Plan Projection, Execution, and Learning For Mobile Robot Control

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Thorsten Belker

aus

Wuppertal

Bonn (Januar) 2004

To my parents

Acknowlegements

The work described in this thesis could not have been done without the support of numerous people. First of all, I wish to thank my advisor Armin B. Cremers and my co-advisor Joachim Hertzberg. I am grateful for their suggestions and their guidance. I would also like to express my gratitude to Armin B. Cremers as well as Michael Beetz for offering me the opportunity to work on the DFG funded research project "Modell- und diagnosebasiertes Transformationslernen von symbolischen Navigationsplänen für mobile Roboter" which inspired me to this thesis in the first place.

This work owes much of its inspiration to Michael Beetz. In many hours of discussion he was always willing to listen and gave valuable suggestions. Joachim Hertzberg as well had an enormous impact on my thinking and helped to improve this work a lot. I thank both of them as well as Malik Ghallab and Martha E. Pollack for organizing the seminars on plan-based robot control which took place at Schloss Dagstuhl in 2001 and 2003. The many fruitful discussions at Dagstuhl helped me a lot in finding a scientific approach to the problems I address in this thesis.

Several other people contributed to this thesis as well. Let me first mention Jürgen Schumacher who has been my office mate for the last four years. I would like to thank him for his patience and serenity. I would also like to thank him and Dirk Schulz for many valuable suggestions which helped to improve this work a lot. Finally, I would like to thank all the other current and former members of the "RHINO team", and especially Wolfram Burgard, Dieter Fox, Dirk Hähnel, Mark Moors, Martin Hammel, and Klaus Schulz for the joyful working atmosphere and their spirit of cooperation.

Abstract

Most state-of-the-art hybrid control systems for mobile robots are decomposed into different layers. While the deliberation layer reasons about the actions required for the robot in order to achieve a given goal, the behavioral layer is designed to enable the robot to quickly react to unforeseen events. This decomposition guarantees a safe operation even in the presence of unforeseen and dynamic obstacles and enables the robot to cope with situations it was not explicitly programmed for.

The layered design, however, also leaves us with the problem of plan execution. The problem of plan execution is the problem of arbitrating between the deliberation- and the behavioral layer. Abstract symbolic actions have to be translated into streams of local control commands. Simultaneously, execution failures have to be handled on an appropriate level of abstraction. It is now widely accepted that plan execution should form a third layer of a hybrid robot control system. The resulting layered architectures are called three-tiered architectures, or 3T architectures for short.

Although many high level programming frameworks have been proposed to support the implementation of the intermediate layer, there is no generally accepted algorithmic basis for plan execution in three-tiered architectures. In this thesis, we propose to base plan execution on plan projection and learning and present a general framework for the self-supervised improvement of plan execution. This framework has been implemented in APPEAL, an Architecture for Plan Projection, Execution And Learning, which extends the well known RHINO control system by introducing an execution layer.

This thesis contributes to the field of plan-based mobile robot control which investigates the interrelation between planning, reasoning, and learning techniques based on an explicit representation of the robot's intended course of action, a plan. In McDermott's terminology, a plan is that part of a robot control program, which the robot cannot only execute, but also reason about and manipulate. According to that broad view, a plan may serve many purposes in a robot control system like reasoning about future behavior, the revision of intended activities, or learning. In this thesis, plan-based control is applied to the self-supervised improvement of mobile robot plan execution.

Arbeiter der Stirn

Ein Mensch sitzt kummervoll und stier
Vor einem weißen Blatt Papier.
Jedoch vergeblich ist das Sitzen –
Auch wiederholtes Bleistiftspitzen
Schärft statt des Geistes nur den Stift.
Selbst der Zigarre bittres Gift,
Kaffee gar, kannenvoll geschlürft,
Den Geist nicht aus den Tiefen schürft,
Darinnen er, gemein verbockt,
Höchst unzugänglich einsam hockt.
Dem Menschen kann es nicht gelingen,
Ihn auf das leere Blatt zu bringen.
Der Mensch erkennt, daß er nichts nützt,
Wenn er den Geist an sich besitzt,
Weil Geist uns ja erst Freunde macht,
Sobald er zu Papier gebracht.

aus: Eugen Roth, Ein Mensch, 1935

Contents

Chapter 1

Introduction

Most state-of-the-art hybrid control systems for mobile robots are decomposed into different layers. While the deliberation layer reasons about the actions required for the robot in order to achieve a given goal, the behavioral layer is designed to enable the robot to quickly react to unforeseen events.

The deliberation layer provides an interface for user interaction. It employs symbolic planning techniques like means-end analysis to compute plans for user specified goals. For this purpose, the planner uses an abstract and mainly static model of the robot's environment and its actions. The resulting action plans are often partially ordered sets of abstract symbolic action specifications. The behavioral layer, on the other hand, is designed for fast feed-back control. It is reactive in that it only relies on the current sensor readings of the robot or a local environment model built from the most recent sensor readings.

This decomposition guarantees a safe operation even in the presence of unforeseen and dynamic obstacles and enables the robot to cope with situations it was not explicitly programmed for. From a software technological point of view, this design is advantageous as the layers can be designed independently by different teams of developers. The layered design, however, also leaves us with the problem of plan execution.

The problem of plan execution is the problem of arbitrating between the deliberation- and the behavioral layer. Abstract symbolic actions have to be translated into streams of local control commands. Simultaneously, execution failures have to be handled on an appropriate level of abstraction. It is now widely accepted that plan execution should form a third layer of a hybrid robot control system. The resulting layered architectures are called three-tiered architectures, or 3T architectures for short.

Although many high level programming frameworks have been proposed to support the implementation of the intermediate layer, there is no generally accepted algorithmic basis for plan execution in three-tiered architectures. In this thesis, we propose to base plan execution on plan projection and learning, and we present a general framework for the self-supervised improvement of plan execution. This framework has been implemented in APPEAL, an Architecture for Plan Projection, Execution And Learning, which extends the well known RHINO control system by introducing an execution layer.

The core of APPEAL's execution layer is the explicit representation of the state of plan execution. This is achieved using Hierarchical Task Networks (HTNs). This representation forms the basis for task decomposition and failure recovery. In contrast to HTN action planning, there are in general many ways to decompose an abstract action into a partially-ordered set of actions and many possible serializations of a partial order. Which expansion is selected and how the tasks are serialized, however, might have a substantial impact on the robot's performance. We consider HTN execution planning as a search for plans with a high expected performance.

To estimate the expected performance of a plan, the execution of the plan is projected. It is one of the main ideas presented in this thesis to base the projection on action models learned from experience. The learned models provide an estimate of the time needed to execute an action. They can be learned from data collected during the plan execution process.

The search for an optimal execution plan is computationally expensive. To mitigate this problem, we interleave planning and execution. This is achieved by applying transformational planning techniques. A default plan is generated by selecting the standard expansion for all pending actions. While the robot starts to execute the plan, the planner can reason about different courses of actions. As soon as a more promising HTN has been determined, the execution plan is transformed accordingly.

This thesis is organized as follows. Chapter 2 describes the RHINO control software. Although mobile robot navigation in general is a Partially Observable Markov Decision Process, the RHINO navigation system heuristically decomposes the POMDP into a state estimation process, a navigation planning process and an active localization process. Following this line of reasoning, we suggest adding another planning process, an execution planning process. This process – as the navigation planning – can be considered as a Markov Decision Process. Experimental results show that execution planning in combination with learned action models bears the potential to improve the robot's performance considerably.

The experiment also demonstrates the need to improve the reliability of the behavioral layer. Chapter 3 describes how the trajectory evaluation for reactive navigation can be improved using local planning techniques.

Chapter 4 introduces the idea to use HTNs to represent the state of plan execution explicitly. Besides the use of HTN planning for task decomposition, we show the benefits of HTNs for handling execution failures on the right level of abstraction. Experiments show that using these techniques the reliability of plan execution can be improved substantially.

Chapter 5 and Chapter 6 describe the application of plan projection to execution planning. There are two main tasks involved in execution planning. First, the sequencing of a partially ordered set of tasks and second, the selection of the most suitable task expansion. A solution to the first task is described in Chapter 5. A solution to the second task is discussed in Chapter 6. The main focus of Chapter 5 is on learning the required action models. The proposed solution is based on features derived from a path planning process and model tree learning. The focus of Chapter 6 is on the application of transformational planning techniques and the application of a slightly different plan projection technique.

Due to the application of transformational planning techniques it becomes feasible to perform online execution planning. However, the action selection process is not transparent to the human observer. In Chapter 7, we demonstrate how action selection rules can be learned from offline execution planning. The action selection rules can be applied in HTN planning at the time of plan generation and turn out to be an alternative to transformational planning. The technique, however, is only applicable in the case of planning problems that do not require a long planning horizon, and not for example to the sequencing problem.

This thesis contributes to the field of plan-based mobile robot control which investigates the interrelation between planning, reasoning, and learning techniques based on an explicit representation of the robot's intended course of action, a plan. In McDermott's terminology [McD92a], a plan is that part of a robot control program, which the robot cannot only execute, but also reason about and manipulate. According to that broad view, a plan may serve many purposes in a robot control system: As Beetz et al. [BHGP02] put it, "the use of plans enables these robots to flexibly interleave complex and interacting tasks, exploit opportunities, quickly plan their courses of action, and, if necessary, revise their intended activities". In this thesis, plan-based control is applied to the self-supervised improvement of mobile robot plan execution.

The idea to have a mobile robot autonomously learn from past experience how its performance can be improved is central to our work. For the task of plan execution, this kind of knowledge is difficult to specify for the human programmer as it requires a deep understanding of the control system. In addition, it is a cumbersome work to specify a good execution policy. It is the aim of APPEAL to automatize this work.

For the application of self-supervised learning techniques to the control of a mobile robot in a human-structured environment, it is important to provide as much domain knowledge to the robot as possible. The use of domain knowledge helps to reduce the amount of data required for learning and confines the space of learnable policies. In APPEAL, this is achieved by providing the task hierarchy of the HTN planning framework. The set of possible tasks and the set of possible task expansions structure the learning problem and therefore simplify learning considerably.

When trying to learn parts of a mobile robot control system, especially if the robot is to work together with people, it is important that the control system is transparent to the human operator. In APPEAL, this is achieved by using symbolic learning algorithms both for the acquisition of the predictive models as well as the acquisition of action selection rules. In this thesis, we propose to apply tree-based inductive learning as well as sequential covering algorithms to both tasks. These algorithms allow to learn symbolic rules which are intelligible for human inspection. In these respects, the approach presented in this thesis differs from other approaches to self-supervised learning for robot control like Q-learning and evolutionary algorithms.

Chapter 2

The RHINO Architecture

This chapter provides an overview of the RHINO architecture. Rather than giving a detailed description of all different aspects and components, it focuses on its main design principles. The RHINO system can be characterized as a modular and hybrid mobile robot control architecture which is based on probabilistic and decision-theoretic algorithms for state estimation, navigation planning, and active localization.

Despite the impressive performance of the RHINO system which has been demonstrated in two long term experiments, we argue that it can still be enhanced substantially by execution planning. To underpin this hypothesis, we describe experimental results obtained by augmenting the RHINO system with a decision-theoretic action selection component. The experiments show that the performance of RHINO can be improved considerably. These results motivate the application of planning and learning techniques to the problem of plan execution.

We argue that the problem of selecting the next target point from the optimal path to the goal can be considered as a Markov Decision Process. The proposed action selection function, however, is limited in various aspects. We discuss the limitations in some detail in order to motivate the improvements suggested in subsequent chapters. We demonstrate how learned action models can be applied to decision-theoretic action selection. The idea that execution planning should be based on learned models of the robot's actions is central to this thesis and further refined in the following chapters.

2.1 Introduction

Purely reactive approaches to mobile robot navigation (please refer to the introductory books of Arkin [Ark98] and Murphy [Mur00] or the article collection of Brooks [Bro99] for details) are based on the assumption that successful mobile robot behavior can be achieved solely by reacting appropriately to sensor stimulus. In contrast to these approaches, the RHINO system relies on an explicit representation of the environment. For navigation, a metric map is used. As described by Thrun et al. [TBB+98, TBB+99] a grid map of an indoor environment can be learned by the robot.

The problem of mobile robot navigation in known environments can be decomposed into three interrelated subtasks: (1) The problem of self-localization, estimating the robot's pose (position and orientation) within the environment, (2) the problem of navigation planning, computing a sequence of actions that when executed lead the robot to its goal position, and (3) the problem of local control, the execution of purely reactive navigation behaviors.

As can be seen from Figure 2.1, a separate module is devoted in the RHINO architecture to each of these three problems. Additional modules exist for sensor interpretation, effector control, map learning, task-level planning, and user interaction.

The localization component [Fox98, FBT99, FBDT99] tries to fit the sensor readings of a mobile robot in a model (a map) of the robot's working environment to estimate the robot's current position or pose within the environment and reduce uncertainty caused by unreliable dead reckoning. In the case that the uncertainty about the current robot position becomes too high, the localization component can make suggestions about navigation actions which might help to reduce the robot's position uncertainty. This process is called active localization [BFT97, BBFC98].

The navigation planning component receives goal points from either a user interface, a task planner or - in the case of active localization - from the localization component. In the RHINO software, navigation planning is performed using a path planner that computes optimal paths to a given goal. Path planning is done using value iteration where the underlying state space is based on a fine-grained two-dimensional grid map of the environment. As explained in Section 2.2.1, value iteration results in a policy which assigns an optimal navigation action to each state in the state space. For the navigation problem, this means that an optimal path to the goal can be computed efficiently for each state in the state space by executing the navigation policy.

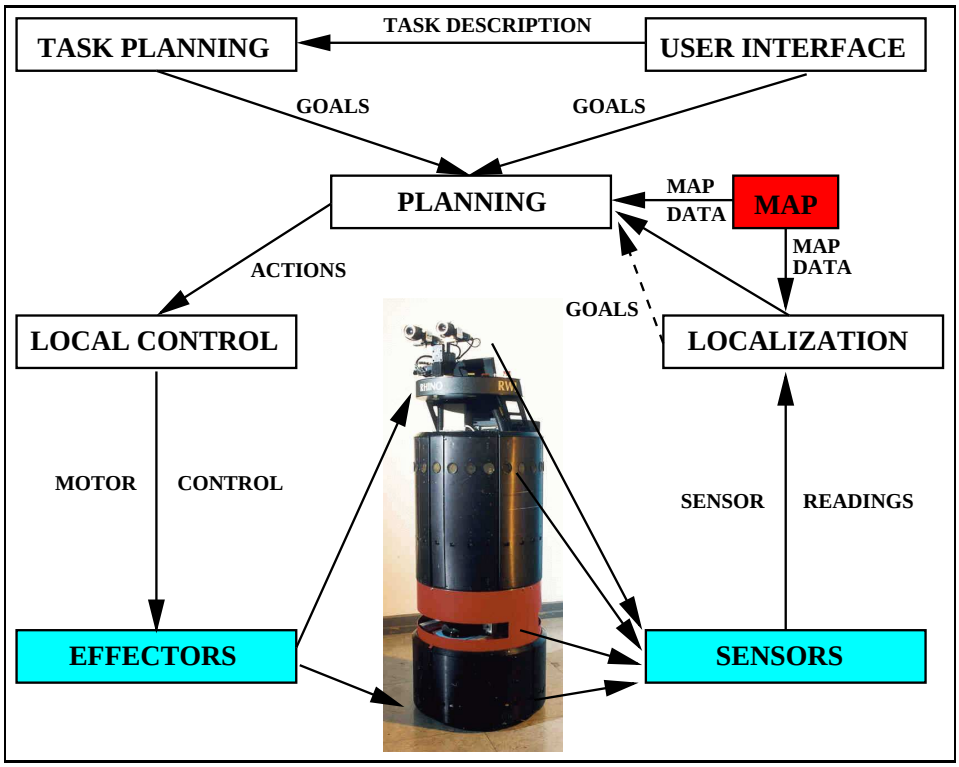

Figure 2.1: Main components of the RHINO control software.

This kind of path planning algorithms support efficient replanning when the robot is detected to deviate from the previously computed path. Plans are executed by computing intermediate points that are passed to the execution layer to be approached reactively.

The local control component ensures a safe navigation even in the case of unexpected or even dynamic obstacles. It is designed to approach a given local target point while avoiding obstacles. It generates a sequence of motor control commands that can be directly executed by the robot while taking its dynamic constraints into account. It is purely reactive, i.e., reacts directly to its sensor readings. This makes the robot independent of a global map and the correct estimation of its position with respect to this map.

Each module is designed as an own process and modules can therefore be distributed over various machines. The asynchronous inter-process communication was originally implemented using the TCX package [Fed93] specifically developed for mobile robot control. In a recent reimplementation of the system and the extensions described in this thesis, however, CORBA was used

Figure 2.2: The museum tour guide robots RHINO (left) and MINERVA (right).

as a middleware platform. Modules are implemented as software components that can be distributed over a local network. Using CORBA, modules can be run on different operation systems and be implemented using different programming languages like C++, JAVA, or LISP.

The use of a modular and component-based architecture allows to quickly reconfigure the system, depending on a particular robot, sensor configuration and application. It also simplifies the software engineering process, as different teams of programmers can extend and modify different modules independently.

The tour guide application [BCF+98, TBB+99] made it necessary to introduce additional modules for user interaction and task planning. A web interface [SBC99, SBF+00, Sch02] makes it possible to control and monitor the robot via the Internet, while simultaneously providing background information and a forum for discussion. The task planner schedules the asynchronously arriving user requests and coordinates various robot activities related to navigation and user interaction. The task planner also monitors the successful execution of plans and modifies plans in response to plan failures.

In the first tour guide project in the Deutsches Museum in Bonn, GOLOG, a first-order logic programming language based on the situation calculus [LRL+97], was used to schedule the user requests for visiting exhibits. The main benefit of GOLOG for mobile robot control is that it allows to integrate programming and planning and therefore makes it easy to specify high-level control programs. The integration of the GOLOG planner into the RHINO system was achieved by an interface module, GOLEX [HBL98], written in PROLOG and C which extends GOLOG in three aspects.

1. It introduces conditional plans and the possibility to react to exogenous events.

2. It allows to monitor the execution of the computed plan and to react to plan failures.

3. It provides a set of abstract actions that are translated into a sequence of actions executable by the lower control layers.

In the second tour guide project in the Smithsonian Museum of American History in Washington, DC, a structured reactive controller (SRC) [Bee99] was applied to the task planning. The SRC could compute default plans efficiently, monitor their execution and detect plan failures or non-standard situations and repair plans or adapt them in reaction to unforeseen situations using plan transformations. The SRC was implemented using RPL, a LISP macro language, and integrated into the RHINO system using HLI, a C-based subsystem of GOLEX. RPL supports the reaction to exogenous events, conditional plans and sub-plan abstractions. RPL is therefore a more powerful mobile robot control language than GOLOG, and is in this respect more similar to a standard programming language like C.

The integration of deliberative and reactive components makes the RHINO system a hybrid mobile robot control system. The navigation software comprises a pilot layer and a navigator layer which is a common decomposition of navigation software [Mey90, CKK96]. The pilot layer comprises the collision avoidance module and the modules for sensor analysis and effector control, while the navigator layer consists of the localization component, the map component, and the path planning module. The task planner and the user interface form a third layer, the deliberation layer. The plan execution modules GOLEX/HLI form an interface layer between navigator layer and deliberative layer.

It is the central hypothesis of this thesis that despite the impressive performance of the RHINO system as demonstrated in the two tour guide experi-

ments, the navigation performance can still be improved considerably when applying learning and planning techniques to the task of plan execution.

We propose in this thesis to extend the RHINO system in three aspects.

- To improve the behavioral layer by introducing a local planning process to guide the search for good trajectories based on Markov Decision Process Planning (Chapter 3).
- To extend the RHINO system by an execution planning system which is based on Hierarchical Task Networks (Chapter 4).
- To enhance the system by a learning component which acquires action models from previous experience. The learned action models form the basis of plan projection and execution planning (Chapter 5 – Chapter 7).

The RHINO system extended in these aspects will in the following be denoted as APPEAL, Architecture for Plan Projection, Execution and Learning.

To underpin the before mentioned hypothesis, we describe in this chapter experimental results obtained by extending the RHINO system with a decision-theoretic action selection component. We suggest modeling the problem of selecting the next target point from the optimal path to the goal as a Markov Decision Process and discuss how the necessary action models can be learned from experience. The experiments show that the performance of RHINO can be improved considerably in this way.

The proposed action selection function, however, is limited in various respects. We discuss the limitations in some detail in order to motivate the improvements suggested in subsequent chapters. Besides the introduction of the basic concepts used throughout the thesis and underlying the design of the RHINO system like (Partially Observable) Markov Decision Processes, the purpose of this chapter is to motivate the application of planning and learning techniques for the task of plan execution.

The chapter is organized as follows. After introducing (Partially Observable) Markov Decision Processes and discussing their application in the RHINO system as well as in other robot control architectures in Section 2.2, we describe how the problem of selecting intermediate points can be modeled as Markov Decision Process in Section 2.3. Section 2.4 deals with the question of how the necessary action models can be learned from experience. In Section 2.5, we demonstrate the usefulness of the decision-theoretic intermediate point selection in both simulator- and real world experiments and conclude in Section 2.6.

2.2 Related Work

In addition to the modular design of the software, the RHINO system is characterized by the application of probabilistic and decision-theoretic algorithms. As will be discussed in this section, the navigation problem is considered as a Partially Observable Markov Decision Process (POMDP), and an approximate solution is computed using a technique called Greedy POMDP Planning. This section provides a brief introduction to Fully and Partially Observable Markov Decision Processes and compares the implementation of Greedy POMDP Planning in the RHINO system to other related robot navigation systems. The application of POMDP planning is well suited for mobile robot control because it allows to model (a) nondeterministic action effects, (b) partial observability, (c) unreliable sensor information, and (d) effector inaccuracy.

2.2.1 Markov Decision Processes

Markov Decision Processes (MDPs) provide a general framework for the specification of simple control problems where an agent acts in a stochastic environment and receives rewards from this environment. A solution of an MDP is a policy, a mapping from states into actions. An optimal policy is a policy that maximizes the expected accumulated future reward.

More formally, an MDP is given by

- a set of states S,
- a set of actions A,
- a probabilistic action model $P(S|S, A)$,
- and a reward function $R : S \times A \rightarrow \mathbb{R}$.

$P(s'|s, a)$ specifies the probability that action a taken in state s leads to the state s'. $R(s, a)$ denotes the immediate reward gained by taking action a in state s. The property that action effects only depend on the action and the state in which they are executed is called the Markov property. The optimal solution to the Markov Decision Problem is a policy π given by the Bellman Equation (for a discrete state space):

$$\pi(s) = \arg\max_{a \in A}[R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)V^*(s')] \tag{2.1}$$

Algorithm 1 Value iteration algorithm

for all $s \in S$ **do**
  $V_1(s) := 0$
end for
$t := 1$
repeat
  $t := t + 1$
  for all $s \in S$ **do**
    for all $a \in A$ **do**
      $Q_t(s,a) := R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_{t-1}(s')$
    end for
    $V_t(s) := \max_a Q_t(s,a)$
  end for
until $|V_t(s) - V_{t-1}(s)| < \epsilon$ for all $s \in S$

The action $\pi(s)$ in Equation 2.1 is the action that maximizes the sum of the immediate reward $R(s,a)$ for taking action a in state s and the discounted expected future reward $\gamma \sum_{s' \in S} P(s'|s,a) V^*(s')$. $V^*(s')$ denotes the utility of being in state s' when acting according to the optimal policy. The constant γ is called discounting factor and weighs the expected future rewards with respect to how far in the future they will occur.

