space. Nevertheless, the value function has some special mathematical properties that simplify the computation of the optimal value function and allow to apply offline policy planning algorithms such as value or policy iteration [41].

The main idea behind value iteration for POMDPs is that a policy $\pi(b)$ can be represented as a set of regions of the belief space. For each of these regions, there is one particular optimal action. The value function assigns a distinct linear function to each of these regions. Each step of value iteration evolves the set of regions by adjusting their boundaries or by introducing new regions.

Policy and value iteration algorithms are known to be able to determine the optimal policy in worlds where the state space, the action space, the observation space and the planning horizon are all finite. However, these exact algorithms turn out to be computationally very expensive. POMDPs with a larger number of states are often already infeasible to solve. In fact, achieving the exact solution is PSPACE-hard for finite-horizon POMDPs [29] and undecidable for infinite-horizon POMDPs [25].

An approach to resolve this computational barrier is Point-Based Value Iteration (PBVI) [31] which selects a small set of points from the belief space and repeatedly applies value iteration to these points. PBVI has been shown to successfully solve problems with much more states (more than 800) than the problems that can be effectively solved with the original value iteration algorithm. In addition, the time required to perform PBVI is significantly lower than the time required to perform standard value iteration. An approach that reduces the time requirements even more are online planning algorithms, presented in the next subsection.

2.3.2 Online Planning

Online planning algorithms approach POMDPs from a completely different angle than the algorithms presented in the previous section. Given the current belief state, an online planner applies forward search to approximate the optimal value function. An online planning algorithm can be divided into a planning

and an execution phase. In the planning phase, the agent's current belief state is passed to the algorithm and the best action to execute in that belief state is determined. In the execution phase, this action is executed in the environment and the agent's current belief state is updated.

Ross et al. [36] provide an extensive survey of the existing online planning methods and a general framework for online planning algorithms that uses an AND/OR-tree in the planning phase. This tree alternates two layers of nodes. The first layer consists of OR-nodes that represent belief states reachable from the agent's current belief state. At these nodes, actions can be selected. The second layer consists of AND-nodes which represent all possible observations given some action selected at the preceding level of the tree. The authors relate this framework to the existing online planning algorithms which apply branch-and-bound pruning [30] and search heuristics [38, 35] to improve the search. While branch-and-bound pruning helps to reduce the number of actions that the algorithm needs to consider, search heuristics help to concentrate the search on possible future belief states that quickly improve the performance of the agent.

Another approach to online planning is provided by Monte Carlo methods which apply repeated random sampling during the planning phase. These methods are introduced in the next section.

2.4 Monte Carlo Methods for POMDPs

Monte Carlo methods are based on repeated random sampling. They originate from the field of statistical physics where they have been used to estimate intractable integrals, and are now used in a wide range of areas such as games [9], engineering and computational biology. In contrast to classical search methods which require heuristics to apply search and pruning effectively, Monte Carlo methods evaluate states by sampling from (quasi) random simulations. The idea behind this approach is that there is not much to be learned from a single random simulation but that from lots of simulations, a successful strategy can be derived. All Monte Carlo methods keep track of the value of an action a in a state s, denoted by $Q(s, a)$, and the number of times action a has been selected in state s, denoted by $N(s, a)$.

Monte-Carlo planning algorithms use a generative model $\mathcal{G}$ of the POMDP. This so-called *black-box simulator* employs the same dynamics as the POMDP. It takes a state $s^t \in S$ and action $a^t \in A$ as input, and returns a sample of a successor state s^{t+1}, observation o^{t+1} and reward r^{t+1} as output. This process is defined in the following equation.

$$\mathcal{G}(s^t, a^t) \rightsquigarrow (s^{t+1}, o^{t+1}, r^{t+1}) \tag{2.11}$$

Here, the operator $\rightsquigarrow$ denotes sampling, e.g., $a \rightsquigarrow b$ means b is sampled from a. The models contained in the underlying POMDP provide the successor variables: the state $s^{t+1} \in S$ is given by the transition model, $T(s, a) \rightsquigarrow s^{t+1}$, the observation $o^{t+1} \in O$ is given by the observation model, $\Omega(s, a) \rightsquigarrow o^{t+1}$, and the reward r^{t+1} is given by the reward function, $r(s, a) \rightsquigarrow r^{t+1}$.

It is also possible to reset the simulator to a start state s. Instead of directly considering the model's dynamics, the value function is updated using the sequences of states, observations and rewards that are generated by the simulator.

Kearns et al. [20] have shown that the amount of samples required to create good policies depends only on the complexity of the underlying POMDP, and not on the size of the POMDP's state space. Their proof is based on the idea of reusable sequences of states, observations and rewards sampled by the generative model. Such sequences provide an accurate estimate for the value of many policies, and it is sufficient to create a small number of these sequences to cover a lot of different policies [20].

The development of Monte Carlo planners for POMDPs has evolved from simple roll-out methods that do not construct a search tree [3] over methods that perform depth-first search with a fixed horizon [20] to a Monte Carlo tree search method [40] that is not limited by a fixed horizon and can compete with offline full-width planners in problems with large state spaces. These algorithms are addressed in detail in the next subsections.

2.4.1 Monte Carlo Rollouts

Monte Carlo rollouts is the most basic Monte Carlo method. Each iteration consists of two steps. The first step, called *simulation*, selects actions according to some probability distribution, usually a uniform distribution, until some terminal condition is reached. Each action is given to the black-box simulator to obtain a sequence of rewards $(r^{d=0}, r^{d=1}, \ldots, r^{d_n})$. Here, d_n denotes the depth of the simulation phase, i.e., the number of steps until the terminal condition is reached. From this sequence, the simulated return $\hat{R}$ is computed as follows.

$$\hat{R} = \sum_{d=0}^{d_n} \gamma^d r^d \tag{2.12}$$

The second step, called *update*, adjusts the values of the action that was selected first during the simulation step according to the following equations.

$$N(s,a) = N(s,a) + 1 \tag{2.13}$$

$$Q(s,a) = Q(s,a) + \frac{\hat{R} - Q(s,a)}{N(s,a)} \tag{2.14}$$

The algorithm repeats the simulation and update steps until some terminal condition is reached. At this point, the action with the highest average value $Q(s,a)$ is chosen.

2.4.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) combines Monte Carlo rollouts with tree search, and was introduced in different versions by Chaslot [9] and Coulom in 2006 [10]. Although MCTS has been mainly applied to the domain of game tree search in classical board games such as *Go* (see, e.g., Chaslot [9] or Coulom [10]), it can theoretically be applied to any domain that can be expressed in terms of state-action pairs, including POMDPs.

The nodes of the MCTS tree can be called *state nodes* because they represent states in the problem. Given an action a and a state s, each node is associated with the two values introduced above: the visitation count, $N(s,a)$, and the

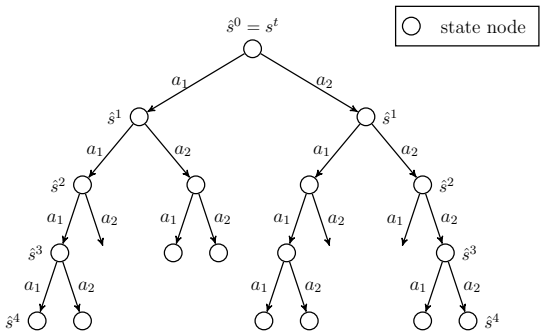


Figure 2.2: The tree build by Monte Carlo Tree Search.

average return of all simulations that passed through this node, $Q(s, a)$. The tree constructed by MCTS is illustrated in Figure 2.2.

An overview of MCTS is given in Figure 2.3. The algorithm iteratively builds a search tree, one node after the other, based on the results of random simulations, until it is interrupted or a computational budget is reached. Each iteration in MCTS can be divided into four different steps: selection, expansion, simulation and backpropagation.

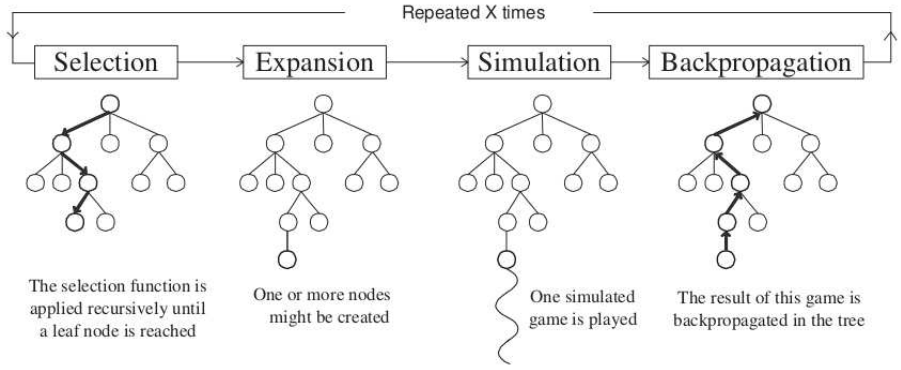


Figure 2.3: An overview of Monte Carlo Tree Search [9].

Selection This step starts at the root of the tree. From there, actions are repeatedly selected according to the statistics stored in the nodes of the search tree until a leaf node L is reached. The selection of optimal actions needs to find a balance between actions that have achieved the best outcomes with respect to the current time step (*exploitation*), and actions that have not been explored (*exploration*).

Expansion One or multiple children of L are added to the tree.

Simulation If L does not correspond to a terminal state in the problem, a random simulation is performed. This simulation starts with the action selected at L and ends when a terminal state in the problem is reached.

Backpropagation The outcome of the simulation step is backpropagated through the internal nodes up to the root of the tree.

When the algorithm reaches a computational budget or is interrupted, the search terminates and an action a is selected at the root node by one of the following criteria, introduced by Chaslot et al. [8].

1. Max child: Select the action with the highest reward.
2. Robust child: Select the most visited action.
3. Max-Robust child: Select the action with both the highest visitation count and the highest reward.
4. Secure child: Select the action which maximizes a lower confidence bound.

MCTS provides three important advantages. The first advantage is the asymmetric growth of the tree developed during the MCTS search. State nodes which appear to be more valuable in the problem are visited more often and more search is spent in the relevant parts of the tree. The second advantage is that the execution of the algorithm can be stopped at any time to return the action that is currently estimated to be optimal. The third advantage is that the algorithm focuses on determining the action that is optimal in the current state because each iteration updates the value of one of the actions available at the current state.

2.4.3 Upper Confidence Bounds for Trees

Several effective strategies have been proposed for the selection step [9]. A well-established strategy for node selection is Upper Confidence Bounds Applied to Trees (UCT), developed by Kocsis and Szepesvári in 2006 [23]. UCT is a generalization of Upper Confidence Bounds, proposed by Auer et al. [1]. Kocsis and Szepesvári studied the multi-armed bandit problem which is analogous to the node selection problem encountered in MCTS. In the multi-armed bandit problem, an agent needs to select a one-armed bandit (slot machine) which gives the maximum estimated outcome in each turn, while in the node selection problem, an agent needs to select an action that maximizes a specific quantity.

At a state node v that represents a state $s(v)$, UCT applies the following equation to select an action $\hat{a}$:

$$\hat{a} = \arg\max_{a} \left\{ Q(s(v), a) + C \times \sqrt{\frac{\ln N(v)}{N(v, a)}} \right\}, \tag{2.15}$$

where the Q function is the same action-value function as in MCTS and the second term of the sum is the so-called UCT bonus. $N(v)$ is the number of visits payed to node v and $N(v, a)$ is the number of visits payed to the child of v that corresponds to action a. Actions that have not been explored at node v receive a value of ∞ in the selection step.

The UCT bonus bounds the regret growth rate by a constant times the best possible regret rate which tackles the exploitation-exploration trade off for bandits. If an action a is selected, the UCT bonus for a decreases because $N(s, a)$ is incremented. At the same time, the UCT bonus for all other actions in

A_s increases because $N(s)$ is also incremented. The parameter C determines the strength of the UCT bonus and is therefore also called *exploration factor*. For rewards in the range $[0, 1]$, Kocsis and Szepesvári found a choice of $C = 1/\sqrt{2}$ to be optimal [23]. For rewards outside of this range, the exploration factor needs to be set through extensive experimentation because the need for exploration varies from domain to domain.

2.4.4 Partially Observable Monte Carlo Planning

Partially Observable Monte Carlo Planning (POMCP), developed by Silver and Veness [40] extends MCTS to POMDPs with discrete observation spaces. By incorporating the observations, the search tree changes from a tree of states to a tree of histories. Given the current history h^t, the algorithm approximates the value function $Q(h^t, a)$.

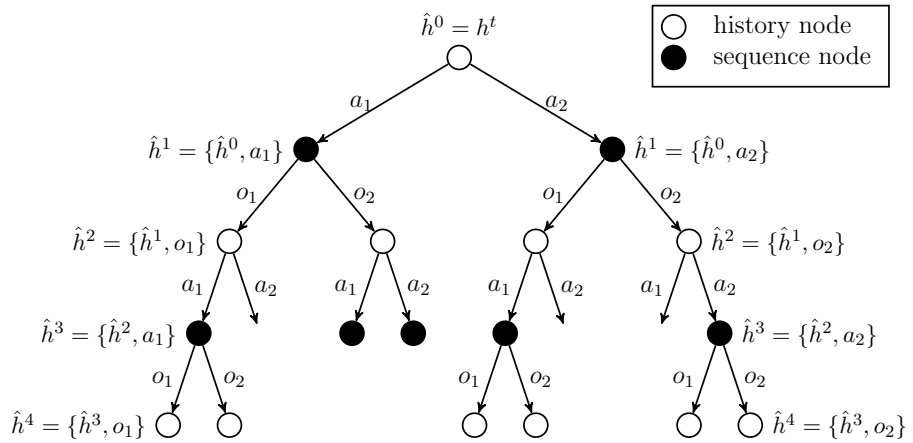


Figure 2.4: The search tree constructed by Partially Observable Monte Carlo Planning for a problem with 2 actions and 2 observations.

POMCP is illustrated in Figure 2.4. In contrast to the MCTS tree in which all nodes represent states, there are two variations of nodes in the POMCP tree. Each *history node* represents a possible history $\hat{h}^k = \{a^0, o^0, a^1, o^1, \ldots, a^k, o^k\}$. The edges emerging from a history node correspond to actions. Based on Koller et al. [24], each *sequence node* represents a *sequence* $\hat{h}^k = \{\hat{h}^{k-1}, a^k\}$ that includes the previous history $\hat{h}^{k-1}$ and the action a^k selected at the preceding history node. The edges emerging from a sequence node correspond to observations.

The algorithm starts from the current history, $\hat{h}_0 = h^t$, by drawing a sample state s from the belief state $b(h^t)$. The black-box simulator $\mathcal{G}$ is then set to state s. While there are no changes to the simulation and backpropagation step of the original MCTS algorithm, the other two steps are changed as follows.

Selection This strategy alternates between the selection of actions at history nodes and observations at sequence nodes. At a history node, an action a is selected according to UCT. This action a is then given to the black-box

simulator to produce an observation o. At the succeeding sequence node, o decides which edge the algorithm needs to follow.

Expansion One new node is added to the tree that corresponds to the first new history encountered during the simulation step.

When the algorithm reaches a computational budget or is interrupted, the search terminates and the agent uses the search tree to selects a real action a for which it perceives a real observation o.

To approximate the belief state, POMCP uses a particle filter for which each particle corresponds to a sample state. At the beginning of each simulation, K particles are drawn from the set of states S according to the current belief state b^t. To update the set of particles, a Monte Carlo procedure is employed that samples observations and compares them to the real observation o. This procedure samples a state s according to the current belief state b^t and passes it to the black-box simulator to receive a successor state s' and an observation o'. If the sampled observation o' is equal to the real observation o, then s' is added to the set of particles. This procedure is repeated until the set of particles consists of K particles.

POMCP is not limited by a fixed horizon search and can compete with best-first, full-width planning methods for POMDPs with a large state space that consist of up to approximately 10^{56} states. In addition, the algorithm requires only a small amount of online computation time to achieve a high performance in such problems.

