TECHNISCHE
UNIVERSITÄT
DARMSTADT

THE UNIVERSITY OF
MELBOURNE

Knowledge Engineering Group
Fachbereich Informatik
Technische Universität Darmstadt

Intelligent Agent Laboratory
Department of Information Systems
The University of Melbourne

**Master's Thesis**

# PLANS AS A MEANS FOR GUIDING A REINFORCEMENT LEARNER

Jens Pfau

Melbourne, Australia, December 19, 2008

Supervisors:

Prof. Liz Sonenberg, Ph.D.
Michael Kirley, Ph.D.
Prof. Dr. Johannes Fürnkranz

# Abstract

The complexity of reinforcement learning problems grows exponentially with the size of the state space, which renders realistic cases unsolvable and underlines the need for guidance. This thesis studies a hybrid agent architecture, in which the top-level module reuses temporal knowledge in the form of plans that it extracts from a concurrently executing low-level reinforcement learner. The first contribution of this work are significant improvements of the original model and implementation of the agent architecture, resulting in a more effective knowledge extraction and reuse. The second contribution is an extensive exploration of the synergy effects that take place between both layers of the architecture. It is shown that the combination of state abstraction and the reuse of plans as temporal abstraction can lead to a significantly shorter learning time of a reinforcement learning agent. Likewise, the number of decisions to be made by the agent is reduced because a plan is a definite commitment to a course of actions that does not require intermediary reasoning. In addition, we demonstrate that the architecture enables the integration of plans as prior knowledge through a clear and convenient interface. Thus, partial and approximate solutions to the problem can be easily specified to significantly decrease learning time even further.

*To my beloved wife Agnieszka Beata — for patience, support and encouragement throughout the course of this thesis.*

# Acknowledgements

I am deeply grateful to my supervisors Liz Sonenberg, Michael Kirley and Johannes Fürnkranz, who provided me with the opportunity to work at an excellent academic institution in an interesting research field. Special thanks go to Liz for inviting me to visit her group and for guiding my work with an immense experience, giving the right advice at the right time. I am equally indebted to Michael, who was always available for fruitful discussions and whose "picky" comments pointed me to the right questions.

I also want to mention Samin Karim, whose work was the base of my thesis. His help and comments especially in the early stages of my research were definitely helpful.

Furthermore, I would like to thank the members of the Department of Information Systems at the University of Melbourne. They welcomed me warmly and supported my research to the best of their knowledge and beyond. The University of Melbourne is an extraordinary place, in which academic excellence and cultural diversity meet to shape the scientific world sustainably. I do not want to miss the unforgettable academic as well as personal experiences I made here.

I appreciate the financial support that I received from the German Academic Exchange Service (DAAD). This definitely facilitated my studies.

Special thanks also go to Stefan Pohl and Stephan Mehlhase, who provided frequent valuable and constructive feedback to my work. Moreover, both of them had a significant stake in the success and shape of my studies. I also want to thank Dhirendra Singh for valuable comments and discussions on reinforcement learning.

I am indebted to my wife and our families, who never stopped to support me in following my dreams. Without their unconditional help, trust and love, my studies and especially this work would not have been possible. I deeply appreciate that.

# Table of Contents

# List of Figures

# 1 Introduction

Recent advances in information technology are impressive. Vehicles are able to cover long distances without the intervention of any driver. House appliances adapt to the habits of their residents, and computer programs defeat chess grand masters. These applications have in common that the problems they tackle are highly complex such that reasonable solutions cannot be defined by deterministic instructions. For instance, developers might have limited information about the dynamics of potential application areas or about the influence of other active entities. Instead, such applications are capable of acting autonomously to a certain degree. Thus, they are able to adapt to their environment without relying on exhaustive problem specifications.

Constructing systems that exhibit some kind of adaptive and autonomous behavior is a research goal of *artificial intelligence* (Russell and Norvig, 2003). The main notion in this area is that of an *agent*. According to Wooldridge and Jennings (1995), this term denotes any software program that follows its own interests autonomously by acting on its environment, reacting to external perceptions and interacting with other agents. Such an agent is called *situated* because it directly affects its environment and is affected by this likewise. Russell and Norvig (2003) consider an agent to act rationally or intelligently if it behaves in such a way to maximize a given performance measure given its currently available knowledge. This definition obviously makes sense for the development of agents that strive to attain goals particularly provided by their human masters. Whether this can cover the notion of intelligence in its entirety, however, is not clear. Other researchers might only deem agents to be intelligent if they exhibit more human-like attributes such as emotions or acting irrationally once in a while (Wooldridge and Jennings, 1995).

The concrete development of a software agent is based on a specific agent architecture, which in turn emerged from a formal agent theory. *Agent theories* allow us to specify the properties of agents and to reason about their interrelations. They arise not only from research in artificial intelligence but also from work in psychology, philosophy and cognitive science. Typically, agent theorists strive to identify characteristics of human behavior and to map them onto a set of clearly graspable concepts.

A particular *agent architecture* provides a methodology for building agents that are compatible with the underlying agent theory (Wooldridge and Jennings, 1995).

Agent architectures generally fall into three categories: deliberative, reactive, and hybrid architectures (Wooldridge and Jennings, 1995). *Deliberative architectures* rely on the assumption that the problems in question can be modeled as symbolic systems and that logical reasoning on these gives rise to intelligent behavior or problem solving (Newell and Simon, 1976). A major product of the symbolic artificial intelligence community are algorithms for automated planning. *Planning*[1] is the process of finding a path of actions to apply in a certain situation in order to reach a situation in which a certain desirable condition holds. STRIPS was one of the early planning systems, which is well known and had a significant impact on subsequent systems (Fikes and Nilsson, 1971). However, it has been recognized that symbolic reasoning is problematic in time-critical applications or agents whose computational resources are restricted, so called *resource-bounded* agents (Chapman, 1987; Brooks, 1986; Bratman et al., 1988). Planning is basically a search in the space of options available to the agent and the states the environment can take. Its complexity thus grows exponentially with the size of both of these sets. Apart from that, it is still not clear how to represent the real world in a symbolic form effectively (Wooldridge and Jennings, 1995).

This has led to researchers arguing for so called *reactive architectures*, which refrain from relying on any complex symbolic reasoning. Rodney Brooks has been a foremost supporter of this idea. His *subsumption architecture*, which solely relies on a simple hierarchical behavior control, has been deployed successfully in various robots (Brooks, 1986). The major advantages of this approach are its simplicity and thus effectiveness. It is questionable, though, whether this kind of architecture can host any higher-level intelligence, which, for example, also reasons about the agent's long-term goals.

*Hybrid architectures* have been proposed by researchers who seek to combine the advantages of both worlds. They usually apply a mixture of different knowledge representations and reasoning techniques. A family of such agent architectures, including for example PRS (Georgeff and Lansky, 1987) and IRMA (Pollack, 1992), are based on the *Belief-Desire-Intention* (BDI) theory of agency, which was initially defined by Bratman et al. (1988). In this framework, reasoning is governed by the beliefs an agent has about its environment, its goals or desires and the intentions it has itself currently committed to. Plans for achieving certain intentions are not calculated on demand at

[1] For a detailed discussion of planning techniques refer to Ghallab et al. (2004).

runtime but are rather specified at development time as approximate recipes. Featuring these high-level concepts, reasoning allows for more complex decision-making and is not only restricted to primitive actions. Because such systems refrain from classical planning and because reasoning is constrained to behavior compatible with the agent's current intentions, they are able to sustain a certain level of reactivity. This has made them popular agent architectures. PRS, for example, was successfully deployed in the failure-handling module of the NASA space-shuttle (Ingrand et al., 1992).

As claimed previously, it is rarely possible to specify the behavior of an agent completely in terms of a program because of the complexity its tasks might exhibit. So far, however, we have only identified planning as a means for agents to behave autonomously to a certain degree. The second major technique for equipping agents with autonomy is *machine learning*. An agent or a computer program can be said to learn if its performance with regard to a certain performance measure improves with its experience (Mitchell, 1997). Reinforcement learning has been the primary method for designing situated, learning agents (Sutton and Barto, 1998). Major motivations for this are that it does not require any prior knowledge about the environment and that it is inherently an *online learning* technique. The latter allows the agent to augment its knowledge after its deployment. An agent employing reinforcement learning simply learns by receiving feedback for the actions it takes in its environment.

## 1.1 Motivation

There has been an impressive amount of research in supporting planning with machine learning techniques, with a major focus on speedup of planning systems (Zimmerman and Kambhampati, 2003). However, limited research has investigated the possibilities for integrating the BDI theory of agency with machine learning techniques, despite its popularity as an agent theory and its focus on reasoning about plans (e.g. Guerra-Hernández et al. (2004) and Olivia et al. (1999)). Conventional BDI architectures still require plans to be specified by developers.

Recently, Karim et al. (2006a) introduced the *Plan-Generation-Subsystem* (PGS) as a possibility for developing learning BDI agents. PGS is the top-level of a hybrid agent architecture, which records courses of actions performed by a bottom-level reinforcement learner. These are stored as reusable plans. It is an online learning approach and hence suited for situated agents. Allowing plans to be acquired during runtime releases developers from having to define suitable plans a priori. Furthermore, the agent's plan base adapts to changes in its environment. Apart from obtaining knowl-

edge automatically within the BDI framework, there are other obvious advantages of extracting plans from low-level behavior. Shifting knowledge from a fine-grained description to a temporally more abstract level yields a compression of knowledge on the one hand and makes it more accessible for inspection by humans on the other hand. It is also reasonable to expect a performance improvement when executing recorded plans. In fact, the knowledge about when to apply a certain plan comprises more information than the knowledge when to apply a single action.

Previous experiments showed useful plans being acquired (Karim et al., 2006a,b, 2008). Yet, there was a substantial bias towards reusing single step plans, which obviously hardly yields any improvement over employing the reinforcement learner alone. Apart from that, results were not entirely as expected. In addition, general plan reuse was low because plans were only reused in the very situation their recording had originally started. A state abstraction was not applied. This suggests the possibility for further analysis of previous results and improving both the model and implementation. It is also worthwhile to study PGS as a possibility for improving the performance of the underlying reinforcement learner. It has long been noticed that deploying a reinforcement learner can be problematic in complex domains with large state spaces, which require the agent to perform excessive, initial exploration (Whitehead, 1991). Thus, speeding up the initial learning phase is a major task for research in reinforcement learning.

## 1.2 Problem statement and research questions

The first part of this work is to identify shortcomings of the current PGS architecture and to relativize previous studies. This naturally leads to investing effort in improving the model and its implementation. In that, this work partly is an extension of what has been done previously. The architecture is simply conceived as a mechanism for extracting knowledge from a low-level learning module and shifting it to a higher level.

Having improved the plan acquisition process, it is then reasonable to ask how the plan execution process can contribute to the overall performance of the underlying learning module. This turns the original view upside down. Plans are no longer only an abstraction from low-level knowledge but they also provide a guidance for the low-level module. They naturally comprise more information then do primitive actions and, in that, their careful reuse can be expected to have a positive influence on the

overall performance. Studying these effects can leverage our understanding of the influence of temporal abstraction in learning and planning in intelligent agents.

This raises a number of questions to be studied in this thesis:

**Under what conditions can PGS successfully acquire effective plans?** A major issue is to clearly identify the requirements of deploying PGS and to assess them against the background of current research.

**How can the reuse of plans, and in particular longer plans, be facilitated?** Both general plan reuse as well as the average length of reused plans were low in previous experiments (Karim, 2009). To justify any discussion about PGS at all, both have to become significantly large enough.

**What is the impact of using plans on the overall performance?** As mentioned previously, plans do represent more information than primitive actions. It might be valuable to exploit this property by a sensible plan reuse in order to improve the overall performance of the agent.

**How can prior knowledge be incorporated into PGS?** Integrating prior knowledge about the problem or environment can substantially improve the time that an agent's learning module requires to converge to a decent behavior policy. A plan represents the knowledge about a probably successful order of primitive actions. It is worthwhile to study the possibility of integrating such knowledge into an agent.

We discuss the first question by reviewing PGS in the context of related research. The second question is addressed by improvements of the model and its implementation. The remaining questions are discussed against the background of experimental evaluation. The second and the fourth question require implementation work to be done, which partly is a change to the original model. Evaluating an agent architecture in a single domain only does not have any value. Research results in this case need to be validated in different domains to allow any conclusions to be drawn. Experiments are conducted within the *pursuit* and the *taxi* domains, which are both grid worlds. In the former, four predators seek to catch a prey by surrounding it. In the latter, a taxi agent is supposed to pickup a passenger at a particular position and to bring him or her to a particular destination.

## 1.3 Thesis outline

The thesis follows two main themes: temporal abstraction and prior knowledge in learning agents. We first discuss both topics in the context of reinforcement learning and hybrid agent architectures, which had a major impact on the development of PGS (Chapter 2). In Chapter 3, we bring together these two concepts when introducing the PGS agent architecture. In particular, we discuss the relationship between PGS and other approaches for exploiting temporal abstraction and prior knowledge in reinforcement learning. Based on that, we identify shortcomings of the original PGS model and develop a number of modifications. These extensions are then touched upon during the description of the system architecture in Chapter 4. However, the concrete implementation is not a major part of this work and its description hence kept rather brief. In chapter 5, we report results of extensive empirical analysis of the extended PGS model. This does not only clarify the restrictions of PGS but also supports the general discussion about temporal abstraction and prior knowledge. We conclude the thesis with a summary of the work conducted and an outlook on future work in Chapter 6.

# 2 Literature survey

PGS as it is adopted in this thesis is mainly driven or inspired by two concepts: reinforcement learning and hybrid agent architectures. The overall architecture derives from other hybrid architectures that extract abstract knowledge from an underlying low-level learning module. The low-level module in PGS is a reinforcement learner, which basically learns which action to carry out in a specific situation. As this also holds for some related agent architectures, reinforcement learning is introduced first in Section 2.1 before hybrid agent architectures are discussed in Section 2.2. This way, we will identify where PGS originally comes from as well as how it relates to other techniques for improving reinforcement learners.

## 2.1 Reinforcement learning

In *supervised learning*, knowledge is compressed or obtained by identifying patterns in labeled input data. Each example consists of a feature vector and an associated output value or class that the example belongs to. The latter have to be provided by an expert. A learning algorithm is supposed to produce generalized rules from this data which predict the output value or class of new examples based on their features. The model is built *offline*, meaning that the learning algorithm requires a sufficiently large example set before it can generate a reasonably accurate model (Mitchell, 1997).

Situated agents as they are of interest in this work, on the contrary, need to adapt to their environment from the very first moment of their life cycle in order to avoid harmful mistakes. They surely cannot wait for enough data being acquired to learn a reasonable behavior. Agents might act for a long time in a particular part of their environment before moving on to another one. In that case, a generally optimal behavior can hardly be obtained in the first place. Likewise, the utility of an action might not become obvious immediately. It might turn out later that a particular action moved the agent onto a path that becomes useful in the long run. It is, however, a general requirement for standard supervised learning algorithms that each example

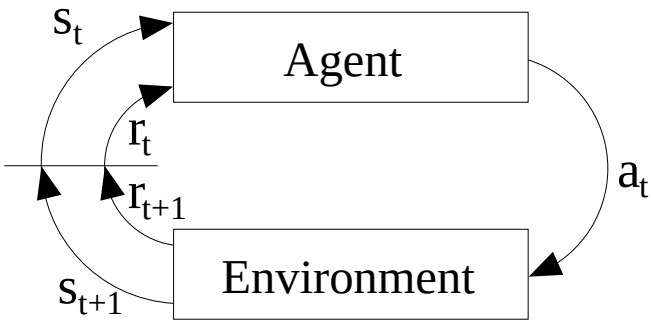



Figure 2.1: The cycle of sensing the environment, acting and receiving a reward in reinforcement learning agents. The agent receives situation $s_t$ and reward $r_t$ at time $t$ and answers with action $a_t$, which leads to a new situation $s_{t+1}$ and reward $r_{t+1}$.

can be evaluated instantly. Hence, they are not particularly suitable for behavior learning in situated agents.

*Reinforcement learning*, in contrast, is an online learning approach. It provides a definition for certain types of problems and a general framework for various algorithms. The agent is assumed to exist within its environment, frequently sensing its surroundings and taking actions to proceed towards its goals. Each action changes the *state* of the agent's environment and results in the agent receiving a *reward*, representing the utility of this state transition. The continuous cycle of sensing, acting and receiving a reward is illustrated in Figure 2.1. While taking an action is the responsibility of the agent, the information obtained from sensing and the reward received are determined by the environment. Even though the agent might have some information about how the reward is produced, it cannot alter the reward's definition. Reinforcement learning algorithms utilize the agent's reward history to estimate the utility of each state, which, in turn, might very well depend on the utility of states that can be reached from the state in question. The ultimate goal of such an agent is to maximize its cumulative long term reward. Therefore, an agent designer can bias the agent towards a particular, goal-directed behavior only by specifying rewards, states and actions appropriately. In particular, the correct utility of actions does not need to be specified explicitly, which is an inherent advantage over supervised learning in the case of situated agents (Sutton and Barto, 1998).

### 2.1.1 The reinforcement learning problem

Much work in reinforcement learning is based on the assumption that the agent's environment has the *Markov property*. This means that the change of the environment and the reward for taking a particular action only depend on the action and the environment state in which it was performed. In particular, the environment's response does not depend on the history of states, actions and rewards. More formally, the following is assumed to hold for all state transitions and histories:

$$\Pr\{s_{t+1} = s, r_{t+1} = r \,|\, s_t, a_t\} = \Pr\{s_{t+1} = s, r_{t+1} = r \,|\, s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\},$$

with $s_t$ denoting the state at time $t$, $a_t$ the action taken at time $t$ and $r_{t+1}$ the reward received as a response to this action. An important observation is that in such an environment all future states and expected rewards can be predicted based on the current state alone. Hence, making a decision in order to maximize the sum of future rewards can be substantially facilitated. It requires, however, that the state description provides a suitable summary of all the relevant information from the history (Sutton and Barto, 1998).

A reinforcement learning problem that fulfills the Markov property can be modeled as a *Markov Decision Process* (MDP), which is a tuple $M = (S, A, T, \gamma, R)$ (definition based on Ng et al. (1999)). $S$ denotes the set of possible environment states and $A$ the set of actions available to the agent. $T = \{P_{sa}(\cdot) | s \in S, a \in A\}$ is the set of probabilities $P_{sa}(s')$ for transitioning to state $s'$ when executing action $a$ in state $s$. $\gamma \in (0,1]$ specifies a discount factor, whose function will become obvious later. $R$ describes the reward distribution, which is usually assumed to be deterministic. In that case, the reward function is defined as $R : S \times A \mapsto \mathbb{R}$. If the agent executes action $a$ in state $s$, the environment yields the reward $R(s,a)$. In this work, we will make the simplifying assumption that the set of states $S$ is finite, which leads to a *finite Markov Decision Process*, respectively.

A *policy* $\pi : S \mapsto A$ for a given MDP denotes that action $a$ is taken in state $s$ with probability $\pi(s,a)$. A reinforcement learning agent seeks to learn a policy that maximizes its future cumulative reward when following this policy. The expected cumulative future reward or *expected return* $E_\pi$ for taking action $a$ in state $s$ and following policy $\pi$ thereafter is described in terms of the *action-value function*:

$$Q^\pi(s,a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|\, s_t = s, a_t = a \right\}. \tag{2.1}$$

The discount factor $\gamma$ determines the influence that later expected rewards have on the overall expected return. The *optimal policy*, whose expected return is larger or equal to that of any other policy for all states, is denoted as $\pi^*$. It can be directly obtained from the *optimal action-value* function $Q^*$, which is defined as (Sutton and Barto, 1998):

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \qquad \forall s \in S, a \in A.$$

By reformulation of Equation (2.1), it can be shown that the action-value function exhibits the following recursive relationship known as the *Bellman equation* (Sutton et al., 1999):

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') \sum_{a' \in A} \pi(s',a') Q^{\pi}(s',a').$$

The optimal action-value function becomes under the same transformations (Sutton et al., 1999):

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') \max_{a' \in A} Q^*(s',a').$$

Applied to every combination of states and actions, these equations yield another set of equations. They could possibly be solved analytically to find $Q^*(s,a)$ and thereby $\pi^*$ if a model of the environment was readily available. This solution, though, requires a huge computational effort and is therefore not feasible in general. It is also a strong assumption that a model is available from the first. More realistically, the model is learned by the algorithm and action selection provided by planning techniques such as *Dynamic Programming* (Sutton and Barto, 1998). However, in this work, only model-free problems and algorithms are of interest. Such reinforcement learning algorithms approximate the optimal action-value function iteratively by updating the value of a state and an action based on the reward received and the value of the successor state. Among the most well-known techniques are arguably Watkin's Q-learning (Watkins, 1989) and Sutton's temporal-difference TD($\lambda$) algorithm, which have been applied in a number of real-world problems (Mahadevan and Kaelbling, 1996). We will focus on the former in the following. Other prominent examples are discussed thoroughly by Sutton and Barto (1998).