Markov Decision Processes can be solved efficiently using value iteration [Bel57] or policy iteration [How60]. Both algorithms are based on the ideas of dynamic programming. In Algorithm 1 we describe the value iteration algorithm.

It can be shown that $\lim_{t \to \infty} V_t = V^*$, i.e. that the value function converges to the optimal value function. In addition, one can prove [WL93] that the following property holds.

$$\forall s \in S \;\; |V_t(s) - V_{t-1}(s)| < \epsilon \Rightarrow \max_{s \in S} |V_t(s) - V^*(s)| < 2\epsilon \frac{\gamma}{1-\gamma} \qquad (2.2)$$

It gives us a criterion for terminating the value iteration with a near-optimal value function.

2.2.2 Partially Observable Markov Decision Processes

Although in the MDP framework there will in general be uncertainty about the effects of an action a taken in state s, there is no uncertainty about the current state of the agent. The agent is assumed to have complete and perfect

perceptual capabilities. Partially Observable Markov Decision Processes are a generalization of MDPs for which this is not the case. In addition to the state space S, the action space A, the reward function R, and the action model $P(S|S,A)$ an POMDP is defined by a set of possible observations O and the observation model $P(O|S)$. $P(o|s)$ denotes the probability of making observation o in state s. In POMDPs actions and observations are assumed to alternate. While an action taken in general increases the agent's uncertainty about its current state, an observation might reduce it again.

POMDPs are much more difficult to solve than MDPs. The intuitive reason for this is that no distinction can be drawn between actions that are taken to change the world and actions that are taken to gather information. Papadimitriou and Tsitsiklis [PT87] have shown that finite horizon POMDPs are PSPACE-complete. Madani et al. have proven that infinite horizon POMDPs are even undecidable [MHC99].

The key to finding solutions for POMDPs is to cast the problem as a belief state MDP. A belief state b is a probability distribution over the state space S and thus the state space of a belief state MDP is continuous. The belief state MDP $(B(S), A, P_{bel}(B|B,A), R_{bel})$ which is equivalent to the POMDP $(S, A, P(S|S,A), R, O, P(O|S))$ is given by

- a set of belief states $B(S)$ over S,
- a set of actions A,
- the action model $P_{bel}(B|B,A)$ given by

$$P_{bel}(b'|b,a) = \sum_{o \in O} P(b'|o,b,a)P(o|b,a) \tag{2.3}$$

- the reward function R_{bel} given by

$$R_{bel}(s,a) = \sum_{s \in S} b(s)R(s,a) \tag{2.4}$$

As for a belief state b the next state b' is determinated by the action a taken in b and the observation o made afterwards, $P(B'|o,b,a) = 1$ for exactly one belief state, b'.

$$P(B'|o,b,a) = \begin{cases} 1 & B' = b' \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

$SE(b, a, o) = b'$ is the estimated belief state of the agent after executing action a in belief state b and observing o. Belief state b' can be computed as follows.

$$\begin{aligned} b'(s') &= P(s'|o, a, b) \\ &= \frac{P(o|s', a, b)P(s', a, b)}{P(o, a, b)} \\ &= \frac{P(o|s', a, b)P(s'|a, b)P(a, b)}{P(o|a, b)P(a, b)} \\ &= \frac{P(o|s', a, b)P(s'|a, b)}{P(o|a, b)} \\ &= \frac{P(o|s', a, b)\sum_{s\in S} P(s'|a, b, s)P(s|a, b)}{P(o|a, b)} \\ &= \frac{P(o|s', a)\sum_{s\in S} P(s'|a, b, s)P(s|a, b)}{P(o|a, b)} \end{aligned} \tag{2.6}$$

Please note that the agent's current belief state contains all information about the agent's past actions and observations. The belief state update computed by the state estimation function is a simple application of Bayes' rule. The dominator $P(o|s, b)$ can be treated as a normalizing factor that guarantees $\sum_s b'(s) = 1$.

$P(o|b, a)$ is defined as follows.

$$P(o|b, a) = P(o|b'') = \sum_{s\in S} P(o|s)b''(s) \tag{2.7}$$

where $b''(s') = \sum_{s\in S} P(s'|s, a)b(s)$.

Based on the idea that a POMDP can be cast as a continuous-valued belief state MDP there are many algorithms that try to approximate the real-valued utility function. However, these algorithms are beyond the scope of this thesis. Please refer to [KLC98, BDH99] for a more detailed introduction to MDPs, POMDPs and the computation of approximate solutions to POMDPs, e.g. using the Witness algorithm [KLC98].

2.2.3 The Navigation Problem as a POMDP

Robot navigation is an obvious example of a Partially Observable Markov Decision Process. Using a map of the environment, the robot repeatedly senses its environment to estimate its current state, e.g. its current position and orientation within the environment. In general the robot's state is not

fully observable, for example because sonar or laser beams have only a limited range and are noisy and due to symmetries in the environment two or more states might be indistinguishable from the current sensor readings (sensor aliasing). Besides the navigation planning for achieving a given goal the robot thus has to plan information gathering actions that help it to re-localize itself. In addition, navigation actions are uncertain due to drift or possible execution failures and are associated with some cost (negative reward). The collision with obstacles is associated with some high negative reward while the event of reaching the goal state is associated with some high positive reward.

In practice, however, it is intractable to model the navigation problem as POMDP. The fact that the planning for navigation actions and the planning for information gathering actions are closely interleaved, causes the high complexity of POMDP planning which makes it infeasible for real-time control. To reduce this complexity, the planning process can be decomposed into a state estimation process and an MDP planning process. This technique is called greedy POMDP planning.

2.2.4 Greedy POMDP planning

The idea of greedy POMDP planning is to (heuristically) assume that the decision process is fully observable. Under this assumption, the POMDP is reduced to an MDP for which an optimal policy π^* can be computed efficiently. To execute the resulting policy the current belief state is updated using the state estimation function $\mathrm{SE}(b, a, o)$ and an action is selected based on the estimated belief state b'. Various strategies for greedy POMDP planning have been discussed by Cassandra al. [CKK96] and Koenig and Simmons [KS98].

The first one is the *most likely state strategy*. It computes the most likely state s with respect to the belief state b' and executes the optimal action a for state s.

$$a = \pi^*(\arg\max_s b'(s)) \tag{2.8}$$

This strategy is used in the Dervish system [NPB95, Nou98] as well as in the original Rhino system [BBC+95, TBB+98] although in both systems path planning is done using a deterministic path planner rather than a path planner based on an MDP.

In the Xavier system [SK95], the belief state b' is used to vote for the

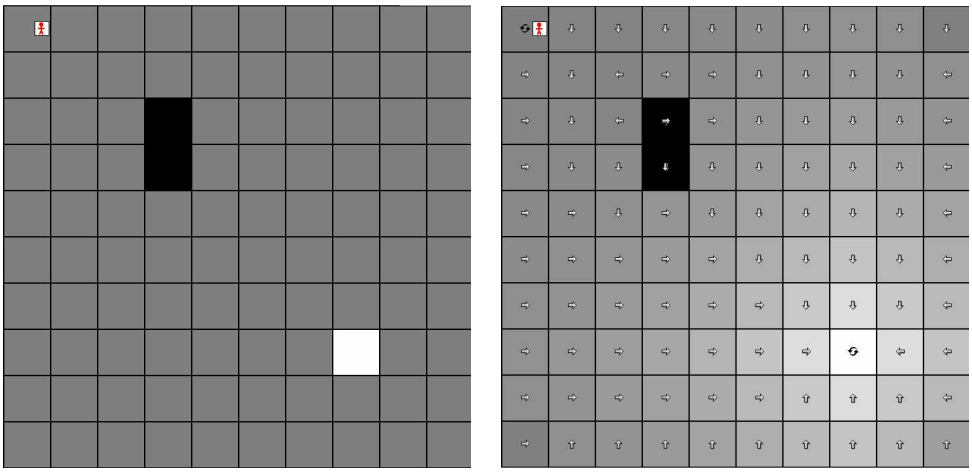

Figure 2.3: Left: The navigation problem. Right: the optimal navigation policy computed by value iteration.

optimal action. This strategy is therefore called *voting strategy*.

$$a = \arg\max_a \sum_{s \in S} b'(s) I(\pi^*(s) = a) \tag{2.9}$$

Littman et al. [LCK95] suggest a refinement of this method which weighs various actions rather than vote for them. This strategy is called *weighting strategy*. It is optimal when the state is fully observable after one step.

$$a = \arg\max_a \sum_{s \in S} b'(s)[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s')] \tag{2.10}$$

While in the DERVISH and in the XAVIER system path planning is done using deterministic path planning algorithms, Cassandra et al. [CKK96] suggest finding the best action in the underlying MDP. Figure 2.3 shows a simple navigation problem with uncertain action effects together with an optimal navigation policy for this problem.

To make the MDP planning feasible, both the state and the action space have to be discretized. The state space is discretized by dividing a metric environment map into grid cells of 1 m^2 resolution with one of four abstract orientations (compass directions). The action space is built by five abstract actions (*move-forward, turn-left, turn-right, no-op* and *declare-goal*).

In a recent reimplementation of the path planning module, we have built a path planner based on a two-dimensional MDP which uses a resolution

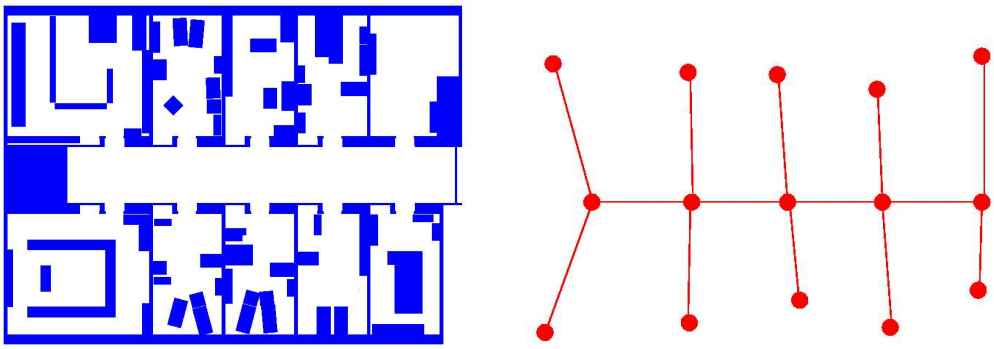

Figure 2.4: A metric and a topological map of an office environment.

of 15 cm^2. A probabilistic action model allows to model drift behavior and keeps the robot away from obstacles.

State Estimation

The problem of robot localization consists of estimating the robot's current pose from the measurements of the odometry sensors, a map of the environment and a history of sensor readings. If the initial pose of the robot is known, this is the problem of *position tracking*, if it is unknown it is the problem of *global positioning*.

In a probabilistic framework, both problems can be cast as state estimation problems and can be solved using the Bayes' formula as described in Equation 2.6. The probability distribution $P(O|S)$ is called sensor model and $P(o|s)$ describes the probability of a sensor reading o at position s. The probability distribution $P(S|S,A)$ is called motion model. $P(s'|s,a)$ describes the probability of the robot being at state s' after executing action a in state s. $P(s'|s,a)$ is computed from the measurements of the odometry sensor and a noise term which is intented as probabilistic model of drift and slippage.

Probabilistic approaches to robot localization differ in the way they model the underlying POMDP which is often closely related to the environment model used. Koenig and Simmons [KS98, SK95] use a topological map of the environment like the one shown in Figure 2.4 (right). A topological map is a graph where the nodes represent robot states (e.g. being in a room) and the edges represent possible transitions between states. The edges in a topological map can be annotated with metric information or with actions that have to be executed to translate from one state to a neighboring one.

The nodes can be annotated with probabilities to store the current belief state. Koenig and Simmons [SK95] discuss in detail how a topological map of the environment can be automatically translated into a POMDP.

The probabilistic action model $P(S|S, A)$ models drift and slippage in navigation. To be useful these transition probabilities should be learned from experience. The same is true for the observation model $P(O|S)$. Observations in the XAVIER system are abstract features like open doors, crossings etc., and their detection may fail. This is modeled in the probabilistic observation model. Koenig and Simmons describe an approach based on the Baum-Welch Algorithm for learning Partially Observable Markov Decision Models [KS96].

The RHINO system does not rely on abstract observations, but considers the raw sensor measurements as observations. The possible states of the robot are the position and orientation in a metric map of the environment. The continuous belief state can be represented discretely using grid maps [FBTC98, FBT99], octrees [BDFC98] or particle filters [FBDT99]. While grid maps and octrees discretize the state space, particle filters represent the belief state by a population of hypotheses which are updated using importance sampling. In contrast to Kalman filters [Kal60], which use a Gaussian distribution to represent the current belief state, these techniques can represent multi-modal distributions as well as uniform distributions and can therefore deal with situations where the robot has more than one likely hypothesis or is completely ignorant about its current position. Markov as well as Monte Carlo localization therefore solve not only the problem of position tracking, but also the problem of global localization.

To compute the probability of an observation o in a state s they use sensor models and an occupancy grid map of the environment [Mor89, Elf89] like the one shown in Figure 2.5. Occupancy grids divide the environment into cells of equal size and assign to each cell the probability that the cell is occupied by an obstacle. Occupancy maps can be autonomously learned as described in [TBB+98]. Figure 2.5 shows a map of the first floor of the Department of Computer Science III at the University of Bonn learned by the mobile robot RHINO using its two laser range finders.

Active Localization

Cassandra et al. [CKK96] propose to use the entropy of the belief state to decide to switch from taking actions to achieve the goal to taking actions to gain information. This idea is refined by Burgard et al. who have proposed

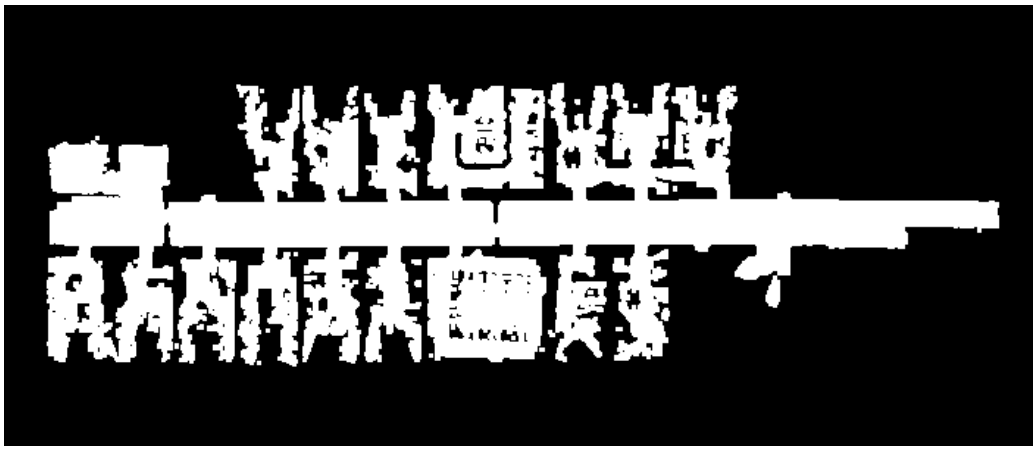

Figure 2.5: A learned occupancy grid of the first floor of the Department of Computer Science III at the University of Bonn. It has been learned by the mobile robot RHINO using two laser range finders.

a method for active localization [BFT97, BBFC98]. Whenever the entropy

$$H(b) = -\sum_{s \in S} P(s) \log P(s) \tag{2.11}$$

in the belief state distribution exceeds a threshold θ, an active localization is performed. For this purpose a set of actions in the robot centric coordinate system is considered and the action a that has minimal costs with respect to $E_a(H(b)) + \alpha c(a)$ is selected for execution. Here $E_a(H(b))$ is the expected entropy of the belief state after executing a and $c(a)$ are the expected costs associated with action a. They are computed using the following formula.

$$c(a) = \sum_{s \in S} P(s) c(a|s) \tag{2.12}$$

Here $c(a|s)$ is the cost of executing action a when the robot starts in state s. $c(a|s)$ can be computed efficiently using e.g. value iteration. To compute $c(a)$ efficiently only states s are considered with a probability $P(s) > \theta_2$.

$E_a(H(b))$ can be computed as follows.

$$\begin{aligned}
E_a[H(b)] &= -\sum_{s'\in S}\sum_{o\in O} P(s'|o,s,a)\log P(s'|o,s,a)P(o|s,a) \\
&= -\sum_{s'\in S}\sum_{o\in O} \frac{P(o|s',s,a)P(s',s,a)}{P(o,s,a)} P(o|s,a)\log P(s'|o,s,a) \\
&= -\sum_{s'\in S}\sum_{o\in O} \frac{P(o|s')P(s'|s,a)P(s,a)}{P(o|s,a)P(s,a)} P(o|s,a)\log P(s'|o,s,a) \\
&= -\sum_{s'\in S}\sum_{o\in O} P(o|s')P(s'|s,a)\log \frac{P(o|s',s,a)P(s',s,a)}{P(o,s,a)} \\
&= -\sum_{s'\in S}\sum_{o\in O} P(o|s')P(s'|s,a)\log \frac{P(o|s')P(s'|s,a)P(s,a)}{P(o|s,a)P(s,a)} \\
&= -\sum_{s'\in S}\sum_{o\in O} P(o|s')P(s'|s,a)\log \frac{P(o|s')P(s'|s,a)}{P(o|s,a)}
\end{aligned} \tag{2.13}$$

Roy and Thrun investigate a related method to account for position uncertainty while avoiding the whole complexity of POMDP planning: coastal navigation [RT99]. They augment the state space with an additional dimension which accounts for the position uncertainty. The position uncertainty is measured as the entropy of the probability distribution of the robot's pose. Path planning is done using value iteration trading-off transition costs and position uncertainty in the reward function of the underlying MDP. The approach is superior to active localization as it can smoothly interleave actions to approach the goal with actions to reduce position uncertainty. However, to be able to compute the entropy of the probability distribution of the robot's pose efficiently, they assume a uni-modal Gaussian distribution which is unrealistic and critical.

2.3 Decision-theoretic Action Selection

In the RHINO system an optimal path to the robot's goal is computed using a deterministic version of value iteration on a grid map of the environment. It takes the distance between two neighbored grid cells as well as the occupancy probability of the grid cells into account. Intermediate target points are computed from the resulting navigation policy using a simple heuristic which selects the last point on the optimal path which is still visible form the robot's current state.

In this chapter, we discuss how the selection of intermediate points can be based on learned models of the robot's behavior. These models include the average time needed by the behavioral layer to approach a given target point. As the robot might fail completely to approach a given target point, it is also necessary to predict the probability of execution failures. In the rest of this section, we describe an action selection function which trades off the expected cost of an action against the probability of a complete execution failure.

We suggest considering the problem of selecting intermediate target points as an MDP, more precisely as a stochastic shortest path problem. We assume full observabilty of the robot's state. This is justified by the same arguments as discussed above, as the action selection MDP is part of the navigation planning process which only takes place when the robot is sufficiently confident about its current position. In the other case, the robot performs an active localization before it proceeds the navigation process. We briefly describe the state space, the action space, the reward function, and the probabilistic action model to specify the action selection MDP.

The States

For the action selection MDP we consider states $s \in S$ that include the robot's position and orientation and the shortest path to the goal, $p = [p_0, p_1, ..., p_n]$. The path can be computed efficiently from the navigation policy generated by the path planning algorithm. The discretization of the path results from the discretization of the environment grid map.

The Actions

We consider as possible actions those actions that ask the local navigation system to approach one of the intermediate points on the path $p = [p_0, p_1, ..., p_n]$. The selection of actions has to consider the following trade-off: Often the choice of target points that are more distant allows the reactive navigation system to drive smoother trajectories. On the other hand, the target point should be close enough so that it is reachable using a local navigation action. The decision between short but difficult versus easy but long paths is already done by the path planning system.

The Probabilistic Action Model

In our action models, the local navigation actions have two possible outcomes: action success and action failure. We consider an action a to be unsuccessful when the robot has not reached its target point within t_{to} seconds, that is, has received a timeout.

In the success case, we assume that the robot has reached the specified target point upon action completion. Specifying the effects if the action has failed is much more difficult. We believe that in this case the exact resulting state is not critical because in the case of action failure the robot is punished with a large negative reward. Thus, in the case of action failure we can simply assume the robot to be where it started to execute action a. We denote the probability that the robot is timed out when executing action a in state s as $P(T = true|s, a) = P^+(s, a)$, and the probability that it is not timed out as $P(T = false|s, a) = P^-(s, a)$.

The Reward Function

The reward for executing action a in state s is defined as follows. For a successful action execution, the agent receives the reward $R^-(s, a) = -l(a)/v(s, a)$ where $l(a)$ denotes the length of the path to the target point of action a. The term $v(s, a)$ denotes the expected average velocity for the execution of action a in state s. For a timed out action the robot gets an immediate reward of $R^+(s, a) = -t_{to}$. In both cases the reward is an estimation of the time the robot loses when executing action a. The robot's expected immediate reward thus is given by:

$$R(s, a) = \sum_{i \in \{+,-\}} P^i(s, a) R^i(s, a)$$

The Action Selection Function

Using the Markov Model depicted above we can use the following decision theoretic action selection function:

$$a_{best} = \arg\max_{a \in A} V^*(s, a) \tag{2.14}$$

where

$$V^*(s, a) = \sum_{i \in \{+,-\}} P^i(s, a)(R^i(s, a) + V^i(s, a))$$

and $P^+(s,a)$, $P^-(s,a)$, $R^+(s,a)$, $R^-(s,a)$ are defined as above. $V^+(s,a)$ denotes the utility of being in state s and $V^-(s,a)$ the utility of being in state s' where s' is the state that results from a successful execution of action a in state s, i.e. the target state of a.

To understand Equation 2.14 recall our assumption that a successful action a leads the robot to the target of a, while an unsuccessful action leaves the robot where it started to execute a. The latter assumption biases the robot to prefer actions with a higher probability of success. We have chosen the discounting factor γ to be 1 because the problem is a finite horizon problem where the agent always reaches an absorbing state after executing a finite number of actions.

The Utility Function

So far we have not specified how $V^+(s,a)$ and $V^-(s,a)$ are computed. In principle, this could of course be done by value iteration [Bel57]. However, in this chapter we use the following approximation: Let $p = [p_0, p_1, ..., p_n]$ be the path from the robot's current position s to its destination and $l(p)$ be $\sum_{i=1}^{n} |p_{i-1}, p_i|$. Further, let v_{avg} be the robot's average velocity while performing navigation tasks. We can then approximate $V^+(s,a)$ and $V^-(s,a)$ like follows:

$$V^-(s,a) = -\frac{l(p) - l(a)}{v_{avg}}, \quad V^+(s,a) = -\frac{l(p)}{v_{avg}}$$

Here the value v_{avg} can be obtained by a simple running average.