2.4.5 Tree Learning Search

MCTS is unable to handle continuous action and state spaces. Recently developed approaches for quasi continuous games perform offline discretization of the action or state space [11]. As an alternative to a priori discretization, Van den Broeck and Driessens [46] introduced Tree Learning Search (TLS) which performs online discretization of the action space. Based on ideas from data stream mining, TLS is a combination of MCTS and incremental decision tree induction which extends MCTS to the continuous domain. It builds discretizations of the search space which are dependent on the state of the problem and discovers high promising regions of the search space.

TLS is based on the assumption that the state transitions of the underlying problem are deterministic. The algorithm searches for an open-loop plan, i.e., the generated policy is a sequence of actions, $\pi = \{a^0, a^1, \ldots, a^n\}$.

The search tree constructed by TLS is illustrated in Figure 2.5. Each state node of the MCTS tree is replaced by a tree that discretizes the action space. The root of this so-called *action tree* represents a state in the problem. Each internal node of this tree represents a constraint that subdivides the action space, and each leaf represents a range of action values in the problem.

Through the replacement of the state nodes of the MCTS tree with action trees, the nodes of the TLS tree do not directly correspond to states anymore. Each leaf node of an action tree corresponds now to a range of states given the assumption of determinism in the state transitions. In the case of non-deterministic transitions, actions might not always lead to the same states, and therefore the leaf nodes of the action trees would need to be extended with additional nodes that correspond to particular states (or state ranges).

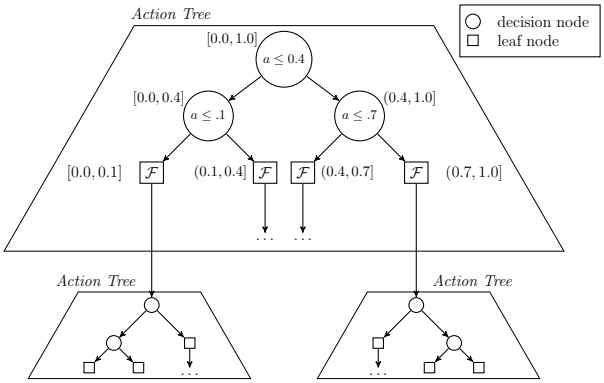


Figure 2.5: An example of the search tree constructed by Tree Learning Search for an action with a range of $[0.0, 1.0]$.

Action trees are constructed by an incremental regression tree learning algorithm that is based on ideas from FIMT [17] and TG [14]. In the beginning, the action tree consists of a single leaf node. For each leaf, the action tree maintains a set of tests $\mathcal{F}$ that decides when the leaf is split. Each test is composed of a set of statistics which are updated for every example that is passed through the tree.

Except for the simulation phase which is not changed, the TLS algorithm modifies the steps of the standard MCTS algorithm in the following way.

Selection The selection of optimal actions in the action trees follows the UCT strategy until a leaf is reached. Each internal node constraints the range of actions from which the algorithm can sample. According to these constraints, the algorithm samples an action a when a leaf is encountered.

Expansion Each node that is added to the tree is connected to a new and empty action tree that represents the next action in the game.

Backpropagation In addition to the update of the visitation count and the total reward, the algorithm also updates the statistics contained in the leaves of the action trees.

TLS has been successfully applied to one-step optimization problems such as function optimization [46]. For multi-step problems, it suffers from a number of problems caused by splitting in the action tree. The reason for these problems is that a split in the action tree is a split of an internal node in the complete tree constructed by TLS. There is a trade-off here between the information contained in the subtree starting from this node and the computational effort required to reuse this information. Deleting the subtree erases all information but has the advantage of being computationally cheap, while duplicating the subtree keeps all information but has the disadvantages of containing moves which become illegal or states which become impossible to reach after a split and of being computationally expensive. Restructuring the tree provides another possibility but also requires additional computation time and space. A study of different restructuring techniques for multi-step problems is given in a recently published master thesis [22].

2.5 Bounding the Horizon Time

This section recapitulates the work of Kearns et al. [21] on Markov decision processes (MDP) with large state spaces. Their study derives theoretical bounds for the depth and the width of a tree required in a tree-based algorithm to generate policies for the agent's behavior for which the value function comes arbitrarily close to the optimal value function (see Section 2.3).

The maximum depth of the tree constructed by MCTS is usually bounded by the maximum number of steps until a terminal state is reached. In addition, this bound determines the depth of the simulation phase. Nevertheless, in some environments, terminal states might not exist or take a very long time to be reached by executing randomly selected actions. This problem is a typical characteristic of domains with an infinite or very large planning horizon. Reasonable bounds for the maximum depth of the COMCTS tree are derived in the following paragraphs.

Because MCTS is based on stochastic simulations, the policies created by the algorithm only approximate the optimal value function. A criterion which ensures that these policies come arbitrarily close to this function is *near-optimality*, defined by Kearns et al. [21] as follows.

Definition 1. *(Near-Optimality) The value function of the policy generated by a tree search algorithm $\mathcal{A}$ satisfies*

$$|V^A(s) - V^*(s)| \leq \epsilon \tag{2.16}$$

simultaneously for all states $s \in S$.

In words, near-optimality guarantees that the difference between the value function of the strategy implemented by a tree search algorithm $V^A(s)$ and the optimal value function $V^*(s)$ is not larger than a very small quantity ϵ. To satisfy near-optimality, the following assumption needs to be made.

Assumption 1. *Rewards generated by the reward function $R(s, a)$ are bounded in absolute value by R_{max}.*

Given this assumption, the required depth H of the tree is selected according to ϵ-horizon time which is computed as [21]

$$H = \log_\gamma \left[\frac{\epsilon(1-\gamma)}{R_{\max}} \right]. \tag{2.17}$$

In words, ϵ-horizon time guarantees that the discounted sum of rewards that is accumulated by considering rewards beyond the horizon H is bounded by ϵ. The proof that leads to Equation 2.17 is given by Kearns et al. [21].

In addition to bounding the required depth of the tree, ϵ-horizon time also delivers a bound for the simulation step in MCTS.

2.6 Continuous Observations

The majority of algorithms for POMDPs assumes the observation space to be discrete. In the real world, however, observations recorded by sensors such

as video cameras, microphones and laser-range finders often provide continuous observation signals. User modeling, event recognition and spoken-dialog systems [37] are examples of applications where the sensor readings result in continuous observations while the states of the underlying system can be represented as discrete features.

Algorithms for POMDPs with discrete observations can normally not be simply extended to continuous observations. The reason for this is that a continuous space consists of uncountably many values and most algorithms require to explicitly consider all possible observations. The usual approach to this problem is a priori discretization of the observation space. Another approach developed by Hoey and Poupart [16] is based on conditional planning. This approach derives a loss-less partitioning of the observation space by considering only those observations that change the agent's policy. Porta et al. [33] extend PERSEUS, a point-based value iteration algorithm originally introduced for discrete state spaces by Spaan and Vlassis [42], to POMDPs with continuous state spaces. This extension is done by using Gaussian mixture models [4] for the representation of the observation, transition and reward models, and by using Gaussian mixture models or particle sets for the representation of the agent's belief state. Given these models, the equations required to perform PERSEUS can be computed in closed form. This extension of PERSEUS can also handle continuous observations by using a sampling approach that is based on the partitioning strategy for the observation space derived by Hoey and Poupart [16].

The algorithm presented in the next section is an extension of POMCP that allows to deal with POMDPs that have a continuous observation space. In contrast to the exact offline planning algorithms presented above, this algorithm is approximate and based on planning online.

Chapter 3

Continuous Observations Monte Carlo Tree Search

POMCP is a state-of-the-art algorithm for POMDPs with discrete observation spaces. Continuous Observations Monte Carlo Tree Search (COMCTS) is a novel online planning algorithm that combines POMCP with incremental regression tree induction, and allows to automatically discretize a continuous observation space. The main idea behind COMCTS is to replace the edges (corresponding to observations) emerging from the sequence nodes of the POMCP tree by so-called *observation trees* which provide discretizations of the observation space. Given the current history h^t, the algorithm estimates the value function $Q(h^t, a)$. As other Monte Carlo methods for POMDPs, the algorithm employs a black-box simulator $\mathcal{G}$ (see Section 2.4).

The remainder of this chapter is structured as follows. Section 3.1 describes the algorithm. Observation trees are discussed in Section 3.2. Strategies to refine the discretizations provided by these trees are given in Section 3.3. Section 3.4 analyzes the algorithm's computational complexity. Finally, a brief overview of related work in Section 3.5 completes the chapter.

3.1 Algorithm

The search tree constructed by COMCTS is illustrated in Figure 3.1. This tree alternates two layers of nodes. The first layer consists of standard MCTS nodes that correspond to possible histories, so-called *history nodes*, or to possible sequences, so-called *sequence nodes*. The second layer consists of tree-based discretizations of the observation space, so-called *observation trees*.

The algorithm is shown in Figure 3.2. Each iteration starts from the current history, $\hat{h}^0 = h^t$. First, an initial state s is drawn from the belief state b^t. The simulator $\mathcal{G}$ is then set to state s. Actions are selected according to UCT until a leaf of search the tree is reached. The observation trees on the way to the leaf are traversed by using observations sampled from the black-box simulator given the selected actions as input. One action available at the leaf is selected, and a new history node corresponding to that action is added to the tree. From here on, actions are selected according to a simulation strategy until the discount horizon H is reached. Each selected action is passed to the black-box simulator

to generate a sequence of observations and rewards. After the simulation, the tree is updated based on this sequence.

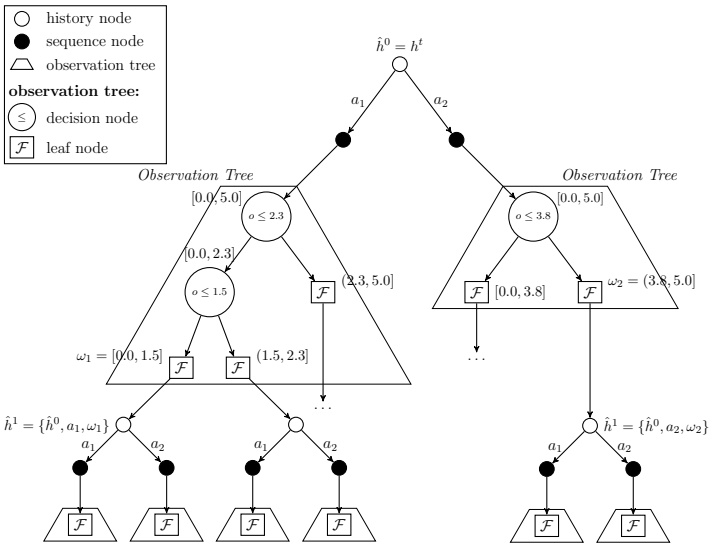


Figure 3.1: An example of the tree constructed by COMCTS.

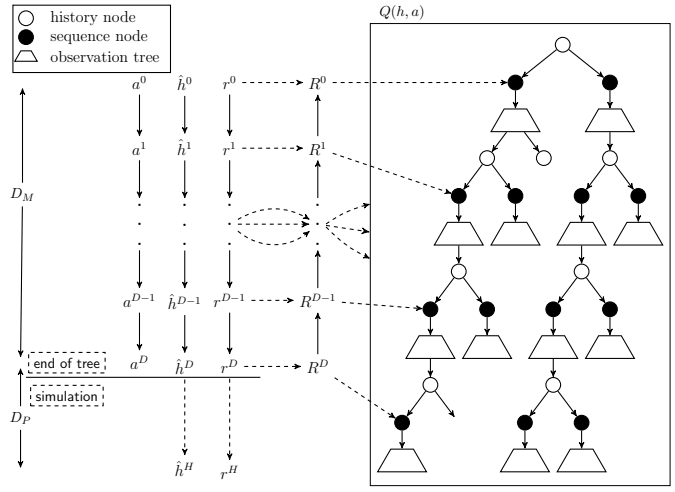


Figure 3.2: A single iteration of COMCTS.

In principle, the algorithm follows the same algorithmic steps as POMCP. While the simulation step is not changed, the modifications to the other steps are given as follows.

Selection This step is extended by a policy for the traversal of the observation trees. Each time the algorithm selects a sequence node, the incoming action a is given to the black-box simulator $\mathcal{G}$ to return a successor state

s', a reward r' and - most importantly for this step of the algorithm - an observation o', $\mathcal{G}(s,a) \rightsquigarrow (s', o', r')$. The observation o' is used to traverse the observation tree that is linked to the currently selected sequence node. Each traversal follows the constraints given by the internal nodes of the observation tree and ends at one of its leaves from which a further sequence node can be selected again and used to sample another observation o'' to traverse the next observation tree. This alternation between the selection of sequence nodes and the traversal of observation trees repeats until an unexplored sequence node is found.

Expansion The unexplored sequence node found in the previous step is added to the COMCTS tree and connected to a new observation tree that consists of a single leaf node. This leaf node represents the complete range of the observation space.

Backpropagation All observations sampled in the selection step and all rewards $(r^0, r^1, \ldots, r^H)$ sampled during both the selection and the simulation step are used to update the tree from the new sequence node to the root of the search tree. The sequence and history nodes are updated in the same way as in MCTS, while the sampled observations are used to update one leaf in each observation tree encountered on the way to the root. To find the correct leaf node in an observation tree, a sampled observation follows the constraints given by the internal nodes of this tree. The basis for the update of some node at depth k of the COMCTS tree is the following simulated return.

$$\hat{R}^k = \gamma^k r^k + \gamma^{k+1} r^{k+1} + \ldots + \gamma^H r^H \tag{3.1}$$

Here, H is the summed depth of the selection and simulation steps (see Figure 3.2), and is given by ϵ-horizon time.

3.2 Learning the Observation Trees

An example of an observation tree is presented in Figure 3.3. Except for its leaves, this tree can be seen as a binary decision tree [28] in which the internal nodes subdivide the observation space into different regions. Each internal node v specifies a *decision* that decides whether a sample observation o passed to the tree belongs to the region of the observation space represented by the left child of v, $o \leq \theta^i$. Here, θ^i is a (split) point of dimension i of the observation function. If the condition $o \leq \theta^i$ holds, then o is passed to the left child of v. Otherwise, o is passed to the right child of v. Usually, external nodes in a decision tree contain the classification for an example that is passed through the tree [28]. In COMCTS, however, each leaf $L(\omega, \mathcal{F})$ of an observation tree T is associated with an interval $\omega = [o_{\min}, o_{\max}]$ of the observation function. The set of all intervals $\Omega = \{\omega_1, \omega_2, \ldots\}$ serves as a discretization of the observation function. In addition, leaves in the observation tree store a set of tests $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots\}$ that is used to decide when the leaf is split. The idea for these statistics is taken from Ikonomovska and Gama [17]. A test $\mathcal{F}_i$ for some leaf L is represented by the following set.

$$\mathcal{F}_i = \left(\theta, \{n_p, n_f, v_p, v_f, v_p^2, v_f^2\}\right) \tag{3.2}$$

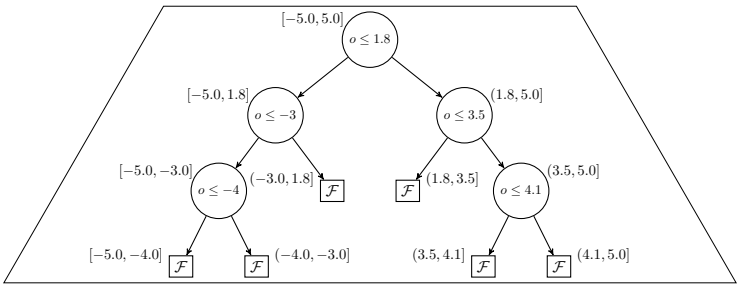


Figure 3.3: An observation tree of COMCTS.

where $\theta \in \omega$ is a possible split point and the remainder of the set is itself a set that is composed of six statistics. Each time a sample observation o reaches L, the sample is classified as either passing or failing the test F_i. The test condition is $o \leq \theta$. The statistics n_p and n_f count the number of examples that reach leaf L, v_p and v_f sum the mean return and v_p^2 and v_f^2 sum the squared mean return of these examples (p denotes passing and f denotes failing the test). These six statistics are sufficient to determine whether the split of a leaf would provide a significant reduction of the variance of the mean return that is accumulated by the examples assigned to leaf L. To make this decision, a standard F-test is used [12]. The F-statistic is defined as follows.

$$F = \frac{\left(\frac{\mathrm{RSS}_1 - \mathrm{RSS}_2}{p_2 - p_1}\right)}{\frac{\mathrm{RSS}_2}{n - p_2}} \tag{3.3}$$

where p_i is the number of parameters used in the regression model, n is the number of data points, and RSS_i is the residual sum of squares, given as

$$\mathrm{RSS}_i = \sum_{j=1}^{n} \left[V_j(o, L) - \bar{V}_j(o, L)\right]^2 \tag{3.4}$$

where $V_j(o, L)$ denotes the value of the variable to be predicted (the value of observation o at leaf node L), and $\bar{V}_j(o, L)$ denotes the predicted value of $V_j(o, L)$ (the average of o's value at L). The F-statistic can be simply computed from the set of the six statistics contained in $\mathcal{F}_i$.