After taking an action $a_t$ in state $s_t$, receiving reward $r_{t+1}$ and transitioning to state $s_{t+1}$, Q-learning applies the following simple update rule to the value-function:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1},a) - Q(s_t,a_t)], \tag{2.2}$$

where $\alpha$ is the learning rate and $\gamma$ the discount factor as described previously. Q-learning has been proved to converge to the optimal action-value function under the condition that states and actions are visited infinitely often and that some other minor restrictions are obeyed (Tsitsiklis, 1994; Jaakkola et al., 1994).

### 2.1.2 Exploration and exploitation

An optimal policy for the action-value function would be to always choose the action $a$ in state $s$ which maximizes $Q(s,a)$. In that case, however, the agent would never explore its options and very likely only exploit a behavior that is not globally optimal. A more subtle and common exploration strategy is the $\varepsilon$-greedy policy. With probability $\varepsilon$, the agent takes a random action and only otherwise chooses the action currently assumed to be optimal. Generally, trading exploration for exploitation effectively is a challenge for reinforcement learning agents. On the one hand, the agent has to exploit its current knowledge in order to achieve high rewards. On the other hand, it might discover better behavior that pays off in the long run if it tries alternative strategies.

The amount of exploration required for attaining an optimal policy and hence the complexity of reinforcement learning grows exponentially with the size of the state space and the number of actions available to the agent. If either of these is large and a reward is not granted until reaching a goal state, initial exploration effectively becomes a random walk in the state space. Not until then, the agent starts to propagate Q-values from the goal state on to previous states. Even traditional speedup techniques such as experience replay (Lin, 1992) or eligibility traces (Sutton and Barto, 1998), in which the propagation of Q-values is accelerated, do not help in that case. Under these conditions, a reinforcement learning problem simply becomes intractable (Whitehead, 1991). The possibility to learn only from rarely provided rewards and the resignation from rating the utility of every action explicitly then becomes a weakness. Hence, exploring the environment blindly or randomly as with the $\varepsilon$-greedy policy is not necessarily a clever approach. The agent could better make use of previously obtained knowledge or knowledge about the problem that is specified by a human domain expert in order to guide its exploration.

This observation basically leads to three major possibilities for improving the learning or exploration phase of a reinforcement learning agent. Originally, the values of the Q-function (*Q-values*) are stored in a table that is indexed by states and actions. In a realistic problem setting, this is not feasible because the state space might simply be too large. In that case, decision-making should take place in an abstracted state space that is created by dropping irrelevant features from the original one or by substituting

the Q-value table with a function approximation. *State abstraction* or generalization is often provided by neural networks. This way, the complexity of the problem can be decreased substantially. Since, an experience made in one state of the original state space will have an affect on the knowledge learned about all other states that exhibit some similarity to this one with regard to the abstracted state space. However, Sutton and Barto (1998) warn against using neural networks in particular because they require non-trivial expertise for their configuration. Apart from that, employing state abstraction can entail losing the possibility to converge to the optimal policy. Li et al. (2006) expand on the question of trading off information loss and state space reduction by giving a more formal analysis of state abstraction.

Complementary to state abstraction is *temporal abstraction*. Enabling decision-making at various levels of temporal abstraction has long been a key research topic in artificial intelligence (Sutton, 1995; Sutton et al., 1999). If an agent is able to solve a problem at one level of detail after the other, it can constrain its options to those relevant at the current level of reasoning. Exploration becomes faster because the agent can effectively take larger steps within the state space. We will study temporal abstraction in detail during the next subsection.

Another opportunity to speed up learning arises from the finding that not even humans tackle a problem without having any *prior knowledge* about how to solve it. We either obtained this knowledge from our own experience or from advice by teachers. Biasing a reinforcement learner with prior knowledge has thus been identified as an important research topic and has been studied widely (Mahadevan and Kaelbling, 1996). We will discuss this topic in Section 2.1.4.

### 2.1.3 Temporal abstraction in reinforcement learning

Temporal abstraction in reinforcement learning mainly has its seeds in two ideas: macro-actions or temporally-extended actions on the one hand, and hierarchical reinforcement learning on the other. *Macro-actions* are closed-loop policies that compete for execution with primitive actions in every state. If a macro-action is chosen for execution, the agent follows its policy until its termination condition is met. The Q-value of the state $s_t$ in which a macro-action $m_t$ was started is updated based on the rewards $r_i$ received during its $n$ steps and the Q-value of the termination state $s_{t+n}$:

$$Q(s_t, m_t) \leftarrow Q(s_t, m_t) + \alpha[\gamma^n \max_{a \in A} Q(s_{t+n}, a) - Q(s_t, m_t) + r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n}]. \quad (2.3)$$

The Q-values of states visited in between are updated according to Equation 2.2. McGovern et al. (1997) define macro-actions as fixed policies prior to their application. They observe that depending on their definition and the task at hand, the impact on performance can either be significantly positive or negative. On the one hand, exploration can be accelerated because the agent can take larger steps in the state space and reach remote states earlier. On the other hand, the Q-value propagation traverses the state space faster. The latter, however, can also be achieved by Lin's (1992) experience replay or eligibility traces, which have a similar effect. These techniques do not have any influence on exploration behavior, though. In a similar way, model-based reinforcement learning algorithms have been augmented to allow planning with temporally-extended actions (Sutton, 1995; Precup et al., 1997, 1998).

Similar to hierarchical task network (HTN) planners (Sacerdoti, 1975), hierarchical reinforcement learning systems allow a problem to be broken down into a hierarchy of subtasks. Thereby, they make use of prior knowledge about potential decompositions of the problem in order to compress the possibly large state space. The agent's execution follows the hierarchy, while its decisions at every choice point are constrained by the guidance of the hierarchy. Then, optimal decisions are only to be learned for the choice points. Such hierarchical problem specifications are described, for example, by finite state machines (Parr and Russell, 1998), partial, non-deterministic programs (Andre and Russell, 2002; Shapiro, 2001) or unordered trees (Dietterich, 1998). These approaches generally drop optimality in favor of a significantly decreased learning time. The difference to macro-actions is that developers anticipate the applicable situations for a certain subtask. Macro-actions, in contrast, can be applied freely in any state. They are considered an augmentation of the learning problem, while a strictly hierarchical decomposition rather provides an abstraction by restraining reasoning to parts of the state space at each hierarchy level (Jong et al., 2008). It is not clear how such a hierarchical decomposition can adapt to dynamics in the environment, for example if the task changes intrinsically.

Sutton et al. (1999) generalize those two streams in the *options* framework. Each option consists of a set of states $I \subseteq S$ called *initiation set* in which it is applicable, a policy $\pi : S \times A \mapsto [0,1] \in \mathbb{R}$, and a stochastic *termination condition* $\beta : S^+ \mapsto [0,1] \in \mathbb{R}$. The execution process is the same as for macro-actions with the slight difference that the applicability of an option can be restricted to a subset of all states. The authors propose an extension which allows policy and termination condition to depend on all states taken previously during the execution of the option. Such options are called *semi-Markov* because their decision-making does not only depend on the current state. Respectively, the learning algorithm as defined in 2.3 is called SMDP *Q-learning*.

The policy of an option can decide to execute another policy, thus giving rise to hierarchically decomposed behavior. The authors also demonstrate how options can be interrupted during their execution in favor of better alternative behavior. The definition of options obviously subsumes the specification of macro-actions and hierarchical reinforcement learning tasks.

**Automatic option discovery**

It still remains an open question how options come to life. Especially the hierarchies in hierarchical reinforcement learning algorithms need to be specified explicitly. Sutton et al. (1999) propose learning the policies of options the same way policies are learned by standard reinforcement learning algorithms. They define the termination states of an option as a subgoal and associate a reward value with them. Then, standard reinforcement learning algorithms can be applied to learn a policy for reaching the subgoal states from given initiation states. However, still initiation conditions and subgoals need to be provided explicitly.

Stolle and Precup (2002) propose an algorithm that discovers these subgoal states automatically. They randomly generate problems in a gridworld problem in order to obtain visitation counts for states. The highest counts indicate possible subgoals. States that are found on trajectories passing by a certain subgoal often enough are considered to be part of the particular option's initiation set. For each pair of initiation set and subgoal, a policy is learned using Q-learning. Only after options have been learned, SMDP Q-learning is applied to learn a policy over these options and primitive actions. Hence, the agent is required to obey a two-stage process, which we notice as unsuitable for situated agents.

McGovern and Barto (2001) propose a similar approach, which learns subgoals online, though. Subgoals are identified by applying data mining techniques to previously experienced trajectories and visitation counts, which are only based on the first visit of a state in each trajectory. Only states that lie on successful trajectories and not on any unsuccessful one are considered for counting. It is assumed that a domain-dependent success condition for trajectories is readily available. The authors observe that subgoals are more clearly identified using these restrictions. The initiation set of an option is created by collecting all states that have been visited prior to reaching the associated subgoal state on any of the relevant trajectories within a certain time frame. The policy for an option is then learned using experience replay with these trajectories.

The general problem with all these approaches is that options are not available until the goal has been reached for the first time. Hence, options cannot be of help during the most critical period of learning, in which the agent is basically conducting a random walk. More recent approaches analyze the structure of the agent's transition graph, either globally (Menache et al., 2002) or locally (Özgür Şimşek and Barto, 2004; Özgür Şimşek et al., 2005). Subgoals are assumed to be bottlenecks in the graph, which can be passed by the agent to enter other strongly connected regions that possibly have not been visited yet. Analyzing the global transition graph has the drawback of requiring a copy of the agent's entire transition history. Only addressing a recent part of this history, on the contrary, requires additional parameters to specify the number of transitions to account for. Both these algorithms rely on experience replay to learn options for discovered subgoals.

**The impact of temporal abstraction**

Jong et al. (2008) conduct a more detailed investigation on the benefits of using options. They conclude that the positive effect of experience replay, which is usually employed to obtain the options' policies, conceals the mere effect of introducing temporal abstraction. To isolate both benefits, they propose an alternative definition of options as subtasks, which explicitly does not include the policy but rather defines an option as a subproblem specification. A policy is not learned ad hoc directly after the subgoal has been discovered, but rather concurrently with the overall policy. The agent basically learns an overall policy and a policy for each subgoal at the same time. The authors note that automatically discovered options can heavily distract exploration similarly to fixed macro-actions. Options are generally selected equally to primitive actions, such that exploration is implicitly biased towards their subgoals. If the goal can only be reached with a number of primitive actions from such a subgoal, it is less likely to be attained by a random walk than without utilizing options. Jong et al. also note that in order to make full use of temporal abstraction, an agent should be able to generalize from the application contexts of temporally-extended actions. This is the main goal of approaches that learn subgoals and useful temporal abstractions by examining policies obtained in related tasks (Thrun and Schwartz, 1995; Pickett and Barto, 2002). They seek to discover subpolicies that can be transferred to future tasks.

Jong et al. do not consider the benefit that options contribute to the understandability of learned knowledge. If a problem is broken down into subtasks, it can be more easily understood by humans. In addition, they do not account for the opportunity to simplify decision-making. Because options are defined over a part of the state

space only, they could actually constrain reasoning to the relevant states and actions. Moreover, the number of complex decisions necessary at the high-level policy can be reduced because control is shared with the options' reasoning processes. This is especially interesting when sensing the environment and subsequent decision-making are costly.

### 2.1.4 Prior knowledge in reinforcement learning

As mentioned above, a reinforcement learner is likely to end up exploring its environment randomly until it receives a reward for the first time (Whitehead, 1991). This surely does not work for large state spaces as found in real-world applications. Incorporating prior knowledge about the task in order to decrease this learning time has hence been recognized as a major requirement for feasible reinforcement learning systems (Mahadevan and Kaelbling, 1996). We already have seen possibilities for guiding a reinforcement learner by exploiting temporal knowledge about the task in the previous subsection. We will expand upon further possibilities in the following.

Integrating prior knowledge into reinforcement learning follows two primary goals: Either guiding exploration towards promising actions or interesting parts of the state space; or reducing complexity by constraining the agent's choices. Prior knowledge is specified about the agent's environment, the problem itself or the goal states (Hailu and Sommer, 1999). It can be obtained from a related task, from another agent or a domain expert. It can be rather intrusive by directly manipulating the knowledge base of an agent or it can give hints to the agent, which it is free to pickup or not. However, coping with inaccurate domain theories remains a problem. Unfeasible prior knowledge can definitely harm an agent's performance (Hailu and Sommer, 1999).

Hailu and Sommer (1999) discuss options and effects of reflecting prior knowledge in the initialization of Q-values when a table-based Q-function is employed. Koenig et al. (1996) show that in order to reduce complexity of a reinforcement learning problem, such a Q-value initialization can be based on heuristics that are consistent or admissible for A$^*$-search[1]. However, this work is restricted to the theoretic properties of reinforcement learning as only a table-based Q-function is assumed. The methods proposed are not feasible for large state spaces, for which the effort of initializing Q-values would be tremendous. Results cannot be directly transferred to systems that deploy function approximation for state abstraction.

[1] For an explanation of $A^*$-search refer to Russell and Norvig (2003).

Artificial neural networks are a common choice for function approximation in reinforcement learning. Influential techniques for priming these are the EBNN and KBANN algorithms (Mitchell and Thrun, 1993; Shavlik and Towell, 1989). Both incorporate domain theory in their processes of learning a network. Prior knowledge is either represented by a set of previously trained neural networks or a set of propositional horn clauses, respectively. Still, domain knowledge has to be encoded on a fairly fine-grained level. In fact, even configuring a function approximator or identifying a suitable state abstraction often requires itself prior knowledge about the specific structure of the learning task.

Instead of directly modifying the agent's underlying knowledge base, Dixon et al. (2000) guide the exploration of a reinforcement learning controller by a number of multiplexed static controllers. The system is able to decide when a particular static controller does not give beneficial advice anymore, for example because the environment has changed inherently or the learning controller has achieved a better policy. The authors observe a substantial decrease in learning time. Price and Boutilier (1999) decouple the source of prior knowledge even more from the learning agent. They propose an algorithm that a reinforcement learning agent can apply to learn by observing a teaching agent. However, the agent is assumed to know its reward model.

Lin (1992) train an agent with one or more possible solutions to the problem. These experience traces or sequences of actions are presented to the learning algorithm repeatedly to increase the impact on the policy being learned. A substantial speedup of learning is noticed. Maclin and Shavlik (1996) introduce a system that takes advice in a simple programming language, which supports higher-level constructs such as loops or decisions. Advice is integrated into the agent's knowledge base by an extended KBANN method and can be injected at any time during the agent's life cycle. Having a more powerful language to specify prior knowledge alleviates the work of domain experts and facilitates their acceptance of the system. In particular, we will refer to prior knowledge that comprises temporal knowledge about a task as *temporal prior knowledge*.

Indirect guidance of a reinforcement learner can also be provided by *shaping* – extending the received reinforcements by a domain-dependent function and thus motivating the agent to visit particular states or to take particular actions (Ng et al., 1999; Wiewiora et al., 2003). The invariance of the optimality of policies under this transformation has been proved by Ng et al. (1999) for certain conditions. Already simple reward transformations can yield a strong speedup. However, they might be difficult to identify.

## 2.2 Hybrid agent architectures

As outlined in the introduction, hybrid agent architectures are characterized by their intention to bring together reactive and deliberative architectures. Agents that apply one of the former are mainly defined by their ability to react to external events instantly. The latter specify agents that carefully reason about their environment first in order to obtain an anticipatory course of actions for reaching their long-term goals. We argued that both have their shortcomings, which are, for example, addressed by BDI architectures. Even though PGS is intended to be embedded into the BDI framework, we will take a more general view here, accounting for its capability to serve as an agent architecture in its own right. We note that PGS has a strong relationship with horizontally layered architectures, in which each layer independently suggests a behavior based on current sensory input (Weiss, 1999). To narrow the scope of this discussion down further, we focus here on architectures in which higher-level layers or modules obtain temporally abstracted knowledge by extraction from lower-level modules. In that, this is closely related to the previous discussion on option discovery in reinforcement learning. A more encompassing discussion of architectures related to PGS can be found in Karim (2009).

CLARION or "Connectionist Learning with Adaptive Rule Induction ONline" is a hybrid agent architecture that stems from work in cognitive psychology and was developed by Sun (1997). It is based on the assumption that human cognition processes generally work at two knowledge levels, one of which is *implicit* and the other *explicit*. The former is assumed to be executable but distributed and inaccessible for manipulation. It is implemented by a neural network, whose knowledge inherently meets the requirement of being distributed over a number of neurons. Explicit knowledge, in contrast, can be more readily interpreted and is accessible for modification. In CLARION, explicit knowledge is implemented by symbolic, propositional state-action rules. At each decision point, both levels provide a weighted recommendation for the next action. The system chooses one of these probabilistically.

Because of the different knowledge representations in CLARION, both levels need to apply different learning strategies. The implicit bottom-level module employs Q-learning to adapt its neural network. Obviously, it can not explicitly manipulate its knowledge. The top-level module learns by extracting symbolic rules from the bottom-level during execution. These rules are later on specialized or generalized based on the outcome of subsequent executions. Sun et al. (2005) note that explicit learning dominates in simple tasks with a small input dimensionality. Otherwise, implicit learning is more feasible because its generalization or approximation capabilities

handle complexity more effectively. This observation is supported by the criticism towards symbolic reasoning as discussed previously. However, the power of symbolic manipulations such as planning or hypothesis testing are only available to the top-level. Having different knowledge representations be manipulated concurrently leads to positive synergy effects. Learning does not only work bottom-up but indirectly also top-down because explicit knowledge guides the agent towards relevant features or information. Both learning speed as well as asymptotic performance are assumed to benefit from this interrelation (Sun et al., 2005). Sun and Zhang (2004) note the possibility to encode prior knowledge as rules in the top-level. The bottom-level would initially rely on the guidance of the top-level and gradually gain more responsibility while it is augmenting its knowledge. However, specifying more complex prior knowledge in the form of state-action rules only is cumbersome.

CLARION does not integrate the possibility for explicit planning even though Sun et al. (2001) found that human subjects were subconsciously applying planning techniques in specific tasks. Because CLARION is outperformed by humans in these tasks, Sun et al. assume the lack of planning to be one of the inherent disadvantages of CLARION. Sun and Sessions (1998) and Sun (1999) propose an algorithm that extracts plans from learned Q-values. They use beam search to find the path that is most likely to reach the goal based on the assumption that a Q-value indicates the probability of reaching the goal from that state. A variant of the algorithm extracts conditional plans, taking into account a certain number of alternative states and actions in each step. The process is run offline after Q-learning has been applied. However, Sun and Sessions note that plan extraction already yields reasonable results with Q-values not having converged yet.