2.4 Learning Action Models

Let us now consider how the velocity function $v : S \times A \rightarrow \mathbb{R}$, the average velocity when executing action a in state s, and the probabilistic action model $P(T|S,A)$ can be learned. To do so, we will first define a suitable feature language to describe state-action pairs (s,a), i.e. a set of observable conditions that are expected to correlate with the navigation performance.

In our learning experiments, we use the following features: (1) clearance towards the target position, (2) clearance at current position, (3) clearance at target position, (4) minimum clearance on path, (5) curvature of the planned path, (6) average clearance on path, (7) maximal minus minimal clearance on the path, and (8) relative length of the path to target position.

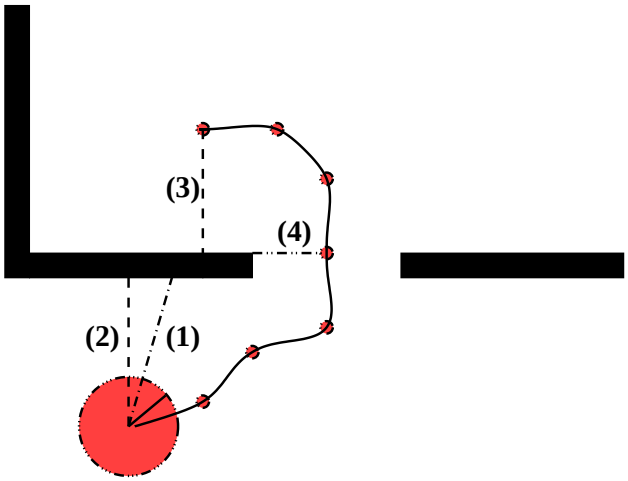

Figure 2.6: The features (1) to (4).

The robot's clearance at any position is the distance to the next obstacle in its environment. It can be computed using the robot's environment map and a ray tracing algorithm. The clearance towards the target position is the distance to the next obstacle in this direction, but relative to the Euclidian distance to the target position. For all k points on the path to the target, we compute the clearance and keep the minimal clearance, the average clearance and the difference between maximal and minimal clearance as features. Another feature is the curvature of the path to the target point a which is $l(a)$ as defined above relative to the Euclidian distance to the target point. We compute the relative length of the path towards a as $l(a)/l_{max}$ where $l_{max} = 800$ cm. Figure 2.6 illustrates the features (1) to (4) graphically. The features are relatively simple and can be computed efficiently.

Having specified the feature language we will now look at the learning mechanisms themselves. We will apply two alternative approaches: neural network learning and tree-based induction. Artificial neural networks [Bis96] are well known as general function approximators. They have therefore often been used to approximate utility and reward functions in reinforcement learning and have the advantage of realizing continuous functions. Tree based induction methods [BFOS84], on the other hand, have the advantage that they provide in addition to a classification (in case of decision trees) or a value prediction (in case of regression trees) an explanation of the results they produce. The learned tree representations are often valuable resources for human inspection and automated reasoning.

2.4.1 Neural Networks

To learn the action model $P(T|S,A)$ we have used a simple feed forward neural network with sigmoidal activation functions which was trained using an epoch back propagation algorithm. To speed up convergence we have scaled the features described above so that their scaled values lie in the interval [0,1]. Each normalized feature vector was associated with either 0 when no timeout was observed or 1 if a timeout was observed. The output of the neural network (after a sufficiently long training) is a value in the interval $[0,1]$ which can be interpreted as the probability of a timeout.

We have used the same neural network structure to learn the function v. Only the training examples differ. The output values in this case are: $p = v_{cur}/v_{max} \in [0,1]$ where v_{cur} is the robot's current, and v_{max} is the robot's maximal velocity.

2.4.2 Tree Based Induction

Decision trees [Qui93a] can be interpreted as a set of Horn clauses that are well understandable for humans. To learn the action model $P(T|S,A)$ with a decision tree we, classify each training example given by the set of features described above as *true* or as *false* depending on whether a timeout occurred or not. In the decision tree framework, a probability can be associated with each classification like this: if n is the number of training examples that are mapped to a decision tree leaf l which is associated with the classification c and m is the number of examples in this set that are classified as c, we can associate the probability $p = m/n$ with this classification (given the observed features).

To learn the function v is a regression and not a classification task. Regression trees [BFOS84] are tree based function approximators and can be applied to the task. Regression trees are similar to decision trees, but differ from them in that leaf nodes are not associated with classifications, but with real values. A regression tree like a decision tree can be translated into a set of horn clauses.

To build a regression tree, a set of training examples associated with a node in the tree is split to minimize some given impurity measure. This impurity measure often is the empirical variance of the output value of the function to learn. For example, to split a set of n examples, S, with a variance in the output value σ^2, into two sets S_1 and S_2, a split is chosen that minimizes $n_1\sigma_1^2 + n_2\sigma_2^2$ where n_i is the number of examples in S_i and σ_i^2 is the variance

in the set S_i with respect to the output value. This process is iterated recursively until some given stopping criterion is met.

We have used the following stopping criterion: For each split we have tested if the split reduces the variance significantly. If the best split does not reduce the variance significantly we stop growing the tree at that node. Whether a split reduces the variance significantly can be tested using a bootstrap test described in some detail in Appendix A. The test computes the probability that a reduction of variance higher than or equal to the observed reduction would be observed under the assumption that the variance has not changed. If this probability, the significance probability, is smaller than some given threshold value θ we say that the split reduces the variance significantly with respect to the significance level θ. The choice of θ allows us to trade-off the prediction accuracy of the tree (on the training examples) against its size. Chapter 5 provides a more detailed introduction to decision- and regression trees.

2.5 Experimental Results

In this section, we will demonstrate (1) that the action selection function defined by Equation 2.14 can be used to execute a navigation plan quickly and reliably and (2) that the necessary models can be learned autonomously. We have performed both simulator experiments and experiments on a real robot to show that the learned action selection improves the robot's performance substantially and significantly.

The setup of the experiments is as follows. After the learning phase, in which the robot has acquired the action models needed, the default methods and the learned action selection policy are compared. The default methods include one that chooses the next target point randomly between 1 and 8 meters ahead on the planned path and a second one that chooses it systematically. The methods are compared based on a sequence of k navigation tasks that is executed n times. We then test whether the learned action selection function improves the robot's performance significantly. This is done with a bootstrap t test (please refer to Appendix A for details).

2.5.1 Learning the Models

To learn the velocity function v and the timeout probability $P(T|S, A)$ we have generated some training and test data using the random action selection

as described above and the simulator. We used a set of 5279 training and 3266 test examples for the classification task (will the robot be timed out when performing a given action?) and 4203 training and 2531 test examples for the regression task (what is the robot's expected average velocity when performing an action?). This data volume corresponds to collecting data from robot runs that take about 24 hours. For the regression task we only consider examples where the robot was not timed out.

Learning the Models with Neural Networks

For the classification task as well as for the regression task we trained a neural network with 8 nodes in the input layer, 8 nodes in the hidden layer and 1 node in the output layer. We performed epoch back propagation to train both neural networks.

For the classification task we used a learning rate of 0.8 and a momentum term of 0.9. After 203590 iterations (a night) we had 88.14% of the training- and 88.77% of the test examples correctly classified.

For the regression task we used a learning rate of 0.8 and a momentum term of 0.95. After 406360 iterations (about 24 hours) we got an absolute error of 3.698 cm/s on the training- and of 3.721 cm/s on the test set where 60 cm/s is the maximal velocity of the robot.

Learning the Models with Tree Induction

We used the same data to learn a decision and a regression tree. Table 2.1 gives the statistics of the training error, the test error and the number of generated rules depending on the significance level used in the stopping criterion for the decision tree learning.

Table 2.2 gives the same statistics for the regression tree where the training and test errors are absolute errors in cm/s.

In our experiments, we have chosen the rules generated from the trees grown with a significance level of 0.05. These rules are well intelligible when inspected by a human operator. The trees are grown within about two minutes.

LEVEL	TRAINING ERROR	TEST ERROR	RULES
0.5	6.6%	13.2%	301
0.4	7.4%	12.6%	202
0.3	8.0%	12.7%	146
0.2	9.6%	12.9%	72
0.1	12.0%	14.0%	21
0.05	12.3%	14.1%	12

Table 2.1: The training error, the test error, and the number of generated rules depending on the significance level used in the stopping criterion for the decision tree learning.

LEVEL	TRAINING ERROR	TEST ERROR	RULES
0.5	2.71 cm/s	3.76 cm/s	743
0.4	3.33 cm/s	3.76 cm/s	256
0.3	3.68 cm/s	3.88 cm/s	96
0.2	3.98 cm/s	4.11 cm/s	48
0.1	4.27 cm/s	4.30 cm/s	29
0.05	4.33 cm/s	4.37 cm/s	25

Table 2.2: The absolute training and test error and the number of generated rules depending on the significance level used in the stopping criterion for the regression tree learning.

2.5.2 Simulator Experiments

In the simulator experiments, we compared the learned and the random action selection mechanism based on a sequence of 14 navigation tasks in a university office environment, which was executed 18 times. Figure 2.7 shows the set of tasks that had to be performed.

Experiment 1

In the first experiment, we compared the performance of the two action selection mechanisms, where the models are learned using neural networks. The average time needed to complete all tasks is 1380 seconds with random action selection and 1107 seconds with the learned action selection. This corresponds to a performance gain of 19.8%. The significance probability computed with a bootstrap t test is 0.0002, and the performance gain there-

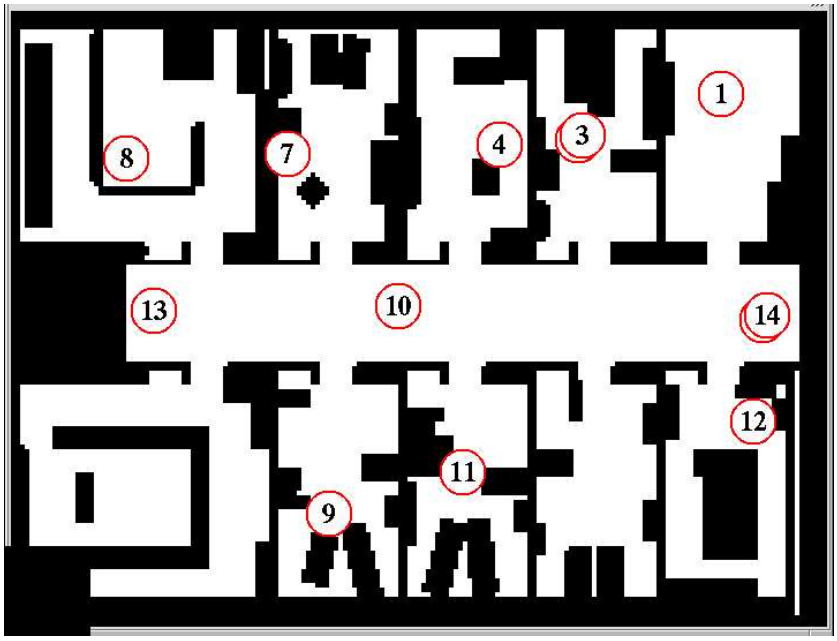

Figure 2.7: The set of target points that define the 14 navigation tasks performed in each iteration of the experiment (three of the points define two navigation tasks depending on the starting position of the robot). Initially the robot is close to target point 13.

fore clearly significant. The significance probability of a reduced variance using the learned action selection mechanism in 0.0247. Table 2.3 summarizes these results.

Experiment 2

In the second experiment, we compared the learned action selection using the tree structured models with the random action selection. With the learned action selection the robot in average needed 925 seconds to complete the whole sequence, in contrast to 1380 seconds with the random action selection. That is a reduction of 33 %. The significance probability of an improved performance is lower than 0.0001. Furthermore, the significance probability of a reduced variance in the execution time is 0.0097. The results are summarized in Table 2.4.

average time with random action selection	1380.44 s
average time with learned action selection	1107.78 s
performance gain	19.75 %
significance probability of a reduced average duration	0.0002
significance probability of a reduced variance	0.0247

Table 2.3: The results of Experiment 1.

average time with random action selection	1380.44 s
average time with learned action selection	925.39 s
performance gain	32.96 %
significance probability of a reduced average duration	< 0.0001
significance probability of a reduced variance	0.0097

Table 2.4: The results of Experiment 2.

Experiment 3

A purely random action selection seems to be a quite weak standard for the performance comparisons. However, as we will show in this experiment, this is not the case. To demonstrate this, we compared a deterministic action selection with the random one: the robot always selects as next target point the last point on the path that is still visible from its current position. This is more or less the strategy that is used in the original navigation system. Surprisingly, the performance is about 11.6 % worse with the deterministic strategy compared to the random action selection. The robot, on average, needs 1541 seconds for the 18 navigation tasks compared to the 1380 seconds with the random action selection. Therefore, the random action selection is significantly better than the deterministic one (with respect to a significance level of 0.001). However, the standard deviation is significantly smaller, 115.2 seconds compared to 237.9 seconds, with respect to the same significance level.

2.5.3 Experiments with the Mobile Robot

In the experiments performed on the mobile robot, we test how well the models learned in simulation generalize when tasks different from those considered in the training phase have to be performed by a real robot. This is an interesting question because it is unrealistic to require state-of-the-art

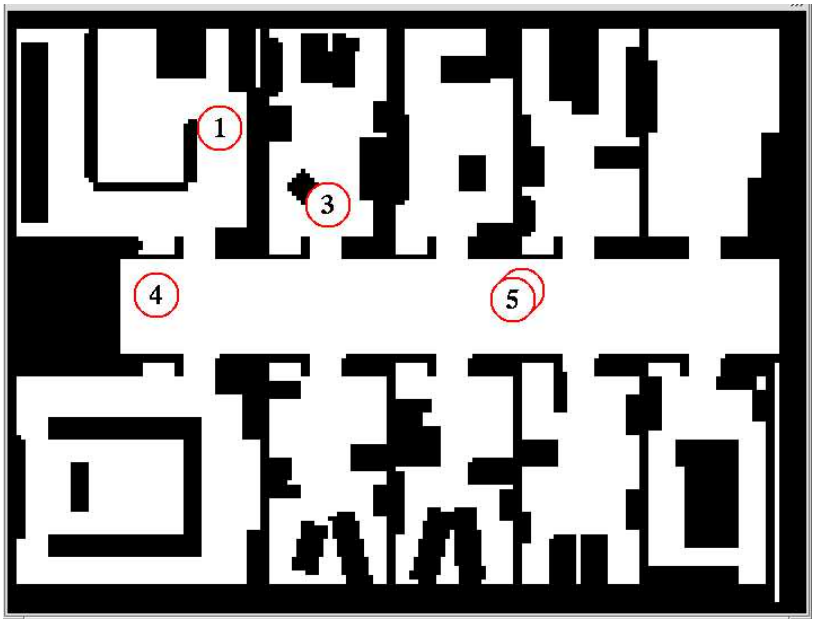

Figure 2.8: The set of target points that define the 5 navigation tasks performed in each iteration of the experiment with the real robot. The robot starts near target point 4.

research platforms to perform the time-consuming learning without keeping them under surveillance. Therefore, it should be possible to learn at least part of the models in simulation.

The robot was to execute a sequence of 5 navigation tasks 18 times both with random action selection and with the informed action selection described above. The experiments were carried out in a populated university office environment. Figure 2.8 shows the environment and the tasks that have to be performed within that environment.

Experiment 4

When using neural networks to learn the models, the robot needed 362 seconds in average to perform the five tasks, opposed to 420 seconds with the random action selection. This is a performance gain of 13.8%. The significance probability of an improved performance when using the action selection function defined in Equation 2.14 is 0.0002. The significance probability of a reduced standard deviation is 0.0541. The robot's performance can be summarized by the descriptive statistics shown in Table 2.5.

average time with random action selection	420.06 s
average time with learned action selection	362.00 s
performance gain	13.82 %
significance probability of reduced average duration	0.0002
significance probability of reduced variance	0.0541

Table 2.5: The results of Experiment 4.

average time with random action selection	420.06 s
average time with learned action selection	306.17 s
performance gain	27.11 %
significance probability of a reduced average duration	< 0.0001
significance probability of a reduced variance	0.0097

Table 2.6: The results of Experiment 5.

Experiment 5

In the last experiment, we have compared the performance of the robot using Equation 2.14 and decision/regression trees for the models with its performance using random action selecticoon: The average time needed to execute the sequence of five navigation tasks is 306 seconds with the learned models as opposed to 420 seconds with the random action selection. This is a reduction of 27.1%. The significance probability of an improved performance is lower than 0.0001 and the significance probability of a reduced variance is 0.0097. The statistics of this experiment are summarized in Table 2.6.

Summary of the Experimental Results

The experiments show that the action selection function defined by Equation 2.14 can be used to execute a navigation plan effectively and reliably and that the necessary models can be autonomously learned by the robot. As shown in the experiments 3 and 4 we can use the simulator to learn an action selection function that performs well on a real robot. Furthermore, it was demonstrated that the learned policy generalizes well to tasks that have not been used for the training. In the domain, tree structured models perform slightly better than neural networks. However, these results do not support a general comparison of the two learning methods as we have used fairly simple implementation of both methods.

2.6 Summary

In this chapter, we have given a short overview of the RHINO system. Its modular, hybrid design and the consequent application of decision-theoretic and probabilistic algorithms for navigation planning, state estimation and active localization make it a robust mobile robot control system which has proven reliable and performant in two long-run experiments.

One of the deficiencies of the RHINO control architecture is the plan execution. This can be demonstrated for the selection of local target points that are used to parameterize the collision avoidance behavior. We have suggested to model the selection of the next intermediate target point as a stochastic path planning problem which is a special type of MDP. Starting from these considerations, an action selection function has been designed that trades off the expected costs for executing an action and the probability of execution failures.

The approach presented is limited in two respects. First, the lookahead is restricted to one action. The values $V^-(s,a)$ and $V^+(s,a)$ are not computed based on forward projection, but estimated heuristically. Chapter 6 shows how to apply value iteration to compute an optimal action selection policy.

Second, we have no accurate model of the robot's new state in case of an execution failure. We assume that in this case, the robot is still where it started to execute the action. It is very difficult to come up with a better model, even if this model is to be learned. One way to deal with this problem is to eliminate the occurrence of complete execution failures altogether. In Chapter 3 we describe how the robustness of the approach point behavior can be improved considerably using local planning techniques for trajectory evaluation. Chapter 4 describes a mechanism to handle execution failures during plan execution. Both techniques help to eliminate probabilistic uncertainty and transform it into metric uncertainty.

Despite these limitations, the performance of the mobile robot RHINO could be improved considerably in the experiments. This is probably due to the learned action models, both for predicting execution failures as well as predicting the average time needed to execute an action. Neural network learning as well as tree-based induction have been demonstrated to achieve substantial performance improvements. For the rest of the thesis, we will use tree-based induction methods, because they can be transformed into symbolic rules and are therefore well understandable for humans. In Chapter 7 we discuss how decision trees can be used for learning action selection rules.

Chapter 5 introduces model trees which generalize regression trees. Their

leafs can contain general linear models and therefore approximate piecewise-linear functions. We also elaborate on the idea to compute a suitable feature language from the shortest path to the goal. These features can be used for attribute tests as well as for prediction functions associated with the tree leafs. Chapter 5 describes an application of model tree learning to the problem of sequencing navigation tasks. In Chapter 6, we come back to the problem of action selection – this time in the context of HTN planning.

The experimental results demonstrate that plan execution is an interesting problem and a good opportunity to improve a mobile robot's navigation performance substantially. MDP-based action selection using learned models of the robot's behavior is a promising approach to overcome these shortcomings. The approach used in the experiments, however, is also limited in various respects. The following chapters will discuss how the limitations can be overcome.

Chapter 3

Local Planning for Collision Avoidance

This chapter deals with the reactive layer of APPEAL which is concerned with the task of approaching given target points while avoiding obstacles and controlling the robot's velocities in a way that takes the robot's dynamic constraints into account. In the RHINO architecture, this is achieved by the *Dynamic Window Approach* which performs a local search in the space of possible control commands. Control commands are evaluated based on a projection of the execution of the control commands using a heuristic evaluation function.

We discuss how the the Dynamic Window Approach can be improved using evaluation functions which are computed by a local planning process. While the search for control commands allows to take the robot's dynamics into account and therefore allows for very smooth navigation behavior, planning is able to determine the optimal path towards the goal and minimizes the likelihood that the robot gets trapped in dead-end situations.

The experiments show that the application of local planning improves the reliability of the behavioral layer considerably. We also introduce different evaluation functions that employ the utility function computed in the planning step to evaluate control commands and compare their performance in extensive experiments. In these experiments, we find one evaluation function to significantly outperform the other evaluation functions, especially the one that models a path following algorithm.

3.1 Introduction

Collision avoidance is a crucial part for all autonomous mobile robots operating in populated and dynamic environments. In order to navigate safely, they must be able to react to unforeseen obstacles like humans crossing their paths. For this reason, virtually all navigation systems comprise some kind of collision avoidance component that controls the robot's motion.

In general, it is also desired that the robot reaches its destination fast. It should therefore take the robot's dynamics into account and move the robot towards the target location most effectively. The optimal approach would be to do an exhaustive search for the sequence of control commands that achieves the shortest time to target, or more generally minimizes a given cost function. However, this approach is computationally expensive and not suited to issue safe control commands at high frequency. An exhaustive search in the space of control commands is therefore not feasible in order to be able to navigate at high speeds.

Many collision avoidance systems use purely reactive approaches that allow for high update rates. They decide on the next control command solely based on different heuristics. However, as no planning is involved, these approaches risk that the robot gets trapped in dead-end situations like for example U-shaped obstacle configurations.

In this chapter, we present a collision avoidance approach that uses local path planning followed by a search for the best next motor control command given the current plan. Local path planning finds the shortest path within a map built from the robot's latest range measurements, and can be carried out much faster than a search for the optimal sequence of control commands. That way, we reduce the risk of getting stuck in a dead-end situation and still achieve the high update rates required for fast navigation.

The crucial aspect of this hybrid reactive approach of course is, how to decide on the control commands based on a plan. A simple approach would be to stick to the planned path as close as possible. However, as path planning does not respect the robot's dynamics, this can result in poor performance. In this work, we consider path planning systems that assign a utility value to each state in the state space. For this purpose, we formalize the navigation problem as a Markov Decision Process which allows us to use dynamic programming algorithms to efficiently compute optimal utility functions for the decision problem.

We introduce different evaluation functions that employ the utility function computed in the planning step to evaluate control commands. We compare

the performance of these functions in extensive experiments carried out with a differential drive robot within our office environment. In these experiments, we find one evaluation function to significantly outperform the other evaluation functions, especially the one that models a path following algorithm.

The algorithm presented here is an extension of the Dynamic Window Approach to collision avoidance [FBT97]. We show that the algorithm outperforms this approach with respect to reliability. This is especially important when trying to learn models of the behavior of the local layer to improve the overall navigation behavior as depicted in Chapters 5-7.

The remainder of this chapter is organized as follows. After discussing related work in Section 3.2, we describe how local navigation planning can be applied to evaluate control commands in Section 3.3. Section 3.4 describes in detail how the evaluation of the performance of control commands is based on a utility function computed in the planning step and we introduce the different evaluation functions we have developed. In Section 3.5, we give results on the performance of our collision avoidance method using the different evaluation functions. Section 3.6 concludes.