3.3 Splitting Strategies

COMCTS suffers from the same issues when splitting a leaf in the observation tree as TLS (see Section 2.4.5). A leaf in the observation tree is an internal node in the complete search tree constructed by the algorithm. When such a leaf is split, the question comes up what to do with the subtree below that leaf. Requesting new samples consumes additional computation time because each new sample requires additional black-box simulations. In particular domains, the number of samples available to the algorithm might also be limited. Van den Broeck and Driessens [46] and Kirchhart [22] study several strategies that can be followed to reduce the amount of information loss in case of a split in

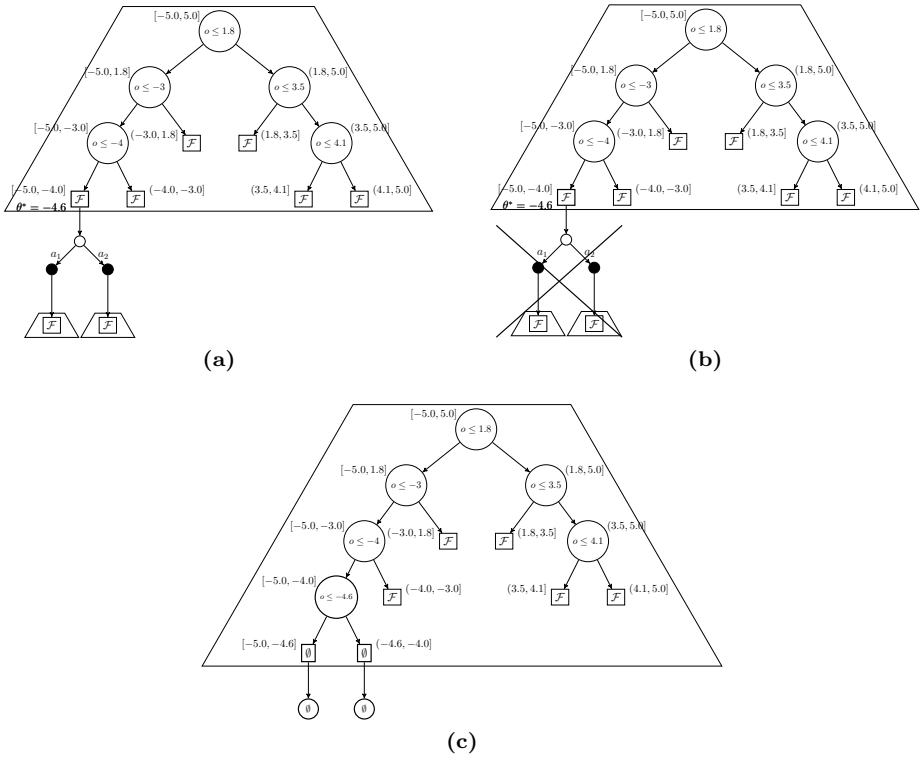


Figure 3.4: An illustration of the deletion strategy. The split point $\theta^* = -4.6$ is found to be significant (**a**). In a first step, the subtree below the split node is removed (**b**). In a second step, this leaf node is replaced by a new internal node which represents the constraint $o \leq -4.6$, and two new leaves are created below the new internal node (**c**).

TLS. A subset of these strategies, consisting of deletion, perfect recall, local recall, and belief-based reuse of knowledge, is used in COMCTS and presented in the following four subsections.

3.3.1 Deletion

Deletion is illustrated in Figure 3.4, and consists of two steps. The first step is to delete the subtree below the split node. The second step is to replace the split node with an internal node for which the constraint is based on the split point, and to create two new leaf nodes below that new internal node.
Deletion is the most inexpensive of the four strategies with respect to time and memory consumption because it does not require to store any samples. The major drawback of this strategy is the loss of possibly valuable information about the expected value of actions at future time steps that is contained in the deleted subtree. The following strategies try to counteract this drawback.

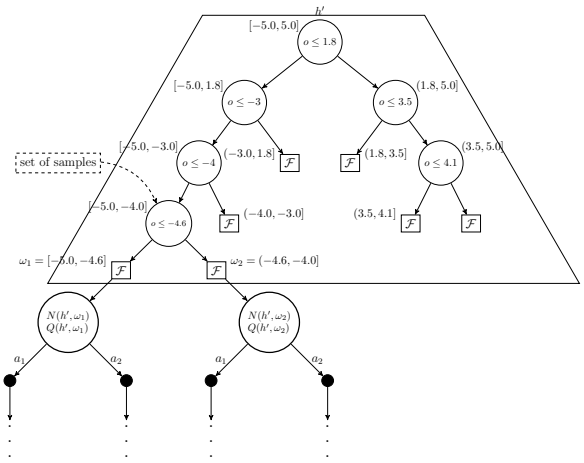


Figure 3.5: Perfect recall maintains a set of samples. In case of a split, this strategy follows the same steps as deletion. After those steps, the subtrees below the split node are recreated based on the set of stored samples.

3.3.2 Perfect Recall

Perfect recall reconstructs the tree by maintaining all samples that passed through the tree in a set $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$. In each iteration of the algorithm, a sample ψ_i is stored as a sequence of action-observation-return tuples, given as

$$\psi_i = \{(a^0, o^0, R^0), (a^1, o^1, R^1), \ldots, (a^H, o^H, R^H)\} \tag{3.5}$$

In the case of a split, perfect recall performs the same two steps as deletion. A third step, illustrated in Figure 3.5, reconstructs the subtrees below the new leaf nodes from the set of samples Ψ. In each step of the reconstruction process, a sample $\psi \in \Psi$ is taken and the new internal node w decides whether this sample ψ belongs to the left or the right child of w. The observation o^0 and the return R^0 contained in the first tuple of ψ are used to create and update a new set of tests for that child node. The subtree of that child node is reconstructed based on the subsequent tuples of ψ.
Perfect recall is computationally very demanding in both time and memory because all samples that passed through the tree need to be stored and because these samples need to be processed through the tree in case of a split. The advantage of this strategy is that no information is discarded from the subtree that existed before the split.

3.3.3 Local Recall

Local recall reconstructs the tree locally, i.e., on the level of the observation tree but not on deeper levels. All samples that passed through the split node are remembered as a set of observation-return pairs Φ. In the case of a split, local recall follows the same two steps as deletion. A third step, depicted in Figure 3.6, uses the set of samples Φ to create and update a new set of tests $\mathcal{F}$ for each new leaf node of the observation tree, and also adds a history node below each of

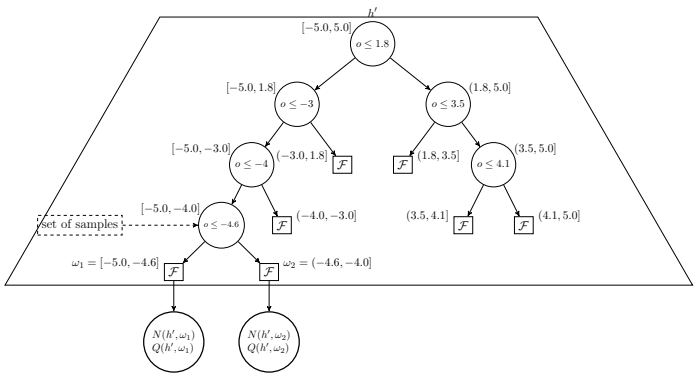


Figure 3.6: Local recall maintains a set of samples. In case of a split, this strategy follows the same steps as deletion. After those steps, the first level of the subtrees below the split node is recreated based on the set of stored samples.

these leaves. Each step of the reconstruction process takes a sample $\phi = (o, R)$ from the set Φ and passes it to the new internal node w to decide whether the sample is passed to the left or the right child node of w. The sample is then used to create and update tests for that child node and to update the history node below that child node.

Before the split, the history nodes below the split node v keep the count $N(h', \omega)$ and the value $Q(h', \omega)$ for some simulated history h' that lead to the observation tree containing v. After the split and applying local recall, the history nodes below the new leaf nodes keep the count and the value for the intervals of the new leaf nodes, i.e., $N(h', \omega_1)$ and $Q(h', \omega_1)$ for the left leaf node's interval ω_1 and $N(h', \omega_2)$ and $Q(h', \omega_2)$ for the right leaf node's interval ω_2.

The advantage of this strategy is that the new nodes do not need to start learning with empty statistics. The main disadvantage is that only the first level of the subtree below the split node is rebuild and that all deeper levels still need to be build again. This strategy has a higher memory and time demand than deletion but is still cheaper than perfect recall.

3.3.4 Belief-Based Reuse of Knowledge

Reuse takes advantage of the Markov property of the agent's belief state. The choice of a good action does only depend on the current belief state and not on any additional information from the past. Particular choices of actions together with particular observations perceived for these actions can result in the same belief. An important example for this are uninformative observations which cause the Bayesian update (see Section 2.2) to uniformly distribute the belief over the state space - no matter which action was executed by the agent. As long as a good policy is known for a specific region in the belief space, the agent can reuse this policy whenever its belief enters this region.

In this study, reuse is restricted to a two-dimensional belief space to allow the following computations. In case of a split, the range in the belief space for each of the new leaf nodes is determined. If these ranges are (nearly) similar,

then splitting is not necessary. To measure the similarity between two ranges, the Euclidean or Manhattan distance over the end points of the ranges can be taken. This distance is then compared to a specific threshold to decide whether splitting is performed.

3.4 Computational Complexity

This section analyzes the time and space bounds of COMCTS. The terms time and running time refer to worst-case running time.

3.4.1 Running Time

In order to analyze the running time of COMCTS, a number of definitions need to be made. These definitions are supported by Figure 3.2. The maximum depth of the search tree M constructed by the algorithm is denoted by D_M, and is defined to be the maximum number of history nodes between the root and any leaf node of the tree. The depth of the simulation step is computed by $D_P = H - D_M$, and is defined to be the number of actions selected from the beginning of the simulation until the end of the simulation, represented by the discount horizon H.

The following definition is supported by Figure 3.3. The depth of some observation tree W is denoted by D_W, and is defined to be the maximum number of nodes between the root and any leaf of W.

Given the definitions above, the running times for each step of the algorithm are summarized in Table 3.1. While the time spend for the simulation step is simply $\mathcal{O}(D_P)$, the times spend for the other steps of the algorithm are derived in detail in the following paragraphs.

In the selection step, the algorithm needs to repeat two traversals until a leaf node is reached. The reason for these traversals is that the COMCTS tree alternates layers of history and sequence nodes with layers of observation trees. The running time for the traversal of history nodes is $\mathcal{O}(|A|)$, where A is the action set of the underlying POMDP, because the algorithm needs to iterate over all children of a history node to select one of them according to UCT. The time required for the traversal of a single observation tree W is $\mathcal{O}(D_W)$. The total running time of the selection step is given as

$$\mathcal{O}(D_M * |A| + D_M * D_W) \tag{3.6}$$

The expansion step spends $\mathcal{O}(1)$ time to add a new sequence node to the search tree and $\mathcal{O}(1)$ time to associate an observation tree containing a single leaf node

Phase	**Running Time**
Selection	$\mathcal{O}(D_M * \|A\| + D_M * D_W)$
Expansion	$\mathcal{O}(1)$
Simulation	$\mathcal{O}(D_P)$
Backpropagation	$\mathcal{O}(D_M * D_W + \sum_{i=1}^{D_M} u_i)$

Table 3.1: The running times of the four phases of COMCTS.

to this new sequence node. Thus, the total time required for this step is $\mathcal{O}(1)$.

The backpropagation step consists of the traversal from the sequence node that is added in the expansion step to the root of the tree. This traversal includes all history and sequence nodes and all observation trees that lie on the path between the new sequence node and the root of the search tree, and runs in $\mathcal{O}(D_M)$ time. While updating history and sequence nodes only requires $\mathcal{O}(1)$ time per node, updating some leaf node of an observation tree W requires to traverse W according to the constraints given by the internal nodes of W. This traversal consumes $\mathcal{O}(D_W)$ time. Creating the first test for a split point at a leaf L in the observation tree also runs in $\mathcal{O}(1)$ but creating further tests requires significantly more time. The reason for this is that the statistics of a new test need to be consistent with the statistics of all other tests maintained at L. The algorithm therefore needs to find the closest test for which it spends $\mathcal{O}(|\mathcal{F}|)$ where $\mathcal{F}$ is the set of tests at L. The time required for updating the tests is also proportional to the number of tests $|\mathcal{F}|$, and is given as $\mathcal{O}(|\mathcal{F}|)$. To check whether one of the tests corresponds to a significant split point, all tests available at L are considered which again results in a time of $\mathcal{O}(|\mathcal{F}|)$. Without taking the time spend for the splitting strategy into account, the total running time of the backpropagation step is

$$\mathcal{O}(D_M * D_W + \sum_{i=1}^{D_M} u_i) \tag{3.7}$$

where $u_i = D_W + 3 * |\mathcal{F}|$ is the time spend for updating a leaf in observation tree i.

In the case of a split, the time spend to update the new leaf nodes depends on the splitting strategy. Deletion runs in $\mathcal{O}(1)$ time because it does not perform any further operations than replacing the leaf node by an internal node and inserting two new leaf nodes. Based on a set of stored samples Φ, local recall creates and updates split tests for the new leaf nodes, and also updates the values maintained at the history nodes below the new leaf nodes. Updating and creating tests requires the same time that is given in the previous paragraph, and iterating through all stored samples requires $\mathcal{O}(|\Phi|)$. Perfect recall also uses a set of stored samples Ψ. Iterating over all samples runs in $\mathcal{O}(|\Psi|)$ time. Recalling one sample $\psi \in \Psi$ has a running time of $\mathcal{O}(|\psi|)$ because the algorithm considers all action-observation-reward tuples in ψ for the reconstruction of the subtrees. Thus, the total running time of the perfect recall strategy is $\mathcal{O}(\sum_{i=1}^{|\Psi|} |\psi|)$.

3.4.2 Space Bounds

The number of nodes of a single observation tree W depends on the number of splits n performed in that observation tree, $|W| = 2^n + 1$. The total number of nodes in all observation trees is given as

$$\sum_{i=1}^{m} |W_i| \tag{3.8}$$

where m is the number of observation trees and $|W_i|$ is the number of nodes in observation tree i.

The number of history nodes connected to some leaf of an observation tree is at most equal to the number of actions in the action set A of the underlying POMDP. Thus, the total number of nodes in the search tree M constructed by COMCTS is at most

$$|M| = \sum_{i=1}^{m} |W_i| + \sum_{i=1}^{m} ((|W_i| - 1) * |A|). \tag{3.9}$$

The space bounds for the splitting strategies vary from no bounds for deletion to $\mathcal{O}(|\Phi|)$ and $\mathcal{O}(|\Psi|)$ for local and perfect recall. The reasons for these bounds are that deletion does not store anything while the two other strategies store a set of samples.