The explicit-implicit distinction is also peculiar to the ACT-R cognitive architecture (Anderson, 1990). On the explicit level, it hosts declarative and procedural or associative knowledge in the form of production rules. The implicit level is comprised of statistical information related to the frequency in which specific explicit knowledge is accessed. Essentially, this affects the likelihood that certain knowledge is retrieved or certain productions are reused. Hence, this is different to knowledge separation in CLARION, where information was simply kept redundantly on both levels. Knowledge in ACT-R is arranged within problem spaces, thus effectively reducing the amount of data to consider in a particular situation. Learning either seeks to assess the utility of declarative knowledge or to identify new knowledge instances or productions (Lebiere et al., 1998). Lebiere and Wallach (2001) describe how temporal knowledge can be encoded on the implicit level by associating consecutive productions. However, it is only an indirect description of temporal knowledge and

not a direct one. ACT-R has been frequently used in cognitive modeling (Bryson, 2000).

Garland and Alterman (2001) propose a technique that allows agents to learn coordinated procedures. Successful execution traces of coordinated activities are analyzed, summarized and stored in a case-base. These cases mainly contain expectations about communication partners and coordination points as well as suitable application contexts. These procedures can be recalled, for example, when an associated communication request is received and a longer interaction is expected. Olivia et al. (1999) also employ case-based reasoning[2] in order to reuse previously applied plans in BDI agents. In a planning situation, previous cases are searched first, possibly a suitable one retrieved and eventually adapted to the current situation. Similarly, concrete planning sequences in the agent architecture PARIS are transformed to abstract cases (Bergmann and Wilke, 1996). The translation process is guided by user-defined rules. On demand, an abstract case can be translated back to a concrete plan instance. Those abstract operators in the abstract case guide the planner to find a concrete implementation for that plan. In general, these approaches rely heavily on classical planning tools and languages.

Common to the architectures presented above is the organization of knowledge on two distinct levels, both of which apply different representations and possibly reasoning techniques. For case-based reasoning approaches, this leads to a reduction in planning time because of the reuse of previous reasoning results. In the case of CLARION and ACT-R, the interactions between implicit and explicit knowledge lead to positive synergy effects, which have a favorable impact on learning performance. However, case-based reasoning requires classical planning techniques, plan extraction for CLARION is rather an offline learning approach, and ACT-R sequence learning does not yield an explicit plan representation. These observations motivate the development of a hybrid architecture that extracts explicit plans from implicit knowledge in an online manner and reuses them competitively with its implicit knowledge. In order to facilitate plan reuse, acquired plans should be abstracted in a way such that they become applicable in a broader context as done in the case-based reasoning approaches.

[2] For an overview of case-based reasoning and case-based planning refer to Mitchell (1997) and Ghallab et al. (2004), respectively.

# 3 The Plan-Generation-Subystem

In the previous chapter, we have discussed the benefits of temporal abstraction and the advantages of hybrid agent architectures. We have identified situated, resource-bounded agents as the main interest of this work. In this chapter, we will introduce PGS as a hybrid agent architecture that extracts temporally abstracted knowledge from implicit knowledge using an online learning approach. We will first describe plans in more detail because they provide the representation of temporal knowledge in PGS (Section 3.1). Then we will present and review the original PGS model in Section 3.2. We will introduce various extensions to the model in Sections 3.3 and 3.4. Finally, we will discuss the integration of prior knowledge to the model in Section 3.5.

## 3.1 Plans

In classical planning agents, a plan is the product of deliberative reasoning and describes a recipe for reaching a certain goal (Ghallab et al., 2004). Particularly in STRIPS-like planning, a plan essentially consists of the following entities (Fikes and Nilsson, 1971):

**Precondition** The precondition determines the situations in which the plan is applicable.

**Goal** The goal specifies the conditions that this plan is supposed to achieve.

**Body** The body defines which actions to execute in sequence from the precondition in order to achieve the goal.

Plans might generally follow more complex control flows, though, for example including conditions, loops and hierarchical decompositions (Ghallab et al., 2004). Likewise, goals might be more descriptive, allowing, for example, to have the execution of entire action sequences as an objective (van Riemsdijk et al., 2008). However, the concrete execution of a plan will generally turn out to be a plain action sequence. This is the case we are interested in here. We note that plans as specified above are

subsumed by the options framework, which was introduced in Section 2.1.3. A plan's precondition can be described by an option's initiation set, the goal by the states in which the option's termination condition holds, and the plan's body by a deterministic policy.

The task of plans in the context of BDI agents is slightly different, which mainly stems from the fact that they are typically not obtained by reasoning at runtime. Instead, a plan is defined by developers and brought to execution once the agent has committed to the intention of performing this plan in order to achieve its associated goal. Further reasoning is restricted to options that are compatible with current intentions, thus effectively focusing and accelerating decision-making (Pollack, 1992; Rao, 1997). This might lead to suboptimal behavior but generally saves the agent time and resources. BDI plans are usually only partial, meaning that they are not supposed to attain a final goal. Instead, they only seek to achieve subgoals, possibly involving the spontaneous execution of other plans (Bratman et al., 1988).

A plan constitutes explicit information about the sequentiality of actions. Hence, the knowledge about when to apply a plan contains more information than the knowledge about the applicability of each of these actions alone. However, it requires less information to be stored because only the plan's application context or precondition needs to be kept. Therefore, a plan is basically a compression of lower-level knowledge, which also makes it particularly suitable for communication to other agents (Sun and Sessions, 1998). Moreover, having temporal knowledge about the intentions of other agents allows more informed reasoning about the coordination of behavior (Pollack, 1992). Having a clear goal and comprising temporal knowledge, plans can also be more readily understood by humans. They have this advantage over simple state-action rules, from which temporal knowledge is difficult to obtain (Karim, 2009).

## 3.2 The original model

PGS serves as the top-level module of a hybrid agent architecture to extract knowledge from a low-level learner (Karim et al., 2006a). It monitors the execution of the bottom-level and records sequences of actions as reusable plans based on domain-dependent heuristics. The bottom-level module can be any system that proposes atomic actions for execution. It is usually a rule-based system with a low-level knowledge representation. Here, the bottom-level is implemented by a reinforcement learning system that learns state-action rules. Plans are not only recorded but also reused when they are expected to be successful.

Plan extraction in PGS relies on the problem being broken down into a set of *subgoals* by a domain-expert, which do not have to relate to each other, though. Each subgoal is assigned a triggering condition called *clue* that determines whether an action would move the agent towards this very subgoal. Actions are generally appended to a currently recorded plan as long as they make its clue condition hold and thus move the agent towards its associated subgoal. The *precondition* of a plan is the situation in which its first action was recorded. To make plans comparable, each one of them is assigned an *expected utility*, which is generally based on the reinforcement learning rewards received during its recording and the confidence that the agent has in this plan. The confidence rises or falls with the plan achieving its original reward again on subsequent executions or not. The expected utility is also supposed to be compared with the expected rewards of actions proposed by the bottom-level learner in order to decide which one of them to execute. Hence, a plan in PGS has a similar definition as a classical plan as described in the previous section. It consists of:

- A precondition, in which it started to be recorded and in which it is assumed to be applicable again.
- A goal or subgoal, describing the intention this plan is supposed to fulfill.
- A plan body, consisting of a simple action sequence.
- An expected utility, making plans comparable with each other and with single actions.

In every step, the system makes a decision on whether it performs the action proposed by the bottom-level module or a plan from its plan library unless it is currently executing a plan anyway. This decision is based on a number of factors, some of which include the availability of any plan for the current situation and the difference of its expected utility and the expected reward of the single bottom-level action. Hence, PGS only requires the current situation, the action proposed by the bottom-level module and its expected reward to be provided for its decision-making. This tuple defines the interface between PGS and the bottom-level module. The system architecture is illustrated in Figure 3.1.

Because the details of PGS have already been described thoroughly by Karim (2009), we will refrain from repeating this information here and instead focus on the issues important for this work. The execution cycle of PGS is depicted in Algorithm 1. If the agent is currently executing a plan, the next action in sequence is chosen to be performed (line 3). Otherwise, the bottom-level module is asked to sense the

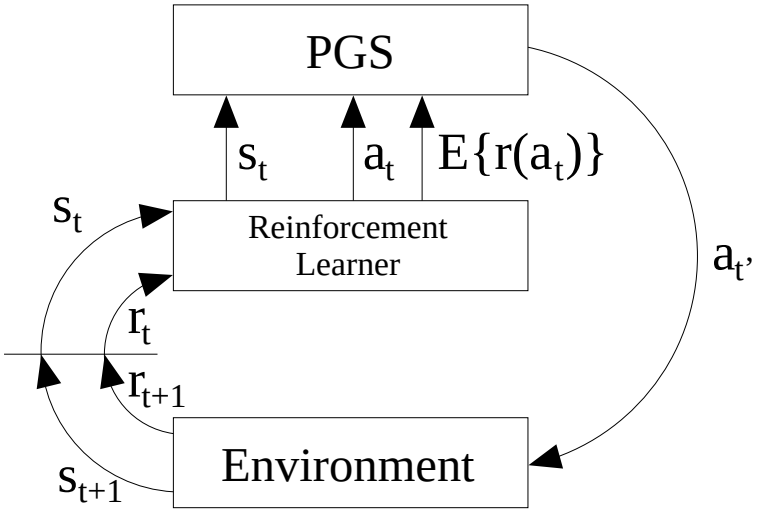



Figure 3.1: The abstract PGS architecture and execution cycle. The agent receives situation $s_t$ and reward $r_t$ at time $t$. The reinforcement learner passes the state information and a proposed action $a_t$ together with the expected reward $\mathrm{E}\{r(a_t)\}$ for that action to the PGS module. This decides on either performing a plan or the bottom-level action and answers to the environment with action $a_{t'}$. This leads to a new situation $s_{t+1}$ and reward $r_{t+1}$. The reinforcement learner is omitted in this process while a plan is executed.

environment and make a proposal for an action (line 6). Then, PGS tries to match the current state with the precondition of any of its plans. If any of the found plans' expected utility exceeds the expected reward of the bottom-level action, that one with the highest expected utility becomes a candidate for execution (line 7).

The *expected utility* of a plan is defined by the factor of the reward that the first action in the sequence received when it was recorded and the confidence the agent has in the plan's success. The confidence is initialized with one on the plan's construction. During subsequent reuses, it is reinforced or penalized depending on its success. Penalizing happens when a plan action turns out to be non-executable or the final reward cannot meet the plan's reward. Otherwise, the confidence is reinforced. The default behavior is that penalizing decrements the confidence value by one if it was two or larger before. The current confidence value is halved if it is smaller than

**Algorithm 1** The PGS execution cycle

**Require:** $C_i = (s_i, a_i, \mathrm{E}\{r(a_i)\})$ is the state-action-reward tuple at execution step $i$, in which $s_i$ is the state, $a_i$ the action and $\mathrm{E}\{r(a_i)\}$ its expected reward. $\mathcal{G}$ is the set of subgoal-clue tuples. $i_{\text{thres}}$ is a threshold number of execution steps, such that plan recording is activated after the $i_{\text{thres}}$-th step. Initialize plan library $\mathcal{P}$, set selected plan $p_s = \text{null}$.

```
1:  for each execution step i do
2:    if IsExecutingPlan() then
3:      ExecutePlan()
4:      continue
5:    end if
6:    s_i, a_i, E{r(a_i)} ← BottomLevel()
7:    p_s ← SelectPlan(s_i, a_i, E{r(a_i)}, G, P)
8:    if p_s ≠ null then
9:      ExecutePlan()
10:     continue
11:   end if
12:   if i ≥ i_thres then
13:     AssociateToPlan(s_i, a_i, E{r(a_i)}, P)
14:   end if
15:   ExecuteAction(a_i)
16: end for
```

two. Reinforcing increments the confidence value by one independent of its current value. A plan is removed from the library if its expected utility drops below a certain threshold. In that, a plan can be assigned a specific *life time*, namely the time between its recording and its removal from the library. The confidence update policy described above was retained from previous work, for which it was empirically obtained (Karim, 2009).

If a plan was selected, its first step is executed (line 9). If no suitable plan was found, the bottom-level action is considered a possible plan action. If it contributes to the subgoal of any currently recorded plan, it is appended. If a plan is not being recorded and the bottom-level action contributes to one of the subgoals, it becomes the first step of a new plan associated to this subgoal with the current situation as its precondition (line 13). If the plan reaches a certain minimum length, it is activated

and thus becomes available for selection later. If no plan action can be performed, the bottom-level one is carried out (line 15).

## 3.3 Discussion

The architecture of PGS is largely inspired by CLARION, which was introduced in Section 2.2. Like CLARION, PGS hosts two different knowledge representations and reasoning techniques on two different layers. Similarly, the main knowledge transfer works bottom-up. However, the top-level of PGS hosts plans, which are more abstract than propositional rules. The external plan extraction capability of CLARION only works offline[1] whereas PGS is an online learning approach, which might foster plan extraction in situated agents. PGS does not apply any top-down learning such that bidirectional synergetic effects as in CLARION cannot be observed. The general approaches of execution monitoring for plan extraction and storing plans in a case-base are also found in architectures that apply case-based reasoning as discussed in Section 2.2.

As shown already by Karim et al. (2006a,b, 2008), PGS is indeed able to extract reasonable and also relatively long plans from the bottom-level reinforcement learner. By extracting plans from low-level rules, PGS obtains temporal and hence more abstract knowledge. The control policy described by the low-level learner is compressed and becomes more accessible for inspection and suitable for communication. By committing to a course of actions once, the agent can omit some part of the otherwise necessary decision-making. This is especially favorable if sensing the environment or reasoning is expensive or only rarely possible.

Plans can be integrated seamlessly to the plan library of a BDI agent as outlined in Section 3.1. Likewise, plans in PGS can be described by the options framework, which was introduced in Section 2.1.3. One of their main restrictions regarding these formalisms, however, is their lack of hierarchical decomposition. This follows naturally from the procedure of plan extraction used in PGS. Another main difference is their notion of goals, which does not consider a goal as a set of particular states to reach but rather as a classification of each action. It is a strong assumption that this knowledge is readily available to the agent designer. In fact, if one is able to classify every action as either helpful or not, the motivation for employing a reinforcement

[1] Taking into account that according to Sun and Sessions (1998) plans can be extracted before the Q-learning algorithm has converged, one might imagine an instance of CLARION that, in fact, extracts and uses preliminary plans before convergence.

learner as the bottom-level module seems questionable. Taking into account that immediate knowledge about the usefulness of every action is available, a supervised learning system might be more suitable as reasoned in Section 2.1. Actually, PGS as described in the previous section even goes a step further. In order to decide whether a proposed action is supportive of any subgoal, its outcome has to be predicted because its actual execution only happens after it is considered to be appended to a plan.

PGS has the capability of discovering options from a reinforcement learner. This process has an advantage over other option discovery algorithms because it proceeds inherently online and does not bias exploration towards particular states. It rather biases exploration towards particular action sequences, which previously have already been found to be useful. This advantage also stems from the fact that plans are not potentially used in every possible state but only in those allowed by the plan selection process. This renders their influence on exploration less substantial and more focused. As opposed to options, plans are not fully integrated to the same decision-making process as primitive actions, which are essentially their competitors. Plans rather override low-level behavior as done in Brooks's subsumption architecture. That makes balancing plan execution and primitive action execution more difficult.

In fact, comparing the expected utility of a plan and an action is not straightforward. According to the description in Section 3.2, the expected utility of a plan is essentially determined by the reward that its first action received. This does not take into consideration that later steps might have much lower or higher rewards. The comparison with primitive actions also does not account for the rewards possibly received after deciding to take the primitive action. A reward is basically not a reliable long-term indicator for the value of a state or an action and hence rather not suitable as an indicator for the value of a plan. Furthermore, assuming that the reward of an action can be predicted is also a strong assumption, which is generally not made in reinforcement learning. In the options framework, in contrast, the value of an option is determined by its Q-value, which does only depend on the agent's experience and not on insight knowledge.

Likewise, the agent designer is assumed to have knowledge about the convergence time of the underlying reinforcement learner. She has to have knowledge about $i_{\text{thres}}$ – the state-action-reward tuple from which on PGS is allowed to record plans. This knowledge is not necessarily easy to obtain nor might it be available at all.

There is space for improvement regarding the recording of plans. So far, plan recording is interrupted if another plan is selected for execution. This will often lead to shorter plans being acquired than would actually be possible.

**Algorithm 2** The extended PGS execution cycle

**Require:** $C_i = (s_i, a_i, \underline{q_i})$ is the state-action-reward tuple at execution step $i$, in which $s_i$ is the state, $a_i$ the action and $\underline{q_i}$ the Q-value of that action. $\mathcal{G}$ is the set of subgoal-clue tuples. ~~$i_{\text{thres}}$ is a threshold number of execution steps, such that plan recording is activated after the $i_{\text{thres}}$ th step~~. Initialize plan library $\mathcal{P}$, set selected plan $p_s = \text{null}$.

1: **for each** execution step $i$ **do**
2: $s_i \leftarrow \text{BOTTOMLEVEL}()$
3: **if** ISEXECUTINGPLAN() **then**
4: $a_i, q_i \leftarrow \text{EXECUTEPLAN}(s_i)$
5: $\text{ASSOCIATETOPLAN}(s_i, a_i, q_i, \mathcal{P})$
6: **continue**
7: **end if**
8: $a_i, \underline{q_i} \leftarrow \text{BOTTOMLEVEL}()$
9: $p_s \leftarrow \text{SELECTPLAN}(s_i, a_i, q_i, \mathcal{G}, \mathcal{P})$
10: **if** $p_s \neq null$ **then**
11: $a_i, q_i \leftarrow \text{EXECUTEPLAN}()$
12: $\text{ASSOCIATETOPLAN}(s_i, a_i, q_i, \mathcal{P})$
13: **continue**
14: **end if**
15: $q_i \leftarrow \text{EXECUTEACTION}(a_i)$
16: **if** ~~$i \geq i_{\text{thres}}$~~ **then**
17: $\text{ASSOCIATETOPLAN}(s_i, a_i, q_i, \mathcal{P})$
18: **end if**
19: **end for**

## 3.4 The extended model

Having identified the shortcomings of the original PGS model in the previous section, we will now describe the major changes we introduced. The extended algorithm is illustrated in Algorithm 2. Obvious changes to the original one are underlined or crossed out.

The first modification stems from the general observation that rewards are not the optimal indication for the value of an action if the bottom-level module is provided by a reinforcement learner. As described previously, Q-values provide more information because they suggest the rewards that are likely to be received when taking a particular

action and following the optimal policy from there on. This extended information is not contained in reward values. Hence, we generally use the Q-value of an action as its "expected reward" in the sense of PGS. We will from hereon denote the reward received by the environment as the *immediate reward*. Thereby, predicting the immediate reward of an action based on insight knowledge about the reward function becomes unnecessary because the Q-value, in contrast, is readily available. This also means that the reward of a plan is determined by a Q-value. In particular, a plan's reward is obtained from the maximum Q-value of the state reached by the last step during recording. In that, a comparison between the value of a single action (its Q-value) and a plan's reward is more reasonable. The plan's reward by being derived from a Q-value denotes the actual cumulative reward expected to be received after executing the plan and following the optimal policy thereafter. The Q-value of an action has the same meaning.

However, this still does not rule out the case in which a plan might have a higher reward than an action but the action would lead to higher rewards received later. Nevertheless, it reduces this risk, especially when a transition to the (only) goal yields the highest possible immediate reward. Then, higher Q-values indicate states closer to the goal, which means that eventually the inferior single action will get a higher Q-value if it really brought the agent closer to the goal than the plan. We generally allow actions to be appended to a plan if their Q-values are higher than the Q-value of the first action of that plan. The approach of exploiting Q-values for plan extraction is related to plan learning in CLARION as proposed by Sun and Sessions (1998) and Sun (1999).

To allow PGS to converge to the optimal policy at all, we modified the SELECT-PLAN method to include some random exploration. Even if a plan had a larger expected utility than the action proposed by the bottom-level, with probability $\varepsilon$[2], which is a parameter to the model, it executes the single action nonetheless. If any plans are found suitable for execution, one of them is chosen randomly. The plan with the highest expected utility has an $n$[3] times higher chance to be selected than the second one. That one, in turn, has an $n$ times higher chance than the third one and so forth and so on. The first step of the selected plan is then executed (line 11).