3.2 Related Work

Most existing collision avoidance methods are purely reactive in the sense that they search for safe robot control commands based on the robot's current proximity sensor data without any projection of the robot's future state. These methods differ in the way this search is carried out. The Vector Field Histogram method [BK91] for example decides on the next movement direction and the speed of the robot based on an angular histogram, which describes the density of obstacles in the surrounding of the robot. Potential field methods [Lat91] achieve collision avoidance behavior by simulating repulsive forces exerted from obstacles and an attractive force towards the target. Both approaches do not explicitly take the constraints imposed by the dynamics of the robot into account. Koren and Borenstein [KB91] identify two major problems of potential field approaches: They often fail to find trajectories between closely spaced obstacles and can produce oscillatory behavior in narrow passages.

Behavior-based navigation systems are composed of at least two behaviors. An approach target behavior and a collision avoidance behavior. While arbitration schemes based on voting are often not able to take dynamic constraints into account, a decision-theoretic arbitration scheme recently pro-

posed by Rosenblatt [Ros00] is.

Another popular method, which is more closely related to our approach is the Dynamic Window Approach to collision avoidance [FBT97, Sim96, Sch98]. This method searches for the trajectory the robot should take within the next time step based on a local map of the robot's surrounding built from the latest sensor measurements. A trajectory is specified by the translational and rotational velocity of the robot and the search is carried out in this velocity space. In order to reduce the search space, the robot's dynamic constraints are taken into account by considering only velocities that can be reached within the next time interval. The approach decides on the best next velocities based on a linear evaluation function which weighs clearance, heading towards the target and speed. Our method adopts this search method, but uses new evaluation functions based on the computation of optimal utility functions from a local map.

The major disadvantage of all purely reactive approaches is that they might not reach the target although a path to the target exists. Therefore, more recently, methods have been developed, that use local planning to overcome this problem. Ulrich and Borenstein [UB00] describe an extension of the Vector Field Histogram method which carries out A-star search on a local map in order to obtain the best sequence of movement directions towards the target location. Konolige [Kon00] uses dynamic programming on the local map to compute the gradient towards the target. To make this approach computationally feasible, only the two dimensional space of possible robot positions is considered for planning. This results in optimal paths with respect to some cost function but does not allow to model the robot's dynamics correctly. In this paper, we suggest a method that simultaneously plans optimal collision-free paths and takes the dynamics of the robot into account. Like Konolige we use a path planner based on dynamic programming to compute a utility value for each state in the state space, but rather than computing an optimal path from the utility function using gradient ascent, we directly use the utility function to evaluate control commands. As our experiments show, our method is able to produce a smoother navigation behavior than a path following algorithm, because it makes a better use of the computed utility function.

Brock and Khatib [BK99] developed a similar method for holonomic mobile robots. They compute a navigation function which labels each cell in the local grid map with the L^1 distance to the goal. They replace the term for the target heading and the clearance term in the evaluation function of the dynamic window approach by two features derived from that navigation

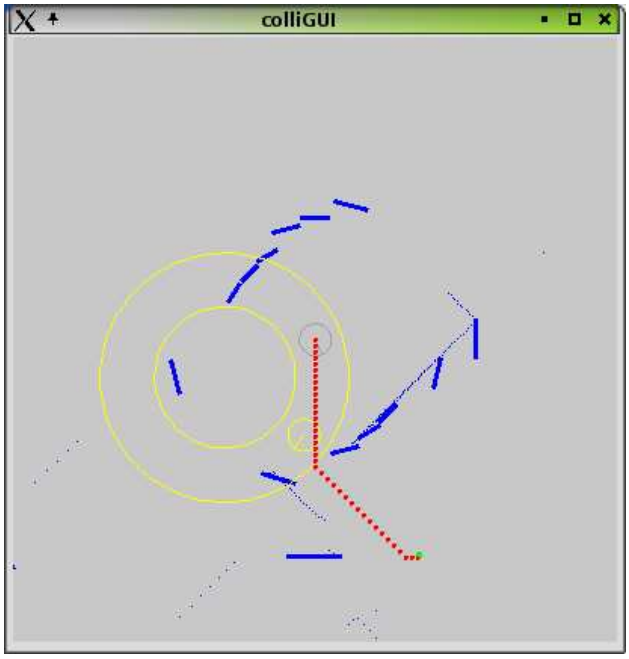

Figure 3.1: An obstacle field constructed from the robot's current sensor readings.

function. Although the computed navigation function is guaranteed to have no local optima, the combined evaluation function is not. In our approach, however, the evaluation function is directly derived from the utility function and thus guaranteed to have no local optima.

3.3 Computing Optimal Utility Functions

In this section, we discuss how navigation problems can be formalized as Markov Decision Problems. Optimal utility functions for these navigation problems can be computed efficiently using dynamic programming algorithms.

As discussed in Section 2.2.1 an MDP is given by

- a set of states S,
- a set of actions A,
- a probabilistic action model $P(S|S, A)$ and

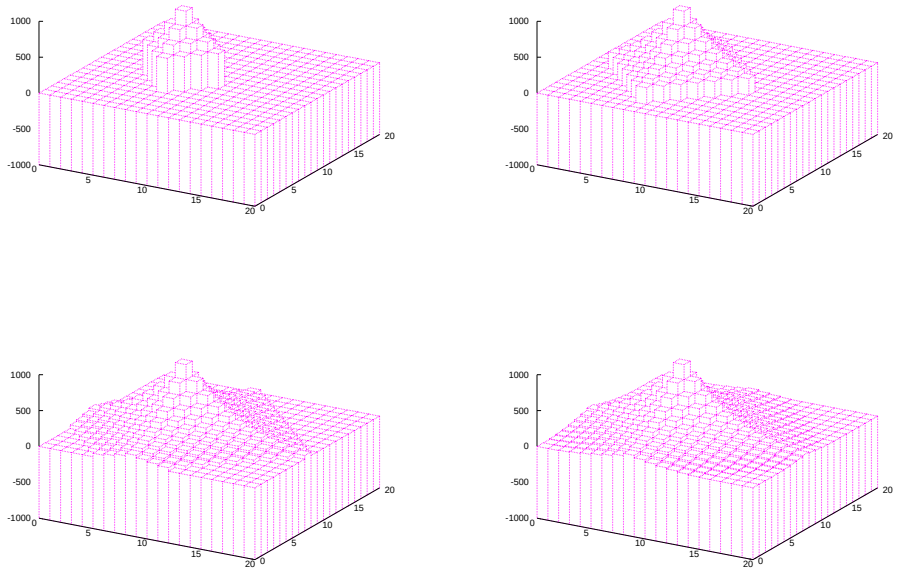

Figure 3.2: The diffusion process for the wavefront algorithm.

- a reward function $R : S \times A \to \mathbb{R}$.

We show in this section how different state and action spaces as well as reward functions result in different utility functions. The next section demonstrates how these utility functions can be used to evaluate local control commands and Section 3.5 compares the performance of the resulting evaluation functions.

We compute the state space of the first utility function which we will call *distance-based utility function 2D* (DUF2D) by discretizing the two-dimensional obstacle field of the robot built from its last sensor readings into quadratic cells. Figure 3.1 shows an example of an obstacle field. In our implementation we used a map of $14\,\mathrm{m}^2$ and a resolution of $10\,\mathrm{cm}^2$. The obstacles are thickened by the robot's radius which allows to treat the robot as a point when computing the utility function.

The possible actions in this case are translations to each of the eight neigh-

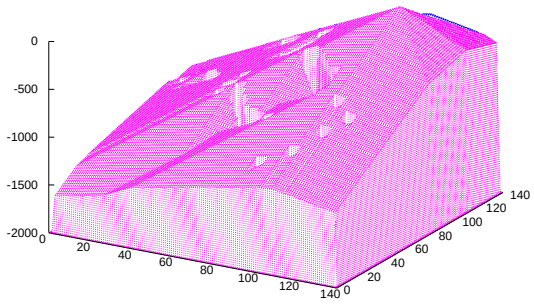

Figure 3.3: The value function computed for the navigation task depicted in Figure 3.1.

boring grid cells. The reward for an action a in state s is

$$r(s,a) = \begin{cases} -2000 & \text{if } s \text{ contains an obstacle} \\ -\text{cost}(a) & \text{otherwise} \end{cases} \tag{3.1}$$

Here, $\text{cost}(a) = \text{dist}(m(s), m(\text{succ}(s,a)))$ where $s(s,a)$ denotes the successor state of s when executing action a, $m(s)$ is the midpoint of the grid cell s and $\text{dist}(p,p')$ denotes the Euclidian distance between points p and p'. The state containing the target point is modeled as an absorbing state.

For efficiency reasons, we assume a deterministic action model where the intended state transition is assumed to always succeed. As we use a deterministic action model, we can use Dijkstra's algorithm instead of value iteration to compute an optimal utility function V^*. For grid-like state spaces the *wave front algorithm* [TBB+98, Kon00] is even more efficient. The algorithm also applies dynamic programming, but is more efficient than value iteration. It only updates the utility values for the states on the fringe of a wave diffusing from the target state(s). Figure 3.2 illustrates the diffusion process in the wavefront algorithm. Figure 3.3 shows the value function computed for the navigation task depicted in Figure 3.1. The path shown there results from a hill climbing search with respect to the value function.

An obvious extension of the MDP considered so far is to make the reward for an action a taken in state s not only dependent on the cost for executing action a, cost(a), but on the cost of being in state s, cost(a).

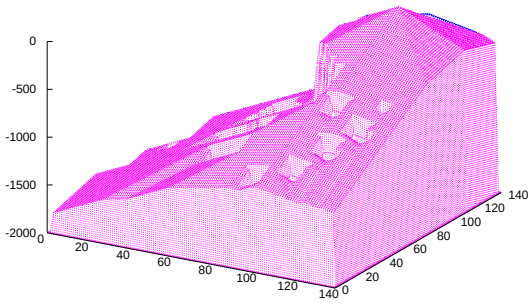

Figure 3.4:

$$r(s,a) = \begin{cases} -2000 & \text{if } s \text{ contains an obstacle} \\ -\text{cost}(a) - \text{cost}(s) & \text{otherwise} \end{cases} \tag{3.2}$$

In the navigation domain, it is reasonable to make $\text{cost}(s)$ dependent on the clearance of the robot in state s. In our implementation, we have used the following formula: $\text{cost}(s) = -\alpha \max(0, \theta_{cl} - \text{clearance}(s))$ where θ_{cl} is a threshold value specifying when the robot has enough clearance. In the experiments, we have used the parameters $\alpha = 10$ and $\theta_{\text{cl}} = 60$. Because of the deterministic action model, we can again compute an optimal value function for the MDP using the wave front algorithm. We call the resulting utility function *distance- and clearance-based utility function* (DCUF). Figure 3.4 shows the DCUF computed for the navigation task shown in Figure 3.1 using the values $\alpha = 20$ and $\theta_{\text{cl}} = 40$. Figure 3.5 shows that the optimal path to the target computed from DCUF significantly deviates from the one computed for DUF2D.

One problem with the utility functions developed so far is that they do not support turns on the spot. As the state space does not contain the robot's orientation, we cannot model that in some situations it is advantageous for the robot to turn to the target before starting to approach it. To be able to do so, we have to introduce an additional dimension to the state space: the robot's orientation. To keep things feasible, the orientation has to be discretized. We consider four discrete orientations and the three actions: forward translation, left turn and right turn. We call the resulting evaluation function *distance-based utility function 3D* (DUF3D).

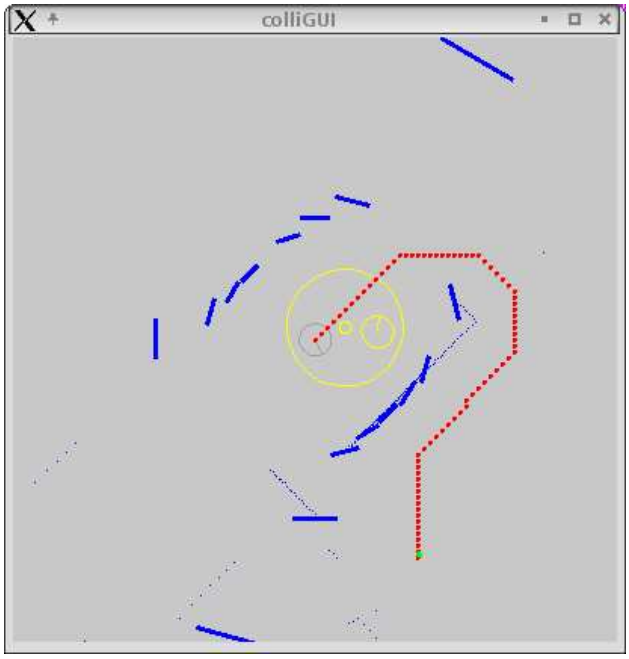

Figure 3.5: The optimal path to the target according to DCUF for the situation shown in Figure 3.1.

3.4 Evaluation Functions for Control Commands

In this section, we describe how we search for admissible control commands which in the case of synchro-drive or differential-drive robots, can be approximated by circular trajectories. In the second part of this section, we examine how these trajectories can be evaluated using the utility functions described in the previous section.

3.4.1 Searching for Control Commands

When using synchro-drive or differential-drive robots a control command is given by a pair (v_o, ω_o), the target translational- and rotational velocity of the robot. When we assume that the robot immediately reaches the new velocities and therefore ignore boundaries on the maximal positive and negative accelerations, we can model these control commands as simple trajectories: (v_o, ω_o) corresponds either to a circular trajectory (if $v_o > 0 \wedge \omega_o \neq 0$), to a

straight line trajectory (if $v_o > 0 \wedge \omega_o = 0$) or to a rotation on the spot (if $v_o = 0 \wedge \omega_o \neq 0$). As Fox et al. show [FBT97] the error we make under this assumption when predicting the position (x, y) of the robot after Δt seconds is bounded by

$$\text{err} = \sqrt{2(\Delta v)^2(\Delta t)^2} = \sqrt{2}\Delta v \Delta t \tag{3.3}$$

where $\Delta v = |v_t - v_{t+\Delta t}|$. We can account for this error by thickening the robot and by keeping the time between two successive control commands small.

If we model control commands and their effects using these simple trajectories, cutting algorithms can be used to check whether a trajectory is admissible, that is whether the robot can stop safely before the next obstacle on the trajectory. It is also straight forward to project the robot state $(x', y', \theta', v', \omega')$ after $\Delta t'$ seconds given an initial state $(x, y, \theta, v, \omega)$ and the trajectory (v_o, ω_o). Assuming that (v_o, ω_o) can be reached in Δt seconds, the projection error is bounded as before.

To select a suitable control command, we have to search for admissible control commands in the space of all possible velocity combinations and then select the one with the highest evaluation with respect to some given evaluation function. To make this search feasible, only the small window of this space is considered that is given by the dynamic constraints of the robot, namely its maximal velocities and its maximal (positive and negative) translational and rotational accelerations together with its current velocity. Given a small constant time Δt, the time the robot needs for one iteration of the algorithm, these constraints limit the possible velocity combinations (v, ω) the robot can reach within this time window. This method to prune the search space for potential control commands has first been proposed by Fox et al. [FBT97] and is called *Dynamic Window Approach*.

3.4.2 Evaluating Control Commands

Under all the admissible trajectories in the dynamic window, one has to be selected according to an evaluation function G. Fox et al. [FBT97] propose to use the class of functions

$$G(v, \omega) = \sigma\left(\alpha \text{head}(v, \omega) + \beta\, \text{dist}(v, \omega) + \gamma\, \text{vel}(v, \omega)\right) \tag{3.4}$$

Here $\text{head}(v, \omega)$ is $180 - \angle(v, \omega)$ where $\angle(v, \omega)$ is the angle to the target after executing action (v, ω) for Δt seconds, $\text{dist}(v, \omega)$ is the distance to the closest obstacle on the trajectory and $\text{vel}(v, \omega)$ is a projection on the translational velocity v. The function σ is a smoothing function intended to increase

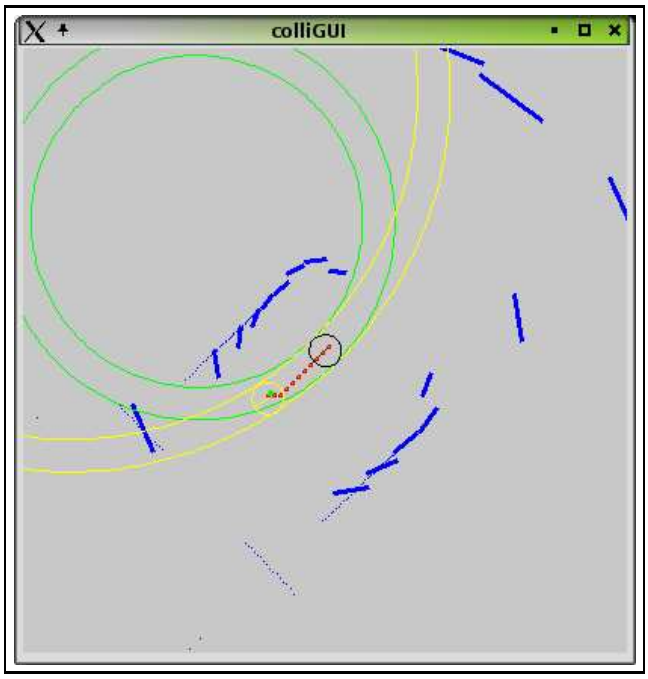

Figure 3.6: Evaluating trajectories by local projection.

the side-clearance of the robot. In the evaluation function, $\text{head}(v, \omega)$ and $\text{vel}(v, \omega)$ take the role of progress estimators, while $\text{dist}(v, \omega)$ accounts for the future utility of a trajectory. In the following, we will call this class of evaluation functions *weighted-sum-based evaluation functions* (WSEF).

This class of evaluation functions, however, might fail to guide the robot towards its target. The robot might for example get stuck in situations like U-shaped obstacle configurations. In the following we will discuss how the utility functions described in Section 3.3 can be used to evaluate trajectories and to avoid this problem.

To evaluate trajectories we project the state $(x', y', \theta', v', \omega')$ after $\Delta t'$ seconds given the initial state $(x, y, \theta, v, \omega)$ and the trajectory (v_o, ω_o). For the two-dimensional utility functions, we map (x', y') to a state s in the MDP's state space and assign the utility $v(s)$ to (v_o, ω_o) where $v(s)$ is computed from V^* by distance-weighted linear interpolation using s and all eight neighbors of s. For the three-dimensional utility function, we do not perform an interpolation. The projected pose (x', y', θ') is mapped to the corresponding state s in the three-dimensional state space and directly assign the utility $V^*(s)$ to the trajectory.

We call the evaluation functions based on DUF2D, and DUF3D, *distance-based evaluation function 2D* (DEF2D) and *distance-based evaluation function 3D* (DEF3D) respectively. Please note that with DEF2D the robot will never consider turns on the spot as they never increase the evaluation function. DEF3D, in contrast, does support turns on the spot.

When using DCUF to evaluate trajectories, the evaluation of a trajectory should not only depend on the projected state after $\Delta t'$ seconds, but on the projected states after $\frac{1}{k}\Delta t', \frac{2}{k}\Delta t', ..., \frac{k}{k}\Delta t'$. We obtain the evaluation as the average evaluation of all the projected states at these times. We call this evaluation function the *distance- and clearance-based evaluation function* (DCEF).

For our experiments, we have developed a fifth evaluation function: the *path following evaluation function* (PFEF). It takes the projected position (x', y') of the robot after $\Delta t'$ seconds given trajectory (v_o, ω_o) and assigns $-\infty$ to the trajectory if the closest point on the shortest path to the goal is more than $\max_{\text{dec}}$ cm (in the experiments $\max_{\text{dec}} = 50.0$) away from (x', y') and DEF2D (v_o, ω_o) in any other case. This evaluation function simulates the computation of robot control commands from a planned path to the goal.

Figure 3.7 shows the evaluation functions (a) WSEF, (b) DEF2D, (c) DEF3D, and (d) DCEF for the situations shown in Figure 3.6. Although in this simple case all the evaluation functions agree that fast straight motion is the best thing to do, the WSEF is much more jagged than the other functions. Sharp turns to the left or to the right seem to be similarly attractive actions as fast straight motion.

In the experiments, we have used a lookahead time of $\Delta t' = 5\,s$. The value of this parameter is crucial for the performance of the robot. The relatively high value ensures that the robot slows down smoothly as it approaches the target position. Ideally, the robot would look ahead for a smaller time period, but would search for a sequence of trajectories rather than a single trajectory. However, this search process results in an explosion of the search space and is in general not feasible. Stachniss and Burgard [SB02] discuss heuristics and pruning strategies to make an A*-like search for sequences of trajectories possible.

3.5 Experimental Results

In this section, we describe two real robot experiments. The first one compares the reliability of WSEF and DEF2D in terms of goal achievement.

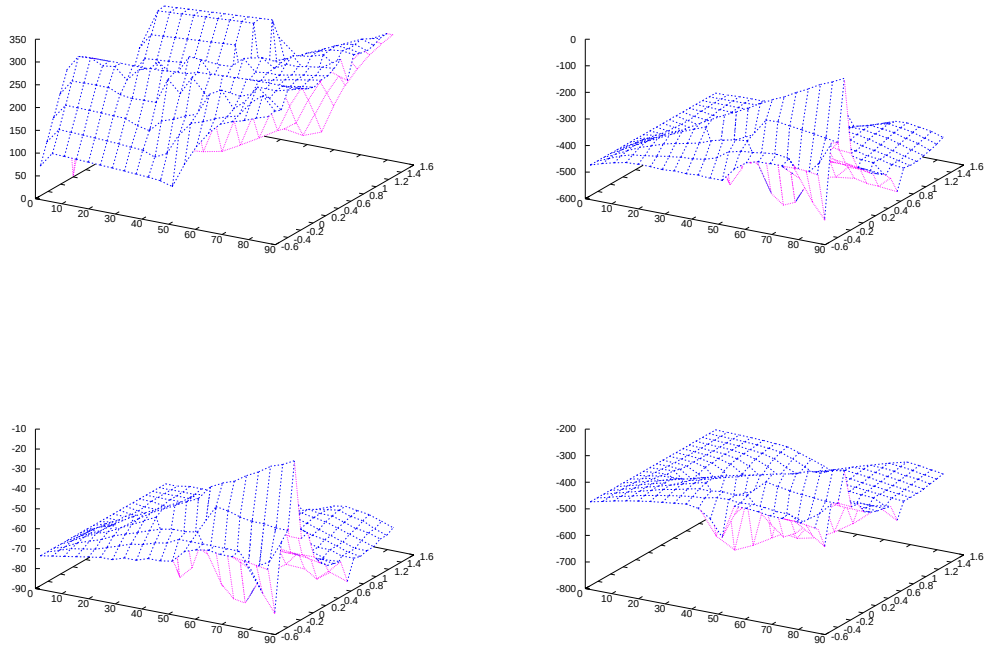

Figure 3.7: The evaluation functions (a) WSEF, (b) DEF2D, (c) DEF3D, and (d) DCEF for the situation shown in Figure 3.6.

The second one compares the performance of the four evaluation functions DEF2D, DEF3D, DCEF, and PFEF. Both experiments have been carried out using a pioneer II platform equipped with a laser range finder as shown in Figure 3.8.