3.5 Related Work

Rather than looking at COMCTS as an extension of POMCP to POMDPs with continuous observation spaces, it can also be seen as a direct extension of MCTS. From this point of view, COMCTS associates each state node of the MCTS tree with a tree-based discretization of the observation space. The selection and backpropagation steps of MCTS are extended by a policy for the traversal of those trees. In addition to the usual update of the state nodes in MCTS, the backpropagation step needs to update the tests maintained at the leaf nodes of those trees.

For an overview of offline planning algorithms that are able to handle POMDPs with continuous observation spaces, the reader is referred back to Section 2.6.

Chapter 4

Belief Based Function Approximation

Belief based Function Approximation using Incremental Regression tree techniques and Monte Carlo (BFAIRMC) is a novel online planning algorithm that is based on a combination of incremental regression tree learning in the belief space and Monte Carlo rollouts. As in other Monte Carlo methods, a simulator $\mathcal{G}$ is used as a black-box model of the underlying POMDP.

Given the current belief state b^t, BFAIRMC approximates the value function $Q(b^t, a)$. The algorithm consists of a two layer architecture. The first layer is called *belief tree* and builds a discretization of the belief space by finding decision rules that subdivide this space into different regions. Each leaf in the belief tree corresponds to a specific region $\mathcal{B}$ in the belief space. The second layer is composed of a set of *action nodes* for each leaf of the belief tree. Each of these action nodes stores the visitation count $N(\mathcal{B}, a)$ and the average return $Q(\mathcal{B}, a)$ for some action a in region $\mathcal{B}$.

4.1 Algorithm

BFAIRMC is illustrated in Figure 4.1. The algorithm maintains a simulated belief state $\hat{b}(s)$. Each simulation starts from the current belief state, $\hat{b}^0(s) = b^t(s)$. An initial state s is drawn from this belief state, and the simulator $\mathcal{G}$ is set to state s. Each iteration can be divided into two steps: simulation and update. The first step follows the constraints contained in the internal nodes of the belief tree to find the leaf L that corresponds to the range of the belief space to which $\hat{b}(s)$ belongs. An action a at leaf L is selected according to UCT, and passed to the black-box simulator to sample an observation o and a reward r. From the simulated belief $\hat{b}(s)$, the action a selected at L, and the sampled reward r, a sample $\chi = (\hat{b}(s), a, r)$ is build. Based on action a and observation o, the third step updates the simulated belief $\hat{b}(s)$, e.g., according to Bayes rule (see Section 2.2). Then, the algorithm goes back to the first step.

After the discount horizon H is reached, the algorithm continues with the second step that updates the tree with all samples encountered during the simulation. This update starts with the last and ends with the first of those samples. The basis for this update is the following simulated return, computed from the

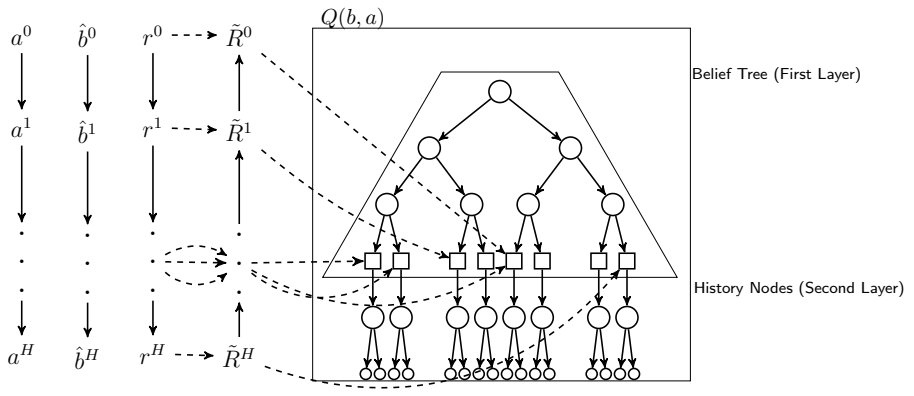


Figure 4.1: A single iteration of BFAIRMC.

rewards sampled during the simulation.

$$R^k = r^k + \gamma r^{k+1} + \gamma^2 r^{k+2} + \cdots + \gamma^H r^H \tag{4.1}$$

where k denotes the depth of the simulation step at which r^k is acquired.

For each sample $\chi = (\hat{b}(s), a, r)$, the algorithm determines the leaf node of the belief tree that represents the region in the belief space to which $\hat{b}(s)$ belongs. This leaf node is found by following the constraints given by the internal nodes of the belief tree. The statistics of the tests maintained at this leaf node and the value of action a maintained at the action nodes below this leaf are then updated according to a specific update rule. This update rule applies bootstrapping [43], i.e., updating the estimate of a value on the basis of other estimates of that value, and is defined in the following equation.

$$\tilde{R^k} = R^k + \gamma^{H+1-k} V(b^{k+1}) \tag{4.2}$$

Here, $V(b^{k+1})$ denotes the bootstrapping term and is given by the maximum average return stored at the action nodes below the leaf node that represents the region of the belief space to which b^{k+1} belongs. Formally, $V(b^{k+1})$ is given by

$$V(b^{k+1}) = \max_a Q(b^{k+1}, a) \tag{4.3}$$

BFAIRMC is similar to Q-Learning [48] for POMDPs which also approximates the value function $Q(b^t, a)$ because it follows similar steps as BFAIRMC. Before diving into the details of how Q-Learning can be used for POMDPs, "standard" Q-Learning for fully observable environments is explained.

Q-Learning for fully observable environments estimates the value function $Q(s^t, a)$. Each iteration of Q-Learning can be divided into a policy evaluation and a policy iteration step. In the first step, the agent observes the current state s^t, selects an action a^t according to some policy that is based on the current estimate of $Q(s^t, a^t)$, executes the action in the environment to receive a reward r^{t+1}, and observes the next state s'. In the second step, the value function is updated according to the following update rule.

$$Q(s^t, a^t) = Q(s^t, a^t) + \alpha \times \left[r^{t+1} + \gamma \times \max_a Q(s^{t+1}, a) - Q(s^t, a^t) \right] \tag{4.4}$$

where α is a learning rate and γ is a discount factor for the expected value of the next state. Q-Learning only updates the value of the selected action. The values for all other actions stay the same. Compared to the update rule of BFAIRMC that considers multiple future rewards, this update rule only takes the immediate reward and the expected value of the next state, expressed by the term $\max_a Q(s^{t+1}, a)$, into account.

Q-Learning can be transformed into an online planning algorithm for POMDPs by employing a black-box simulator $\mathcal{G}$. At the beginning of the policy evaluation step, $\mathcal{G}$ is set to a sample state s drawn from the current belief state b^t. An action a is selected in the same way as before, and thrown into $\mathcal{G}$ to receive a sample of a reward, observation and next state, (r', s', o'). In the policy iteration step, this sample is then used to update the value function.

BFAIRQ replaces the original update rule of BFAIRMC with the update rule of Q-Learning. The algorithm also follows the same steps as Q-Learning but applies UCT to select the actions during the policy evaluation step. To achieve comparability between the two algorithms, BFAIRQ performs as many steps as BFAIRMC in each iteration, given by horizon H.

4.2 Learning the Belief Tree

Similar to the observation trees in COMCTS, the belief tree can be seen as a binary decision tree [28] in which the internal nodes subdivide the belief space into different regions. Each internal node v specifies a *decision* of the form $\hat{b} \leq \theta^i$ where $\hat{b}$ is the simulated belief given to the belief tree, and θ^i is a (split) point of the i-th dimension of the belief space. If the condition $\hat{b} \leq \theta^i$ holds, then $\hat{b}$ is passed to the left child of v. Otherwise, $\hat{b}$ is passed to the right child of v. Each leaf of the belief tree keeps a set of tests $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \dots\}$ that determines when the leaf is split, and that is similar to the set of tests used in COMCTS. In contrast to COMCTS where tests consist of statistics based on **all** actions, the tests tried out for BFAIRMC in this thesis consist of statistics based on **each** action. Such a test $\mathcal{F}_i$ is defined as

$$\mathcal{F}_i = (\theta, \{\mathcal{F}_i^{a_1}, \mathcal{F}_i^{a_2}, \dots, \mathcal{F}_i^{a_m}\}) \tag{4.5}$$

where m is the number of actions and $\mathcal{F}_i^{a_i}$ is the set of statistics for action a_i, containing the same six measures that are used for the tests in the observation tree in COMCTS. A leaf is split when there is a significant difference in the return accumulated by any of the actions. As in COMCTS, a standard F-test [12] is used to perform this check (see Section 3.2).

The belief space of a POMDP typically consists of many more dimensions than the observation space. Finding beneficial split points in a high dimensional belief space is not a simple task. Therefore, this thesis restricts to settings where the belief can be summarized in a compact form. In particular, two variants of BFAIRMC are used. The first variant is restricted to beliefs that can be summarized in one probability. The second variant is able to handle higher dimensional belief spaces that can be approximated by a bivariate Gaussian distribution with three parameters, $b^t(s^t) \sim \mathcal{N}(\mu, \Sigma)$. Here, the mean vector is $\mu = (\mu_X, \mu_Y)$ and the covariance matrix is

$$\Sigma = \begin{pmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{pmatrix}$$

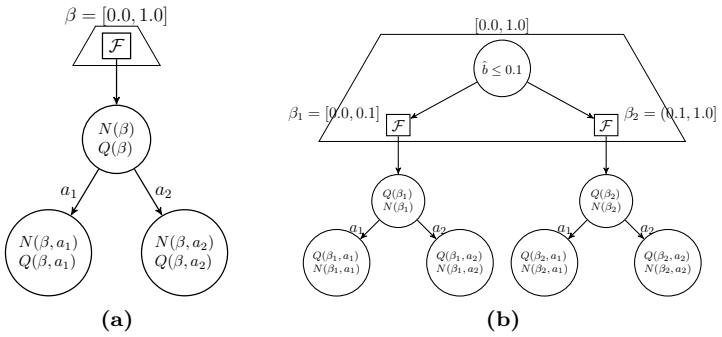


Figure 4.2: Splitting in the belief tree. The split point found to be significant is $b(s) = 0.1$.

where the random variables X and Y have the same standard deviation σ.

4.3 Splitting Strategies

The splitting process in BFAIRMC is shown in Figure 4.2. The leaf node that is selected for splitting is replaced by an internal decision node whose key value is the splitting point that is found to be significant. Two new leaf nodes are created and connected to the new internal node. Splitting causes the same issues as in COMCTS and TLS. Three different strategies to tackle these issues are discussed in this study: deletion, insertion and perfect recall.

4.3.1 Deletion

Deletion is equivalent to the deletion strategy for COMCTS. The algorithm assigns empty statistics to the new leaves, and empty values to the new action nodes of the second layer. Therefore, the disadvantage of this strategy is that all knowledge gathered previously is lost. Because no samples need to be maintained, the advantage of this strategy is that it is computationally cheap in both time and memory.

4.3.2 Insertion

Insertion takes advantage of the information contained in the winning test $\mathcal{F}^*$ that determines the split point. Before a leaf node in the belief tree is split, it represents a region in the belief space that is given by some interval ω. After a split, each new leaf node represents a subregion of this region, $\tilde{\omega} \subset \omega$, defined by the split point θ. An action-based test $\mathcal{F}_i = (\theta, \{\mathcal{F}_i^{a_1}, \mathcal{F}_i^{a_2}, \ldots, \mathcal{F}_i^{a_m}\})$ contains the count and the summed return for each action a in $\tilde{\omega}$. In case of a split, insertion follows the same steps as deletion. An additional step incurs the count and the average return of each action from the winning test $\mathcal{F}^*$ to the new action nodes. Figure 4.3 illustrates this step.

Insertion has two main advantages. The first advantage is that the new nodes do not need to start from empty values as is the case in deletion. The second advantage is that no additional memory is demanded because the values

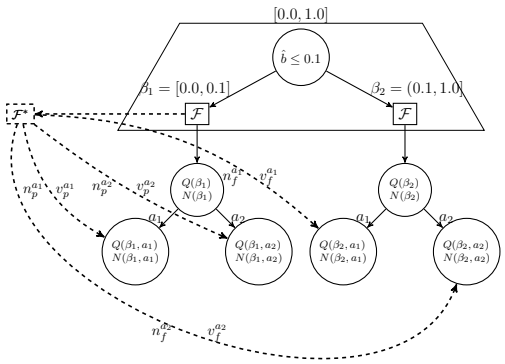


Figure 4.3: The insertion strategy infers the values of the new nodes directly from the winning test.

for the new history nodes are already available in form of the statistics of the winning test. Furthermore, the computation time required by this strategy is nearly equivalent to the time required by deletion because the additional step does not require any complex operations.

4.3.3 Perfect Recall

Perfect recall, depicted in Figure 4.4, is similar to perfect recall for COMCTS but differs in the composition of the samples. The algorithm remembers each sample χ that has been used to update the tree as a tuple of a simulated belief $\hat{b}(s)$, the action a selected for that belief, and the return R used to update the tree, $\chi = (\hat{b}(s), a, R)$. All of those samples are maintained in a set $X = \{\chi_1, \chi_2, \ldots, \chi_n\}$. In the case of a split, the algorithm recalls each sample χ that belongs to one of the new leaf nodes to recreate the values for the new nodes below these leaf nodes. To determine whether a sample $\chi = (\hat{b}(s), a, R)$ belongs to one of the new leaf nodes, it is checked whether the belief $\hat{b}(s)$ fulfills the constraints provided by the internal nodes on the path from the root of the belief tree to the new internal node. If the sample reaches one of the new leaf nodes, the new nodes on the second layer (below the new leaf nodes) are updated in the usual way based on χ.

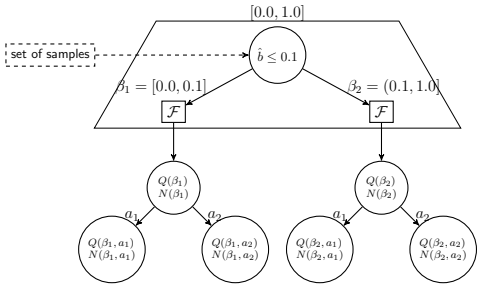


Figure 4.4: The perfect recall strategy maintains a set of samples. In case of a split, this set is used to avoid information loss.

Compared to the previous two strategies, this strategy requires significantly more memory because it needs to store each sample that passed through the tree, and it requires more computation time because each of these samples needs to be processed through the tree in case of a split. The advantage of this strategy is that no information is lost from the subtree that existed before the split.

4.4 Computational Complexity

This section analyzes the time and space bounds of BFAIRMC. The terms time and running time refer to worst-case running time.

4.4.1 Running Time

A single iteration of BFAIRMC consists of three steps relevant for its running time: traversing the tree until a leaf is found, recomputing the simulated belief state $\hat{b}^t$, and updating a specific leaf node of the tree. The time spend to traverse the belief tree U until a leaf is found is proportional to the tree's depth D_U, and is expressed as $\mathcal{O}(D_U)$. Recomputing the belief state $\hat{b}^t$ requires the time that is needed for an exact belief state update (see Section 2.2), $\mathcal{O}(|S|^2)$, where $|S|$ denotes the number of states of the underlying POMDP. Updating a specific leaf node of the tree runs in the same time that is spend for updating a leaf in some observation tree in COMCTS (see Section 3.4.1). The running times for each step of BFAIRMC are summarized in Table 4.1.