Another major modification addresses the general problem that decisions about appending actions to plans were originally depending on predictions of the actions'

[2] In this work, $\varepsilon$ is generally set to 0.1, which is a frequently used value in the $\varepsilon$-greedy reinforcement learning exploration strategy.

[3] In this work, $n$ is set to 4 because this was empirically found to be a good heuristic.

outcome. To know which subgoal is supported by an action, its outcome needs to be considered. To overcome this issue, actions need to be carried out first before any decision about their recording can be made. This is reflected in Algorithm 2 on lines 15 and 17, which are swapped as opposed to the original algorithm. As mentioned earlier, the recording of a plan is stopped once another plan is selected for execution. It would be preferable, though, that recording continues in that case such that the old plan becomes a suffix of the new one and a longer plan is obtained. This change is reflected on lines 4 and 5 as well as 11 and 12. The EXECUTEPLAN method returns the action performed by the plan and the Q-value of the new state. This information is fed into the ASSOCIATEPLAN method.

As previously discussed in Section 2.2, hybrid architectures can benefit from synergy effects between their layers. Originally, PGS only conducts bottom-up learning – extracting plans from the bottom-level module – although a possible top-down learning approach is straightforward. The reinforcement learner is now enabled to learn from the execution of plans, which thereby act as static controllers. Transfer of knowledge is simply achieved by updating Q-values during the execution of plans. This approach is similar to those of Maclin and Shavlik (1996); Lin (1992); Dixon et al. (2000) as depicted in Section 2.1.4. Making the knowledge transfer bidirectional is assumed to facilitate the performance of the overall system. This assumption will be subject to empirical investigations in Chapter 5. One trade-off implicated by this change needs to be mentioned upfront: Having the reinforcement learner learn during the execution of plans requires additional sensing of the environment, which could have been saved otherwise. This is reflected in lines 2 and 4, where the bottom-level is also required to sense the environment even if a plan is executed. However, no decision-making is necessary during plan execution.

We assume the plan confidence update process to be effective enough to foster the use of effective plans over that of less effective ones, with effectiveness being determined by the comparison of taking single actions and using the plan. Hence, we consider the restriction to start using PGS only after a certain number of state-action-reward tuples unnecessary. This way, another component that required additional knowledge is rendered obsolete. This is in line with Sun and Sessions (1998). They observed that plan extraction in CLARION is useful even if Q-values have not converged yet.

## 3.5 Prior knowledge in PGS

In Section 2.1.4, we discussed the definition of prior knowledge in a higher-level language, generally considering that prior knowledge was necessary to render any reasonably complex reinforcement learning task solvable. We propose here that plans as in PGS can be deemed to be such a language. In fact, BDI agents are typically defined by prior knowledge plans. Sun and Zhang (2004) observe the benefit of defining prior knowledge at a higher level to feed it down to a lower level knowledge base. We argue that the plan execution process of PGS provides a convenient hook for injecting prior knowledge into an agent. We will briefly discuss these options here and exploit them later on during the empirical evaluation. We will then also allude to the risks of priming an agent with prior knowledge.

The outline of Algorithm 2 immediately suggests four points that can benefit from prior knowledge:

**Subgoal-clue tuples** The more comprehensive the problem can be covered by distinct subgoals, the more plans can be recorded and the more accurate they are going to be.

**Plan library** The plan library can be primed with plans that serve as preliminary, partial solutions to the problem. Because the reinforcement learner is also learning during the execution of a plan as described in the previous section, it is likely to benefit from guidance by plans.

**SELECTPLAN method** Depending on the domain, it might be feasible to select plans for reuse in states that are similar to their original preconditions. This is in line with the argument of Jong et al. (2008) that temporal abstraction and state abstraction should go hand in hand.

**ASSOCIATETOPLAN method** Once the recording of a plan has been started, it is necessary to decide how long subsequent actions are to be appended. The decision could rely on the development of subsequent rewards or on knowledge about the domain-specific semantics of actions and their interrelations.

All these points provide non-intrusive and intuitive ways to specify prior knowledge, meaning that the fine-grained knowledge base and parameters of the underlying reinforcement learner do not have to be modified. They allow for encoding insight knowledge either about the problem or the environment. Preliminarily defined plans can be easily exploited to roughly guide the agent towards promising states or actions.

The impact of varying subgoal-clue tuples has been studied by Karim (2009). Here, actions will be appended to plans according to the specification in the previous section, which is led by Q-values. We will study priming the plan library and guiding the plan selection process in detail during the evaluation of the model in Chapter 5.

# 4 Implementation

In the previous chapter, we have introduced PGS and proposed a number of changes and extensions. In this chapter, we will adapt an engineering point of view and describe PGS on a more technical level. In particular, we will outline its architecture and depict possible extension points. However, implementation details are not of particular interest to this thesis because they do not contribute to answering the research questions. They are hence omitted. The following discussion will thus be rather concise.

## 4.1 Architecture

The implementation is based on previous work by Karim et al. (2008) and is provided in the JAVA programming language[1]. During the course of this thesis, the original source code was extensively re-factored and corrected. Effectively, a reimplementation of the system was conducted. The goal here was to increase the understandability and extensibility of the architecture in order to facilitate further research. The architecture with its main components and their interrelations is depicted in Figure 4.1. Most of these components correspond to entities introduced with the PGS algorithm in Section 3.2. We will present them briefly in the following.

### The `StateAction` class and the `State` and `Action` interfaces

The `StateAction` class corresponds to the main interface between PGS and the low-level module – the state-action-reward tuple. It is a container for concrete, domain-specific instances of the `State` and `Action` interfaces. The former holds the current state of the environment as it was sensed by the agent's low-level module. The latter corresponds to the action that the low-level module proposed for execution. It also contains its expected reward value. A `StateAction` object is passed to the `PGS` class in every execution step unless a plan is executed.

[1] http://java.sun.com

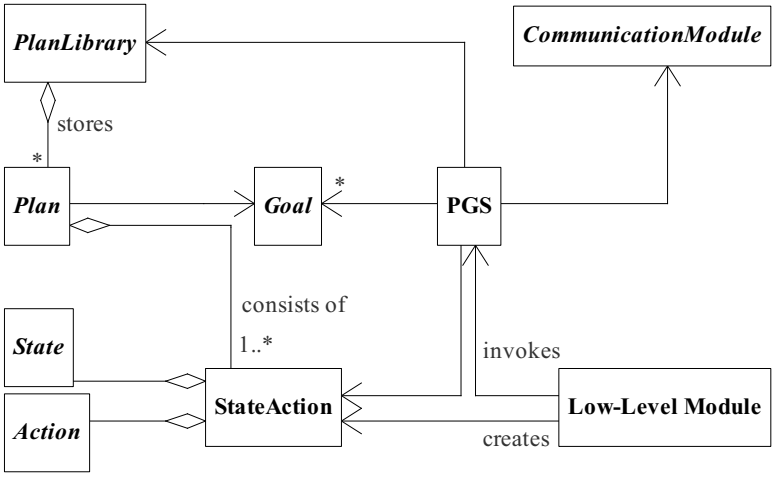



Figure 4.1: The concrete architecture of PGS with its main components.

### The `PlanLibrary`, `Plan`, and `Goal` interfaces

Plans are specified by concrete implementations of the `Plan` interface. Each plan is associated with a particular, domain-specific goal, represented by a concrete implementation of the `Goal` interface. In addition, a plan holds a list of `StateAction` objects, which correspond to the plan's action sequence. All plans of an agent are hosted in a `PlanLibrary` instance, which basically acts as a container.

### The `PGS` class

The `PGS` class is the central component of the system. It wires together all other components including the low-level module. From that, it receives a `StateAction` instance in every execution step, which is then fed to the PGS execution cycle as described in Section 3.2.

### The `CommunicationModule` class

Karim (2009) elaborated on strategies for communicating plans between agents. The `CommunicationModule` class is responsible for deciding on whether to send a particular plan to other agents or not and which received plans to keep for further reuse.

## 4.2 Extension points

From the architectural view described above, some possible extension points follow naturally. It should be noted that most of them were only uncovered by significant refactoring. We focus on those that are of general interest or relevant to the evaluation of PGS in the next chapter. In particular, we will omit those that only vary with the domain at hand and that have no general value, namely the `State`, `Action`, and `Goal` interfaces.

### Plans and plan management

Plans are central to PGS, so is their organization. Plans need to be created and deleted if they turn out to be useless. They also need to be indexed for fast access. It is essential that the system is able to determine plans that are applicable in a certain situation quickly. The plan library also might treat plans differently according to their age, for example. These requirements depend on the specific application of the system and are hence encapsulated in a separate module. Plans themselves might come in different shapes. In particular, each plan determines the states in which it is applicable itself. This allows for state abstractions be defined specifically for each plan. Likewise, plans are supposed to handle confidence updates autonomously. This gives rise to various opportunities. For example, prior knowledge plans are defined by the same interface as ordinary plans, which allows their seamless integration into the system.

### Communication

PGS itself is a single agent architecture. However, a PGS agent might possibly find itself in a multi agent scenario. In that case, plans can serve as a convenient way to exchange behavior policies. We elaborated on that in Section 3.1. Because communication is generally an important part of agent systems, its logic is encapsulated in a separate module. This module might, for example, use a threshold on the confidence values of plans to decide which ones to communicate. It could also decide on sending plans in bulk to reduce time spent on establishing communication connections. However, this component is not only responsible for sending plans but also for receiving them. It could apply reasoning about which received plans to incorporate into the plan library and which ones not.

# 5 Evaluation and discussion

In the previous chapters, we have introduced PGS, discussed implementation details, described the plan acquisition process, and identified limitations of the model. We have also proposed extensions to the model, which we seek to evaluate empirically in this chapter in order to support our claims. In general, we provide a careful and thorough hands-on analysis of the PGS architecture to depict its very peculiarities. We also expect new questions to arise that might encourage further investigations. First, we will briefly review the general hypotheses we seek to test.

Even though plan extraction observed in earlier work satisfied the expectations, plan reuse did not (Karim, 2009). No significant plan reuse was observed and mostly plans consisting of only one step were reused, which is basically nothing else than a single action. We hypothesize that the extensions proposed in Section 3.4 and the general re-engineering conducted lead to longer plans being reused and a generally high use of plans compared to single primitive actions.

Earlier, we argued for a plan as temporal knowledge comprising more information than the sum of its actions. Therefore, we hypothesize that reusing plans in situations that are beyond their original preconditions has a larger positive effect on performance than reusing a single action. We presume that this effect especially holds for the time of learning, when the low-level reinforcement learner still explores the environment. In this phase, it can particularly benefit from guidance.

In the same way, we expect temporal prior knowledge in the form of plans to facilitate the performance of such a system. We hypothesize that injecting plans as prior knowledge can increase performance even further. We have already demonstrated that the integration of temporal prior knowledge in PGS is straightforward, which we will make use of here.

In the next section, we provide the rationale for the choice of the domains used for the evaluation of the model. Subsequently, we will describe in detail the methodology that guided the experiments (Section 5.2). We will present and discuss results separately for the chosen domains in Sections 5.3 and 5.4. This chapter is concluded with a short summary (Section 5.5).

## 5.1 Domain choice

Estimating the plan reuse and performance of agents reliably requires that experiments are repeatable in a controlled way. Valid conclusions cannot be drawn from a single observation alone. Moreover, testbeds need to be free of external influences in order to facilitate the analysis. For these reasons, experiments are conducted using *toy domains* rather than real-world problems. This allows the explicit setting of test parameters and the controlled measurement of the variables under observation, which simplifies reasoning about results significantly.

The value of executing a plan as generated by PGS depends on the uncertainty in the domain, which is according to Decker (1995) comprised of the uncertainty about changes in the environment itself, about other agents' actions, and about the outcomes of actions. The predictability of the results of a sequence of actions decreases with its length and the uncertainty of the domain. This is a serious problem for classical planning. If a goal is supposed to be reached by a single plan, this plan has to account for all possible uncertainties it could encounter during its execution. The extreme case is a completely randomly behaving domain, in which the best policy is necessarily restricted to look ahead only a single step. In that case, temporal knowledge cannot be of any value. We seek to study the feasibility of PGS in domains of varying uncertainty and thus varying degrees of realism. This enables a reasonably thorough analysis of the architecture's value. However, to render plan reuse useful at all, the domains studied here exhibit certain temporal patterns, which can indeed be covered by temporal knowledge. This is supposed to shed more light on the advantages and disadvantages of temporal abstraction in general.

The first test bed is the *pursuit domain*, which was originally defined by M. Benda and Dodhiawala (1986) and has since been studied in various configurations (Stone and Veloso, 2000). In this domain, four predators seek to hunt down a prey on a grid without explicitly coordinating their actions. In order to win, the predators have to surround the prey. In the original task, the prey was moving randomly. Here, we vary its behavior from random to deterministic in order to study the impact of determinism. This domain hence covers two of the three factors of uncertainty defined by Decker: uncertainty about the environment and uncertainty about other agent's actions. The task is reasonably complex to be interesting because it waives one of the strong assumptions made in classical planning that the world is only changing due to the agent's own actions (Ghallab et al., 2004). It is generally amenable to the application of PGS because subgoals in the definition of PGS can simply be specified for the problem. Because the state space of this problem is large, state abstraction needs to

be applied, which renders plans reusable in different situations. This allows us to study the impact of plan reuse on the agent's exploration, with plans being learned or specified a priori.

In the second domain, the *taxi domain*, an agent is supposed to pickup a passenger at a certain position on a grid and deliver her to a particular destination. The problem has been studied frequently in literature on temporal abstraction because it exhibits distinct states that can be identified as subgoals easily (e.g. Dietterich, 1998; Parr and Russell, 1998; Özgür Şimşek et al., 2005). However, PGS cannot clearly be configured for this domain as we will see later. This allows us to demonstrate the inherent restrictions of the approach. However, we will modify the PGS model in order to record and reuse action sequences at least. The problem neither exhibits uncertainty about the environment nor about other agents. We also refrain from adding uncertainty about the agent's own actions. Hence, the advantage of plan acquisition and execution should become even more evident than in the pursuit domain.

## 5.2 Methodology

In both domains, agents have to reach a goal within a particular number of steps. We seek to evaluate the impact of deploying PGS on the reinforcement learner's initial exploration or learning phase. To assess performance, mainly the following dependent variables are observed during the experiments and considered in the discussion:

**Steps** The average number of steps that the agents need to reach their goal indicates their performance, with better performing agents requiring less steps. This is a more informed indicator for performance than the ratio of trials in which the agents reached their goal successfully. As opposed to success rate, it does not depend on the maximum number of steps allowed per trial, which itself is a parameter to the system.

**Decisions** The average number of decisions indicates the computational costs expected to be spent on reasoning. In this work, we are primarily interested in situated, resource-bounded agents. They need to restrict costly decision-making in order to obey time-constraints. Minimizing decision-making is a goal of the extensions to PGS introduced in this thesis. The number of decisions directly depends on the following two variables because no decisions are to be made while following a plan.

**Plan length** The average length of executed plans indirectly indicates the benefit of using PGS. If plans are rather short, the number of decisions can hardly be reduced and PGS itself becomes obsolete. Therefore, this number is sought to be maximized here.

**Plan use** The ratio between actions entailed by the use of plans and low-level actions determines the amount of time in which PGS has the control over the agent. Since more plan reuse generally implies less decisions to be made, this variable is also to be maximized.

Because there are strong relationships between these variables, we will seek to discuss them concurrently during the experiment analysis. Other experiments will be conducted to support or explain these primary results.

Each experiment configuration is run for a certain number of trials, with each trial running until the agents reached their goal or the maximum number of steps was executed. A certain turn in the concatenation of trials is called a sequence, thus uniquely identifying a certain time point within the entire experiment. Considering that a trial runs for at most 150 steps and a certain number of trials $n$ is performed for every experiment, there are at most $150 \times n$ sequences in an experiment.

Statistics are calculated after every 100 trials and values averaged over these 100 trials. This enables studying the trend of performance, in which we are particularly interested. Each experiment configuration is executed 30 times with different, randomly determined initial situations for every trial. However, all configurations share the same sequences of initial situations. This allows to estimate confidence intervals and enables statistically valid comparisons of configurations. Significance tests are conducted at a confidence level of 0.95 with *Welch's t-test*, which does not require that the two samples to be compared exhibit equal variances. We assume that a sample size of 30 is sufficient to expect a normal distribution for the sampling distribution of the mean (Cohen, 1995). Detailed statistical results will be reported during the discussion for those results that are not obvious from the figures. Confidence intervals are drawn on most of the line graphs. For some points they are too small to be visible, though.

The results presented in this chapter are not likely to hold for every kind of domain. They might vary quantitatively or even qualitatively. We will thus point out restrictions and assumptions of the experiments conducted here that should be considered by the reader. Additional domains were not examined because of time-constraints.

## 5.3 Pursuit domain

The *pursuit* or *predator-prey* domain is one of the original testbeds of PGS. Parameters and domain rules were mostly retained from previous work (Karim, 2009). In the pursuit domain, four predators seek to chase down a prey on a non-toroidal grid ($13 \times 13$ tiles). Learning is only implemented for predators. The simulation is time-discrete: Each predator can chose to move up, down, left or right once every time step. The prey, in contrast, only moves every second step. The predators win if they manage to surround the prey on its north, east, south and west side at the same time. The prey wins if it reaches any of the boundaries or if the maximum number of steps (here 150) is exceeded. The predators start at different random positions on the board in each trial.

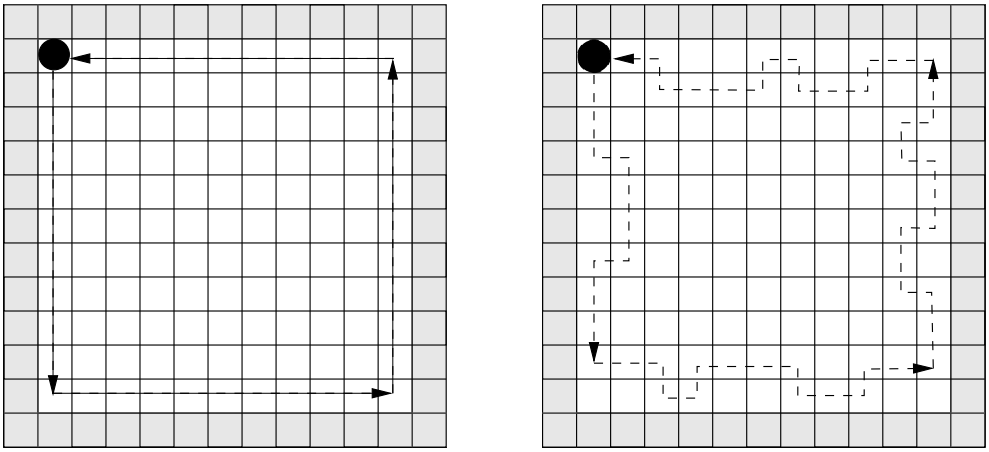
The following three different types of prey are deployed:

**Deterministic circle prey** This prey starts at the upper right corner and runs anti-clockwise along the largest square on the grid that does not touch any of the boundaries. Its path is depicted in Figure 5.1a. The prey freezes as long as any predator is blocking its way. It is obvious that this prey cannot win by its own effort because it never reaches a boundary.

**Nondeterministic circle prey** This prey follows the same rules as the deterministic circle prey but randomly leaves its trajectory. The prey has a 60% chance in each turn of moving off its track one step towards the grid's center and following the same movement pattern there. It has a 30% chance of moving back to its original path. The path of this prey is depicted in Figure 5.1b.

**Random prey** This prey starts each trial randomly on one of the four tiles at the upper left corner of the grid's center and from there on only takes random steps. It is the only prey of these three that has a chance of winning by reaching a boundary.