3.5.1 Experiment 1

In the first experiment, the robot repeatedly executes planned paths between the four goal positions shown in Figure 3.9. The path planner generates intermediate goal points 2 m ahead on the path until the goal point itself is within 2 m distance. In case that the robot finds no admissible trajectory, the planner tries three methods to handle the exception. 1.) The robot turns to the target before it continues to approach it. 2.) The robot moves backward a bit and then tries to approach the target again. 3.) The robot turns to free space, moves forward 50 cm, turns back to the target and continues to

Figure 3.8: The Pioneer II platform

evaluation function	# failures	# tasks	failure rate
WSEF	37	101	36.63
DEF2D	2	107	1.87

Table 3.1: The number of execution failures for the two evaluation functions WSEF, and DEF2D.

approach it. If all three handlings are not successful, the robot tries them again with the target point only 1 m ahead on the path. If the three handlings fail again, the robot gives up and starts to execute the next task. The exception handling is not built into the behavioral layer, but implemented using the HTN planner introduced in Chapter 4.

As you can see from table 3.1 using WSEF the robot had a failure rate of 36.63 % where the robot with DEF2D had a failure rate of only 1.87 %. In the latter case, at least one of the failures was caused by a human teasing the robot by repeatedly blocking its way. With WSEF the failure rate was with over 60 % particularly high for the second navigation task ($2 \rightarrow 3$). For this task, WSEF is often not able to find the door opening.

The experiment clearly demonstrates that DEF2D is significantly more reliable than WSEF for difficult navigation tasks like entering or leaving a room. Please note that the experiment is designed to compare two evalua-

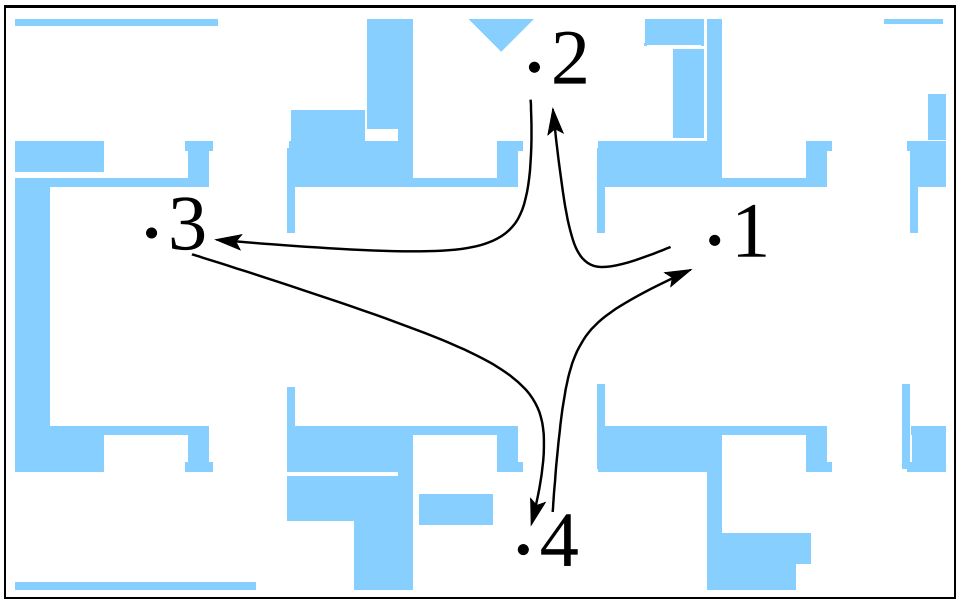

Figure 3.9: The experimental setup.

tion functions rather than two systems. It does not make any claims about the performance of the original RHINO system in comparison to APPEAL.

3.5.2 Experiment 2

In the second experiment, the path planner randomly generates intermediate goal points for the collision avoidance which are between 1 m and 5 m ahead on the planned path. Note that this is a quite challenging setup to test a local navigation algorithm as the goal points tend to be quite far away. We draw the distance to the next target point on the path randomly to ensure that the robot faces many different situations. Please also note that we performed our experiments in a populated university office where humans cross the robot's way occasionally. The sampling of target points introduces many variance in the robot's behavior. Therefore each course of the experiment consisting of the four navigation tasks shown in Figure 3.9 was repeated 12 times. Figure 3.10 shows the average time it took the collision avoidance system to complete the course using the four evaluation functions together with the 95% confidence interval of the mean.

As we can see from Figure 3.10, DCEF performs significantly better than all other evaluation functions. For example, it is 15% better than DEF2D,

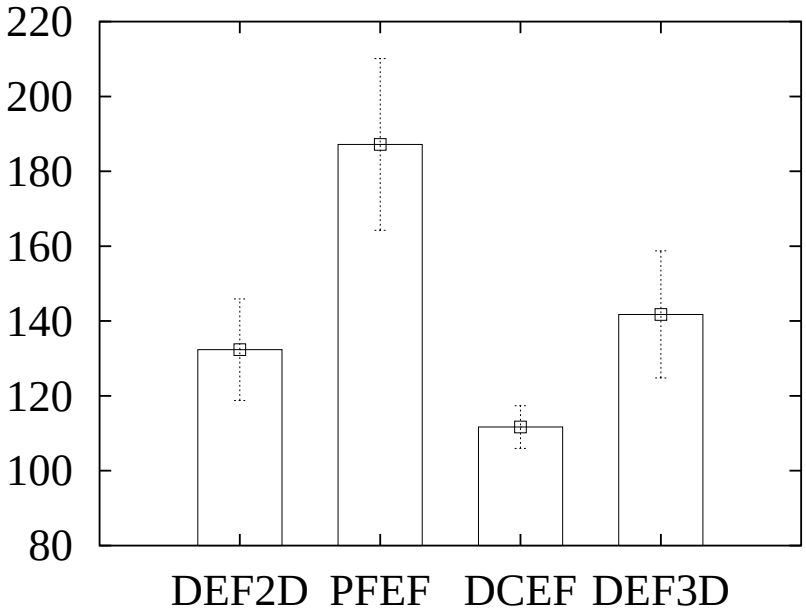

Figure 3.10: The average time it took the collision avoidance module to complete a sequence of four navigation tasks using different evaluation functions. The bars visualize the 95% confidence intervals of the mean.

the second best evaluation function. DEF2D and DEF3D do not differ significantly in the experiment. PFEF is by far the worst evaluation function and performs significantly worse than the other three functions.

It is not surprising that the robot performs better using DCEF than using DEF2D alone because DCEF takes the side clearance of the robot as well as the progress towards the target into account. In another experiment, we drove the robot up and down the corridor of our department building with a speed of up to 100 cm/s where the corridor is blocked by two unknown obstacles. In Figure 3.11 you can see typical traces produced by these two evaluation functions during this experiment. As illustrated in the figure, DEF2D tends to come closer to obstacles and then has to reduce its velocity. In the figure the robot's current velocity is indicated by the width of the light grey background of the robot's trajectory. In additional simulator experiments using the same setting, DCEF on average required 11.4% less time for completing a sequence of four navigation tasks than DEF2D.

Surprisingly, it turned out that DEF3D does not perform significantly better

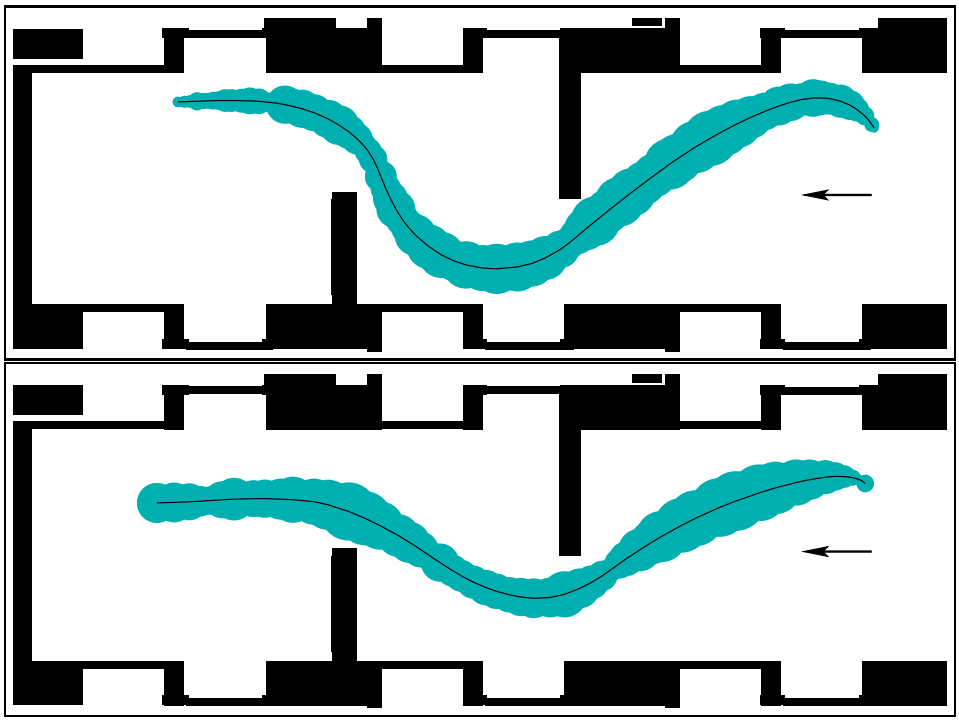

Figure 3.11: Paths produced by DEF2D and DCEF. The plotted positions are estimated by the localization component.

and in fact even slightly worse than DEF2D. However, this can be explained by the very coarse model used, and in addition, the evaluation of this function takes more than 1 second per iteration using our current implementation rather than about 0.25 seconds with DEF2D. As the robot adapts to these long update times, it is in general still possible to navigate reliably, but the robot drives more carefully. However, for high speed navigation in the hallway the update interval for DEF3D is much too large and results in unreliable navigation behavior. Here DEF2D clearly outperforms DEF3D, although not for principal reasons but due to the speed limitations of current computers.

PFEF performs significantly worse than the other evaluation functions. This is caused by the type of map used for planning. We use a local map of the robot's environment that is built from one sensor reading only. Due to occlusion, the map can change considerably during a short time period resulting in a new shortest path which differs drastically from the previous one. This can lead to unstable navigation behavior when using PFEF. Figure 3.12 demonstrates this effect. After a small rotation of the robot the shortest path to the target point has changed completely.

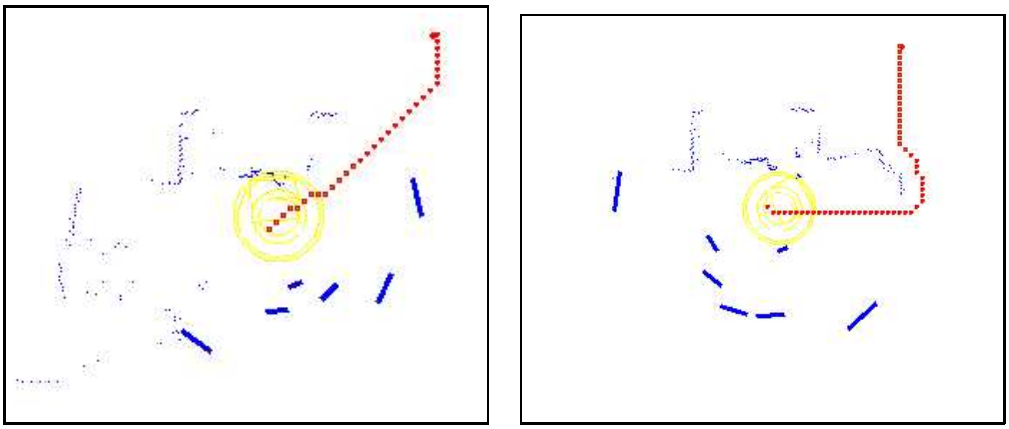

Figure 3.12: The instability of paths.

3.6 Summary

In this chapter, we have presented an approach to collision avoidance for mobile robots that uses local path planning within a map built from the robot's latest range measurements followed by a search for control commands to steer the robot towards the goal safely and efficiently. While the search for control commands allows to take the robot's dynamics into account and therefore allows for very smooth navigation behavior, planning is able to determine the optimal path towards the goal and minimizes the likelihood that the robot gets trapped in dead end situations.

In extensive real robot experiments, we have compared the distance-based evaluation function with the weighted-sum-based evaluation function and found the former to significantly outperform the latter in terms of reliability. This result is important when trying to learn models of the behavior of the behavioral layer to improve the overall navigation behavior as depicted in Chapters 5–7. The rare cases where DEF2D does not find an admissible trajectory, are in general not due to local maxima of the evaluation function, but to unexpected obstacles. We will discuss in Chapter 4 how these execution failures can be handled in the execution layer.

We have also compared four approaches to utilize the results of path planning in the search for good control commands and show that the navigation performance achieved with the *distance- and clearance-based evaluation function* is significantly better than the performance achieved by the other three evaluation functions, especially the one that models a path following behavior.

Chapter 4

HTNs for Plan Execution

Most state-of-the-art systems for mobile robot control comprise a deliberative and a reactive layer. The deliberative layer computes mission- or task plans based on a static or slowly changing world model and abstract models of the robot's possible actions. In contrast to the deliberative layer, the reactive layer performs fast feed-back control and is thus able to act in highly dynamic environments and react to unforeseen changes in the robot's surroundings.

One of the problems of mediating between the two layers is the problem of plan execution. Abstract tasks have to be decomposed in terms of behaviors that can be executed by the reactive system. Plan execution also has to deal with execution failures, e.g., due to dynamic obstacles in the robot's environment or out-dated environment models. In three-tiered robot control architectures the plan execution constitutes an own layer.

This chapter introduces the execution layer of APPEAL. We show how *Hierarchical Task Networks* (HTNs) can be used for the transparent representation of the state of the execution process. In contrast to classical HTN planning, tasks are considered as objects that can perform considerable computation like, e.g., path planning on initialization. This helps to keep the task decomposition process simple. The explicit representation of choice points in the plan execution process helps to handle unexpected failures on an adequate level of abstraction and to reason about plan improvements. We demonstrate these benefits of the HTN representation for plan execution in three real world experiments.

4.1 Introduction

Abstract plan- or task layers have been used in robot control since SHAKEY's times [Nil84], and they are essential in hybrid robot control architectures [Mur00, KBM98]. One of the main problems when applying symbolic plans to mobile robot control is their robust execution. Abstract symbolic actions have to be decomposed into sequences of reactive behaviors provided by the behavioral layer of the control system. This is the reason why plan execution layers in three-tiered architectures are often named sequencing layers, or *Sequencers* for short.

In contrast to task or mission planning where it is in general a hard problem to find a correct solution to a given problem in the first place, in plan execution there are often many possible task decompositions that achieve a given goal but which might be more or less appropriate in a given situation. To find the task decomposition that maximizes plan quality – either with respect to robustness or the robot's expected performance – is the problem of action selection, and a plan execution system has to deal with it.

Besides the decomposition of actions, plan execution systems have to account for execution failures, e.g., due to changes in the environment. It is important that these failures are handled on the right level of abstraction, that is, on the right level of the decomposition hierarchy. A closed door has to be handled by planning a new path to the goal based on an update of the environment map, while a failure caused by a dynamic obstacle might be handled by a sequence of reactive behaviors to circumvent the obstacle.

In this chapter, we describe the execution layer of APPEAL as implemented in the RHINO system and discuss how it accounts for these problems. It is built on top of the reactive layer described in the previous chapter and executes abstract navigation tasks. The tasks can be specified by a human user using a graphical user interface. The tasks could, in principle, also be provided by a symbolic planning system which performs some kind of task- or mission planning.

APPEAL's execution system is based on the idea to represent the decomposition hierarchy and the state of the decomposition process explicitly as a plan. We use HTNs [NCLMA99] as the plan format. The hierarchical nature of HTN plans makes them a handy substrate for dealing with execution failure by jumping to higher levels of abstraction within the current plan and pick alternative task expansion strategies. This 'failing upwards' was one of the reasons for developing it for the archetype of HTN planners, NOAH [Sac77].

The approach to HTN planning presented in this chapter deviates from the

original approach in various aspects. First, tasks are considered as objects that know about their possible decompositions. They keep track of the state of their execution, for example to avoid the repeated execution of decompositions that have failed before. They can also perform considerable computation on their initialization. This computation can itself be a planning process like the computation of the shortest path to a given goal.

The execution system should also be able to replan opportunistically in order to improve the robot performance. This requires the explicit representation of the state of the execution process in order to remove tasks from the plan and choose new expansions where appropriate. In APPEAL, this is guaranteed by the application of the HTN plan representation.

The reasoning about the benefits of different courses of actions as required for the replanning can be based on plan projection. This, however, requires informative models of the tasks which are subject to replanning. As these models might be difficult to acquire, especially for tasks that are directly executed by the behavioral layer, it is useful to learn the required action models autonomously. To this end, the robot has to be able to monitor and log execution data for the tasks. The logging functionality is another important feature of APPEAL's execution layer.

The chapter is organized as follows. After discussing related work in the field of plan execution and hybrid robot control architectures, we give a short overview of HTN planning in Section 4.3. Section 4.4 applies HTN planning to the problem of navigation task execution and shows how the plan representation supports task decomposition, action selection, and failure handling. Section 4.5 gives a short overview of the execution planning system of APPEAL and its three main components. In Section 4.6, we demonstrate the use of HTN-based execution planning in three real robot experiments and conclude in Section 4.7.

4.2 Related Work

APPEAL is an example of a three-tiered architecture. Three-tiered architectures organize the control software in three layers.

1. A reactive layer. It controls the effectors of the robot and does the sensor interpretation. It provides simple reactive behaviors like wall following and collision avoidance. It is also called control layer.

2. A deliberative layer. It computes mission- and/or task plans from ab-

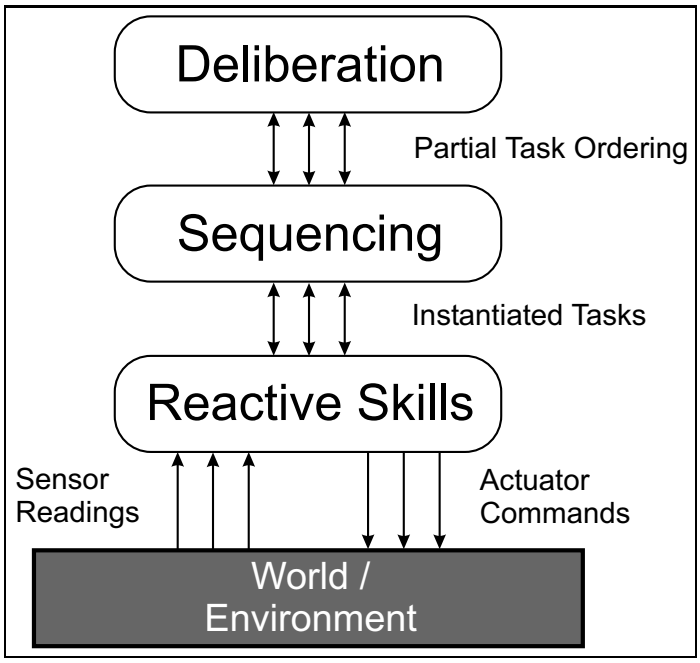

Figure 4.1: Interaction of the three levels in a 3T architecture.

stract task descriptions and abstract, often static models of the robot's working environment and its possible actions. This layer is also named deliberation layer.

3. An execution layer. This layer mediates between the reactive and the deliberative layer. Abstract plans are decomposed in terms of reactive behaviors while taking execution failures and exceptional situations into account. In the literature, this layer is often named sequencing layer as computing a totally-ordered sequence of behaviors from a partially ordered plan was originally considered the main problem of plan execution. However, we will use the more general term execution layer.

Figure 4.1 sketches the layering of a typical three-tiered (3T) architecture and describes the interaction between the three layers. The picture is taken from [BFG+97]. The idea to organize the control software of a mobile robot in three layers has been suggested independently by three research groups [Gat92, Con92, Bon91] at about the same time. Gat [Gat98] provides a detailed overview of successful three-tiered architectures for mobile robot control and discusses their differences.

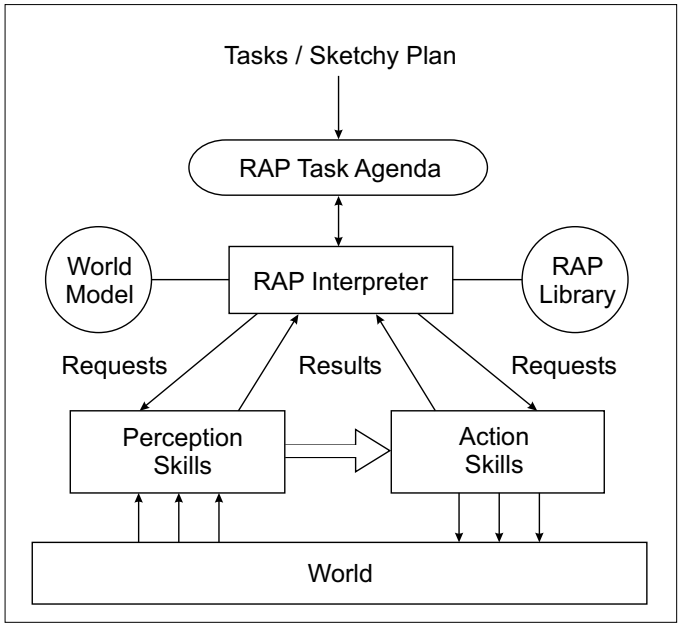

Figure 4.2: The Animate Agent Architecture

The suggested architectures were all intended to overcome the shortcomings of the Sense-Plan-Act (SPA) approach to robot control [Nil84] and of the purely reactive architectures like the subsumption architecture [Bro86, Bro99]. The SPA architecture as applied to control the mobile robot SHAKEY did not scale well to highly dynamic environments. It turned out that for mobile robot control the robust execution of a plan is significantly more difficult than its generation. Subsumption-style architectures can be regarded as a radical reaction to the problems of SPA architectures. Hierarchically layered behaviors make robot control very fast and reliable, but the approach lacks the possibility to integrate some kind of mission planning. Three-tiered architectures can be considered the most successful approach to combine purely reactive control and planning.

In the research on three-tiered architectures many different plan execution languages and frameworks like RAPs [Fir89, Fir95], REX [Kae87], PRS [GL87], and ESL [Gat97] were suggested. Bonasso et al. give a comprehensive overview of the research on plan execution languages and frameworks [BFG+97].

RAPs (Reactive Action Packages) [Fir87] are a set of rules implemented in a macro language of LISP that define how actions are to be decomposed into sequences of behaviors executable by the reactive system. It is possible to

have more than one method to decompose an abstract action into sequences of behaviors. Which decomposition method is to apply depends on the current state of the world. If a decomposition fails, the next applicable rule can be applied if there is one. RAPs can be arranged into hierarchies, which allows to handle failures on the right level of abstraction. This kind of *failing upwards* is supported by the HTN-style execution system of APPEAL as well.

RAPs have been successfully integrated into the Animate Agent Architecture [FKPS95] and applied to control the mobile robot CHIP which was developed at the University of Chicago. Figure 4.2 gives an overview of the Animate Agent Architecture. The Animate Agent Architecture is not a three-tiered architecture as there is no planning layer. Instead, sketchy plans given by a user are refined by task decomposition and by adding ordering constraints for the tasks. Like in APPEAL, this is done until a behavior is determined that can be directly executed by the reactive system. ESL (Execution Support Language) [Gat97] is a macro language of LISP developed to ease the implementation of plan execution layers in 3T architectures. ESL is based on the ideas of RAPs, but provides more sophisticated methods for failure recovery and action monitoring by language extensions.