As in COMCTS, the time consumed for each splitting strategy differs. The time spend for the deletion strategy is equivalent to the time spent for the same strategy in COMCTS, i.e., $\mathcal{O}(1)$. The insertion strategy incurs the statistics of the test that determines the split point to the new nodes. This strategy thus runs in $\mathcal{O}(|A|)$ time where A is the action set of the POMDP for which the algorithm is applied. The perfect recall strategy consists of two operations. Iterating through the set of stored samples X depends on the number of samples, and requires $\mathcal{O}(|X|)$ time. Updating the action value for an action node of the second layer in one of these iterations costs $\mathcal{O}(1)$ time. Updating a leaf's test statistics requires the same time as updating a leaf in some observation tree in COMCTS (see Section 3.4.1)

Phase	**Running Time**
Selection of a leaf node	$\mathcal{O}(D_U)$
Recomputation of belief state	$\mathcal{O}(\|S\|^2)$
Update of a leaf node	$\mathcal{O}(D_U + 3 * \|\mathcal{F}\|)$

Table 4.1: The running times of the three phases of BFAIRMC.

4.4.2 Space Bounds

The number of leaf nodes in the belief tree depends on the number of splits n, and is calculated in the same way as the number of leaf nodes in a single observation tree is calculated for COMCTS, 2^j. The number of nodes in the belief tree is $2^j + 1$. For each leaf of the belief tree, there are $|A|$ nodes connected

to it on the second layer, where A is the action set of the underlying POMDP. Thus, the total number of nodes in the tree U constructed by BFAIRMC is

$$|U| = 2^j + 1 + 2^j * |A| \tag{4.6}$$

The space bounds for the splitting strategies vary from no bounds for deletion and insertion to $\mathcal{O}(|X|)$ for perfect recall. The reason for the bound of deletion is the same as in COMCTS: it does not remember any samples. Insertion directly takes the action values for the new nodes from the test that determines the split point, and therefore no additional space is demanded. The space bound for perfect recall depends on the number of stored samples $|X|$.

In Table 4.2, the space bounds of BFAIRMC and COMCTS are compared against each other.

Component	**COMCTS**	**BFAIRMC**
Single tree-based discretization	$2^n + 1$	$2^j + 1$
All tree-based discretizations	$\sum_{i=1}^{m} 2^{n_i} + 1$	$2^j + 1$
Complete tree	$\sum_{i=1}^{m} 2^{n_i} + 1 + \sum_{i=1}^{m} (2^{n_i} * \|A\|)$	$2^j + 1 + 2^j * \|A\|$

Table 4.2: Space bounds of COMCTS and BFAIRMC.

4.5 Related Work

A method that builds a similar tree-based discretization of a continuous space as BFAIRMC is Continuous U Tree [45]. This method applies incremental regression tree techniques from Chapman and Kaelbling [7] and is an extension of the U Tree algorithm [27]. While BFAIRMC discretizes the belief space for a POMDP, Continuous U Tree discretizes the (continuous) state space of an MDP through a so-called *state tree* that is structurally similar to the belief tree. Instead of an F-test, a Kolmogorov-Smirnov test or a test based on squared sum error [12] are used. Instead of Monte Carlo rollouts, the estimates for the actions are computed by creating an MDP over the discretization given by a split. This MDP is constructed from a number of samples maintained by the algorithm. Each sample consists of a state information, the action performed for that state information, the resulting state information, and the reward obtained for the corresponding states and the performed action.

In a broader context, BFAIRMC is related to function approximation techniques in reinforcement learning, especially those techniques that are based on decision or regression trees (see, e.g., [34, 47]). Q-Learning [48] with a piecewise constant function approximator also estimates $Q(b, a)$ but differs in the way how it updates this value. Rather than updating $Q(b, a)$ according to the "bottom-to-top" approach in BFAIRMC, Q-Learning updates the value in a "top-to-bottom" approach. BFAIRMC updates the value after a fixed amount of steps in the simulator and starts from the last sample that it found before the discount horizon was reached. Q-Learning updates the value after each step in the simulator and updates the value based on the outcome of the current step and the expected optimal value of the next step. In addition, the update rule of Q-Learning differs from the update rule for BFAIRMC in including a learning rate.

Chapter 5

Empirical Evaluation

The main goal of the experiments is to gain insight into the behavior and performance of the algorithms presented in the previous two chapters. To acquire this insight, Section 5.1 introduces a set of benchmark problems and a set of additional methods against which the algorithms are compared.

Four different experiments were conducted for this thesis. The first two experiments investigate the performance of the algorithms given a specific number of samples or a specific amount of time. The third experiment examines the practical running time of the algorithms. The fourth experiment looks at the approximation of the one-step value function generated by COMCTS. The results of these experiments are presented in Section 5.2.

5.1 Experimental Setup

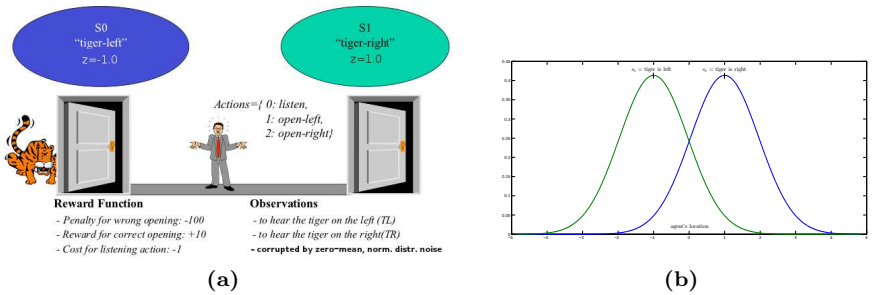
This section presents two environments that are used to analyze the algorithms proposed in this thesis. Each environment is first presented in its discrete form by defining the states, actions, observations and rewards. Then an explanation is given how the observation model is changed from a discrete to a continuous function. An overview over all environments with respect to their complexity is given in Table 5.1. After the presentation of the environments, a set of algorithms is introduced that are used to set the experimental analysis of COMCTS and BFAIRMC into a more general context.

5.1.1 The Tiger Problem

The discrete version of this environment was originally introduced by Kaelbling et al. [18] and has been extended to continuous observations by Hoey and Poupart [16]. Figure 5.1 illustrates this environment. The story describing this

Environment	$\|S\|$	$\|A\|$	$\|O\|$	O Function	O Noisy?
Tiger	2	3	1	1D-Gaussian	yes
Discrete Light Dark	#rows $*$ #columns	4	2	2D-Gaussian	yes
Continuous Light Dark	∞	4	2	2D-Gaussian	yes

Table 5.1: The characteristics of the environments.



(a) **(b)**

Figure 5.1: The Tiger problem with a 1D continuous observation space. **(a)** The characteristics of the environment. **(b)** An example of the observation function where the standard deviation is $\sigma = 0.965$.

famous POMDP goes as follows. A man is situated in a hallway with two doors. One of the doors leads to a small treasure of money, while the other leads to a dangerous tiger.

States: At the beginning, the tiger is randomly placed behind one of the doors. Behind the other door, a small treasure can be found. Thus, the state is determined by the location of the tiger.

Actions: The agent can select one of three actions: open left, open right and listen. When the agent opens one of the doors, the tiger changes its position with probability 0.5. If the agent listens, the tiger stays where it is.

Rewards: The agent receives a reward of $+10$ when it opens the door with the treasure behind it, and suffers a penalty of -100 when it opens the door with the tiger behind it. Listening costs -1.

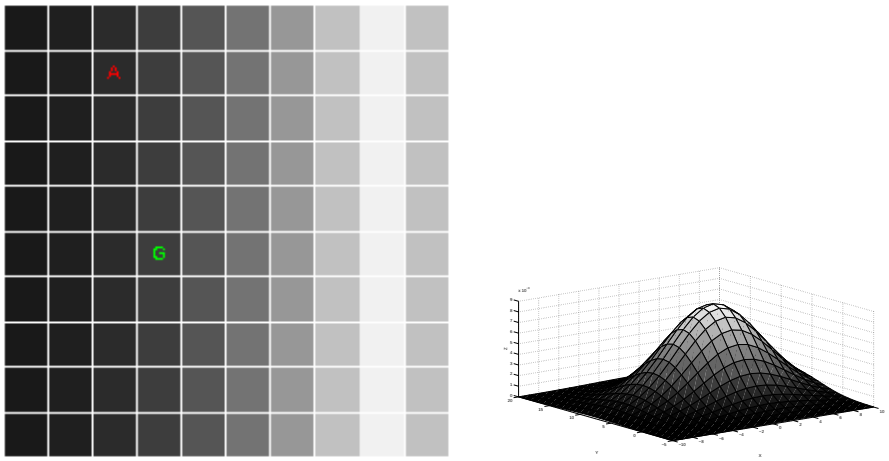
Observations: There are two observations that describe the position of the tiger: left and right. When the agent listens, it receives the correct position of the tiger. When the agent opens one of the doors, it receives either observation with equal probability (independent of which state the agent is in).

The discrete observation signal is extended to the continuous domain in the following way. The doors are located at $z = -1$ (left) and $z = +1$ (right). The agent is at $z = 0$ and has one sensor in form of a binary microphone. This microphone reports the exact location of the tiger but suffers from a lack of accuracy which corrupts the observation signal by some zero-mean, normally distributed noise $\mathcal{N}(0, \sigma^2)$ for which the standard deviation σ is predefined.

The Tiger Problem is not meant to be a model of some real-world domain but rather to be a toy problem that can be used to gain basic insight into the behavior and the performance of the algorithms.

5.1.2 The Light Dark Domain

A variant of this environment with a continuous state space was introduced by Platt et al. [32]. Figure 5.2a presents this environment. The agent is required



(a) **(b)**

Figure 5.2: The discrete state space light-dark domain. **(a)** An example of the environment with a size of 10×10. The agent's location is indicated by the symbol A and the goal's location is indicated by the symbol G. The light is located in the previous to last column. **(b)** An example of the observation function when the agent is located at $(x = 2, y = 8)$.

to localize itself before it can approach the goal. There is a light source somewhere in the environment. The intensity of the light depends on the horizontal coordinate of the agent's location. The closer the agent is to this source, the better is the agent's ability to localize itself. The idea here is that the agent might need to move away from the original goal to be able to perform a sufficient localization.

The environment's continuous state space is transformed to a discrete state space by placing the agent in a grid world with finite distances between each cell. The following lines describe this variant of the light dark environment.

States: The state space consists of all cells of the grid. Special states are the goal state and the agent's starting state.

Actions: The agent can select one of four actions: move up, move down, move left or move right. Actions are deterministic. If an action would end outside of the grid, the agent ends up on the cell on the side which is "opposite" to the cell where it began the movement. For example, when the agent would leave the grid on the right side, it enters again on the left side.

Rewards: The agent suffers a penalty of -1 for each move it makes.

Observations: The observation is the location (x, y) of the agent, corrupted by some zero-mean normally distributed noise $\mathcal{N}(0, \sigma^2)$, with a standard deviation σ that depends on the intensity of the light at the current location of the agent, and is calculated as

$$\sigma = \sqrt{0.5 \times (x_L - x)^2 + K} \tag{5.1}$$

where x_L is the horizontal coordinate of the light source's location, x is the horizontal coordinate of the agent's location and K is a constant that denotes the minimum amount of noise.

A variant of the light-dark domain with a continuous state space is also used in this thesis. This variant is similar to the variant introduced above, but differs in the state space and the actions as follows.

States: The state space is a subspace of the plane $\mathcal{R}^2$ with boundaries $[-1, 7]$ on the horizontal axis and $[-2, 4]$ on the vertical axis. The goal state is the area given by the boundaries $[-1.0, 1.0]$ on the horizontal axis and $[-1.0, 1.0]$ on the vertical axis.

Actions: Actions move the agent by one unit on the grid. This movement is corrupted by some zero-mean normally distributed noise $\mathcal{N}(0, \sigma^2)$, with a fixed standard deviation σ.

5.1.3 Additional Algorithms

To provide a more general analysis, the algorithms proposed in this study need to be compared to existing methods from the literature and to heuristics that are conceptually much simpler than the algorithms. These additional algorithms are introduced in the following paragraphs.

There are two existing methods that the algorithms are compared to. The first method is called **Random**, and selects actions randomly based on a uniform distribution over the set of possible actions. The second method is called **Monte Carlo (MC)**, and selects actions based on the simulation of random rollouts during which the actions are chosen according to a uniform distribution. The action with the highest average return is then selected for execution in the real environment.

The heuristics used in this study are developed for a specific environment and cannot directly be applied to other environments. For the Tiger problem, there are two heuristics. The first heuristic, called **H.1**, relates to the optimal one-step value function for the tiger problem (see Figure 5.13 and [18]). Under H.1, the agent plays according to the following strategy.

$$a^t = \begin{cases} \text{open left door} & \text{if } b^t(s^t = \text{tiger is right}) \geq 0.9 \\ \text{open right door} & \text{if } b^t(s^t = \text{tiger is left}) \geq 0.9 \\ \text{listen} & \text{otherwise.} \end{cases} \tag{5.2}$$

The second heuristic, called **H.2**, is similar to the previous strategy but can request the environment for its actual state s^t. This heuristic assumes that the agent can directly observe the state of the environment. Under H.2, the agent plays according to the following strategy.

$$a^t = \begin{cases} \text{open left door} & \text{if } b^t(s^t = \text{tiger is right}) \geq 0.9 \ \wedge \ s^t = \text{tiger is right} \\ \text{open right door} & \text{if } b^t(s^t = \text{tiger is left}) \geq 0.9 \ \wedge \ s^t = \text{tiger is left} \\ \text{listen} & \text{otherwise.} \end{cases} \tag{5.3}$$

It is possible that the two heuristics relate to bounds on the optimum for the infinite horizon Tiger problem, with H.1 giving a lower bound and H.2 giving

an upper bound. Intuitive reasons for this relation are that H.1 is based on the optimal one-step value function, and that H.2 augments H.1 with full observability. Given partial observability, the agent can obviously never reach the same performance as given full observability. H.1 might thus be a lower bound because sometimes the agent still opens the incorrect door (with the tiger behind it) although the agent is quite certain in which state it is. Playing H.2, the agent will always open the correct door, given that the agent's belief is high enough. The proof for the relation between these heuristics and the bounds on the optimal value function is not the focus of this thesis and is therefore omitted.

For the light-dark domain, the heuristic is called **H.3**. It takes the state s^t with the highest belief $b^t(s^t)$, and considers all successor states that can be reached from s^t by performing one action. For each successor state s', the Euclidean distance between the locations of the goal state and the state s' is calculated. The action that leads the agent to the successor state with the shortest distance is then selected for execution in the real environment. The drawback of this heuristic is that the agent might believe that it is in a certain state with a high probability although in the real world, it is in a completely different state.

5.2 Experimental Results

This section reports the results of four experiments. The first experiment investigates the performance of the algorithms in sample-based settings, the second experiment examines the performance of the algorithms under time-based settings, the third experiment gives an insight into the time consumed by each algorithm, and the fourth experiment looks at the approximation of the one-step value function generated by COMCTS.

5.2.1 Sample-Based Performance

This experiment investigates the performance of COMCTS and BFAIRMC by looking at the cumulative reward achieved by the agent over a finite number of steps in the environment given a fixed amount of rollouts. First of all, the algorithms are compared against each other, against the heuristics and against MC. This results in proposing the following hypothesis for both COMCTS and BFAIRMC.

Hypothesis 1. *The algorithm outperforms Monte Carlo rollouts.*

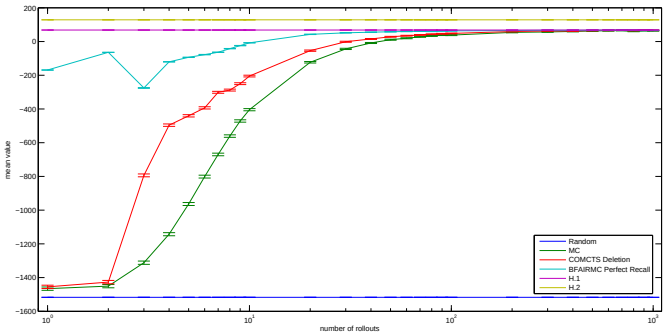
Secondly, the variations of each individual algorithm are compared against each other. This leads to posing the following hypotheses for both COMCTS and BFAIRMC.