These prey types vary with regard to their predictability, which will allow us to make statements about the value of plans given different degrees of uncertainty in the environment. If not mentioned differently, the deterministic circle prey is used for all experiments. Considering that predators have to implicitly coordinate to surround the prey simultaneously, this task is already reasonably complex. However, the problem is simplified by an immediate reward being available for every step, as we will see later. In fact, this is a major simplification of the reinforcement learning task. Nevertheless,



(a) deterministic (b) nondeterministic

Figure 5.1: The unique path of the deterministic circle prey and a possible path of the nondeterministic one.

this problem, which is sometimes called *immediate reward learning*, has often been studied in literature (Schuurmans and Greenwald, 1999). Another reason for using the pursuit domain as a testbed for PGS is that subgoals can be defined quite clearly. In that, the domain can serve as a first presentation of a properly deployed PGS.

Next, we will describe the configuration of the predators. Subsequently, we will present and discuss the experiments. Those are divided into the following parts:

- Control experiments with basic configurations, which explore the inherent characteristics of PGS (Section 5.3.2).
- Prior knowledge experiments, which investigate the impact of integrating prior knowledge into the predators (Section 5.3.3).
- Further experiments, which complement and enrich the other experiments (Section 5.3.4).

### 5.3.1 Predator configuration

The predator setup requires the bottom-level reinforcement learner and the top-level PGS module to be configured, both of which are described in this section.

#### Configuration of the reinforcement learning modules

Each predator is driven by a reinforcement learning module provided by a system called FALCON, "a Fusion Architecture for Learning, COgnition, and Navigation" (Tan, 2004). For this work, the exact definition of FALCON is not essential. It is enough to note that it applies standard Q-learning with an additional scaling term $(1-Q(s,a))$ in order to provide a smooth normalization of the Q-values and to restrict them to the interval $[0,1] \in \mathbb{R}$ (Tan, 2007):

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1},a) - Q(s_t,a_t)](1-Q(s,a)).$$

The learning parameters are the same as in Tan (2004). The Q-learning parameters, however, are set as follows: $\varepsilon = 0.6, \alpha = 0.5, \gamma = 0.01$. $\varepsilon$ is decreased by 0.0006 after every trial. These parameters were taken from previous work and have proven to perform well. The decrease of $\varepsilon$ was changed to 0.0004, though. We refrained from further cumbersome search for optimal learning parameters apart from testing some minor variations (see Figure 5.2). The configuration described here turned out to be the best one as for long term performance and performance during exploration with regard to the average steps needed to catch the prey. We will show later that our claims also hold for a configuration that does not reduce $\varepsilon$ but that relies on the following fixed values frequently used in literature: $\varepsilon = 0.1, \alpha = 0.05, \gamma = 0.9$. This latter configuration shows the best performance during initial exploration (see Figure 5.2). However, it does not reach optimal performance in the long run because it does not cease to take exploratory actions. In fact, however, it is a more realistic setting because it does not rely on information being available about the time needed for the reinforcement learner to converge to an optimal policy. This knowledge would usually not be available to an agent designer.

FALCON itself does not employ any function approximation for state abstraction. However, the original state space is large. For a $13 \times 13$ tiles grid and 5 agents, there are $(13 \times 13)^5 \approx 1.38 \times 10^{11}$ possible states. It would take the predators a long time to fill a table-based Q-function and converge to the optimal policy. To render the problem solvable, the state information passed to every predator is restricted to only

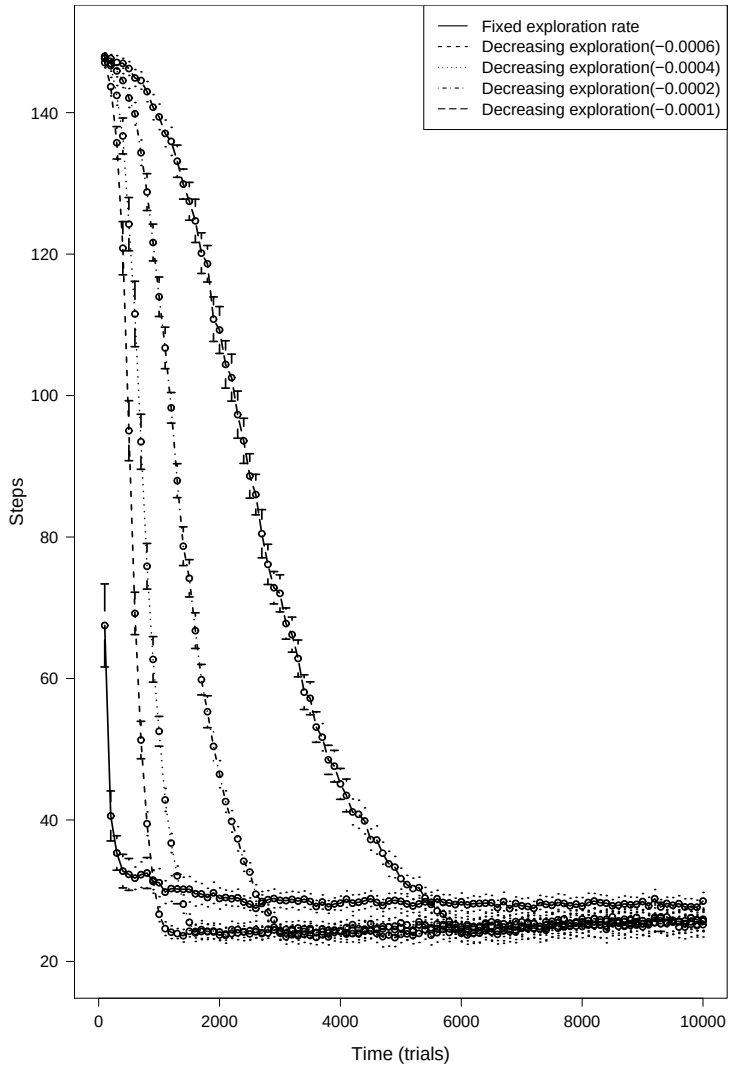



Figure 5.2: Performance of different Q-learning parameter settings for FALCON in the pursuit domain. For the fixed exploration rate, parameters are: $\varepsilon = 0.1, \alpha = 0.05, \gamma = 0.9$. For the decreasing exploration rate, initial parameters are: $\varepsilon = 0.6, \alpha = 0.5, \gamma = 0.01$, with $\varepsilon$ decreasing after every trial by the amount in brackets.

two parameters. One is an indicator for the *bearing* of the predator towards the prey. It can take eight different values corresponding to the main points of the compass. The other one is the *exposure* of the prey, which determines how well the prey is covered by the predators. It can take four values which gradually rate the exposure. It is defined as the number of sides (north, south, east, west) of the prey that are not occupied by any predator. Predators at intermediary directions such as north-west count both for the northern and western sides. If all predators, for example, are located on tiles directly above the prey, the exposure is 3. If all tiles directly adjacent to the prey are occupied by predators, the exposure is 0. This abstraction reduces the state space to $8 \times 4 = 32$ possible states.

Rewards are provided to a predator after every action and lie in the range $[0,1] \in \mathbb{R}$. They depend on the change of distance and exposure between the situation in which the action was performed and the situation in which the action resulted. Minimizing the distance between each predator and the prey is obviously a necessary condition for catching the prey. It is, however, not a sufficient one. In order to win a trial, the predators also need to cover the prey such that it cannot move anymore. This is reflected by the second factor in the reward stimulus – the change of the prey's exposure. The reward $r$ for a certain predator at time step $t$ given the distance $d_s$ between predator and prey and the exposure $e_s$ at time step $s$ is as follows:

$$r = \begin{cases} 1.0 & \text{if the prey is captured,} \\ 0.8 & \text{if } e_t < e_{t-1} \wedge d_t < d_{t-1}, \\ 0.6 & \text{if } d_t < d_{t-1}, \\ 0.4 & \text{if } e_t < e_{t-1}, \\ 0.0 & \text{otherwise.} \end{cases}$$

**Configuration of the PGS modules**

The PGS module of the hybrid architecture is configured as specified in Section 3.4. It is equipped with the following two subgoal-clue tuples, which naturally follow from the problem and the reward definition:

1. **Subgoal**: Minimize distance.
   **Clue:** An action is moving the predator towards this subgoal if it reduces the distance between the predator and the prey.

2. **Subgoal**: Minimize exposure.
   **Clue:** An action is moving the predator towards this subgoal if it increases the predators' coverage of the prey.

Note that these subgoals make use of the same domain knowledge as the reward calculation. Hence, they do not require any additional prior knowledge assuming that the reward definition is provided by the agent designer anyway. Actions are attached to a currently recorded plan as long as they move the agent towards the plan's subgoal and towards Q-values larger than that one of its first step. This conforms to the default implementation. The minimum length of plans to be activated is two steps. Hence, no knowledge about the domain that is not known to the reinforcement learner anyway has been exploited so far.

### 5.3.2 Control experiments

We first demonstrate that the adoption of PGS itself without any additional prior knowledge in this domain already leads to improved performance, indicated by the observed variables that were identified in Section 5.2. The following predator configurations are evaluated:

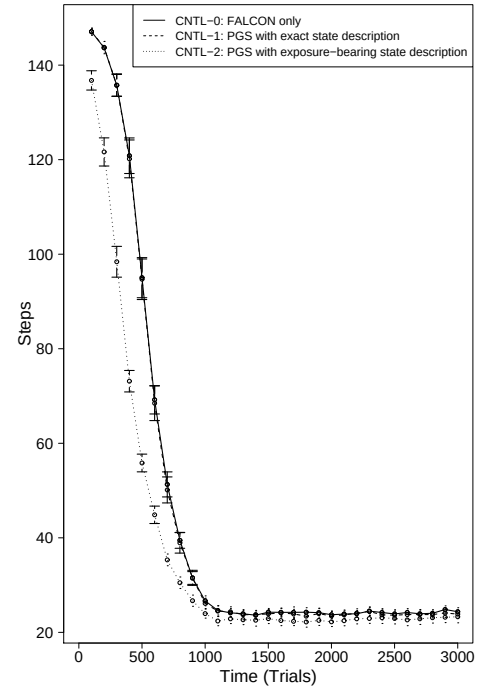
**`CNTL-0`** A plain FALCON setup as described above.

**`CNTL-1`** A setup with FALCON and PGS, where the latter applies a fully informed state description that consists of the exact positions of all agents on the grid. This is the configuration of original work on PGS.

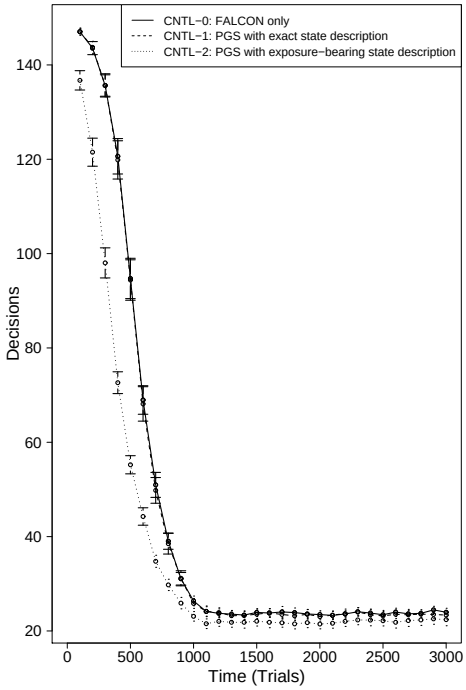
**`CNTL-2`** A setup with FALCON and PGS, where the latter applies the same generalized state description as the former, namely a tuple of exposure and bearing. Note that this PGS configuration does not require any additional knowledge that is not available to FALCON. Also the subgoal-clue tuples do not require any prior knowledge as discussed in the previous section.

From Figure 5.3, it is obvious that overall plan reuse for `CNTL-1` is low and that the number of steps and decisions never deviates significantly from `CNTL-0`. In fact, the difference in steps and decisions is too small to be visible on the graphs. In contrast, `CNTL-2` requires significantly less steps to be taken for the learning phase than `CNTL-0`.[1] Moreover, it requires significantly less decisions to be taken throughout

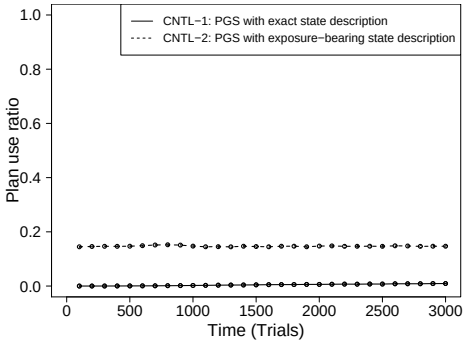
[1] The difference is statistically significant until trial 1,100 and for some scattered measuring points later with $p < 0.05$.



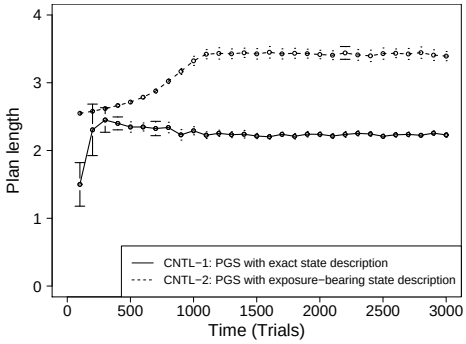

(a) Average number of steps




(b) Average number of decisions




(c) Average plan reuse




(d) Average plan length

Figure 5.3: The performance of basic predator configurations in the pursuit domain. Values are averaged over the most recent 100 trials.

almost the entire experiment.[2] However, the differences are rather minor. Around 15% of the actions performed by `CNTL-2` are due to plan execution, which is a significant difference to the plan reuse of `CNTL-1`. Likewise, the average length of executed plans for `CNTL-2` increases gradually until it levels off at about 3.5 steps.

Plan use for `CNTL-1` is low because its state space is very large and a particular situation unlikely to occur twice. Hence, its behavior does not deviate significantly from `CNTL-0` because mostly actions proposed by the reinforcement learner are performed. In case of `CNTL-2`, the state space is mapped to a set of only 32 states, such that significant plan reuse can indeed be observed. This configuration outperforms both former significantly with regard to the average number of steps performed, especially during the learning phase until convergence to a stable performance. We claim that this is due to more plans being executed and due to their execution guiding the agent procedurally. An action sequence that was successfully applied in a particular state is likely to be successful in a different but similar state. Furthermore, the average length of executed plans for `CNTL-2` is significantly longer than the minimum plan length of 2. This represents a major improvement over previous results, in which most plans only consisted of one step.

These observations show that PGS, after re-engineering, is in fact able to reuse plans sensibly, in that it does use a significant amount of plans without reducing its performance. The execution of plans leads to significantly less steps needed to catch the prey even though (a) PGS relies solely on the bottom-level module for knowledge acquisition and (b) it records plans while the bottom-level module has not converged yet. The number of decisions is related to the average number of steps and average plan reuse as reasoned previously. Therefore, this performance measure also benefits from adopting PGS on top of the plain reinforcement learner. The number of decisions for `CNTL-2` is significantly lower than for `CNTL-0` and hence computational costs are saved. However, the advantage gain over the number of steps is minor because plan reuse in general is still only about 15% for `CNTL-2`.

One could argue that the improvements in learning time indicated by the average steps needed to catch the prey could have possibly been achieved by tuning the FALCON module itself. We cannot disprove this claim because we refrained from optimizing the 13 real-valued parameters apart from what is depicted in Figure 5.2. Note that the PGS configurations introduced above, in fact, do not require any prior knowledge. It can be concluded that PGS by itself can be a valuable addition to a

[2] The difference is statistically significant for all measuring points apart from those at trials 2,500 and 2,700 with $p < 0.05$.

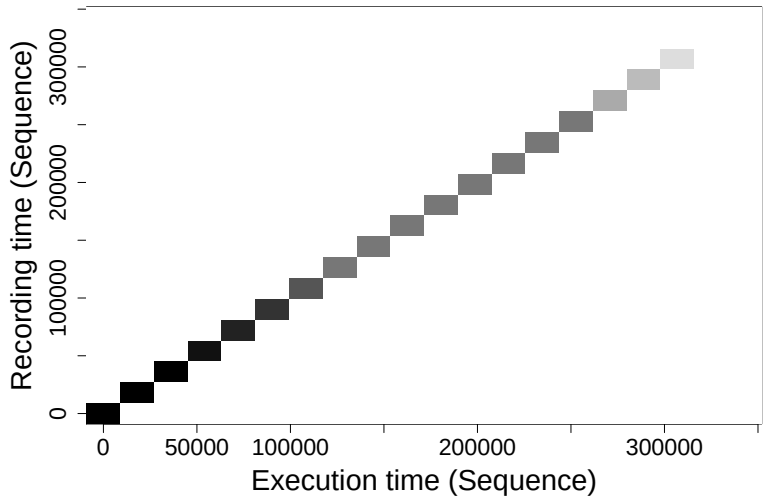



Figure 5.4: The ration of plans recorded in a specific time interval and reused in another specific time interval for CNTL-2.

reinforcement learning algorithm as a substantial improvement for learning time. This should especially be valuable if a function approximator is used to represent the value function. Function approximators usually come with a larger number of additional parameters to be considered for optimization. It is surely worthwhile to save time on this possibly cumbersome effort. Nevertheless, we note that observations could result from the specific task definition used here alone.

Still, there is surely space for further improvements. One of these arises from a closer observation of the relationship between recording time and execution time of plans. Figure 5.4 depicts the ratio of plans recorded in a specific time interval of the experiments *and* reused in another specific time interval. On both axes, the graph shows the sequences[3] of CNTL-2. Both scales are split into 20 equally large intervals. Each of the shaded boxes belongs to a certain combination of two of these intervals. The color of a box indicates the ratio of plans that were (a) recorded in the time interval denoted by the position of the box on the y-axis *and* (b) reused in the time interval

[3] As explained in Section 5.2, a sequence is a particular turn in the concatenation of all trials that were run in an experiment.

denoted by the position of the box on the x-axis. Darker values correspond to a larger number of plans for that interval combination. For example, the sixth box from the left indicates the ratio of plans that were recorded roughly between sequence 80,000 and 100,000 and were reused sometime later in the same interval. The data was generated for the conjunction of all 30 experiment runs.

In Figure 5.4, there are only shaded boxes on the main diagonal. That means that plans were never reused in another time interval than in that in which they were originally recorded. In that, only recently recorded plans are reused throughout the entire life cycle of the predators. This means that plans in general do not have a long life time, not even after the reinforcement learner has converged to an optimal policy. Then at the latest, PGS should, in fact, start recording plans that are effective in the long run. We assume that the plan confidence update as well as the plan selection mechanism require improvement in order to allow useful plans to reside in the plan libraries for a longer time. We reason that the state description used in `CNTL-2` is too abstract. Obviously, it does not capture all relevant state information. For example, the positions of other predators are not accounted for at all. This renders a plan reusable in a number of situations, in which its application is actually not worthwhile. This will lead to the plan being executed in an inappropriate situation soon, thus resulting in a negative confidence update early. In general, more sensible plan selection can be expected to lead to a more efficient plan reuse. Such improvements might also lead to increased plan length and plan reuse growing over time, which cannot be observed yet. They will be – amongst others – the topic in the following subsection.

### 5.3.3 Integrating prior knowledge

In Section 3.5, we identified four distinct parts of the PGS algorithm that we consider amenable to the integration of prior knowledge. We decided to investigate two of these in detail: One of them concerns the selective reuse of plans while the other one concerns the prior content of the agent's plan libraries. Both are to be evaluated in this section.

#### Guiding the plan selection

So far, PGS was restricted to the same state description that FALCON applies in order to show that it is a valuable addon in this domain without exploiting any inherent advantage. The usefulness of plans, however, is more closely tied to their specific context condition than is the usefulness of a single action as already noted by Jong

et al. (2008). Hence, it is of interest to apply a more fine-grained state description and thus to achieve a more sensible plan reuse. A fully descriptive state, however, cannot reasonably be adopted because in more realistic state spaces it is very unlikely that a certain situation exactly occurs again. As recognized before, this already holds in the pursuit domain as well. Therefore, methods of state abstraction should be applied in order to enable a proper reuse of plans. It is not the intention of this work to discuss this field but rather to show that PGS exposes a convenient hook for integrating knowledge about state similarity into the plan selection process.