The early execution systems like RAPs and PRS are based on a set of symbolic rules that are processed based on the current situation. More recent plan execution systems emphasize the hierarchical nature of tasks and their decomposition. TCA (Task Control Architecture) [Sim94] provides commonly needed control constructs like task decomposition, task synchronization, execution monitoring, and exception handling by managing task trees. Task trees can be considered as execution traces of hierarchical plans and are created dynamically at run time. Like in APPEAL, plan execution is performed by expanding abstract tasks into tasks directly executable by the behavioral system where the expansion can be dependent on the robot's state. In contrast to APPEAL, not only tasks are explicitly represented in the task tree, but also monitors, exception handlers, and temporal constraints on the execution order. TCA has been successfully applied to control many different mobile robot platforms like e.g. XAVIER [KS98]. TDL (Task Description Language) [SA98] provides the functionality of TCA as an extension of the programming language C++.

RAPs, ESL, TCA, and TDL as well as the other plan execution frameworks suggested in the literature [BFG+97] are well suited to select behaviors and their parameterizations. When selecting the next behavior, the execution system can take the robot's current state into account and is therefore reactive. In addition, it can maintain a state history to avoid the repeated

execution of unsuccessful actions. However, these sequencing systems are not intended to do search, some kind of planning, or learning. The execution layer of APPEAL deviates from earlier sequencing layers in this respect. This chapter will show how path planning using the MDP framework can be made part of the HTN planning. Chapters 5–7 demonstrate the integration of search, temporal projection, and learning into APPEAL's execution system.

CLARAty (Coupled Layer Architecture for Robotic Autonomy) [VNE+01] is another more recent architecture that tightly integrates planning and execution in an execution layer. In contrast to APPEAL, however, CLARAty is intended as a two-layer architecture.

In the ROBELS system [MG02b, MG02a], HTNs are used for the specification of sensor-motor modalities. Sensor-motor modalities specify how different self-localization and navigation methods can be combined in order to complete a task. ROBELS is in some aspects very similar to APPEAL, especially in the attempt to combine learning and planning techniques for plan execution. Regarding the use of HTNs the main difference is that in ROBELS planning processes are only allowed in leaf nodes of the AND/OR tree while in APPEAL they can be part of inner nodes as well. Chapter 6 provides a more detailed discussion of ROBELS.

4.3 Hierarchical Task Networks

HTN planning specifies a planning problem as a task network, i.e. a set of tasks together with constraints on the order in which they can be performed and restrictions on how variables may be bound. Tasks may be *elementary* or *compound*. The expansion, in general, is not unique. Planning is performed by expanding compound tasks until only elementary tasks remain.

HTN planning can be considered as extension of least-commitment planning where a plan is given by (a) a set of plan steps, (b) a partial order on the set, (c) a set of variable bindings, and (d) a set of causal links between the plan steps[1]. Hierarchical decomposition extends partial-order planning by abstract operators. A plan p correctly implements an abstract operator o if it is a complete and consistent plan for the problem of achieving the effects of o given the preconditions of o if

[1]Please refer to the introductory textbook of Russel and Norvig [RN95] or the survey article of Weld [Wel94] for an introduction to least-commitment and partial-order planning

1. p is consistent: there is no contradiction on the ordering- or variable binding constraints,

2. every effect of o is asserted by at least one step of p and is not denied by some other, later step of p, and

3. every precondition of the steps in p is achieved by a step in p or is one of the preconditions of o.

Although the selection of expansion methods for non-primitive tasks introduces a new kind of choice point to least commitment planning, it is in general helpful to focus the search. Furthermore, hierarchical decomposition can help to prune the search tree if one of the following two properties holds.

Downward Solution Property If p is an abstract solution to the planning problem, then there is a primitive solution of which p is an abstraction.

Upward Solution Property If an abstract plan is inconsistent, then there is no primitive solution of which it is an abstraction.

The downward solution property can be used to prune away all other abstract plans as soon as an abstract solution has been found. The upward solution property on the other hand can be used to prune away all descendents of an inconsistent abstract plan in the search tree.

HTN planning originates from early work by Sacerdoti [Sac77]. Since then, it has been used as a technique in several domain-independent and special-purpose planners. Milestones include the SIPE-2 system [Wil88] and its various applications, as well as BRIDGEBARON [SNT98], the winner of the 1997 computer bridge world championship. The SHOP planner [NCLMA99] is a modern, domain-independent HTN planner incorporating the BRIDGEBARON design knowledge.

4.4 Execution Planning using HTNs

This section will describe the use of the HTN representation for plan execution. We will discuss that HTNs provide an elegant way to represent choice points in the execution process. This is especially important when handling failures. It might for example be appropriate to backup several levels in the expansion hierarchy to handle a failure on a more abstract level.

However, the execution planning process in APPEAL deviates from standard HTN planning in various aspects. First, in the implementation presented here, we deal with fully instantiated tasks all the time. Second, we consider only fully ordered sequences of tasks as possible expansions. Third, we do not consider other planning operators than the expansion of compound tasks. These three restrictions basically reduce HTN planning to a search in an AND/OR tree. This proves to be sufficient for our application domain, for other applications, however, it might be useful to have the full HTN planning framework.

There is another difference with respect to the search process. In our implementation, the AND/OR tree is searched depth-first and the execution process stops as soon as a *plan stub* has been determined. We call plan stub an HTN with an elementary task as the first task in its ordering. The strategy of stopping to reduce compound tasks in an HTN as soon as a plan stub has been found, is called the *lazy expansion principle*. This principle is used in our case as the robot may start navigating even before a complete solution plan has been found. The rationale is that we cannot assume that no unforseen events occur during plan execution (which would make plan suffixes unexecutable or irrelevant) and that plans may include sensing actions whose results may not be known at planning time. The lazy expansion principle just reduces waste of planning time in these cases. After each successful execution of a task, the search backs up in the tree until a task is found that is not yet completed.

Please note, that with the lazy expansion principle it becomes possible to expand a compound task into a sequence of tasks whose length might not even be known before expanding the task for the last time. This is central for using HTNs for path execution as will become clear shortly.

In our framework tasks are not purely syntactical constructs, but are objects. These objects know about their possible expansions and about the state of their execution. In addition, these objects can initiate substantial computation like MDP path planning on construction. This is true not only for elementary actions, but also for compound tasks.

Before dealing with these issues in some more detail, we give an example of execution planning using HTNs. For this purpose, we first introduce part of the operator inventory for our execution planner. We have the following schemata for elementary tasks:

SetTarget(x, y, d) sets the target point (x, y) for the low-level routine for collision-free drive control, which has to be approached up to a precision

of d cm.

TurnTo(x, y) causes the robot to rotate on the spot until heading towards the point (x, y).

TurnToFree() causes the robot to rotate on the spot until heading towards some free space.

MoveForward(d) causes the robot to move by d cm straight forward.

MoveBackward(d) causes the robot to move by d cm straight backward.

All elementary tasks must have an implementation in terms of behaviors provided by the behavioral layer, so that executing an elementary task means calling the respective routine. That does, of course, not guarantee that each and every elementary task instance, or its corresponding control routine, respectively, can be successfully executed. Failure is possible, as usual. This issue will be addressed below.

We have two types of compound tasks, the schemata of which are:

ApproachPoint(x, y, d) drives the robot to position (x, y) within an approach distance of d. In terms of the expansion hierarchy, **ApproachPoint** is a middle-level task that serves for dealing with self-generated intermediate target points.

MDPgoto(x, y) drives the robot to the user-specified target at position (x, y). Over the time, it gets expanded into a sequence of **ApproachPoint** operators.

In the experiments, we consider the following expansions. The compound task **MDPgoto**(x, y) can be expanded into **ApproachPoint**(t_x, t_y, c) with the target point (t_x, t_y) either 2 m (default), 1 m, or 4 m ahead on the optimal path to the goal point (x, y) and with c=1 m, c=0.5 m or c=2 m respectively. **ApproachPoint**(x, y, d) is either expanded into **SetTarget**(x, y, d) (default), into the sequence **TurnTo**(x, y), **SetTarget**(x, y, d), into the sequence **MoveBackward**(30), **TurnTo**(x, y), **SetTarget**(x, y, d), or into the sequence **TurnToFree**(), **MoveForward**(30), **TurnTo**(x, y), **SetTarget**(x, y, d). Figure 4.3 gives a schematic summary of the expansion hierarchy used in the example.

Within a plan, an instance of any task has all arguments fully instantiated, as we are dealing with purely propositional plans here. Moreover, all task instances have an additional argument saying whether they are PENDING,

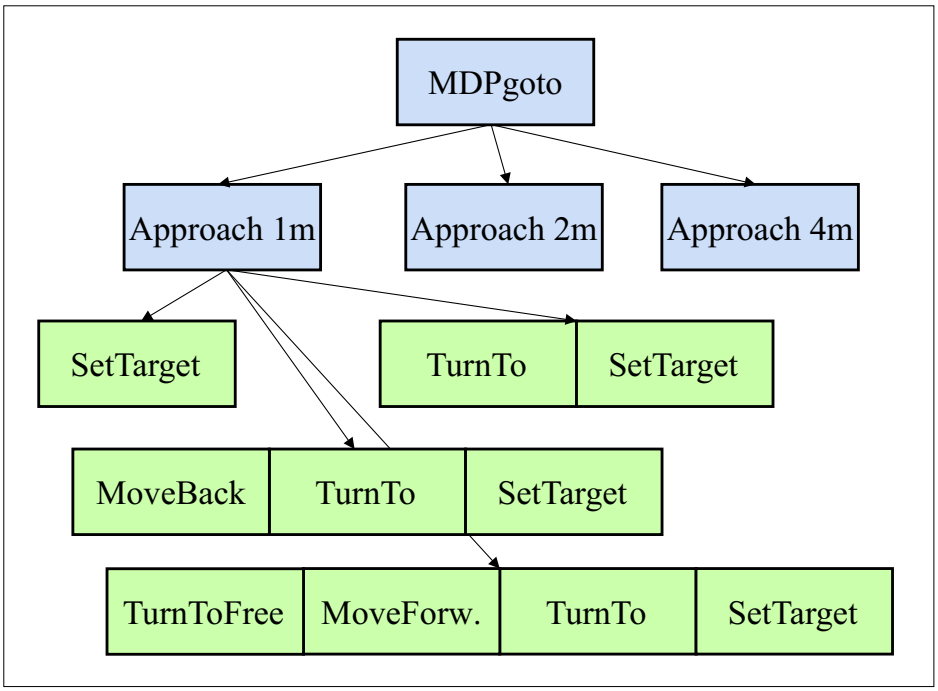

Figure 4.3: The expansion hierarchy.

i.e., not yet executed (for elementary tasks) or EXPANDED in the case of compound tasks. Executed tasks are simply deleted from the current plan.

Representing a plan as a stack, here is an example. Assume the navigation target is the position $(1521.31, 1563.8)$ on some given floor map. This would be transformed into the one-task plan

$$\textbf{MDPgoto}(1, \text{PENDING}, 1521.31, 1563.8)$$

where the first two arguments are the task instance ID and the status, respectively, and the following ones are like in the task schema descriptions given above. (This pattern will re-appear in all other task instances to follow.)

Dealing with the top task of the stack means expanding it, in this case, since it is compound. Using the default expansion, this yields

$$\begin{gathered}\textbf{ApproachPoint}(2, \text{PENDING}, 1434.38, 1009.38, 100)\\ \textbf{MDPgoto}(1, \text{EXPANDED}, 1521.31, 1563.8)\end{gathered}$$

and, expanding the **ApproachPoint** task,

SetTarget(3, PENDING, 1434.38, 1009.38, 100)
ApproachPoint(2, EXPANDED, 1434.38, 1009.38, 100)
MDPgoto(1, EXPANDED, 1521.31, 1563.8)

As the topmost task is elementary, this is a plan stub, and according to the lazy expansion principle, this operator gets immediately executed by the robot, activating an approach target behavior.

Assuming that all goes well, the control routine implementing the **SetTarget** task terminates successfully. This task pops out, and so does **ApproachPoint** in consequence. The **MDPgoto** task, however, is not yet finished, as its target point is not yet reached according to the robot's self-localization. In consequence, it gets re-expanded, yielding

ApproachPoint(4, PENDING, 1476.88, 1158.12, 100)
MDPgoto(1, EXPANDED, 1521.31, 1563.8)

the topmost task of which would get expanded into the respective **SetTarget** task and executed as before. If all keeps going well, this cycle of expand-execute-pop is repeated until the final target point is reached and **MDPgoto** pops out. Figure 4.4 summarizes such an expand-execute-pop cycle.

Failures to execute an elementary task are reported by the low-level control routines by raising exceptions of different types. Assume that executing the top task in the plan

SetTarget(5, PENDING, 1476.88, 1158.12, 100)
ApproachPoint(4, EXPANDED, 1476.88, 1158.12, 100)
MDPgoto(1, EXPANDED, 1521.31, 1563.8)

results in an exception of type NO-ADMISSIBLE-TRAJECTORY, i.e., the low-level execution cannot find an unoccluded local path from the current position to the point (1476.88, 1158.12), based on the recent sensor readings. As a result, the failed task would pop out and the next expansion alternative for the **ApproachPoint** task would be patched in, resulting in

MoveBackward(6, PENDING, 30)
TurnTo(7, PENDING, 1476.88, 1152.12)
SetTarget(8, PENDING, 1476.88, 1158.12, 100)
ApproachPoint(4, EXPANDED, 1476.88, 1158.12, 100)
MDPgoto(1, EXPANDED, 1521.31, 1563.8)

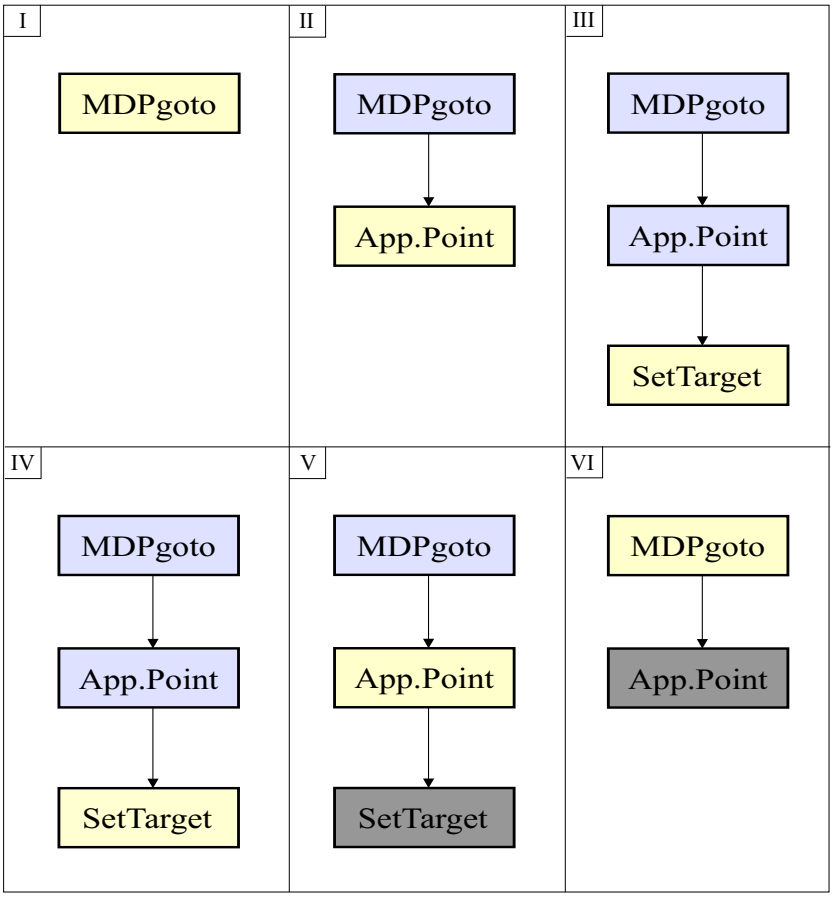

Figure 4.4: Task decomposition using HTNs.

Assume that after the successful execution of **MoveBackward** and **TurnTo** the execution of the **SetTarget** task fails again. The planner has to backup to the **ApproachPoint** task in the tree. If we assume that there is no other expansion applicable in this situation, the planner can even backup to the pending **MDPgoto** task in the stack. Only if no more backtracking is possible, the execution of the main task fails and permanent failure is reported. Figure 4.5 summarizes the example.

The example nicely demonstrates the *lazy expansion principle* for task decomposition and the handling of exceptions caused by unexpected obstacles or inaccurate effectors. It can also be used to illustrate some of the advantages of the object-oriented representation of tasks.

MDPgoto performs a full path planning on a grid map of the environment when created. The path planning process results in an MDP policy which

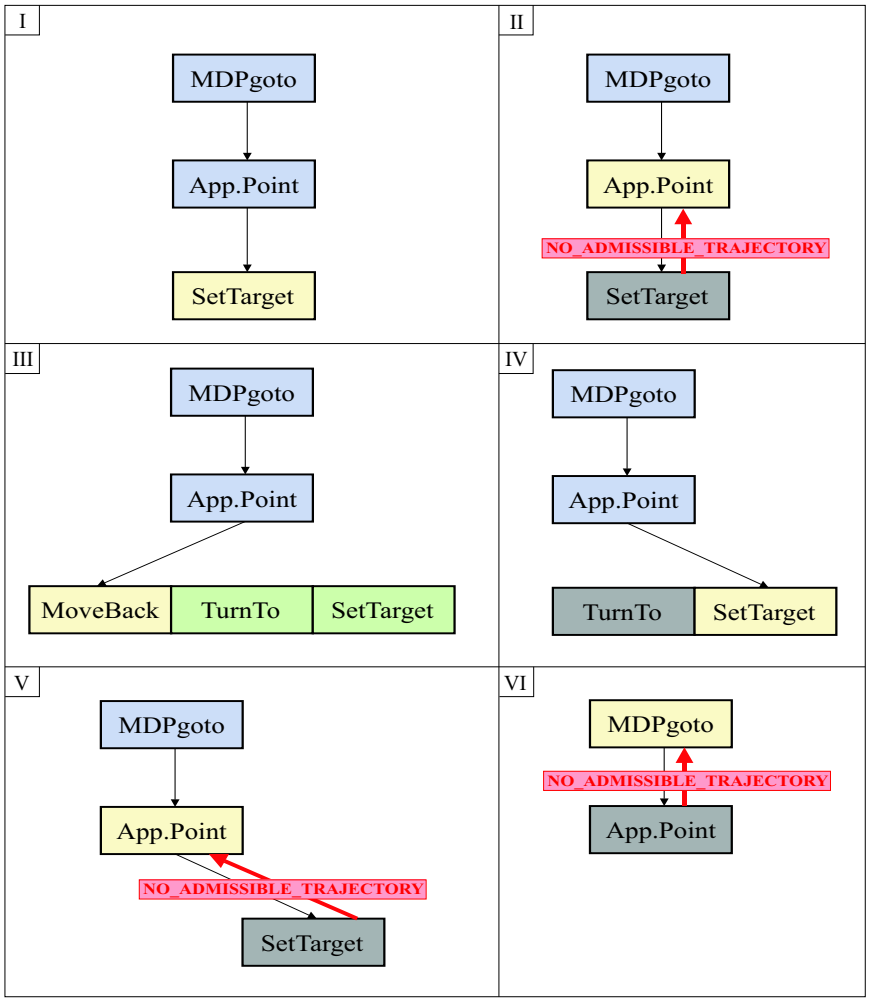

Figure 4.5: Failure recovery using HTNs.

assigns to each possible state of the robot, that is, a cell in the grid map, the optimal action with respect to the MDP. From this policy, the optimal path from the robot's current position to the goal can be computed efficiently. From the optimal path to the goal, instances of the **ApproachPoint** command can be generated where the target point is 1 m, 2 m, or 4 m ahead on the path. In contrast to classical HTN planning, the expansion depends on the current state of the robot when expanding the **MDPgoto** task. This simplifies planning as we deal with fully instantiated tasks all the time. To represent tasks as objects is an elegant way to integrate path planning and HTN planning in a hybrid planning process.

Which expansion to select from a set of possible expansions, can have a considerable influence on the robot's performance. However, the choice of the best action in the current situation, might also mean heavy computations. In general, however, the robot has to wait for the results of this computation before it can act. This time loss might in some cases even outweigh the performance improvement gained by the reasoning. To mitigate this problem, the execution planner always selects the default expansion of a task when expanding it for the first time. Like this, the resulting default plan can be computed very fast and the robot can start acting immediately. The time needed for the execution of the planned sequence of elementary actions, however, can be used to reason about plan improvements that can be achieved by the selection of alternative task expansions on some level of the current plan.

If for some compound task in the task tree an alternative expansion has been found which is projected to result in a better performance, the current plan has to be transformed accordingly. As the construction of plan stubs is very fast, this can be achieved by pruning all nodes in the tree up to the node that is to be expanded in a different way and then expand the tree again until a new plan stub has been generated. Chapter 6 will discuss this application of transformational planning to the problem of selecting the best expansion for a given task in some more detail.

4.5 Execution Planning in APPEAL

Figure 4.6 gives an overview of the execution planner of Appeal which comprises three main components. 1.) The plan module which represents the current state of the plan execution as an HTN. 2.) The world model which collects the data from various modules of the overall system like the localization module, the mapping module, and the behavioral system. It also provides an interface to these modules which can be used to execute elementary tasks. 3.) The observer module which observes the current state of the world to react to changes either by modifying the current plan or by logging state changes in a database system.

The plan module represents the current state of the plan execution as an HTN. Abstract tasks specified by a human or some task planning system form the input of the plan module. They are decomposed using default expansions until a plan stub has been generated. The elementary tasks of the plan stub can be executed directly by the behavioral system.

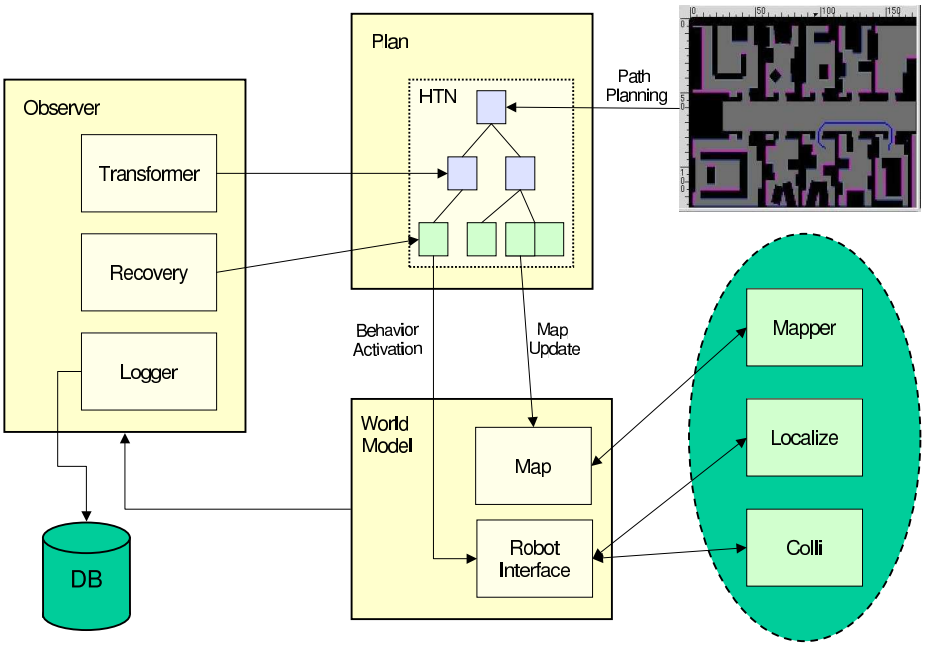

Figure 4.6: The APPEAL architecture

The world model provides a uniform interface to the other modules of the system that are necessary for a robust navigation. This is the localization system briefly introduced in Chapter 2, the behavioral system introduced in Chapter 3, and a mapping system used to perform updates of a static world map.