Hypothesis 2. *The variants are ordered ascending in performance according to their degree of avoiding information loss as follows: deletion, local recall (insertion), perfect recall.*

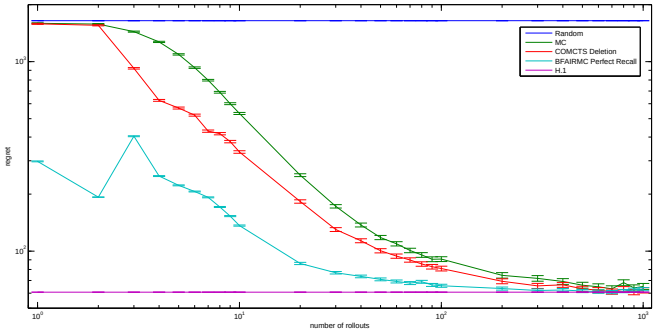
The main evaluation metric in this experiment is mean value which is defined to be the average cumulative reward achieved by the agent over a finite number of steps in the environment given a fixed number of rollouts. An additional metric

for the tiger problem is regret which is defined to be the difference between the optimum and the mean value. The assumption is made that the optimum for the tiger problem is the mean value accumulated by an agent that employs heuristic H.2.

Figure 5.3 shows the online performance of the different algorithms in the infinite horizon Tiger problem with a one-dimensional continuous observation space. The standard deviation of the noise that corrupts the observation signal is $\sigma = 0.965$ [16]. The execution of the agent is stopped after 50 steps in the environment. The numbers are acquired by performing 5,000 runs with uniformly distributed starting states. The discount factor for the algorithms is set to $\gamma = 0.5$.



(a) Average cumulative reward



(b) Regret

Figure 5.3: The performance achieved over 5,000 simulations of 50 steps each for the different algorithms in the infinite horizon Tiger problem with 1D continuous observations.

The results indicate that COMCTS and BFAIRMC converge faster to the possible lower bound on the optimum given by heuristic H.1 than MC. For this problem, the best performing algorithm turns out to be BFAIRMC with perfect recall. The reason for this is that whenever a door is opened, the agent's belief state is reset uniformly over the state space. Only when listening, the agent's belief state shifts away from this uniform distribution. The more rollouts the algorithms can perform, the less different is their performance. After approximately 300 rollouts, all three algorithms approach the same value that H.1 achieves. From this point on, the difference between the algorithms is negli-

gible. A possible reason for this indifference is that H.1 is indeed a lower bound on the optimum in this problem and therefore hard to compete with.

In Figure 5.4, BFAIRMC is compared against BFAIRQ in the tiger problem with the same settings as before. For BFAIRQ, the learning rate is set to $\alpha = 0.1$ and the discount factor is set to $\gamma = 0.9$.

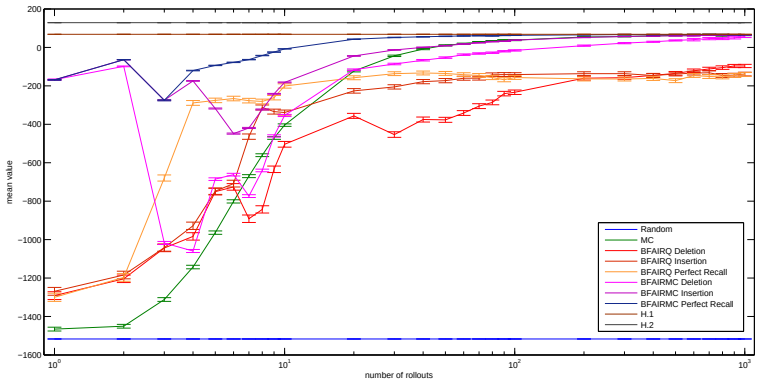


Figure 5.4: The average cumulative reward achieved by BFAIRMC and BFAIRQ in the infinite horizon Tiger problem with 1D continuous observations.

The results point out a large difference in performance between BFAIRMC and BFAIRQ given a sufficient number of rollouts. From 20 rollouts on, BFAIRMC clearly outperforms BFAIRQ. The reason for this is that the update of the value in BFAIRMC considers all rewards accumulated in a simulation, while BFAIRQ only considers the immediate reward of the current step. There are significant differences in the performance of the different variants of both algorithms. Perfect recall always achieves the highest value, followed by insertion and after that followed by deletion. This result comes from the difference in avoiding information loss.

Figure 5.5 illustrates the online performance of COMCTS in the light-dark domain with a discrete state space of size 10×10 and a two-dimensional continuous observation space. As discussed in Section 4.1, BFAIRMC cannot be directly applied to problems with a large state space and is thus omitted for the discrete state space light dark domain. After 20 steps in the environment, the execution of the agent is stopped. The numbers are acquired by performing 1,000 runs with uniformly distributed starting states. The discount factor for the algorithms is set to $\gamma = 0.95$.

The results show that there is not much difference between the performance of COMCTS and MC given a low (less than 10) or a high number of rollouts (more than 600). In between these points, the mean value achieved by COMCTS is slightly higher than the mean value achieved by MC. This indifference might be caused by a high number of rollouts needed to learn the observation trees in COMCTS: at the point where there are enough rollouts for this algorithm to generate a good policy, there are also enough rollouts for MC to generate such a policy. Another possible reason lies in the manual tuning of the algorithm. Other parameter settings, e.g., a different choice of the exploration factor in

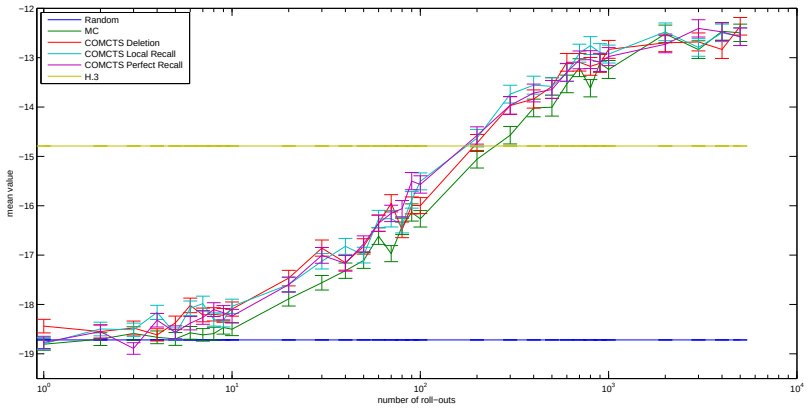


Figure 5.5: The average cumulative reward achieved over 1,000 simulations of maximal 20 steps each for COMCTS in the discrete state space light-dark domain with 2D continuous observations.

UCT or the significance level in the regression tree component could possibly lead to better results. In contrast to the heuristics for the tiger problem, the heuristic for the light-dark domain can be beaten given a sufficient amount of rollouts. The reason for this, given in Section 5.1.3, is that the agent might belief to be in a certain state with a high probability although in the real environment, the agent is in a completely different state. Following H.3, the agent might then select actions that move it away from the goal's location. Furthermore, the number of rollouts from which each algorithm outperforms the heuristic are very close. This result might be explained by the same fact that might cause the indifference between MC and COMCTS. To outperform H.3, COMCTS requires so many rollouts that MC can also create a good policy.

To compensate that BFAIRMC cannot be directly applied to the discrete state space light-dark domain, a continuous version of this environment is briefly investigated. In this environment, BFAIRMC relies on an approximation of the agent's belief state through a Gaussian distribution and a Kalman filter (see Appendix A). Figure 5.6 illustrates the online performance of BFAIRMC in this domain. The numbers are acquired by performing 1,000 runs with uniformly distributed starting states. The execution of the agent is stopped after 50 steps in the environment.

The results indicate a clear improvement of BFAIRMC over the random agent, but the behavior of the algorithm given more than 10 rollouts is very unstable and its performance varies a lot. There are three possible reasons for this. The first reason is that a restricted form of the multivariate Gaussian is used to approximate the agent's belief in this problem. The second reason is that, in contrast to all other experiments, the parameters of the algorithm have not been tuned for this problem. The third reason is that, given many rollouts, the algorithm finds so many refinements for the discretization of the belief space that the values for the actions in some parts of this discretization become less reliable than they were with a coarser discretization given a lower number of rollouts.

Figure 5.7 compares the online performance of the different variations of

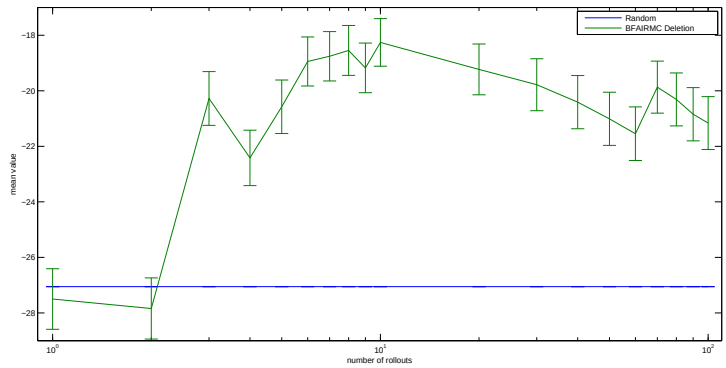


Figure 5.6: The average cumulative reward achieved over 1,000 simulations of maximal 50 steps each for BFAIRMC in the continuous state space light-dark domain with 2D continuous observations.

BFAIRMC in the infinite horizon tiger problem with the same settings as before. Their order in performance reflects their ability to avoid information loss: deletion does not reconstruct anything and achieves the worst performance, insertion incurs some amount of knowledge from the test that determines the split point and achieves a higher value than deletion, and perfect recall rebuilds as much knowledge as possible from its stored rollouts and achieves the highest value of all three variations. The first dip that appears in all three variants is caused by the first refinements of the discretization of the belief space. With deletion, a good policy might have been learned before the refinement, and after the refinement, this policy is lost and needs to be learned again. With insertion, the information incurred from the test could be unreliable because it might be based on a insufficient amount of rollouts. For both variants, there might not be enough rollouts left after a refinement for learning a good policy again. The second dip that can be seen in deletion and insertion is possibly caused by the same reason of loosing and being required to find a good policy again. Interestingly, perfect recall does not show this second dip in performance. This result can be explained by the fact that perfect recall can benefit from a much larger amount of information, in form of the rollouts that this strategy maintains, than insertion and deletion. The intensity of the dips makes the difference in their ability to avoid information loss apparent again. Deletion looses most information and shows the most intense dips, insertion looses less information and shows less intense dips, and perfect recall looses no information and even does not show the second dip.

In Figure 5.8, the different variants of COMCTS are compared against each other in both the infinite horizon tiger problem and the discrete state space light-dark domain. The settings for the environments are the same as before. From these results, it can be seen that the difference in performance between the three variations of COMCTS is negligible. A possible explanation for this is that deletion already achieves a good performance in the two domains investigated in this experiment. A more general explanation could be that the first layer of history and sequence nodes of the search tree build by COMCTS is not changed in case of a refinement of the discretization. The nodes in this first

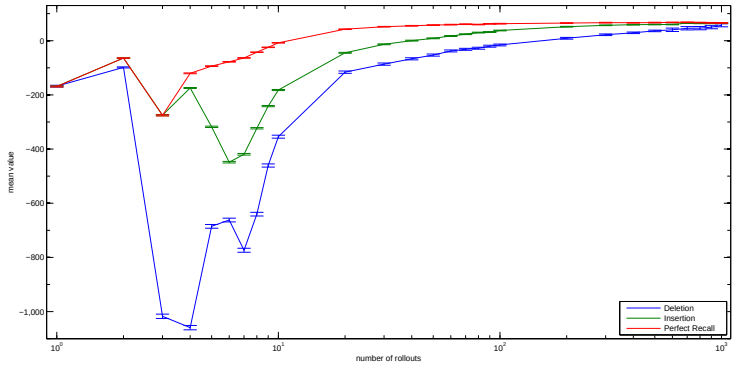
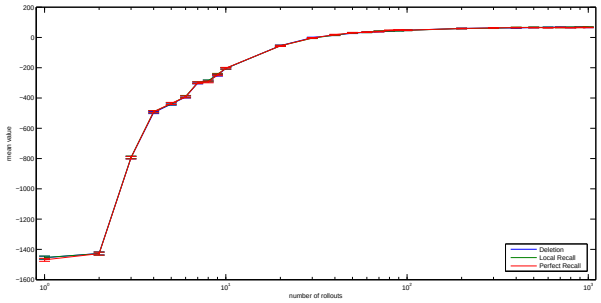
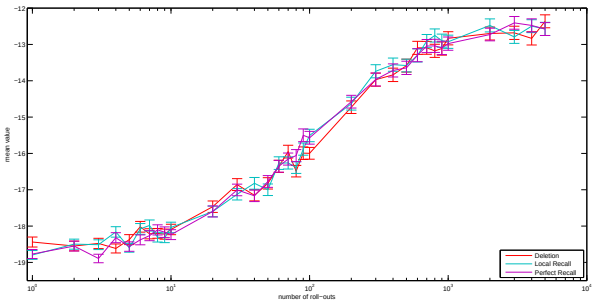


Figure 5.7: The average cumulative reward achieved over 5,000 simulations of 50 steps each for the different variations of BFAIRMC in the infinite horizon Tiger problem with 1D continuous observations.

level maintain the values that decide the final action selection of the agent which, in consequence, influences the agent's performance in the real world.



(a) Tiger problem



(b) Light-dark domain

Figure 5.8: The average cumulative reward achieved by the different variations of COMCTS in the infinite horizon Tiger problem with 1D continuous observations and the discrete state space light-dark domain with 2D continuous observations.

To summarize the results of this experiment, the hypotheses posed at the beginning of this section are reconsidered. For the first few rollouts in the tiger

problem, the results show a significant improvement of BFAIRMC and COMCTS over MC in the tiger problem. However, this improvement is not supported by the results for COMCTS in the light-dark domain, and given enough rollouts in the tiger problem, this difference is not apparent anymore. Therefore, Hypothesis 1 is refuted. Hypothesis 2 is supported for BFAIRMC because there is a significant difference between the performance of the variants of this algorithm, but renounced for COMCTS because there is nearly no difference between the performance of the variants of this algorithm.

5.2.2 Time-Based Performance

This experiment examines the time-based performance of COMCTS and BFAIRMC by looking at the cumulative reward achieved by the agent over a finite number of steps in the environment given a fixed amount of time.

Figure 5.9 shows the online performance of the different algorithms in the infinite horizon Tiger problem with a one-dimensional continuous observation space. The settings are the same as before, only the execution of the agent is already stopped after two steps in the environment. Because the performance is very similar for all variations of COMCTS, only the deletion variant is part of the figure.

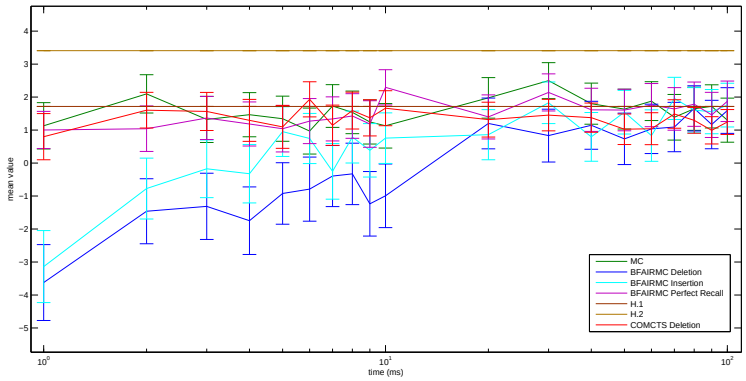


Figure 5.9: Time-based performance in the infinite horizon tiger problem.