We implemented three simple state abstraction mechanisms that guide the decision-making in the plan selection process on whether a certain plan is applicable in the current situation. The first approach, later referred to as `STATE-0`, is based on a fully informed state description consisting of the exact positions of all agents. We assume that different features in the state description are of different importance for decision-making. For example, in order to reuse a plan in a certain situation, it is less important that the positions of other predators match the original precondition than that the predator's own and the prey's position match with the original precondition. To implement this idea, we slightly altered the definition of confidence as introduced in Chapter 3. A plan's confidence is now also defined for states similar to the original precondition, with similarity holding if:

- the Manhattan distance between the predator's actual position and its position in the plan's precondition is smaller than 3, and
- the Manhattan distance between the prey's actual position and its position in the plan's precondition is smaller than 3, and
- the sum of Manhattan distances between all other predators' actual positions and their positions in the plan's precondition is smaller than 6.

The second state abstraction approach, which we will refer to as `STATE-1`, requires a more detailed introduction. It also relies on a redefinition of the confidence of a plan but is only based on the abstracted state description used in `CNTL-2`. From a domain-expert's point of view, states are similar if they exhibit similar values for exposure and bearing. To make use of that knowledge, we define the confidence value of a plan as a two-dimensional, asymmetric Gaussian in the state space with its center at the plan's context condition. Note that this is reasonable here because similar states are indeed defined by similar numerical values in the state description. This approach allows the agent to have the most confidence in reusing a plan if it finds itself in

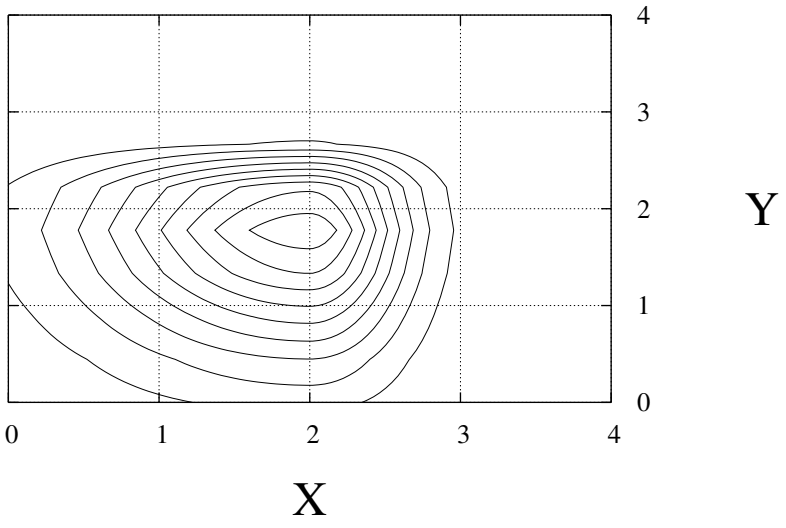



Figure 5.5: Illustration of the Gaussian state abstraction. This example shows a two-dimensional state space and the Gaussian confidence of a plan visualized by a contour plot. For different directions from the center of the plan's precondition, different confidence values apply. For example, the plan is more likely to be reused in states with smaller x- and y-values than with larger ones.

the same situation as during plan recording. It is somewhat inspired by radial-basis-function networks, which typically map an input vector to a number of Gaussians, thus defining its membership to the clusters or classes described by these Gaussians (Moody and Darken, 1989). The idea is illustrated in Figure 5.5. It leads to the following definition for a plan's confidence:

**Definition** Let $x$ be the state vector for the current situation, $\mu$ the state vector for the plan's precondition, and $\Sigma^{-1}$ a placeholder for two different covariance matrices

(entries unequal to zero only on the main diagonal), one of which is applied if $x_i < \mu_i$ and the other elsewise, then

$$\text{confidence} = \text{amplitude} \times exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}$$

determines the confidence in executing the plan in situation $x$.

It is obvious that the update rule for reinforcing and penalizing the confidence of a plan has to be adjusted. The calculation proposed relies on the assumption that a successful or unsuccessful execution of a plan should have a two-fold implication: First, it should reinforce or penalize the overall confidence in the plan. Second, it should reinforce or penalize the confidence in applying the plan in situations similar to this execution. An execution further away from the original precondition should have a larger impact on the *Gaussian's variances* of dimensions in which it differs from the plan's precondition. An execution of a plan in a situation rather similar to its precondition, in contrast, should have a larger effect on the overall confidence in the plan, represented by the *amplitude* of the Gaussian. The update rule for incrementing the confidence is presented in Algorithm 3. The decrement rule is similar. A confidence update basically generalizes or specializes the application context of a plan. This is similar to the rule refinement at the top-level module in CLARION (Sun et al., 2005). If a plan execution achieves a reward that is larger than that one gained when it was recorded, the precondition of the plan is changed to the more successful situation.

Note that there is no learning rate. This turned out to yield better results in this domain. Yet, it should generally lead to an adverse, oscillating behavior because a predator would repeatedly try a plan in a situation more different from its original precondition even though that one was already found unsuitable. The Gaussian amplitude of every plan is initialized with 1 and the variances with 10. The former reflects the assumption that initially the confidence in the plan's value is absolute. The latter was chosen because it yielded the best results.

`STATE-1` relies on the imprecise exposure-bearing state description. We expect that a more informed description can lead to a further improvement of plan reuse. Hence, we define another experiment, `STATE-2`, in which the Gaussian approach as described above is applied to the exact state description consisting of the positions of all agents, which is also used by `CNTL-1`. Obviously, similar numerical values for a certain dimension correspond to similar positions and hence similar states, which renders the Gaussian state abstraction suitable to this state description. The Gaussian

**Algorithm 3** The reinforcement method for plans with Gaussian confidence

**Require:** The real-valued vector *newSituation* describing the situation for which to test the confidence in applying this plan and the real-valued vector *precondition* of same length describing the situation in which the plan was recorded.

1: $amp = \frac{\text{confidence}(newSituation)}{\text{confidence}(precondition)}$
2: $var = 1.0 - amp$
3: $amplitude = \min(1, amplitude + amp)$
4: **for** $i$ **in** $[0..(\textbf{length}(newSituation) - 1)]$ **do**
5: **if** $precondition_i > newSituation_i$ **then**
6: $\sum_{l,i,i}^{-1} := \sum_{l,i,i}^{-1} \times e^{-var}$
7: **else**
8: $\sum_{r,i,i}^{-1} := \sum_{r,i,i}^{-1} \times e^{-var}$
9: **end if**
10: **end for**

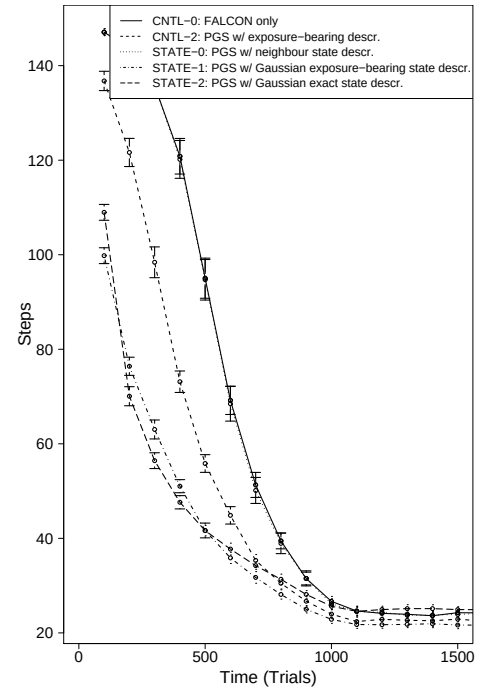
amplitude of every plan is initialized with 1 again but variances with 0.5. The latter was found to be reasonable in exploratory experiments.

The performance of STATE-0, STATE-1 and STATE-2 in comparison to CNTL-0 and CNTL-2 is shown in Figure 5.6. It is obvious that the number of steps and decisions for STATE-0 does not deviate significantly from CNTL-0. In fact, the difference is too minor to be visible clearly. For STATE-1 and STATE-2, however, a significant improvement in the average number of steps can be observed for the learning phase.[4] The improvement in the number of decisions is eminent throughout the entire experiment.[5] STATE-0 shows hardly any plan reuse at all while STATE-1 and especially STATE-2 show a significant plan reuse throughout all times of the experiments. For both configurations, plan reuse starts with optimistic values that slightly decrease. It levels off at around 30% for STATE-1 and around 38% for STATE-2. Average plan length converges to about 2.3 for both configurations that apply the Gaussian abstraction. It converges to about 3.5 for those configurations that apply the exposure-bearing state description.

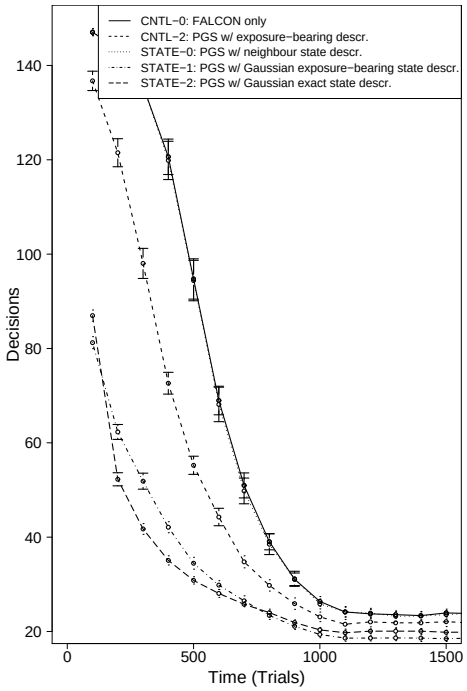
We conclude that the general performance of STATE-0 is not significantly different to that of CNTL-0 because hardly any plan reuse is observed. Hence, there is no improvement over CNTL-1 with regard to plan execution. In contrast, dynamically

[4] For STATE-1, the improvement is statistically significant for the first 900 trials with $p < 0.05$. For STATE-2, it is significant for the first 700 trials.

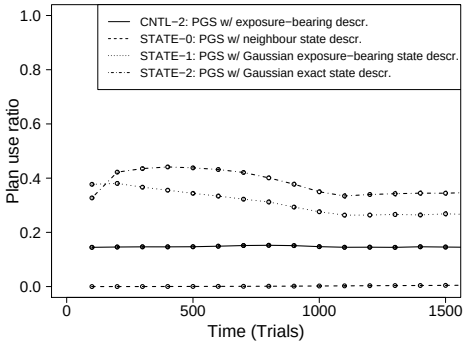
[5] The improvement is statistically significant with $p < 0.05$ for all measuring points.



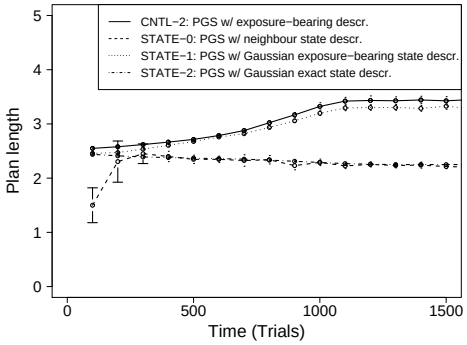

(a) Average number of steps




(b) Average number of decisions




(c) Average plan reuse




(d) Average plan length

Figure 5.6: The performance of basic predator configurations and configurations that apply state abstraction in the pursuit domain. Values are averaged over the most recent 100 trials.

adapting the conditions under which a plan is applicable seems promising. The results for STATE-1 and STATE-2 show a large improvement especially in plan reuse, which, in turn, leads to a shortened exploration phase. This supports the claim that exploration can be guided by a proper reuse of plans. However, plans in STATE-1 still do not have a long life time as can be seen in Figure 5.7a. As reasoned previously, we assume this to be a general problem of the abstraction degree of the exposure and bearing state description.

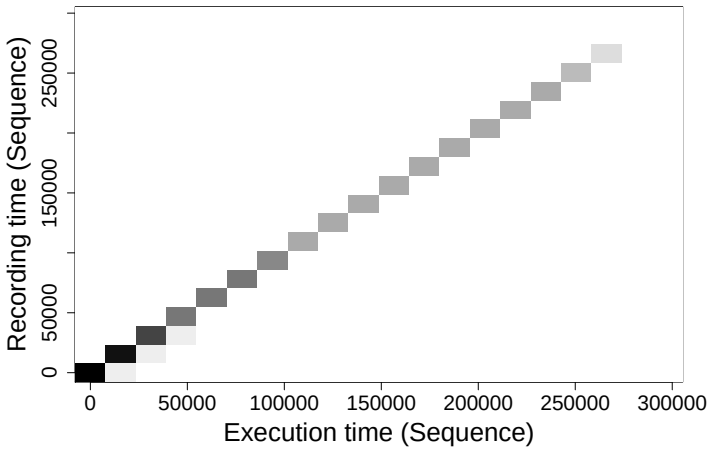
However, in Figure 5.7b, there are not only shaded boxes on the main diagonal for STATE-2. For most of the boxes on the main diagonal, there are also three or four boxes shown for later intervals on the x-axis but for the same interval on the y-axis. That means that plans in general are not only reused in the very same interval they were recorded, but also in three or four later intervals. Still, plans show the largest reuse more recently after their recording, which can be inferred from the fading colors of boxes at later execution intervals. Obviously, the likelihood of a plan to be reused decreases with its age. Nevertheless, plans for STATE-2 have a generally longer life time than plans for CNTL-2 and STATE-1. We argue that this is caused by their usage of a more informed state description. The state description leads to plans being less often reused in situations majorly different from the plan's original and successful recording than is the case for less informed and more abstract state descriptions.

**Priming the plan library**

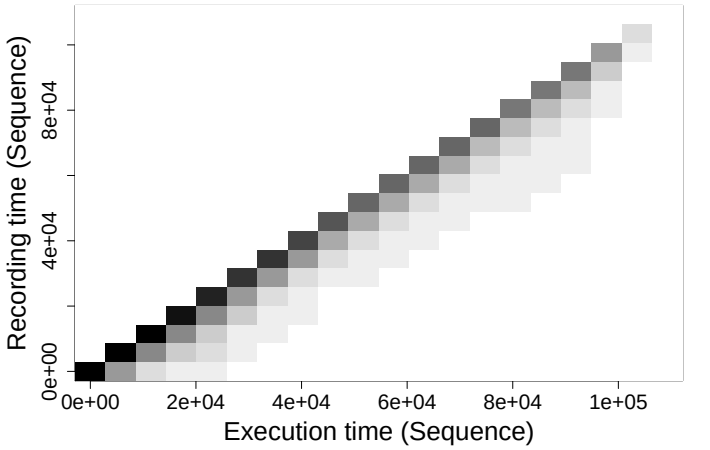
The intention of the second prior knowledge extension is to decrease the number of average steps required instead of increasing plan reuse. We primed the predators' plan libraries with the following two different prior plans in separate experiments. Both plans make use of the fully informed state description and exploit insight knowledge of the path of the prey. They make use of the weakness of the prey that it only moves if it can follow its deterministic path. If a predator blocks the prey's way, the prey simply does not move in that turn.

**PRIOR-0** This plan has a very high confidence if the predator is the only one on the prey's path. In that case, it basically overrides the low-level behavior and does not move anymore. The predator will block the prey, once it is eventually reached by it. That makes surrounding the prey easier for the predators.

**PRIOR-1** This plan has the same activation condition as PRIOR-0 but it calculates a plan that moves the predator towards the prey along its trajectory. The predator




(a) STATE-1




(b) STATE-2

Figure 5.7: The number of plans recorded in a specific time interval and reused in another specific time interval for STATE-1 and STATE-2.

freezes once it stumbles upon the prey. It will basically lead to a blocking position earlier.

Neither of these plans is reinforced nor penalized because we wanted to encourage their reuse solely for the sake of a faster learning time. It would be generally possible, though, to "unlearn" such prior plans once they turn out to lead to worse results than the reinforcement learner would achieve. This is the actual purpose of the update of the plan's confidence. Note that explicit communication for both plans is not necessary. The activation is only triggered on the basis of the other predators' positions. For both experiments, the state abstraction for all other learned plans is `STATE-1` because it exhibits the best performance with regard to the average steps. The indicators for plan reuse, plan length and number of decisions are less interesting here because plan reuse is artificially boosted. Hence, we compare the average number of steps between the `STATE-1` configuration and both prior knowledge configurations (see Figure 5.8).

It can be seen that predators applying these prior knowledge plans require less steps than previous configurations during the crucial learning phase. However, as soon as the other configurations reach a better policy, this advantage turns into a disadvantage. This underlines the need for a possibility to "unlearn" prior knowledge. This is in line with the results of McGovern et al. (1997) who observed that the influence of prior knowledge macro-actions or plans can be adverse or supportive depending on their quality. The initial speedup observed here might seem obvious because we "told" the predators how to solve the problem but it has to be emphasized that only very little knowledge was injected to the system. In fact, the task requires four predators to implicitly coordinate their actions to reach their goal. We only provide them with a hint about the prey's behavior, which had a large impact on their initial performance. Moreover, to specify this plan requires much less effort than tweaking the reinforcement learner's parameters and Q-value initializations to achieve the same behavior. It should become clear from this example that adding knowledge to a reinforcement learner in a higher-level language might be valuable and is worth further investigations.

### 5.3.4 Further considerations

We conducted further experiments, three of which we consider to be of interest here. The first one addresses the question of what are the advantages the reinforcement learner gains from plan execution. The second one specifically investigates the influence of varying degrees of uncertainty in the environment on the benefit of deploying

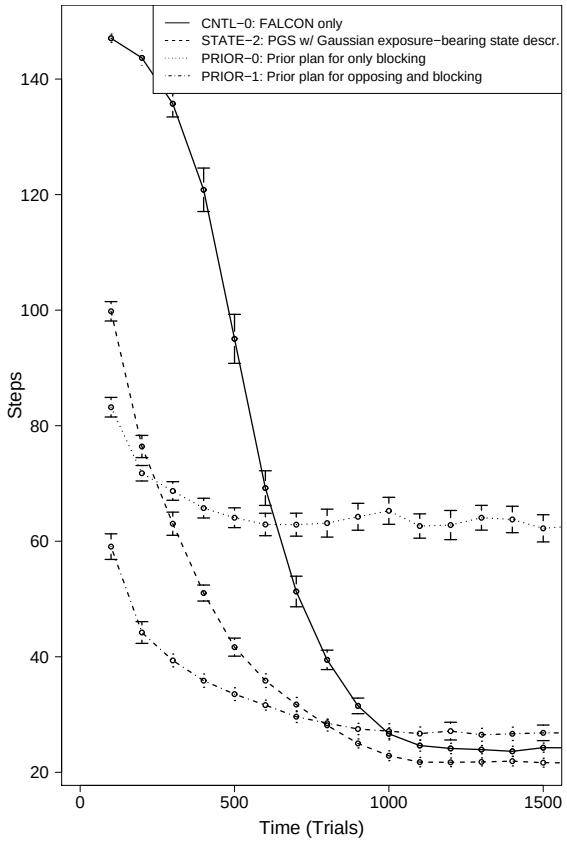



Figure 5.8: The average number of steps of previous predator configurations and configurations that are supplied with prior knowledge plans in the pursuit domain. Values are averaged over the most recent 100 trials.

PGS. The third one applies PGS to a FALCON module that uses alternative Q-learning parameters to support our results.