The observer module comprises three submodules. The recovery module is responsible for modifying the plan in reaction to execution failures. The transformation module reasons about the current plan of the robot in order to detect opportunities for plan improvements. The reasoning is based on learned models acquired from the data collected by the logging system. This will be explained in some more detail in Chapter 5.

4.6 Experimental Results

This section presents results of three experiments that have been performed to evaluate the design of the execution planner of APPEAL. All experiments

demonstrate the use of HTN-like representations for failure recovery. In the first two experiments, a PIONEER II robot is faced with an unexpected obstacle. Backtracking in the task tree helps the robot to successfully recover from the failure in both cases. The failures, however, are handled in different ways, based on a simple diagnosis of the failure. The third experiment demonstrates how a failure on a low level of abstraction (executing a navigation behavior) is handled on a much higher level of abstraction by replanning a path.

4.6.1 Failure Recovery using HTNs

The first experiment is intended to demonstrate how the execution planner of APPEAL supports the handling of exceptions caused by unexpected moving obstacles. Figure 4.7 shows how a human steps into the way of the robot. As the obstacle moves fast towards the robot, the robot can do nothing but an immediate stop. After the stop, the **SetTarget** behavior fails by throwing the exception NO-ADMISSIBLE-TRAJECTORY and the exception handling is activated. The situation is similar to the one shown in 4.5. The execution planner removes the failed **SetTarget** from the plan and has now to select an alternative expansion of the **ApproachPoint** task. As the direct way to the target is blocked, the second possible expansion of **ApproachPoint** is not an option in this case (please refer to Figure 4.3 for an overview of the possible expansions in this case). The robot thus selects the third expansion. It moves backward until it is free again and continues its approach target behavior. However, the human decided to do the same and thus all possible trajectories are blocked again. The failure recovery decides to try the last possible expansion. The robot turns into the direction of highest clearance, moves a bit forward in this direction, turns back towards the target and restarts the approach point behavior. The robot succeeds in passing the human.

Figure 4.8 shows another example of successful failure recovery. In this scene, a wastebasket has been placed near the entry of a door passage. The robot tries to drive round the obstacle but the controller overshoots and the robot gets too close to the wall. This again causes an immediate stop and the robot throws the exception NO-ADMISSIBLE-TRAJECTORY. In this case, however, the exception can be handled by selecting the second expansion of **ApproachPoint**. The robot just turns to the target and can then successfully approach the target again.

The two experiments show that HTNs provide an elegant framework for han-

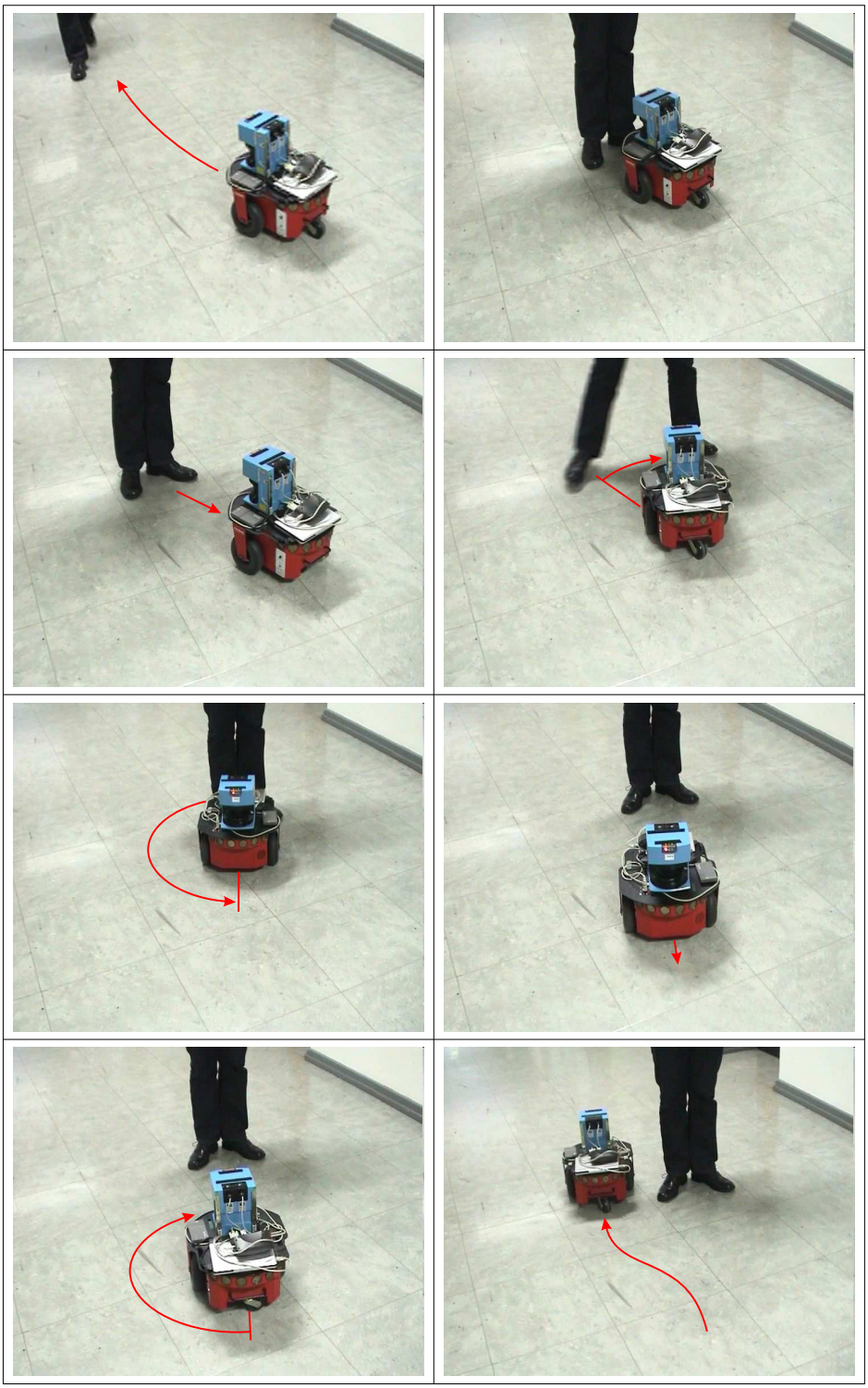

Figure 4.7: A PIONEER II robot handles a sequence of exceptions caused by a moving human obstacle.

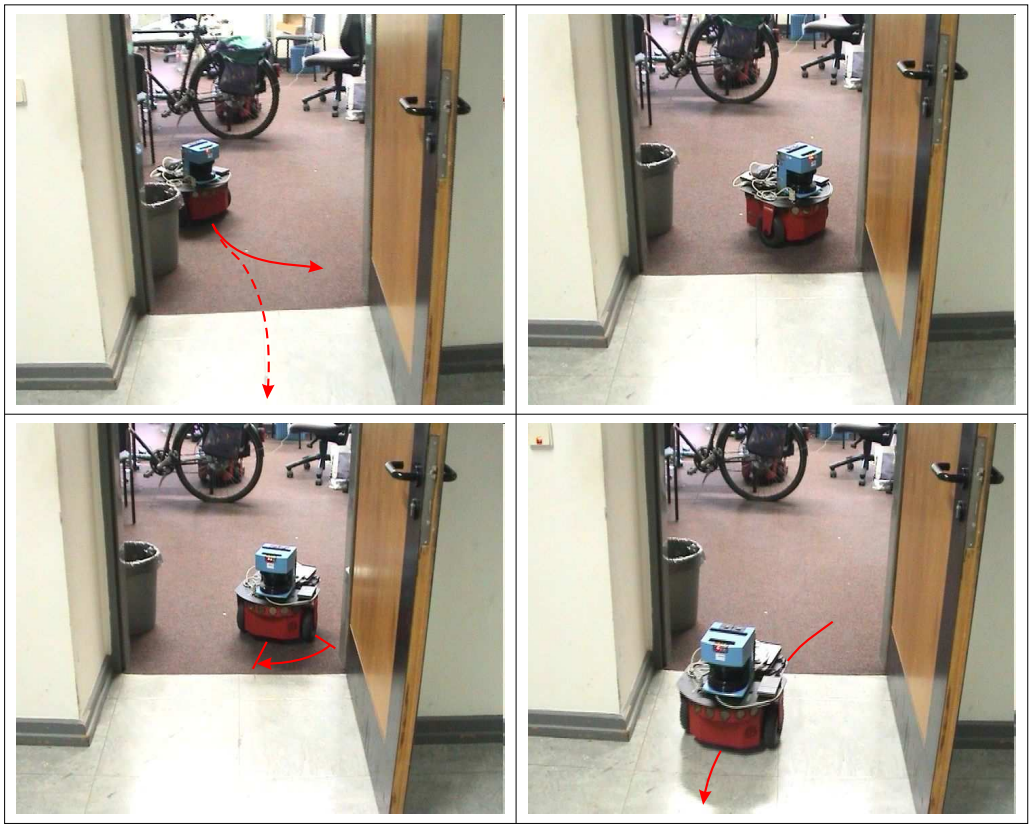

Figure 4.8: A robot handling a failure caused by an unexpected obstacle.

dling execution failures caused by unexpected and possibly moving obstacles. They demonstrate that failures can be handled by backtracking in the task tree and then selecting an alternative expansion on some appropriate level of abstraction. The action selection process for exception handling might include some kind of diagnosis. In our experiments a few hand-coded rules have been sufficient for this.

In the experiments presented so far, we only had to backtrack one level in the expansion hierarchy to successfully handle execution failures. In the following section, we will present an experiment where an execution failure is handled by backtracking multiple levels in the task tree which implements some kind of replanning.

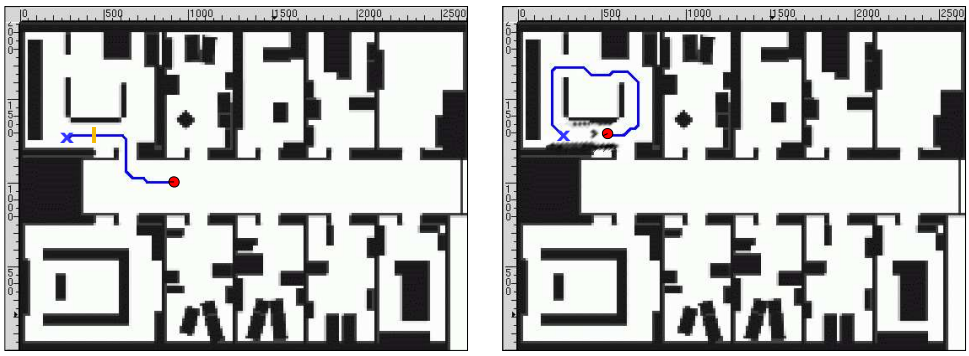

Figure 4.9: Replanning in the presence of a blocked passage.

4.6.2 Failure Recovery by Replanning

Consider the situation depicted in Figure 4.9 where the robot is asked to go to the point marked by a cross. The robot computes the path shown in Figure 4.9 (left) although there is an alternative path which, however, is considerably longer. As soon as the robot gets to know that an obstacle (marked by a yellow bar) blocks its way, it should replan and consider the longer way (right). The experiment described in this section shows that this kind of replanning can be achieved by HTN-based execution planning.

For this purpose, we have to extend the task hierarchy by an additional level shown in Figure 4.10. The **Goto** tasks can be expanded in two ways. First, by an **MDPgoto** or second by a **MapUpdate** followed by an **MDPgoto**. **MapUpdate** is an elementary task which is executed by collecting a sequence of laser range scans that are then integrated into the robot's global environment map. **MapUpdate** is a sensing action/task. This is deliberately so as the computationally demanding task should only be performed when required.

As can be seen from Figure 4.11, using the extended task hierarchy the mobile robot RHINO shows exactly the expected behavior. It first tries to navigate to the goal using the computed shortest path (1). It is then faced with an unexpected obstacle (2). As the obstacle blocks a narrow passage, no alternative expansion of the **ApproachPoint** task is expected to be useful in this case. The planner thus backtracks further. For the **MDPgoto** as well no alternative expansion is considered to be useful and the planner backtracks again. On the top level of the task hierarchy, the planner finally selects an alternative expansion of the **Goto** task. **Goto** is expanded into a sequence of **MapUpdate** and **MDPgoto**. The map update includes the wastebasket

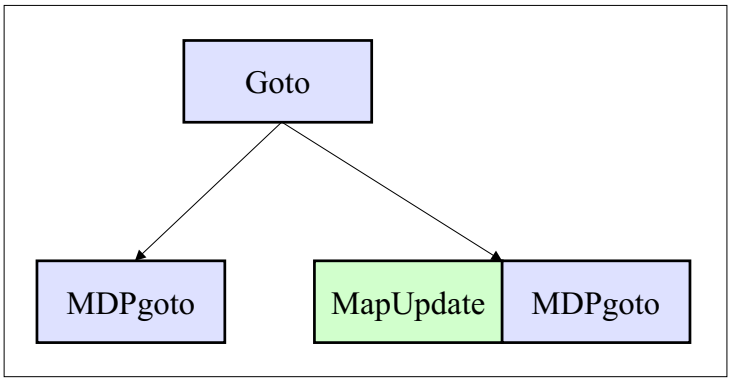

Figure 4.10: Extension of the expansion hierarchy for implementing map updates.

into the map and the **MDPgoto** task performs path planning on the updated map. It finds another path, which is longer, but avoids the blocked passage. The new **MDPgoto** is again decomposed in a sequence of **ApproachPoint** tasks (3-5) until RHINO finally arrives at its goal position (6). Figure 4.12 summarizes the experiment.

The experiment shows that using the HTN framework, execution failures on the lowest level of the task hierarchy might be handled on a much higher level of abstraction if appropriate. To this end, the exception handling mechanism has to be able to detect that a failure cannot be handled appropriately on a given level of abstraction. In the experiment, this has been achieved by using the concept of a narrow passage which can be computed from a clearance map (please refer to Chapter 5 for more details). However, we do not claim to have a general story about this kind of diagnosis and it might well be an interesting topic for future research in this area.

4.7 Summary

The sequencing layers of three-tiered architectures like APPEAL have three main responsibilities.

1. Decomposing abstract tasks into a sequence of elementary tasks that can be executed by a behavioral layer. This decomposition process should be efficient to allow for fast replanning.

2. Handling failures reported by the behavioral system. The failure han-

Figure 4.11: The mobile robot RHINO performing replanning when confronted with an unexpected obstacle.

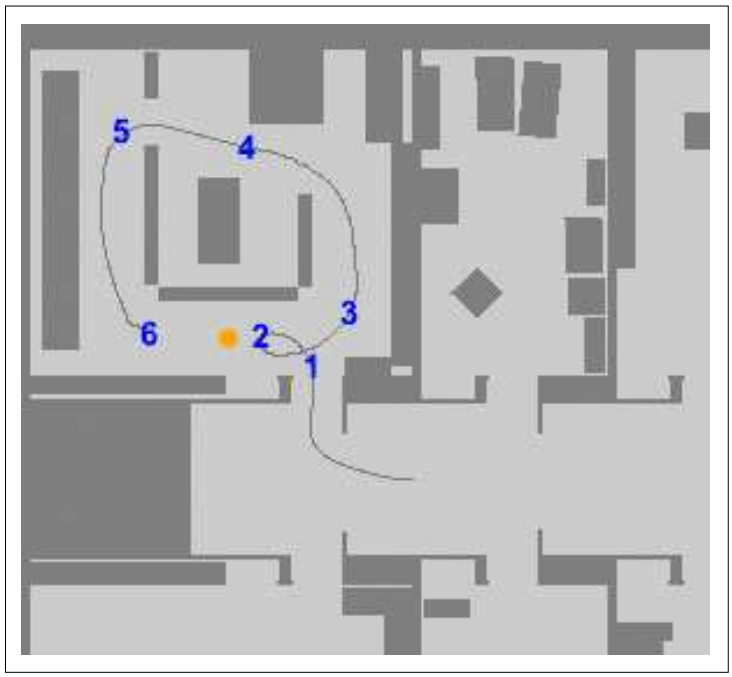

Figure 4.12: Summary of the experiment. The numbers denote where the pictures of Figure 4.11 have been taken.

dling should be performed on the right level of abstraction, that is e.g., by choosing a different behavior or by computing a new path on an updated map depending on the severity of the failure.

3. Optimizing plan quality with respect to some given performance measure. The optimization of plan quality is an important aspect of plan execution, as an abstract task or a partially ordered set of such tasks generally has many different valid execution plans which might differ considerably with respect to their quality, that is e.g., their robustness or effectiveness.

In this chapter, we have introduced the plan execution layer of Appeal, which meets the three above-mentioned requirements. It is based on the idea to use Hierarchical Task Networks for the transparent representation of the state of the execution process. In contrast to the task- and mission planning approaches, which are based on HTNs, tasks are considered as objects. They keep track of the state of their execution, for example to avoid the repeated execution of decompositions that have failed before. They can also perform considerable computation on their initialization like planning

the shortest path to a given goal. This helps to integrate different planning techniques like hierarchical decomposition and path planning in a hybrid planning system.

The explicit representation of choice points in the plan execution process helps to perform failure recovery on the appropriate level of abstraction (failing upwards). This has been demonstrated in three real world experiments with two different robot platforms, the PIONEER II platform and the RWI B21 platform. The experiments show how different failures caused by unexpected obstacles are handled on different levels of abstraction depending on the situation in which they occurred. While in two cases the exception could be handled by selecting a different sequence of elementary tasks, the blockage of the robot's path by an obstacle in a narrow passage was handled by a replanning based on an update of the robot's environment map.

It has also been argued that the execution planner of APPEAL supports some kind of transformational planning. It computes a default plan, which can be generated very fast by always selecting the default expansion of a task, and immediately starts to execute it. This default plan is only modified if some alternative course of action is projected to result in a better performance. Chapter 5 and Chapter 6 deal with this issue in some more detail and discuss how the relevant projective models can be learned from the data collected by APPEAL's execution system.

Chapter 5

Plan Projection for the Online Scheduling of Navigation Tasks

This chapter demonstrates the application of plan projection to the online scheduling of navigation tasks. In our approach, plan projection is based on models of the robot behavior where the models are learned from data collected by the execution system. We apply model tree learning, a generalization of regression tree learning, to predict the time it takes the robot to execute an action. Model trees are well suited for learning action models as they combine the benefits of memory-based learning and general function approximation.

We introduce the projection-based scheduling system PBS. PBS schedules asynchronously arriving user requests for navigation tasks in order to maximize a given reward function taking deadlines, expected execution time, and task priorities into account. PBS performs a heuristic search in the space of all possible task orderings to find the schedule that maximizes the robot's expected reward, where which is computed based on projecting the schedules. PBS is integrated in the HTN-based execution system introduced in the previous chapter by adding a new level to the task hierarchy.

The experimental evaluation of projection-based scheduling shows that it outperforms both first-in-first-out scheduling and urgent-first scheduling in two different scheduling scenarios. For a third scheduling scenario, we investigate how different prediction functions and projection methods impact the robot performance.

5.1 Introduction

The idea of plan projection is to forecast the physical robot performance on the basis of the abstract, symbolic plan representation. Plan projection, however, is not a simulation, and deliberately so. Plan projection tolerates all the abstractions and inaccuracies of the plan level domain description for the sake of efficiency of reasoning. Its purpose is to find with negligible effort possible alternative courses of action that look better in some respect than the currently available plan. If such a plan is found, it is swapped into the robot controller instead of the previous one.

We borrow the term plan projection from McDermott [McD92b] and from its refinements later developed by Beetz [Bee00]. Their work on plan projection has focused on making the robot performance more robust by forestalling certain critical situations that some plan might lead the robot into. Our focus is more on predicting the expected performance of a robot executing a navigation plan.

We show in this chapter, how a projection algorithm can be based on models of the robot behavior learned from previously collected execution data. The main challenge here is to effectively compute a feature language that is suited to compactly describe a navigation action and allows for precise predictions of the time it takes to execute the action. To solve this problem, we use the optimal path to the action's goal point together with a clearance map of the environment to compactly describe the action.

Based on the feature language computed from this path, we can apply standard learning techniques to estimate the time it takes the robot to execute an action. In this chapter, we introduce model tree learning, which combines the benefits of linear regression and regression trees. While linear regression is a technique for function approximation well known from the statistics literature, regression trees and model trees are less known. Regression trees are similar to decision trees, but contain real-valued predictions in the tree leafs. Model trees combine both concepts and allow for general linear functions in the tree leafs. Model trees are well suited for learning action models as they combine the benefits of memory-based learning and general function approximation. In addition, they can be transformed into sets of rules that are well suited for human inspection and interpretation.

As application scenario for plan projection based on learned models, we consider in this chapter the online scheduling of navigation tasks. We introduce the projection-based scheduling system Pbs. Pbs schedules asynchronously arriving user requests for navigation tasks in order to maximize a given re-

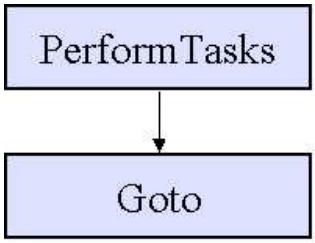

Figure 5.1: The only possible expansion of **PerformTasks** is **Goto**. **PerformTasks** keeps a list of **Goto** tasks and selects the first task in the list for expansion.

ward function taking deadlines, expected execution time, and task priorities into account. PBS performs a heuristic search in the space of all possible task orderings to find the schedule that maximizes the robot's expected reward where the expected reward of a schedule is computed based on its projection.

The sequencing of a partially ordered or completely unordered set of tasks for a single robot is a relatively simple scheduling problem. In contrast to other scheduling tasks, e.g. for a fleet of robots, we consider task sequencing to be part of the plan execution problem.

The online scheduling is integrated in the HTN-based execution system by introducing a new task to the task hierarchy. The task **PerformTasks** makes the top level of the task hierarchy. It is responsible for executing all user requests that are currently pending. For this purpose, **PerformTasks** keeps a list of all pending user requests. In each expansion step, the first task in the list is selected.

New tasks are added to the list in the order of arrival (or alternatively in the order given by the urgency of the tasks). PBS reschedules the tasks in order to maximize the robot's expected reward. In contrast to the simple heuristics used to generate a default ordering, PBS can account for the probability of missed deadlines, expected execution time, and task priorities simultaneously.