Except for BFAIRMC with perfect recall, the results point out a large difference between the performance of COMCTS and the other two variants of BFAIRMC given a low amount of time. With only one millisecond of computation time, COMCTS approaches the near optimal performance of H.1. BFAIRMC with deletion or insertion needs around ten milliseconds more to reach a similar performance. However, BFAIRMC with perfect recall and MC can both compete with COMCTS from the first millisecond on. The performances of the different variants of BFAIRMC reflect what is seen in sample-based settings. Both BFAIRMC with deletion and insertion need some time until they reach the same performance as perfect recall. This behavior is caused by the loss of information of these two variants. In particular, a partial loss for insertion and a total loss for deletion.

Figure 5.10 shows the online performance of the different variants of COMCTS in the discrete state space light-dark domain with a two-dimensional continuous observation space. The settings are the same as before.

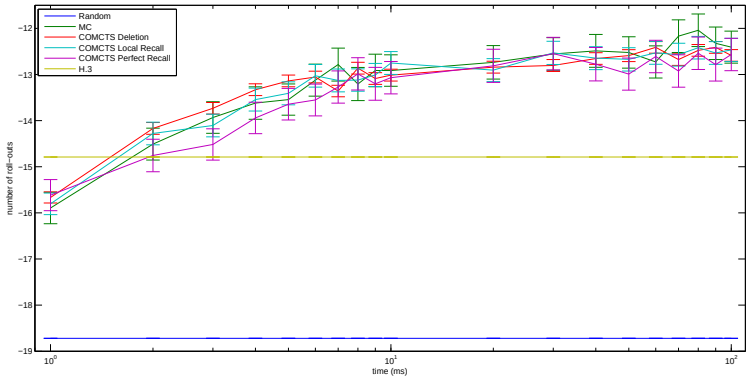


Figure 5.10: Time-based performance of COMCTS in the discrete state space light-dark domain.

The results reflect the same similarity in performance between the variants of COMCTS that is demonstrated by the sample-based experiment in the previous section (compare Figure 5.8): there is no significant difference between the variants of this algorithm. There are two possible reasons for this. The first reason is that deletion already achieves a high average cumulative reward that cannot be outperformed any more by the other two variants. The second reason is that the other two variants only effect the deeper levels of the search tree and not the final decision at the first level of the action to be executed by the agent in the real world. Furthermore, the performance of MC and any variant is very similar.

5.2.3 Practical Running Time

This experiment analyzes the performance of COMCTS and BFAIRMC under time constraints. The theoretical analyses of the running time of the novel algorithms (see Sections 3.4.1 and 4.4.1), give rise to the following hypothesis.

Hypothesis 3. *COMCTS requires significantly less running time than BFAIRMC.*

In both COMCTS and BFAIRMC, perfect recall requires to store and to process all samples that passed through the search tree. This leads to the following hypothesis.

Hypothesis 4. *Deletion requires significantly less running time than perfect recall.*

The metric of performance is the number of roll-outs that is achieved by the algorithm in the environment given a fixed amount of time. The numbers are acquired by performing 2,000 runs with uniformly distributed starting states.

The results for the infinite horizon Tiger problem are presented in Figure 5.11. MC is the best performing algorithm with respect to the number of rollouts, and acquires more than three times as many rollouts as the next best algorithm, COMCTS. All variants of COMCTS turn out to achieve more rollouts than any variant of BFAIRMC. There are three reasons that cause this difference in computation time. The first reason is that the rollouts in COMCTS are partially random, while the rollouts in BFAIRMC are following UCT. The second reason is that BFAIRMC requires to recompute the belief state for each step during a rollout which is quite computationally expensive. The third reason is that each step during a rollout in BFAIRMC provides a sample that is used to update the belief tree, while only those samples that are gathered during the selection and expansion phase in COMCTS are used to update the observation trees (compare Figures 3.2 and 4.1).

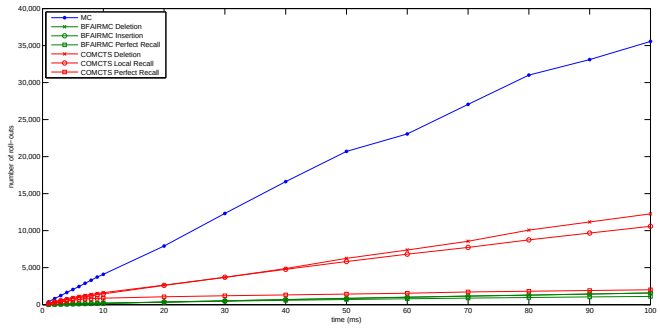


Figure 5.11: The average number of rollouts achieved by each algorithm in 2,000 simulations over time in the infinite horizon Tiger problem with 1D continuous observations.

As discussed in Section 4.2, there is not much difference in the number of rollouts achieved by the splitting strategies deletion and insertion of BFAIRMC. Slightly less rollouts are achieved by perfect recall. Interestingly, the number of rollouts reached by COMCTS with perfect recall comes close to the deletion and insertion variants of BFAIRMC. The difference in demand for computation time between COMCTS with local recall and perfect recall is emphasized by the former variant achieving more than four times as many rollouts than the latter variant. The reason for this difference is that local recall only rebuilds the tree on the level of one observation tree, while perfect recall also rebuilds the subtrees below that observation tree. COMCTS with deletion beats the other variations of the same algorithm and also all variations of BFAIRMC. The number of rollouts accumulated by this variant differs by a factor of more than five to BFAIRMC with the same splitting strategy. The results for the discrete state space light-dark domain are shown in Figure 5.12. Again, MC turns out to achieve the highest number of rollouts. The difference in performance between the splitting strategies of COMCTS is similar to what is observed for the tiger problem, with deletion acquiring five times more rollouts than perfect recall. In general, the number of rollouts is two times lower than in the tiger problem. This decrease in performance is caused by the black-box simulations which are much more complex for the light-dark domain than for the tiger problem. Looking back at the results for the time-based performance of COMCTS with perfect

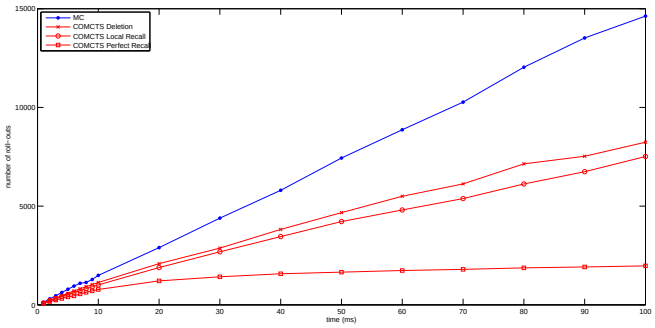


Figure 5.12: The average number of rollouts achieved by the different variants of COMCTS in 2,000 simulations over time in the discrete state space light-dark domain with 2D continuous observations.

recall and local recall (see Figures 5.9 and 5.10), there is no indication that these variants improve the performance of the agent. Taking the additional high costs with respect to running time into account, there does not remain a positive argument for these variants in practice. Nevertheless, these variants might show a difference in performance in more complex domains.

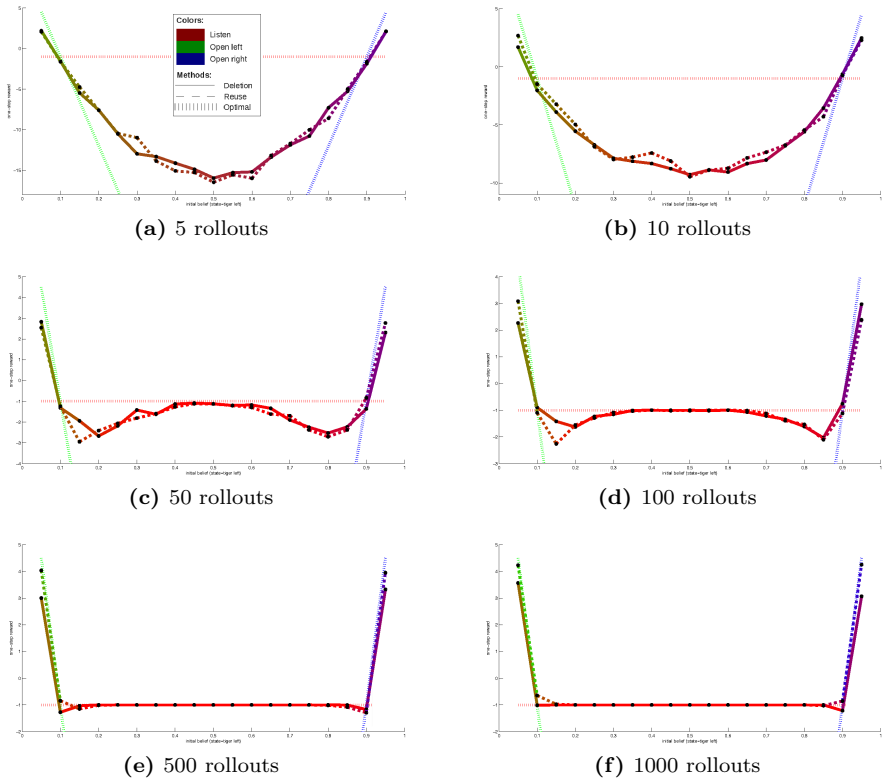
The results of this experiment lead to the support of both hypotheses proposed at the beginning of this section. Hypothesis 3 is largely supported because there is a large difference in running time between the deletion and local recall variants of COMCTS and any variant of BFAIRMC. Only COMCTS with perfect recall comes close to the running time of BFAIRMC. Hypothesis 4 is supported because the results show a large difference between perfect recall and deletion for both COMCTS and BFAIRMC in the tiger problem, and for COMCTS in the light-dark domain.

5.2.4 One-Step Value Function in the Tiger Problem

This experiment examines the development of the approximation to the one-step value function that is generated by the deletion and the belief-based reuse of knowledge variants of COMCTS in the infinite horizon tiger problem with one-dimensional continuous observations. Because the latter variant can reuse a lot of information within the observation trees, the following hypothesis is proposed.

Hypothesis 5. *The development of COMCTS to the optimal one-step value function is faster with belief-based reuse of knowledge than with deletion.*

In this experiment, each data point is acquired by performing 5,000 runs with starting states distributed according to the agent's initial belief state $b(s = \text{tiger left})$. The sequence $\{0.05, 0.1, 0.15, \ldots, 0.95\}$ determines the value for the agent's initial belief state $b^0(s = \text{tiger left})$. After one step in the environment, the execution of the agent is stopped. The value of a data point is the average reward accumulated by the agent. The color of a line segment between two points (b_1, b_2) indicates which action is selected most for that region in the belief space: red corresponds to *listen*, green corresponds to *open the left door*, and blue corresponds to *open the right door*. To indicate the approximation to



(a) 5 rollouts **(b)** 10 rollouts

(c) 50 rollouts **(d)** 100 rollouts

(e) 500 rollouts **(f)** 1000 rollouts

Figure 5.13: The development of the approximation towards the optimal one-step value function that is achieved by the deletion and the reuse of knowledge variants of COMCTS in the infinite horizon tiger problem over 5,000 simulations with 1D continuous observations.

the optimal one-step value function, the three deterministic policies of always selecting the same action are also part of this experiment. The optimal value function is the line given by the highest values achieved by these three policies. The results of this experiment are shown in Figure 5.13. Given 5 or 10 rollouts, the action selection and the reward accumulated by both variations are similar. Given more rollouts, the performance of reuse breaks in and deletion achieves a higher reward and comes also closer to the optimal one-step value function than reuse. Arriving at 500 rollouts, reuse can be seen to improve over deletion, and provides an even closer approximation to the one-step value function than deletion does for the other numbers. This outcome of close approximation is supported by the results given 1000 rollouts. Despite the correct action selection in nearly all regions of the belief space by reuse for 500 and more rollouts, the difference in reward to deletion is relatively small. An exception from this outcome is the region in the belief space around the point where the optimal action switches from listen to open the correct door.

From the shape of the approximation in the graphs, it can be clearly pointed out in which regions of the belief space the value function is more easy and in which it is more hard to learn. The region where the learning is simpler is

approximately represented by the interval $[0.3, 0.7]$, and the regions where the learning is difficult are approximately represented by the intervals $[0.1, 0.3]$ and $[0.7, 0.9]$.

The general result of this experiment is that a lot of rollouts are required to correctly learn a close approximation to the optimal one-step value function in the infinite horizon tiger problem with continuous observations. Furthermore, the regions of the belief space can be coarsely divided into two categories with respect to how fast the development of the approximation towards the optimal value function is. The regions surrounding the points from which on listening is not the optimal action anymore are hardest to learn, while all other regions are simpler to learn. Since the development of deletion clearly approaches the optimal one-step value function faster than deletion, Hypothesis 5 is refuted.

Chapter 6

Conclusion

This chapter summarizes and discusses the main findings of this thesis. After the discussion, answers to the research questions are given. Finally, recommendations for future work are outlined.

6.1 Contributions

This thesis contributes two novel algorithms to the field of online planning in POMDPs with continuous observations. Both algorithms are based on a combination of incremental regression tree induction and a particular Monte Carlo method. The Monte Carlo method is used to exploit and explore the agent's environment, while incremental regression induction is used to discretize a continuous space. The following paragraphs address those two algorithms in detail.

Continuous Observations Monte Carlo Tree Search is an extension of Monte Carlo Tree Search to POMDPs with continuous observations. Each node of the search tree of MCTS is associated with a tree-based discretization of the POMDP's observation space. This discretization is build automatically by an incremental regression tree learning algorithm that adjusts the discretization whenever there is significant evidence given by a set of test statistics that a refinement of the discretization would effect the agent's performance.

Belief based Function Approximation using Incremental Regression tree techniques and Monte Carlo is a two layer algorithm that combines a tree-based discretization of the belief space with Monte Carlo rollouts. This discretization is again build by an incremental regression tree learning algorithm that is similar to the same component used in Continuous Observations Monte Carlo Tree Search.

The splitting strategies that are employed by each algorithm to refine the tree-based discretizations represent another contribution of this thesis. Two of these strategies, deletion and perfect recall, can be applied to both algorithms. Deletion simply remembers no samples and does no more operations than are required by the splitting procedure. In contrast to deletion, perfect recall remembers all samples that passed through the algorithm and uses them to reconstruct the search tree in the case of a split. The other strategies are local recall which rebuilds the tree of the first algorithm on the level at which

the split occurs, and insertion which directly incurs knowledge about actions from the test that determines the split point in the second algorithm.

A further contribution of this thesis is an analysis of the algorithms in two distinct environments. The tiger problem serves as a toy environment to test the algorithms, while the light-dark domain is a more complex environment that represents a simplified robot localization problem. The algorithms are compared to domain-specific heuristics and existing methods such as Monte Carlo rollouts. The analysis indicates that both novel methods converge faster to a high performing policy than Monte Carlo rollouts with a uniform action selection but cannot outperform this algorithm given a lot of samples. Because the set of problems investigated for the analysis is small, no conclusions about the performance of the algorithms in other domains can be drawn.

6.2 Answers to the Research Questions

This thesis focused on the development of two novel online planning algorithms for POMDPs with continuous observations. Before a general conclusion about the problem statement of this thesis is drawn, the research questions presented in Section 1.1 are recapitulated and answered.

The first two research questions relate to the use of the incremental regression tree component of Tree Learning Search for the discretization of other continuous spaces than the action space. The two algorithms presented in this thesis employ variations of this component to successfully discretize continuous observation spaces and continuous belief spaces.

With respect to the third research question that asked how information loss can be avoided in the new algorithms, three different strategies that recreate knowledge are discussed. Perfect recall is a generally applicable strategy that remembers each sample to rebuild the information. Local recall is a similar strategy that also remembers samples but only reconstructs the information on the first level. Insertion is a strategy that is related to action-based test statistics from which some amount of information can be directly inferred in the case of a split.

The fourth question inquired for a theoretical analysis of the time and space bounds of both techniques. This analysis is given in Sections 3.4 and 4.4. Continuous Observations Monte Carlo Tree Search requires less computation time than the belief based function approximation algorithm but demands significantly more space. This theoretical analysis is supported by a practical investigation of the running time, provided in Section 5.2.3.