**The impact of plan execution on the reinforcement learner**

We saw previously that applying PGS has a positive influence on the performance of the predators. We also noted in Section 3.4 that FALCON is in fact learning during the execution of plans. Hence, we hypothesize that its value-function benefits from a pre-

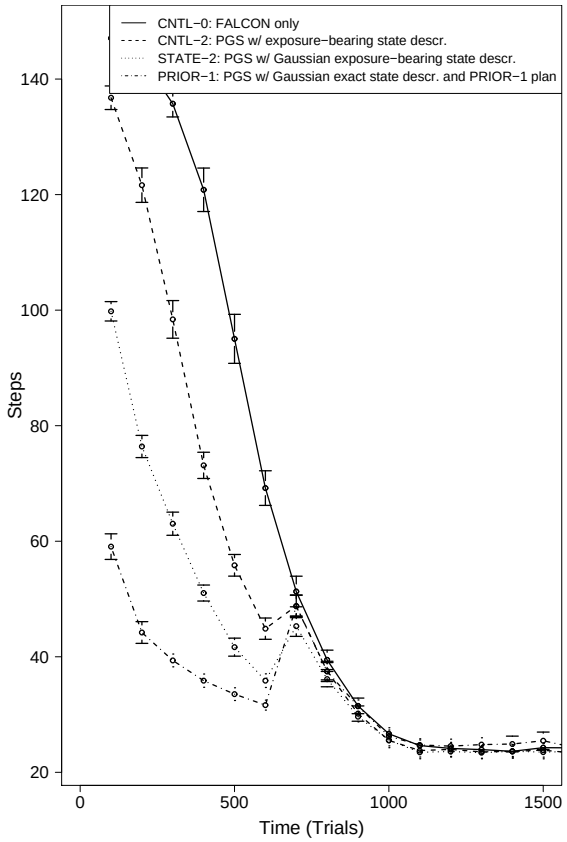


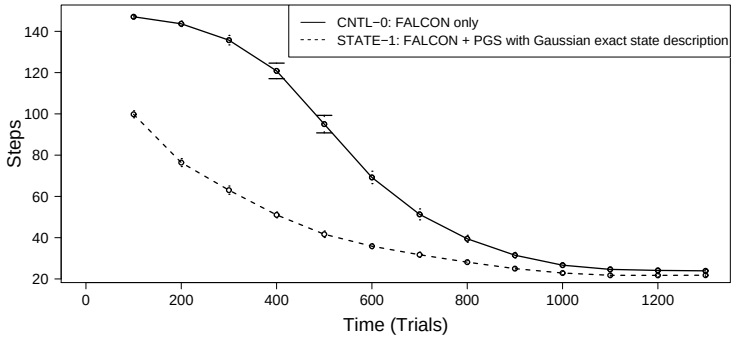

Figure 5.9: The average number of steps of various predator configurations with PGS switched off after 600 trials in the pursuit domain. Values are averaged over the most recent 100 trials.

vious application of plans. To test this hypothesis, we deployed `CNTL-2`, `STATE-1` and `PRIOR-1`, and switched the PGS module off after 600 trials such that from then on only the FALCON module was working. According to our assumptions, the performance of these predators should still outperform `CNTL-0` – the plain FALCON configuration. The values for average steps are shown in Figure 5.9. The average steps needed for the PGS configurations is significantly smaller than for `CNTL-0` but only until the PGS modules are switched off.

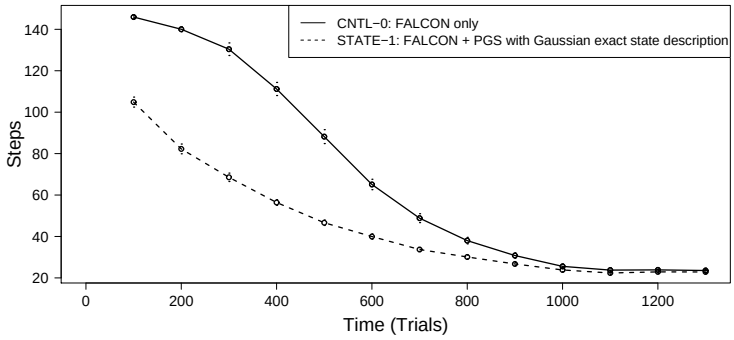
The simulation results do not support our hypothesis – none of the PGS configurations exhibits a better performance than CNTL-0 after the application of plans. Surely, a problem in the implementation cannot be ruled out. However, Dixon et al. (2000) provide an explanation for this observation. They note that a reinforcement learner needs to be able to represent the knowledge that is transferred from another controller, which is in this case specified by plans. Two factors need to be considered: *state-space deficiency* and *representational deficiency*. The first one denotes the failure of a reinforcement learner to benefit from prior knowledge that applies a more powerful state description. In fact, this holds for STATE-1 and PRIOR-1 because the state description originally introduced for STATE-1 is more powerful than that of CNTL-0. A reinforcement learner suffers from a representational deficiency if its decision-making is not powerful enough to represent that one used by the injected prior knowledge. This especially holds for PRIOR-1, whose execution follows comparably complex rules. In fact, actions in any plan are selected based on the original decision to execute the plan. This means that decision-making waives the Markov property, which the reinforcement learner, in contrast, assumes to hold. Hence, we can assume that the reinforcement learner cannot learn from any plans in the above mentioned configurations. All speedup observed previously is thus only a result of interrupting the execution of the reinforcement learner's policy. The advantage of temporal abstractions reported by McGovern et al. (1997) cannot be achieved here because the agent's reinforcement learner does not backup Q-values from the plans' last states to their first ones. Only intermediary one-step standard Q-learning propagations are performed. Nevertheless, the general speedup observed here is real even though it does obviously not affect the reinforcement learner.

**The impact of varying uncertainty on the benefit of plan reuse**

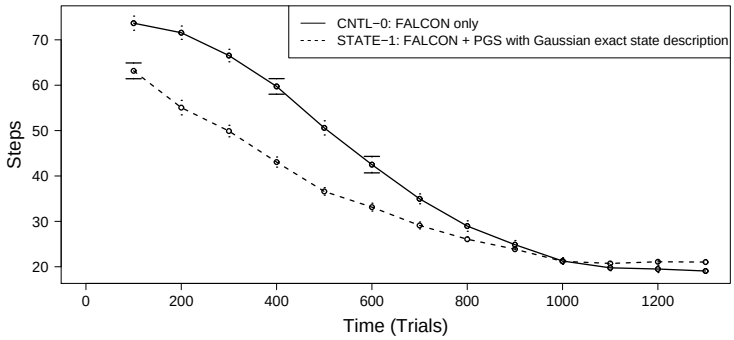
In the experiments presented previously, the advantage of the PGS configurations to the FALCON-only one varies with the availability and quality of prior knowledge as seen when priming the plan library as well as with plan reuse in general. In the case of prior knowledge plans being used, the advantage later turns into a disadvantage. We reasoned previously that the success of plan execution also depends on the predictability of the environment. In contrast to plans, open-loop policies can regularly sense the environment and decide on a new course of actions, which gives them an advantage when coping with uncertainty in the environment. Constant sensing and reasoning, however, might not be possible in resource-bounded agents that are required to behave in a reactive manner. A closed-loop policy such as a plan, in contrast, commits to




(a) Deterministic circle prey




(b) Nondeterministic circle prey




(c) Random prey

Figure 5.10: The average number of steps of `CNTL-0` and `STATE-1` for different degrees of predictability of the prey in the pursuit domain. Values are averaged over the most recent 100 trials.

multiple actions at once and follows this decision blindly. It saves the agent resources for other purposes than reasoning.

This lets us assume that if the prey exhibits a less predictable behavior than considered previously, we expect plans to be less effective. In the end, it becomes less likely that a previously successful action sequence is valuable again. To test this hypothesis, we deployed the three prey types that are described in Section 5.3 and that vary with regard to their predictability. For each of them, we compared the FALCON-only configuration (`CNTL-0`) with the PGS one that applies a Gaussian state abstraction on the exposure-bearing state description (`STATE-1`). We hypothesize that the benefit of adopting PGS additionally to a reinforcement learner is proportional to the predictability of the prey with regard to the average steps needed to catch it.

The results are shown in Figure 5.10. Indeed, the advantage of the hybrid approach over the FALCON-only model varies with the degree of predictability of the prey. The more predictable the prey behavior is designed, the larger is the overall advantage of the hybrid approach. It is noteworthy, though, that the performance advantage gained by adopting PGS is almost as large for the nondeterministic circle prey as it is for the deterministic one. This lets us assume that plans can even provide guidance and partial solutions if the behavior of the environment is not completely predictable.

Hence, we can identify three factors that affect the advantage of the hybrid approach over the pure reinforcement learner: The first one concerns the amount and quality of prior knowledge added to the system, the second one relates to the degree of uncertainty in the environment, and the third one to plan reuse in general. A fourth factor of more theoretical nature shall be mentioned briefly. In every step, the FALCON-only predators have to sense the environment and come to a decision on which action to take next. There are, however, domains that require a highly reactive behavior. In that case, constant sensing and reasoning would not be possible. Making a decision might then have a cost that exceeds that one for taking an action. If such a cost was reflected in the model, the advantage of the hybrid approach would be increased. In contrast, if taking an action was considered to be potentially expensive compared to making a decision, the advantage would be decreased.

### Alternative FALCON parameters

Finally, we show that qualitatively similar results for the experiments above can be obtained with more realistic Q-learning parameters that do not depend on knowledge about the convergence time of the reinforcement learner. These are $\varepsilon = 0.1, \alpha = 0.05, \gamma = 0.9$, with $\varepsilon$ being fixed. A comparison of `CNTL-0`, `CNTL-2`, and `STATE-1`

for these parameters is given in figure 5.11. Obviously, the learning speedup as for the average number of steps needed is less substantial than for previous results.[6] However, with plan reuse and plan length being considerable, the improvement in the number of decisions is still significant.[7] `STATE-1` does not show a larger plan length than `CNTL-2` and even has a smaller plan reuse. This is contradicting previous observations. As of the time of writing, we do not have any explanation for this observation.

## 5.4 Taxi domain

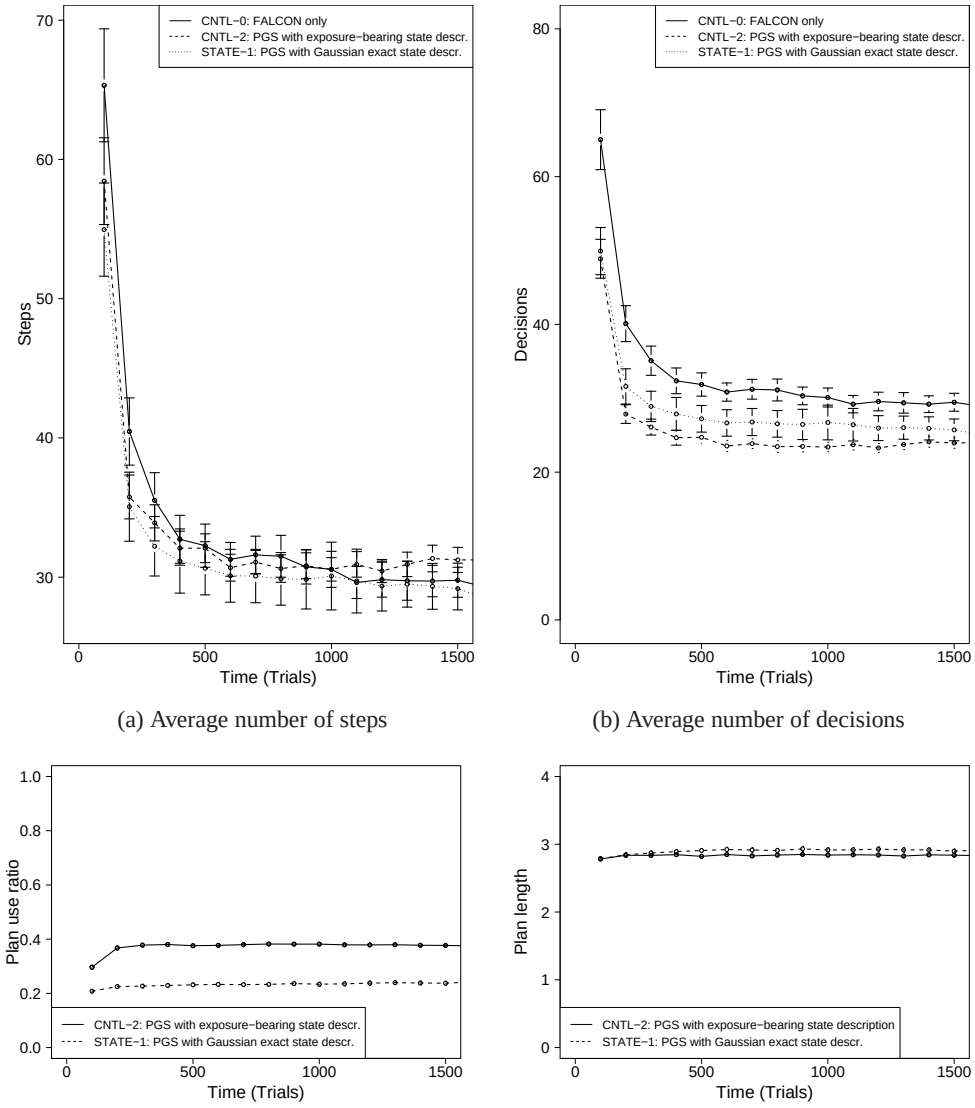
The taxi domain, which is a $5 \times 5$ tiles gridworld, is illustrated in Figure 5.12. A taxi agent starts at a random position on the grid in each trial. A passenger appears at one of four distinct locations and seeks to go to one of these four locations. Her initial location and destination are determined randomly for every trial. The taxi agent has to learn to pickup the passenger and deliver her to the destination within a certain number of steps (here 80). In each turn, the taxi can take a step north, south, east or west or pickup, or unload the passenger. As can be seen in the figure, there are barriers on the grid which cannot be passed. This makes the task more difficult on the one hand, but provides for distinct subgoals on the other. All traffic between locations on both halves of the grid has to pass the central tiles, which therefore obviously become bottleneck states. Moreover, no trial can be successful without solving the following two subtasks: navigating to the passenger and picking her up, then navigating to her destination and unloading her.

Research in option discovery has studied this domain to evaluate algorithms that identify such subgoals autonomously (see Section 2.1.3). The task was originally defined by Dietterich (1998). It has since been used in a number of hierarchical reinforcement learning studies (e.g. by Parr and Russell (1998), Andre and Russell (2002) or Özgür Şimşek et al. (2005)).

The notion of subgoals is also related to PGS. However, we have noticed in Section 3.3 that this follows a different idea, which makes its application in the taxi domain particularly difficult. Plan recording relies on continuous feedback about the value of actions taken. In the pursuit domain, for example, an action was attached to a plan if it reduced the exposure of the prey or the distance between the predator and the prey.

[6] For `CNTL-2` and `STATE-1`, the improvement is statistically significant for the first 200 trials with $p < 0.05$. For `STATE-1`, it is also significant for the trials 600 to 1,200.

[7] Both for `CNTL-2` and `STATE-1`, the improvement is statistically significant for all measuring points with $p < 0.05$.




(a) Average number of steps

(b) Average number of decisions

(c) Average plan reuse

(d) Average plan length

Figure 5.11: The performance of `CNTL-0`, `CNTL-2`, and `STATE-1` for alternative Q-learning parameters in the pursuit domain. Values are averaged over the most recent 100 trials.

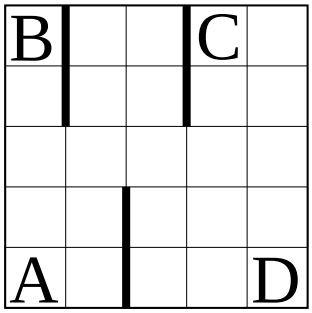



Figure 5.12: The taxi domain with the 4 possible passenger locations and destinations.

Such information is not readily available for all state transitions in the taxi domain. Because of the barriers, the shortest way, for example, is neither necessarily the best one nor possible at all. To reach the overall goal, the agent has to take actions that seem to be adverse at first. This would especially be a problem if the grid had an even more complex, maze-like structure. In these cases, the reinforcement learner also has to rely on backing up Q-values from the goal alone because intermediary rewards cannot be defined reasonably. The agent needs to reach the goal first and receive a reward to start exploring the state space selectively. Because we cannot clearly identify subgoals in the sense of PGS, we have to slightly bend their definition here. We specify a single subgoal that records a plan as long as actions move the agent upwards the Q-value gradient:

1. **Subgoal**: Maximize Q-value.
   **Clue:** An action is moving the predator towards this subgoal if the Q-value of the new state is greater or equal than that of the previous one.

This means basically that the agent extracts trajectories from the Q-table. It also entails that plans do not have any meaningful subgoal anymore, which waives the advantage of easier knowledge inspection by experts. The definition of the subgoal-clue tuple makes use of the fact that transitioning to a state with a higher Q-value conforms to transitioning to a state from which on larger rewards are expected. In this scenario, this means getting closer to the goal. This subgoal-clue definition simply constrains the general assumption that only actions leading to states with higher Q-values than the *first* action are appended even further. Effectively, it forces plans to walk along a Q-value gradient, which, however, does not need to be the optimal one.

This definition renders the process in these experiments even more similar to learning plans in CLARION, which we discussed in Section 2.2.

Apart from not being able to provide immediate feedback to actions, the problem is simple compared to the pursuit domain. The environment is static. The taxi agent is the only active entity and the outcomes of its actions are deterministic. With 5 possible passenger locations including the passenger being on board, 4 possible destinations and $5 \times 5 = 25$ possible taxi positions, the state space consists of only $5 \times 4 \times 25 = 500$ states. Because of that, neither the reinforcement learner nor PGS have to apply a state abstraction to cope with this task. PGS is configured as defined in Section 3.4. The reinforcement learner is provided by standard Q-learning with a table-based Q-function.[8] Its learning parameters are again: $\varepsilon = 0.1, \alpha = 0.05, \gamma = 0.9$. If the agent reaches the goal, it gets a reward of 1.0. If it performs an unsuccessful pickup or unload, it receives a reward of $-0.5$. In all other cases, the reward is $-0.05$. This follows the reward scheme of Andre and Russell (2002).

### 5.4.1 Experiments

Because of the absence of uncertainty, we hypothesize that plan reuse should be large. The same should apply to the length of plans. An action sequence that was successful once, is very likely to be successful again. If it is not the optimal path, then eventually a competing action or plan will gain advantage over the plan because of the exploration of PGS described in Section 3.4. Because of that, the asymptotic performance with regard to the number of steps should be the same for a FALCON-only and a PGS configuration. It should also let plan reuse and length grow continuously because smaller plans should be substituted by longer ones frequently. However, since no state abstraction is employed, PGS will not contribute to the agent's exploration such that we do not expect any speedup during learning time.

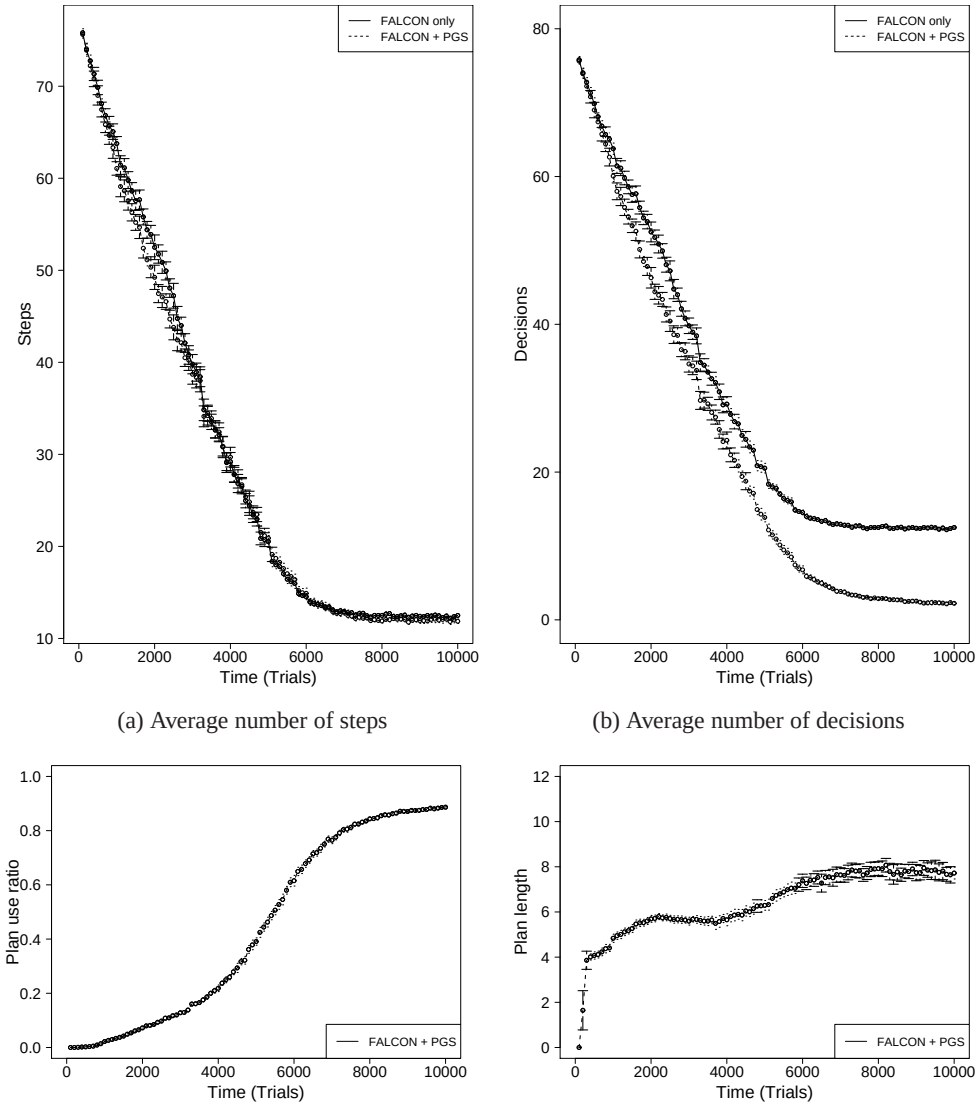
To test these hypotheses, we deploy the following agent configurations:

**CNTL-0** A plain Q-learner setup as described above.

**CNTL-1** A setup with the Q-learner and PGS with both applying a fully informed state description.

The results are depicted in Figure 5.13. Indeed, plan reuse grows continuously to around 90% and average plan length to around 8 steps. There is no significant difference observable for the average number of steps needed to solve the problem.

[8] The original implementation is by Stephan Mehlhase and was restructured and adapted for this work.




(a) Average number of steps

(b) Average number of decisions

(c) Average plan reuse

(d) Average plan length

Figure 5.13: The performance of `CNTL-0` and `CNTL-1` in the taxi domain. Values are averaged over the most recent 100 trials.

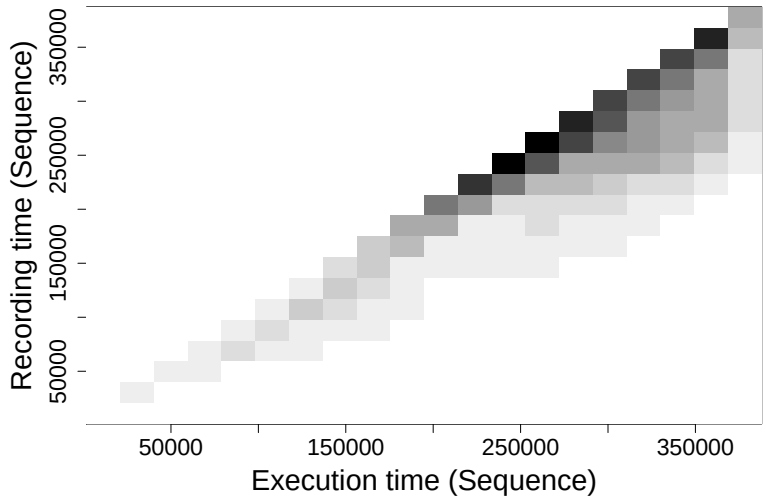



Figure 5.14: The number of plans recorded in a specific time interval and reused in another specific time interval for `CNTL-1`.

However, the average number of decisions can in the long run be reduced from around 15 per trial to around 4 by deploying PGS. In general, plan reuse is much more effective than in the pursuit domain, which is a result of the absence of uncertainty and the largely reduced size of the state space. This argument is also supported by the observation that the general life time of plans increases with time (see Figure 5.14). This indicates that plans in a way converge to a competitive behavior, which we were not able to achieve in the pursuit domain. Of course, these results are to be expected. However, they confirm that the model is indeed working and is reusing plans correctly. In addition, the initial discussion on the definition of subgoals demonstrates the restrictions of PGS on a practical example.

## 5.5 Summary

The experiments show that the benefit of temporal abstraction by plans mainly depends on the predictability of the environment and the quality of the state abstraction applied. The former is backed by common sense and previous studies. The latter sup-

ports the need for combining temporal abstraction and state abstraction. We expand on this discussion during the conclusion in the next chapter.

# 6 Conclusions and future work

In this thesis, we brought together a number of concepts: We discussed temporal abstraction in reinforcement learning and in hybrid agent architectures. We also discussed the integration of prior knowledge in reinforcement learning. Based on this discussion, we presented a hybrid agent architecture, PGS, which extracts temporal knowledge in the form of plans for later reuse from a reinforcement learner. The PGS architecture was reviewed carefully and its peculiarities were identified. This led to a number of improvements to be proposed. In particular, we sought to increase both plan reuse as well as the length of extracted plans without implying a major loss in performance. In a second step, we evaluated empirically the impact of plans as temporal abstractions and as prior knowledge on the performance of PGS agents. In particular, we posed a number of research questions. We will revisit them in the next section and summarize the answers that arose during the course of this thesis. Thereafter, we will touch upon opportunities for future work in Section 6.2.

## 6.1 Research questions revisited

In the following, we will review the research questions posed in Section 1.2 by summarizing the results of this thesis, both theoretical as well as empirical ones.

### Under what conditions can PGS successfully acquire effective plans?

There are a number of assumptions built into PGS, some of which are rather strong and are not commonly made in related work such as option discovery in reinforcement learning. In particular, PGS implicitly categorizes actions with regard to their contribution to certain subgoals. This implies that the usefulness of actions has to be decidable immediately. In fact, this is a requirement of supervised learning that is explicitly not made in reinforcement learning, which is thereby rendered particularly suitable for the application in situated agents. In domains where the usefulness of actions is not directly obvious, this assumption renders PGS unfeasible. In such a case, we would only be able to deploy PGS by using a particular subgoal that is not readily

interpretable by humans. However, this workaround also prevents the extraction of meaningful plans, which is actually a particular benefit of PGS.

Moreover, the original model needed to be able to predict the outcome of actions to decide about their contribution to subgoals. We managed to waive this assumption by reorganizing the original execution cycle. Anyway, learned plans were generally applied in experiments without a performance loss compared to a plain reinforcement learner. Only in the case of a totally randomly behaving environment, the reuse of plans naturally became a general disadvantage.

### How can the reuse of plans, and in particular longer plans, be facilitated?

We were able to increase plan reuse by applying different state abstraction strategies, which allowed plans to be reused in situations beyond their original preconditions. Generally, plan reuse in experiments was significantly greater than in previous work without a trade-off in performance. Likewise, the average length of reused plans was increased significantly by improving the plan recording mechanism. In particular, the utility information for states of the reinforcement learner was exploited to provide a more informed utility definition for plans. However, plans were only reused a few times before they were dropped from the plan library again. We reason that our state abstractions were not entirely appropriate and led to plans being reused in unsuitable situations too soon. Indeed, with a more carefully designed state abstraction, plans started to have a longer life time. This is in line with the general assumption that plans or action sequences are naturally more sensitive to their place of application than single actions. Plan reuse and plan length increased with the predictability of the environment, becoming quite impressive for the totally predictable case. This showed that the model indeed works as intended.

### What is the impact of using plans on the overall performance?

Generally, the reuse of plans yielded a performance improvement as long as there were temporal patterns in the behavior of the environment and as long as a state abstraction was applied. Larger improvements were noted when the environment was more predictable and a more advanced state abstraction was used. In general, the largest improvement was observed during the exploration phase. This follows from our argument that plans comprise more information than single actions and are thus a more powerful tool for exploration in unknown parts of the state space. This observation underlines the need to intertwine temporal and state abstraction. However,

the reinforcement learner does not benefit from the general performance improvement at all. In fact, knowledge was not transferred from the top-level module to the bottom-level, even though the reinforcement learner is learning during plan execution. We reason that such effects previously observed in the options framework do not occur in PGS because of two restrictions: The reinforcement learner does not apply value updates over plans and the knowledge representation at the top-level is too powerful to be mapped by the bottom-level. In addition to performance improvements, the number of decisions to be taken decreased significantly, of course depending on the level of plan reuse. This allows resource-bounded agents in particular to spend time and resources on other processes than reasoning.

### How can prior knowledge be incorporated into PGS?

We have investigated two possibilities for integrating prior knowledge into PGS in this thesis: guiding the plan selection process and priming the plan library with prior plans as partial solutions to the problem. The former was already implicitly exploited during the use of state abstraction. The latter allows for the injection of temporal prior knowledge, which can be defined on a higher level than primitive actions. This makes it generally more suitable as a language for specifying prior knowledge. We showed that prior knowledge plans can be defined under the very same interface as ordinary PGS plans such that they can be seamlessly integrated into the system. The experiments show that such knowledge – even if it is very basic – can substantially increase performance during the learning phase of the agent. However, as soon as the reinforcement learner has reached a better policy, the advantage turns into a disadvantage. This encourages the development of "mechanisms" for unlearning prior knowledge, which PGS exhibits but which were not explicitly exploited in the course of this work.

## 6.2 Future work

In the following, we will outline a few possible themes for the extension of PGS.

PGS could be re-engineered to become a fully compliant option discovery algorithm. This would generally require only two steps, namely integrating extracted plans back into the reinforcement learner as options and rendering the reinforcement learner capable of learning over these options. A suitable learning algorithm was discussed in Section 2.1.3. Such work would shed light on the question of whether plans, as

generated by PGS, are suitable options at all. It is interesting because this approach for option discovery would follow a completely different idea than previous work. The latter typically identifies a possible subgoal first and then learns a policy for reaching it.

We have touched upon strategies for state abstraction in Chapter 5. However, there is surely space for improvement. Applying more powerful state abstraction techniques could lead to a more effective plan reuse. In a second step, entire sets of similar plans could be analyzed in order to find a common representative and a combined precondition. This idea could be taken further by generating a representative that consists of more powerful instructions such as loops or conditions, which would allow a more informed mapping of its parents' policies. These representatives could be manipulated or evolved further, for example involving genetic programming methods.

Furthermore, PGS could be deployed to more sophisticated environments, in which more actions are available and particular actions need to be carried out in sequence in order to have any meaning to the domain. Likewise, the assumption of deterministic action outcomes could be waived to study the impact of another component of uncertainty.

# Bibliography


Anderson, J. R. (1990). *The Adaptive Character of Thought*. Erlbaum, Hillsdale, NJ.

Andre, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 119–125. AAAI Press.

Bergmann, R. and Wilke, W. (1996). PARIS: Flexible plan adaptation by abstraction and refinement. In *Proceedings of the ECAI 1996 Workshop on Adaptation in Case-Based Reasoning*.

Bratman, M. E., Israel, D., and Pollack, M. (1988). Plans and resource-bounded practical reasoning. In Cummins, R. and Pollock, J. L., editors, *Philosophy and AI: Essays at the Interface*, pages 1–22, Cambridge, Massachusetts. The MIT Press.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23.

Bryson, J. (2000). Cross-paradigm analysis of autonomous agent architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(2):165–190.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377.

Cohen, P. R. (1995). *Empirical methods for artificial intelligence*. MIT Press, Cambridge, MA, USA.

Decker, K. S. (1995). *Environment centered analysis and design of coordination mechanisms*. PhD thesis, University of Massachusetts Amherst. Director-Victor R. Lesser.

Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126. Morgan Kaufmann.

Dixon, K. R., Malak, R. J., and Khosla, P. K. (2000). Incorporating prior knowledge and previously learned information into reinforcement learning agents. Technical report, Carnegie Mellon University.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

Garland, A. and Alterman, R. (2001). Learning procedural knowledge to better coordinate. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1073–1083.

Georgeff, M. P. and Lansky, A. L. (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Guerra-Hernández, A., Fallah-Seghrouchni, A. E., and Soldano, H. (2004). Learning in BDI multi-agent systems. In *CLIMA IV*, pages 218–233.

Hailu, G. and Sommer, G. (1999). On amount and quality of bias in reinforcement learning. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 1491–1495.

Ingrand, F. F., Georgeff, M. P., and Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and their Applications*, 7(6):34–44.

Jaakkola, T., Jordan, M. I., and Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201.

Jong, N. K., Hester, T., and Stone, P. (2008). The utility of temporal abstraction in reinforcement learning. In *Proceedings of the 7th international Joint Conference on Autonomous Agents and Multiagent Systems*, pages 299–306, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Karim, S. (2009). *Extracting Plans in Situated, Resource-Bounded Agents: A Hybrid Approach*. PhD thesis, The University of Melbourne. submitted.

Karim, S., Sonenberg, L., and Tan, A.-H. (2006a). A hybrid architecture combining reactive plan execution and reactive learning. In *Pacific Rim International Conferences on Artificial Intelligence*, Lecture Notes in Computer Science, pages 200–211, Heidelberg. Springer Berlin.

Karim, S., Subagdja, B., Pasquier, P., and Sonenberg, L. (2008). Plans: A target representation for learned knowledge.

Karim, S., Subagdja, B., and Sonenberg, L. (2006b). Plans as products of learning. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 139–145, Washington, DC, USA. IEEE Computer Society.

Koenig, S., Simmons, R. G., and Kaelbling, P. (1996). The effect of representation and knowledge on goal-directed exploration with reinforcement learning algorithms. In *Machine Learning*, pages 227–250.

Lebiere, C. and Wallach, D. (2001). Sequence learning in the ACT-R cognitive architecture: Empirical analysis of a hybrid model. In *Sequence Learning - Paradigms, Algorithms, and Applications*, pages 188–212, London, UK. Springer-Verlag.

Lebiere, C., Wallach, D., and Taatgen, N. (1998). Implicit and explicit learning in ACT-R. In Young, I. F. R. . R., editor, *Cognitive Modeling II*, pages 183–193. Nottingham University Press, Nottingham.

Li, L., Walsh, T. J., and Littman, M. L. (2006). Towards a unified theory of state abstractions for MDPs. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, pages 531–539.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. In *Machine Learning*, pages 293–321.

M. Benda, V. J. and Dodhiawala, R. (1986). On optimal cooperation of knowledge sources - an empirical investigation. Technical Report BCS-G2010-28, Boeing Advanced Technology Center, Boeing Computing Services, Seattle, Washington.

Maclin, R. and Shavlik, J. W. (1996). Creating advice-taking reinforcement learners. In *Machine Learning*, pages 251–281.

Mahadevan, S. and Kaelbling, L. P. (1996). The NSF workshop on reinforcement learning: Summary and observations. *AI Magazine*, 17:89–97.

McGovern, A. and Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 361–368, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

McGovern, A., Sutton, R. S., and Fagg, A. H. (1997). Roles of macro-actions in accelerating reinforcement learning. In *Proceedings of the 1997 Grace Hopper Celebration of Women in Computing*, pages 13–18.

Menache, I., Mannor, S., and Shimkin, N. (2002). Q-Cut - Dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, pages 295–306, London, UK. Springer-Verlag.

Mitchell, T. (1997). *Machine Learning*. McGraw Hill.

Mitchell, T. M. and Thrun, S. B. (1993). Explanation-based neural network learning for robot control. In *Advances in Neural Information Processing Systems 5*, pages 287–294. Morgan Kaufmann.

Moody, J. and Darken, C. J. (1989). Fast learning in networks of locally tuned processing units. *Neural Computation*, 1:281–294.

Newell, A. and Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126.

Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the 16th International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann.

Olivia, C., Chang, C.-F., Enguix, C. F., , and Ghose, A. K. (1999). Case-based BDI agents: An effective approach for intelligent search on the world wide web. In *Proceedings of the AAAI-99 Spring Symposium on Intelligent Agents in Cyberspace*, USA.

Özgür Şimşek and Barto, A. G. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, page 95, New York, NY, USA. ACM.

Özgür Şimşek, Wolfe, A. P., and Barto, A. G. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the Twenty-second International Conference on Machine learning*, pages 816–823, New York, NY, USA. ACM.

Parr, R. and Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press.

Pickett, M. and Barto, A. G. (2002). PolicyBlocks: An algorithm for creating useful macro-actions in reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 506–513, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Pollack, M. E. (1992). The uses of plans. *Artificial Intelligence*, 57:43–68.

Precup, D., Sutton, R. S., and Singh, S. (1997). Planning with closed-loop macro actions. In *In Working Notes of the 1997 AAAI Fall Symposium on Model-directed Autonomous Systems*, pages 70–76. MIT Press.

Precup, D., Sutton, R. S., and Singh, S. (1998). Theoretical results on reinforcement learning with temporally abstract behaviors. In *Tenth European Conference on Machine Learning*, pages 382–393, Chemnitz, Germany. Springer Verlag.

Price, B. and Boutilier, C. (1999). Implicit imitation in multiagent reinforcement learning. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 325–334. Morgan Kaufmann.

Rao, A. (1997). A unified view of plans as recipes.

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall International, 2 edition.

Sacerdoti, E. (1975). The nonlinear nature of plans. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 206–214. Morgan Kaufmann.

Schuurmans, D. and Greenwald, L. (1999). Efficient exploration for optimizing immediate reward. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 385–392, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Shapiro, D. (2001). Using background knowledge to speed reinforcement learning in physical agents. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 254–261. ACM Press.

Shavlik, J. W. and Towell, G. G. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science*, 1:233–255.

Stolle, M. and Precup, D. (2002). Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pages 212–223, London, UK. Springer-Verlag.

Stone, P. and Veloso, M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383.

Sun, R. (1997). An agent architecture for on-line learning of procedural and declarative knowledge.

Sun, R. (1999). Knowledge extraction from reinforcement learning. In *International Joint Conference on Neural Networks*, volume 4, pages 2554–2559.

Sun, R., Merrill, E., and Peterson, T. (2001). From implicit skills to explicit knowledge: A bottom-up model of skill learning. *Cognitive Science*, 25:203–244.

Sun, R. and Sessions, C. (1998). Learning plans without a priori knowledge. In *World Congress on Computational Intelligence / INNS-IEEE International Joint Conference on Neural Networks*, volume 1, pages 1–6, Piscateway, NJ. IEEE Press.

Sun, R., Slusarz, P., and Terry, C. (2005). The interaction of the explicit and the implicit in skill learning: A dual-process approach. *Psychological Review*, 112:159–192.

Sun, R. and Zhang, X. (2004). Top-down versus bottom-up learning in cognitive skill acquisition. *Cognitive Systems Research*, 5(1):63–89.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Sutton, R. S. (1995). TD models: Modeling the world at a mixture of time scales. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 531–539. Morgan Kaufmann.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

Tan, A.-H. (2004). FALCON: A fusion architecture for learning, cognition, and navigation. In *Proceedings of the 2004 IEEE International Joint Conference on Neural Networks*, volume 4, pages 3297–3302.

Tan, A.-H. (2007). Direct code access in self-organizing neural networks for reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 1071–1076.

Thrun, S. and Schwartz, A. (1995). Finding structure in reinforcement learning. In *Advances in Neural Information Processing Systems 7*, pages 385–392. MIT Press.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and q-learning. *Machine Learning*, 16:185–202.

van Riemsdijk, M. B., Dastani, M., and Winikoff, M. (2008). Goals in agent systems: A unifying framework. In *Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England.

Weiss, G. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press.

Whitehead, S. D. (1991). A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 607–613.

Wiewiora, E., Cottrell, G., and Elkan, C. (2003). Principled methods for advising reinforcement learning agents. In *Proceedings of the International Conference on Machine Learning*.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10:115–152.

Zimmerman, T. and Kambhampati, S. (2003). Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine*, 24:73 – 96.

# Declaration

This thesis is my original work and has not been submitted, in whole or in part, for a degree at this or any other university. Nor does it contain, to the best of my knowledge and belief, any material published or written by another person, except as acknowledged in the text.

Melbourne, Australia, December 19, 2008 Jens Pfau