The answer to the last research question that asked for the performance of the algorithms in a set of benchmark problems is provided by the results of the experiments given in Section 5.2. In the tiger problem, the belief based function approximation algorithm performs best. The difference in performance between the variations of COMCTS turns out to negligible, while there is a significant difference in performance between the variations of BFAIRMC.

In relation to the initial problem statement, the conclusion can be drawn that Continuous Observations Monte Carlo Tree Search is a possible extension of Partially Observable Monte Carlo Planning to POMDPs with continuous observations. The discretization component of this new algorithm allows to automatically discretize the observation space. Although the experiments con-

ducted for this thesis point out a clear improvement of the algorithm over a uniform random method and a slight improvement over uniform Monte Carlo rollouts, there is not enough evidence yet to decide whether this algorithm is effective in practice.

6.3 Recommendations for Future Work

The most important direction for future work is a more systematic evaluation of the algorithms. This evaluation should include two components.

The first component is a new set of benchmark problems that is larger than the current set and that might clarify how the novel algorithms presented in this thesis perform in comparison to existing methods. On the current set of problems, uniform Monte Carlo rollouts reach a high performance that is hard to beat, but for more difficult problems, Monte Carlo rollouts might work worse than the new algorithms. In addition, the current set contained problems with a relatively small state space. Therefore, problems with larger state spaces could be examined. BFAIRMC could also be studied on POMDPs with discrete observations because the algorithm is not restricted to continuous observation spaces.

The second component is a larger set of algorithms that is based on existing methods. For Continuous Observations Monte Carlo Tree Search, this set includes Partially Observable Monte Carlo Planning [40] with an a priori discretization of the observation space. This would allow for a comparison between the automatic discretization by COMCTS and a priori discretization. For the belief based function approximation algorithm, this set includes other forms of Q-Learning such as Q-Learning with an update rule that is based on the entire sequence of rewards sampled during the policy evaluation step and comes therefore closer to the update rule used in BFAIRMC. A further extension would be to weight each update, e.g., as in $Q(\lambda)$ [43]. Hypothetically, this would improve and move the performance of BFAIRQ closer to the performance of BFAIRMC.

Considering that the two algorithms use different kinds of tests to find subdivisions of a continuous space, it should be probed whether these tests can be exchanged against each other. Furthermore, this experimental investigation could be supported by a theoretical analysis that clarifies in which kind of settings which kind of test is optimal.

6.4 Other Future Work

Other future work could go in four possible directions. The first direction is to combine Partially Observable Monte Carlo Planning [40] with Tree Learning Search [46] to develop an algorithm that handles POMDPs with discrete observations and continuous actions. POMDPs with continuous action and continuous observation spaces could be approached by a combination of Tree Learning Search and Continuous Observations Monte Carlo Tree Search. The second direction is to replace the discretization component of the two algorithms developed in this thesis by another component, e.g., any kind of incremental decision or regression tree learning algorithm [7, 13] or Hierarchical Optimistic Optimization [5] which refines the discretization of a continuous space in each

iteration. The third direction is to employ this discretization component in other algorithms for POMDPs. The fourth direction is to modify the belief state estimation that is used in the function approximation algorithm proposed in this study. A first modification could be Gaussians with the full covariance matrix instead of the current diagonal matrix. Further modifications could involve other approximations such as a particle filters.

Bibliography

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, May 2002.

[2] Richard Ernest Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] Dimitri P. Bertsekas and David A. Castanon. Rollout algorithms for stochastic scheduling problems. *Journal of Heuristics*, 5(1):89–108, 1999.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[5] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. Online optimization in x-armed bandits. In Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou, editors, *NIPS*, pages 201–208. Curran Associates, Inc., 2008.

[6] Anthony R. Cassandra. A survey of POMDPs applications. In *AAAI Fall Symposium: Planning with POMDPs*, pages 17–24, 1998.

[7] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: an algorithm and performance comparisons. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 2*, IJCAI'91, pages 726–731, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[8] Guillaume M. J-B. Chaslot, Mark H. M. Winands, H. Jaap Van Den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation (NMNC)*, 4(03):343–357, 2008.

[9] Guillaume Maurice Jean-Bernard Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik. Monte-carlo strategies for computer go. In P-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 83–91. Namur, 2006.

[10] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H.H.L.M. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[11] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-carlo tree search in poker using expected reward distributions. In Zhi-Hua Zhou and Takashi Washio, editors, *ACML*, volume 5828 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2009.

[12] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences (7th Ed.)*. Thomson Brooks/Cole, 2007.

[13] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM.

[14] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In Luc De Raedt and Peter A. Flach, editors, *ECML*, volume 2167 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2001.

[15] N.J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. *IEEE Proceedings F, Radar and Signal Processing*, 140(2):107–113, 1993.

[16] Jesse Hoey and Pascal Poupart. Solving POMDPs with continuous or large discrete observation spaces. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 1332–1338. Professional Book Center, 2005.

[17] Elena Ikonomovska and Joao Gama. Learning model trees from data streams. In Jean-François Boulicaut, Michael R. Berthold, and Tamás Horváth, editors, *Discovery Science*, volume 5255 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 2008.

[18] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, May 1998.

[19] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[20] Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. Approximate planning in large POMDPs via reusable trajectories. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *NIPS*, pages 1001–1007. The MIT Press, 1999.

[21] Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.

[22] Lukas Kirchhart. Reusing knowledge in tree learning search. Master's thesis, Department of Knowledge Engineering, Maastricht University, 2012.

[23] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[24] Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel. Fast algorithms for finding randomized strategies in game trees. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 750–759. ACM, 1994.

[25] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In Jim Hendler and Devika Subramanian, editors, *AAAI/IAAI*, pages 541–548. AAAI Press / The MIT Press, 1999.

[26] Andrei A. Markov. *Theory of Algorithms*. Academy of Sciences, USSR, 1954.

[27] Andrew Kachites Mccallum. *Reinforcement learning with selective perception and hidden state*. PhD thesis, 1996.

[28] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., 1997.

[29] Christos Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Math. Oper. Res.*, 12(3):441–450, 1987.

[30] Sébastien Paquet, Ludovic Tobin, and Brahim Chaib-draa. An online pomdp algorithm for complex multiagent environments. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 970–977, New York, NY, USA, 2005. ACM.

[31] Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for pomdps. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1025 – 1032, 2003.

[32] Robert Platt, Russell Tedrake, Leslie Kaelbling, and Tomas Lozano-Perez. Belief space planning assuming maximum likelihood observations. In *Robotics Science and Systems Conference (RSS)*, 2010.

[33] Josep M. Porta, Nikos A. Vlassis, Matthijs T. J. Spaan, and Pascal Poupart. Point-based value iteration for continuous POMDPs. *Journal of Machine Learning Research*, 7:2329–2367, 2006.

[34] Larry D. Pyeatt and Adele E. Howe. Decision tree function approximation in reinforcement learning. Technical report, In Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models, 1998.

[35] Stéphane Ross and Brahim Chaib-Draa. Aems: an anytime online search algorithm for approximate policy refinement in large pomdps. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pages 2592–2598. Morgan Kaufmann Publishers Inc., 2007.

[36] Stéphane Ross, Joelle Pineau, Sébastien Paquet, and Brahim Chaib-draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32:663–704, 2008.

[37] N. Roy, J. Pineau, and S. Thrun. Spoken dialogue management using probabilistic reasoning. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL-2000)*, Hong Kong, 2000.

[38] J. K. Satia and R. E. Lave. Markovian decision processes with probabilistic observation of states. *Management Science*, 20(1):1–13, 1973.

[39] Colin Schepers. Automatic decomposition of continuous action and state spaces in simulation-based planning. Master's thesis, Department of Knowledge Engineering, Maastricht University, 2012.

[40] David Silver and Joel Veness. Monte-carlo planning in large POMDPs. In John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, editors, *NIPS*, pages 2164–2172. Curran Associates, Inc., 2010.

[41] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088, 1973.

[42] Matthijs T. J. Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.

[43] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[44] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[45] William T. B. Uther and Manuela M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 769–774, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[46] Guy Van den Broeck and Kurt Driessens. Automatic discretization of actions and states in Monte-Carlo tree search. In Tom Croonenborghs, Kurt Driessens, and Olana Missura, editors, *Proceedings of the ECML/PKDD 2011 Workshop on Machine Learning and Data Mining in and around Games*, pages 1–12, 2011.

[47] Xin Wang and Thomas G. Dietterich. Efficient value function approximation using regression trees. In *In Proceedings of the IJCAI Workshop on Statistical Machine Learning for Large-Scale Optimization*, 1999.

[48] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

Appendix A

Belief State Approximation

The belief state can be approximated by model estimation techniques such as Kalman filters or particle filters that break these two barriers. These techniques are introduced in the following sections.

A.1 Kalman Filters

The Kalman filter [19] is a recursive state estimator that belongs to the class of Gaussian filters. It is one of the most used and historically earliest implemented filters for continuous spaces. Gaussian filters represent the belief state as a multivariate normal distribution with two sets of parameters defined as follows [44].

$$\Pr(s) = |(2\pi\Sigma)|^{-0.5} e^{-0.5(s-\mu)^T \Sigma^{-1}(s-\mu)} \tag{A.1}$$

where s is a vector that represents the state, μ is the mean, and Σ is the covariance, represented by a quadratic, symmetric and positive-semidefinite matrix.

The following three conditions are sufficient to ensure that the belief $b(s^t)$ is always a Gaussian.

1. The state transitions are linear with additive Gaussian noise.

$$s^{t+1} = A^t s^t + B^t a^t + \epsilon^t \tag{A.2}$$

   Here, s^t is a state vector and a^t is the vector that expresses the action. The matrices A^t and B^t ensure that the state transition model becomes linear. ϵ^t is a vector that denotes the randomness or noise in the state transition and is distributed according to a normal distribution with zero mean and covariance matrix R^t.

2. The observations are also linear with additive Gaussian noise.

$$o^t = C^t s^t + \delta^t \tag{A.3}$$

   Here, the matrix C^t is again responsible for linearity, and δ^t denotes the randomness in the observations. The noise δ^t is distributed according to a normal distribution with zero mean and covariance matrix Q^t.

3. The initial belief $b(s_0)$ is normally distributed.

$$b(s_0) = \Pr(s_0) = |(2\pi\Sigma_0)|^{-0.5} e^{-0.5(s_0-\mu_0)^T \Sigma_0^{-1}(s_0-\mu_0)} \tag{A.4}$$

The algorithm of the Kalman filter is given in Algorithm 1. The next mean μ^{t+1} and the next covariance matrix Σ^{t+1} are recursively constructed from the current mean μ^t and the current covariance matrix Σ^t. This construction is similar to the exact belief state update given in Equation 2.6. The algorithm takes the current mean μ^t, the current covariance matrix Σ^t, the current action a^t and the current observation o^t as input, and outputs the next mean μ^{t+1} and the next covariance matrix Σ^{t+1}. It starts by computing a temporary mean $\bar{\mu}^{t+1}$ based on the state transition model and the action a^t, and a temporary covariance matrix $\bar{\Sigma}^{t+1}$ based on the state transition model and the covariance of the noise in the state transitions, R^t. From the temporary covariance matrix $\bar{\Sigma}^{t+1}$ and the observation model, the algorithm computes the matrix K^{t+1}, also known as the *Kalman gain*. The Kalman gain determines the importance of the observation for the estimation of the state. The algorithm continues by adjusting the temporary mean $\bar{\mu}^{t+1}$ according to the Kalman gain and according to the difference between the actual observation o^t and the prediction given by the observation probability. Finally, the covariance matrix is adjusted according to the information that is gained from the observation.

Algorithm 1 The Kalman filter algorithm.

Require: current mean μ^t, covariance matrix Σ^t, action a^t, observation o^t
  function KALMAN_FILTER($\mu^t, \Sigma^t, a^t, o^t$)
    $\bar{\mu}^{t+1} \leftarrow A^t\mu^t + B^t a^t$
    $\bar{\Sigma}^{t+1} \leftarrow A^t\Sigma^t(A^t)^T + R^t$
    $K^{t+1} \leftarrow \bar{\Sigma}^{t+1}(C^t)^T(C^t\bar{\Sigma}^{t+1}(C^t)^T + Q^t)^{-1}$
    $\mu^{t+1} \leftarrow \bar{\mu}^{t+1} + K^{t+1}(o^t - C^t\bar{\mu}^{t+1})$
    $\Sigma^{t+1} \leftarrow (I - K^{t+1}C^t)\bar{\Sigma}^{t+1}$
  return μ^{t+1}, Σ^{t+1}
  end function

Whether modeling the belief state as a Gaussian distribution is reasonable depends on the characteristics of the underlying problem. Because Gaussians have a single maximum, i.e., they are uni-modal, they are ideal for problems where the belief state is centered around the true state with a small amount of uncertainty. However, Gaussians are not a good representation for problems where the belief state is more complex.

A.2 Particle Filters

As an alternative to Gaussian filters, particle filters [15] belong to the class of nonparameteric filters. Instead of basing the belief state on a fixed distribution type, the particle filter provides an approximation of the belief state by a finite number of values. Each of these values is associated with a region in the state space. The belief state is represented by a set of random state samples, called particles, which are drawn from the probability distribution that underlies the belief state. The distribution can be multi-modal, i.e., have multiple maxima,

and it can also be non-Gaussian. Although such a representation is not exact, it can be used for a large variety of distributions because it is non-parametric.

Algorithm 2 The particle filter algorithm.

Require: current set of particles X^t, action a^t, observation o^t
  function PARTICLE_FILTER(X^t, a^t, o^t)
    $\bar{X}^{t+1} \leftarrow X^{t+1} \leftarrow \emptyset$
    for $n = 1 \rightarrow N$ **do**
      sample $x_{[m]}^{t+1} \sim \Pr(x^{t+1}|a^t, x_{[m]}^t)$
      $w_{[m]}^{t+1} \leftarrow \Pr(o^t|x_{[m]}^{t+1})$
      $\bar{X}^{t+1} \leftarrow \bar{X}^{t+1} + \langle x_{[m]}^{t+1}, w_{[m]}^{t+1} \rangle$
    end for
    for $n = 1 \rightarrow N$ **do**
      draw i with probability $\propto w_{[i]}^t$
      add $x_{[i]}^t$ to X^{t+1}
    end for
  return X^{t+1}
  end function

The algorithm of the particle filter is given in Algorithm 2 [44]. The next set of particles X^{t+1} is recursively constructed from the current set of particles X^t. This recursive construction is analogous to the belief state update represented in Equation 2.6 where the next belief state is also recursively computed from the current belief state. The algorithm takes the current set of particles X^t, the current action a^t and the current observation o^t as input, and outputs the next set of particles X^{t+1}. It starts by constructing a temporary set of particles $\bar{X}^{t+1}$ which is an approximation of the belief b. New particles are sampled from the next state distribution $\Pr(x^{t+1}|a^t, x^t)$ and for each of them the so-called importance factor or weight $w_{[m]}^{t+1}$ is calculated. The importance factor is defined as the probability of the observation o^t given the particle $x_{[m]}^{t+1}$. The particles and their importance weights are added to the temporary particle set $\bar{X}^{t+1}$. The algorithm continues with a technique called importance resampling that draws with replacement N particles from the temporary set $\bar{X}^{t+1}$ according to their importance factor, and adds them to the output set X^{t+1}. Before the resampling step, the particles are distributed according to $b(x^t)$ but after resampling, they are distributed approximately according to the distribution of the next belief state, comparable to Equation 2.6, $b(x^{t+1}) = \alpha \Pr(o^t|x_{[m]}^{t+1}) b(x^t)$. There are two reasons why importance resampling is applied. The first reason is that only samples from the set $\bar{X}^{t+1}$ are available while the set of interest is X^{t+1}. The second reason is that the distribution of the new samples needs to be biased according to the weights of the samples to ensure that samples with a high weight appear more frequently in X^{t+1}.