The rescheduling is organized as a transformational planning process. The robot starts to execute the default schedule computed by one of the above mentioned heuristics. Parallel to execution, the scheduler searches for schedules that promise a higher expected reward. As soon as such a schedule is found, the default plan is replaced by the new schedule. When the next task has to be selected by expanding **PerformTasks**, this is done with respect to the new schedule.

The remainder of this chapter is organized as follows. The next chapter sketches related work in the field of plan projection, projection-based planning, scheduling, and the learning of action models. Section 5.3 briefly reviews the concept of decision trees and introduces regression trees and model trees. In Section 5.4 we discuss in some more detail the feature language used to describe navigation actions and learn their expected duration. Section 5.5 describes the scheduling system PBS, which is then experimentally evaluated in Section 5.6. Section 5.7 concludes the chapter.

5.2 Related Work

We borrow the term plan projection from McDermott [McD92b] and from its refinements later developed by Beetz [Bee00]. Their work on plan projection has focused on forestalling plan failures caused by exogenous events and is part of a framework for transformational planning. The idea of transformational planning is to interleave planning and execution. Rather than computing a correct plan in the first place, a default plan to carry out the user-specified tasks can be assembled quickly just by retrieval from a plan library. The default plan, however, is neither guaranteed to be optimal nor even to be correct. To forestall potential plan failures, the default plan is repeatedly projected and transformed as suggested by plan critics. Both plan projection as well as plan repair can be done during execution, and plan execution and plan projection can therefore be closely interleaved this way.

McDermott [McD94] has suggested a probabilistic rule language that allows to specify what can happen when an event occurs, as well as what events can occur when certain propositions are true. The language supports a Monte Carlo style of projection, in which event sequences are sampled randomly using the probabilities specified by the rules. This means that the same plan has to be projected many times in order to obtain statistically valid propositions. The idea of this Monte Carlo style projection is very similar to logic sampling, a stochastic simulation method to compute probabilities from complex Bayes nets [RN95, Jor98].

Beetz et al. [BBG99] describe a transformational planning approach for scheduling navigation tasks. It is based on the transformational planning framework of McDermott [McD92b] and his probabilistic rule language [McD94], but is applied to scheduling navigation tasks for real and simulated robots. They also give a criterion for probabilistically approximately accurate predictions which determines the number of sample execution scenarios necessary to detect a failure that occurs with a probability $\geq \theta$

with a probability of at least β.

McDermott and Beetz use plan projection to find a probably correct plan for a given problem. We, in contrast, consider planning problems for which many valid solutions exist, but where plans differ considerably in their quality. Plan projection is thus used to improve plans rather than repair them. Both approaches have in common that plan projection is used for interleaving planning and plan execution. Another important difference to their work are the techniques used for plan projection. While they are using a plan projection based on Monte Carlo simulation, which is computationally very demanding, we apply learned action models, which allows for faster projection.

In this chapter, we apply plan projection to schedule navigation tasks. More precisely, we deal with the problem of sequencing a partially ordered or completely unordered set of navigation tasks for a single robot. This is a relatively simple scheduling task and generally considered part of the plan execution problem. Alami et al. [AFH+98] as well as Surman and Morales [SM02] describe solutions to the more general problem of scheduling tasks for fleets of robots. Both approaches, however, do not attempt to learn action costs.

To learn the action models required for the plan projection, we apply model tree learning, a generalization of regression tree learning that combines the advantages of memory-based learning with the advantages of function approximation. Regression trees have been applied to learn action models for navigation planning, e.g. by Balac [Bal02] and Haigh [Hai98]. Balac uses regression tree models to predict action costs for navigation actions of an outdoor robot and applies these action models to do path planning which enables the planner to take different terrain conditions into account.

Haigh et. al. have also applied regression tree learning to acquire cost models for navigation actions. They are applied to compute the best route in an indoor environment taking into account that the crowdedness of floors and hallways varies with respect to the time of day. In another application, regression tree rules are transformed into search control rules for the STRIPS-like planning system PRODIGY [VCP+95].

Haigh as well as Balac argue that regression tree learning is well suited for learning action models and can be used by path planning and task planning systems for various reasons. (a) Tree-based induction methods can learn conditional action effects, (b) they are applicable to continuous variables, (c) they tolerate noisy data, and (d) they produce explanations well understandable to human users. In this chapter, we will argue that model trees are well suited for the same reasons, but are more expressive models as they

are not limited to approximating piecewise-constant functions.

Wang and Dietterich [WD99] discuss the use of regression trees in combination with TD(λ)-learning to solve job-shop-scheduling problems. They utter the hope that regression tree learning might be suitable to become the "basis of a function approximation methodology that is fast, gives good performance, requires little-or-no tuning of parameters, and works well when the number of features is very large and when the features may be irrelevant or correlated".

Balac [Bal02] provides a comparative study of different learning algorithms and concludes that regression trees are best suited for learning action models. In the experimental comparison of neural networks and regression trees described in Chapter 2 we obtain similar results. Sridharan and Tesauro [ST00] applied Q-learning in combination with regression trees in multi-agent scenarios. They point out that regression trees have been superior to neural networks both in the policies learned and the reduced training effort.

5.3 Tree-based Induction

This section describes tree-based induction methods. We briefly review the concept of decision trees and then introduce regression trees and model trees, which generalize tree-based learning techniques to numeric target attributes.

5.3.1 Decision Trees

Decision tree learning is a method for approximating functions of the form $y = f(x_1, ..., x_k)$ where the k attributes $x_1, ..., x_k$ can be nominal or numeric and the target attribute y is nominal. Although decision tree learners are weak learners, decision tree learning belongs to the most popular learning schemes. This popularity is due to the fact that decision trees can be interpreted as sets of horn clauses. Therefore decision trees do not only allow to predict values of the target function, but in addition help to understand the structure of the target function. This makes them particularly attractive for data mining tasks [WF00].

Figure 5.2 shows a simple example of a decision tree used by Russell and Norvig in their AI textbook [RN95] to introduce decision trees. It predicts whether a person (say R.) will wait for a table in a restaurant. The inner nodes of decision trees are labeled with attribute tests, each of the leaving edges is labeled with one of the possible outcomes of the test and each leaf

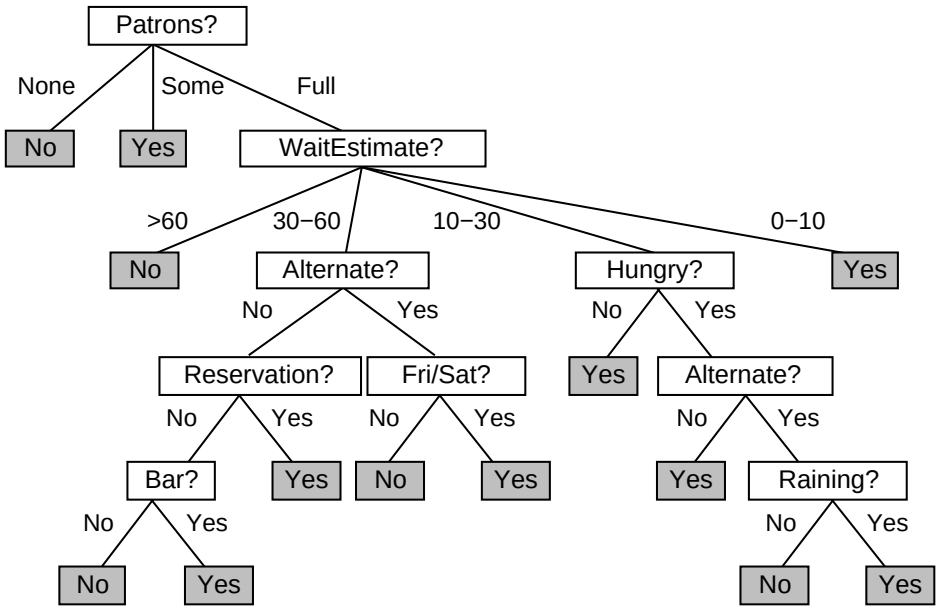

Figure 5.2: A simple decision tree

node with a value of the target concept.

Each path from the root node to a leaf node corresponds to a horn clause. In the example, the rule that predicts whether R. will wait for a table in the situation given by the feature vector (Alternate=No, Bar=Yes, Fri/Sat=Yes, Hungry=No, Patrons=30-60, Price=cheap, Raining=No, Reservation=Yes, Type=Burger, WaitEstimate=30-60) is

IF	Patrons=Some	AND	WaitEstimate=30-60	AND
	Alternate=No	AND	Reservation=Yes	
THEN	WillWait=Yes			

Decision trees can represent arbitrary boolean functions, as every row in the truth table can be represented by a path from the root node to a leaf node in the decision tree. Unfortunately, such a representation is not always compact as can be demonstrated for the parity function

$$\text{parity}(\vec{x}) = \begin{cases} 1 & \sum_i x_i \text{ is even} \\ 0 & \text{else} \end{cases}$$

To represent this function we need $2^n - 1$ inner nodes and 2^n leaf nodes, because every row in the truth table has to be represented with a path containing n attribute tests.

Standard decision trees are propositional in nature, as all attribute tests refer to a single object and expressions of predicate calculus like

$$\forall x, y[\exists z P(x, z) \wedge P(z, y) \Rightarrow G(x, y)]$$

therefore cannot be represented. However, extensions of decision tree learning to first-order logic exist [BuR98].

From the fact that decision trees can represent any boolean function by storing the complete truth table of the function, it follows that a decision tree learner should prefer simpler trees against more complex trees (if they have the same prediction quality) to achieve a suitable generalization. As a complete search in the space of all possible decision trees is not feasible (the problem of growing optimal decision trees is known to be NP-complete [HR76]), decision tree learning is carried out by a general-to-specific hill-climbing search in this space. Simple decision tree learners perform no backtracking during this search. More advanced decision tree learning algorithms perform post-pruning (see below) which can be regarded as a form of backtracking.

Although the aim of the search is to increase the prediction quality of the tree, it is not a good strategy to select the tree that maximally reduces the classification error in each iteration of the greedy search. As was shown experimentally [BFOS84, Qui93a] it is a better strategy to select in each iteration the tree that maximally reduces the impurity of the training data.

A descriptive statistics that measures the impurity of numeric attributes is the variance. For nominal attributes y with possible values $y_1, ..., y_k$ such an impurity measure is the entropy:

$$\text{entropy}(S) = -\sum_{i=1}^{k} p(y_i) \log_2 p(y_i)$$

Here $p(y_i)$ is the proportion of examples with target value y_i in S, that is $p(y_i) = \frac{p_i}{n}$ where p_i is the number of occurrences of y_i in S and n is the number of examples in S.

Together with the language bias of decision trees this search bias to greedily reduce impurity within the training data constitutes the inductive bias of decision tree learners.

Algorithm 2 Generic Recursive Partitioning Algorithm

Require: A set of n data points $T = \{< \vec{x}_i, y_i >\}$
  if termination criterion is met **then**
    create a leaf node l and assign value $g(T)$ to it
  else
    find best splitting test s^*
    create node t with split s^*
    partition T into T_L and T_R according to s^*
    apply the algorithm recursively to T_L and T_R
  end if

Growing Decision Trees

Decision trees are built with recursive partitioning algorithms like the one given in algorithm 2. This algorithm partitions the input space into regions and assigns a class to each region.

The complexity of the decision tree built by such algorithms depends on five decisions that have to be made.

Stopping Criterion When to stop the tree growing process.

Possible Splits What kind of attribute tests to consider.

Splitting Criterion How the best attribute test is chosen from a set of candidate splits.

Possible Predictions What kind of predictions are used in the leaf nodes.

Prediction Function How the predictions for the leaf nodes are computed.

The choice of the stopping criterion is considered as not very critical when using a suitable post-pruning method (see below). A possible stopping criterion is that the number of examples p that are assigned to a certain leaf node is less than a constant threshold value c. Often a threshold of $c = 2$ is chosen. This results in huge decision trees which tend to overfit the training data. An alternative stopping criterion is to limit the depth of the tree.

Which splits are considered for an attribute y depends on the type of the attribute. If the attribute is categorical with the possible values $y_1, ..., y_k$ splits of the form $y = y_i$ are constructed. If the attribute is ordered, that is discrete-valued with an order defined on the attribute values, splits of the form $y < y_i$ are supported as well. For numeric attributes splits of the form

$y < \theta$ are considered. This produces decision boundaries that are parallel to the main axes. Other splits like multi-variate splits have been investigated in the machine learning literature [MKSB93, BU95].

Breiman et. al. [BFOS84] suggest selecting the best split as follows. Let T be a set of training samples and $T_L(s)$ and $T_R(s)$ be partitions of T according to some split s. Let $I(T)$ be some measure for the impurity of an example set T. Then the split s^* is considered best that maximizes the impurity reduction:

$$s^* = \arg\max_s \left[I(T) - \left(\frac{|T_L(s)|}{|T|} I(T_L(s)) + \frac{|T_R(s)|}{|T|} I(T_R(s))\right)\right] \tag{5.1}$$

where $|T|$ is the number of examples in set T. As impurity measure they suggest to use the Gini index [BFOS84]. Quinlan [Qui93a] suggests to use the entropy (see above) as impurity measure for nominal attributes.

In standard decision trees, predictions are of the form $y = y_i$ where y_i is one of the possible values of y. It is simple to compute a confidence factor that can be interpreted as the probability that a classification is correct. It is just the proportion of examples in a leaf node that are correctly classified. Instead of constant-valued predictions it is also possible to have predictions of the form $y = h(\vec{x})$ where the function h could for example be represented by a neural network. Quinlan [Qui93b] discusses how decision tree learning can be combined with instance-based learning by using a distance-weighted prediction.

In the simplest case, the function for computing the predictions is a majority vote of the classifications of all the examples in the leaf node. In case of non-constant predictions of the form $y = f(\vec{x})$, the prediction function is itself a learning scheme like a neural network learner.

Post-pruning Decision Trees

A major problem when growing decision trees is to decide when to stop the growing process. Stopping too late causes huge trees that tend to overfit the training data and therefore achieve a poor generalization on the test data. A stricter stopping criterion, e.g. to stop when no more split can be found that significantly reduces the impurity, might prevent to find a combination of splits that can significantly reduce impurity. To circumvent this problem, Quinlan [Qui87] has suggested growing trees using a trivial stopping criterion (e.g. until each leaf node only contains examples that belong to the same class) and then do reduced-error post-pruning. For reduced-error pruning a part of the data that can be used for training (e.g. a third of it) is split from

Algorithm 3 Reduced-error post-pruning

Require: A set of n data points $V = \{< \vec{x}_i, y_i >\}$ independent of the set used for growing the tree
Ensure: l is the root of a binary decision (sub)tree
  if l is leaf node **then**
    return
  end if
  Apply the algorithm recursively to the two sons of l, l' and l''
  if l' and l'' are leaf nodes **then**
    if the fusion of l' and l'' reduces the error on V **then**
      l=fuse(l',l'')
    end if
  end if
  return

the training data and stored as validation set. Then a tree is built using the reduced training set, the growing set, and a trivial splitting criterion. The validation set is then used for the post-pruning of the tree. Two leaf nodes are fused again if the prediction error on the validation set is less with the resulting smaller tree than it was with the bigger tree. Here, fusing means that the data stored in each leaf node is merged and the prediction function is applied to the merged data to generate a new prediction. This pruning step is repeated until the prediction error on the validation set cannot be further reduced. The pseudo code of the reduced-error pruning method is given in Algorithm 3.

Breiman et.al. [BFOS84] consider another post-pruning mechanism, cost-complexity pruning, which assigns a cost to each split and uses it to maximize the utility of the tree which trades off the complexity of the tree for the prediction error it produces. Quinlan and Rivest [QR89] describe a post-pruning method based on the Minimum Description Length principle. The latter two methods have the advantage that they do not need to withhold data from the training set for the validation set.

An alternative to pruning the decision tree to avoid overfitting phenomena, is to first derive rules from the tree and then prune the rules in the rule set by removing preconditions until this does not further improve the quality of the rule. One way to evaluate the quality of a rule is measuring the prediction error for a separate validation set (reduced-error pruning). Quinlan [Qui93a] describes a heuristic rule pruning method.

When translating a decision tree into a set of rules, the rules have the fol-

lowing property: Every possible example is covered by exactly one rule. In the rule pruning process this property is in general not maintained. A single example can be covered by more than one rule or no rule. This has two consequences: a default rule has to be introduced and the rules have to be ordered and processed in that order. An advantage of rule pruning over tree pruning is that the resulting rules are often much simpler and more intelligible. A test relevant in one context (rule) but irrelevant in another context (rule) can be removed from the first rule while being maintained in the other rule. Because of the tree structure of decision trees this kind of pruning is not possible for tree pruning.

5.3.2 Regression Trees

Regression trees [BFOS84] are similar to decision trees but predict a continuous valued function rather than classify an object into discrete classes.

Regression trees are built with recursive partitioning algorithms like the one given above, but the predictions, the prediction function, and the stopping criterion differ. In the simplest case the prediction is of the form $y = \theta$. For learning a function $f(\vec{x}) = y$ with $f : \mathbb{R}^n \rightarrow \mathbb{R}$ using univariate threshold splits this results in a piecewise constant regression model where the input space is partitioned into regions with boundaries parallel to the main axes and where a constant value is assigned to each region. In this case the constant prediction is computed as the average target value of the examples in a leaf node, the prediction function is thus the mean function. The splitting criterion is again impurity reduction where the impurity measure in this case is the empirical variance.

Kramer [Kra96] introduces structural regression trees, a generalization of regression trees where the inner nodes can contain first-order literals or their conjunction. The algorithm requires a background theory to make the growing process feasible and to allow for non-determinate literals (literals that introduce new variables that can be bound in several alternative ways).

5.3.3 Model Trees

When using a linear regression rather than the mean function as the prediction function, the resulting trees are called model trees. This term can also be applied to tree-based models with more complicated prediction functions like a neural network learner, although this is seldom the case.

Different methods have been proposed to grow and prune model trees. Karlic [Kar92] proposes to use linear regression in the growing as well as in the pruning process. For the growing process this means to use another impurity measure I:

$$I(S) = \sum_{i:s_i \in S} (y_i - g(\vec{x}_i))^2 \tag{5.2}$$

where $g(\vec{x}_i)$ is the predicted value for the measurement vector $\vec{x}_i$ according to the regression plane and $y_i = \vec{x}_i$ is the observed value. The same impurity measure is used during the post pruning process where $\vec{x}_i$ and y_i are taken from the validation set

As this method is quite computationally demanding, Quinlan [Qui92] suggests introducing the linear models only during the pruning stage while the tree is grown using the mean target value as prediction function. This is much more efficient as the linear model has only to be computed for each leaf node and not for each candidate leaf node during the growing process. However, this method is less consistent from a theoretical point of view as the choice of the splits in the tree should depend on the prediction function used.

Torgo [Tor99] has proposed a third alternative. A regression tree is grown and pruned using the standard method described above. After both steps, a model is fitted to the training data in the leaf nodes. This procedure of course only makes sense if a variety of different models like linear models, polynomial models or kernel models can be used. Using a validation set or cross-validation, it can be experimentally determined for each leaf node which model predicts the data best.

Figure 5.3 shows a simple model tree. To get an impression of the performance of the above mentioned algorithms in comparison with standard regression trees for ideal data, we have generated 4000 training and 2000 test examples from this model tree by randomly choosing the variables x,y and z in the interval [0,100]. Table 5.1 summarizes the performance of the following four learning algorithms on this data: (1) using linear predictions for the growing as well as for the pruning process [Kar92], (2) using constant predictions in the growing as well as the pruning process (using a standard regression tree) [BFOS84], (3) using a constant prediction in the growing as well as in the pruning process and replace it with a linear prediction afterwards [Tor99], and (4) using a constant prediction in the growing process and a linear prediction in the pruning process [Qui92]. We used a depth limit stopping criterion with a maximal depth of three and allowed for regression over all three input variables x, y and z

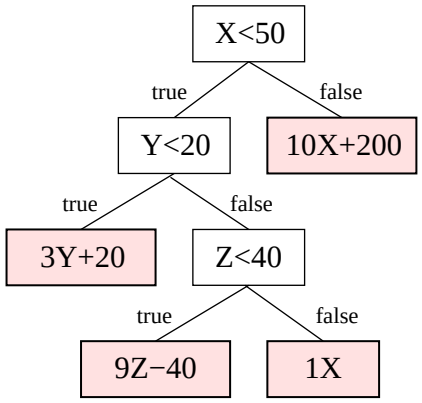

Figure 5.3: A simple model tree.

	1	2	3	4
absolute training error (s)	0.45	45.1	8.49	8.49
absolute test error (s)	0.67	46.8	9.49	9.49
quadratic training error (s^2)	42.10	3563	425	425
quadratic test error (s^2)	214	4082	663	663
learning time (s)	325	75	75	75
number of rules	5	6	6	5

Table 5.1: Performance comparison of the four different regression-/model tree learning algorithms described in the text.

On the synthetic data, algorithm 1 gives by far the best predictions, but takes more than four times as long as the other algorithms. Algorithm 2 results in the weakest prediction, while algorithms 3 and 4 do not differ significantly with respect to the prediction quality.

5.3.4 Implementation Details

For the experiments, we have implemented a generic tree learning algorithm. The algorithm can be parameterized by different stopping criteria, splitting criteria, prediction functions and post-pruning strategies. This design allows to view decision-, regression- and model tree learning as different instances of the same tree learning algorithm as suggested by Breiman et al. [BFOS84].

The package is implemented as C++ class library and contains classes for the depth limit stopping criterion, the minimal support stopping criterion,

the impurity reduction splitting criterion, the error reduction splitting criterion, the majority vote prediction function, the mean prediction function, the single-variate linear regression prediction function, the multi-variate linear regression function, and reduced-error tree post-pruning.

Our implementation can generate lists of rules from a learned tree, prune these rules and store them in a text file. The package also provides a rule interpreter which can be used to load and process the rules, e.g. in order to estimate execution costs of various navigation actions.

5.4 Learning Action Models

Projecting navigation plans involves the prediction of the cost, that is the duration of navigation actions. This is straight forward for most actions. The costs of a **TurnTo** command for example can be estimated by a linear function of the angle the robot is to turn. More difficult, however, is the prediction of the costs for an **MDPgoto**, an **ApproachPoint**, or a **SetTarget** command. The time the robot needs to execute these tasks depends on the distance to the target or goal point and the local surroundings of the robot. In free space, for example, the robot will translate faster than in a narrow passage.

The main challenge here is to efficiently compute a feature language which is suited to compactly describe a navigation action and allows a learning algorithm to predict the time it takes to execute the action with sufficient accuracy. We address this problem by using the optimal path to the action's goal point together with a clearance map of the environment to derive features that compactly describe the action.

The time the robot needs to arrive at the goal position (within some given accuracy) has to be estimated based on previous experience with executing this kind of task. As the time the robot needs to execute the navigation task depends on the shape and length of the path to the goal state G, we use a set of features derived from that path.

The set of features contains the path length, the path curvature, that is the ratio of the Euclidian distance to the target and the path length, the initial velocities of the robot, and the initial angle of the robot towards the path and the target. In addition, we use features derived from the main axis clearance map of the environment.

To compute the main axis clearance map for a grid map like the one shown

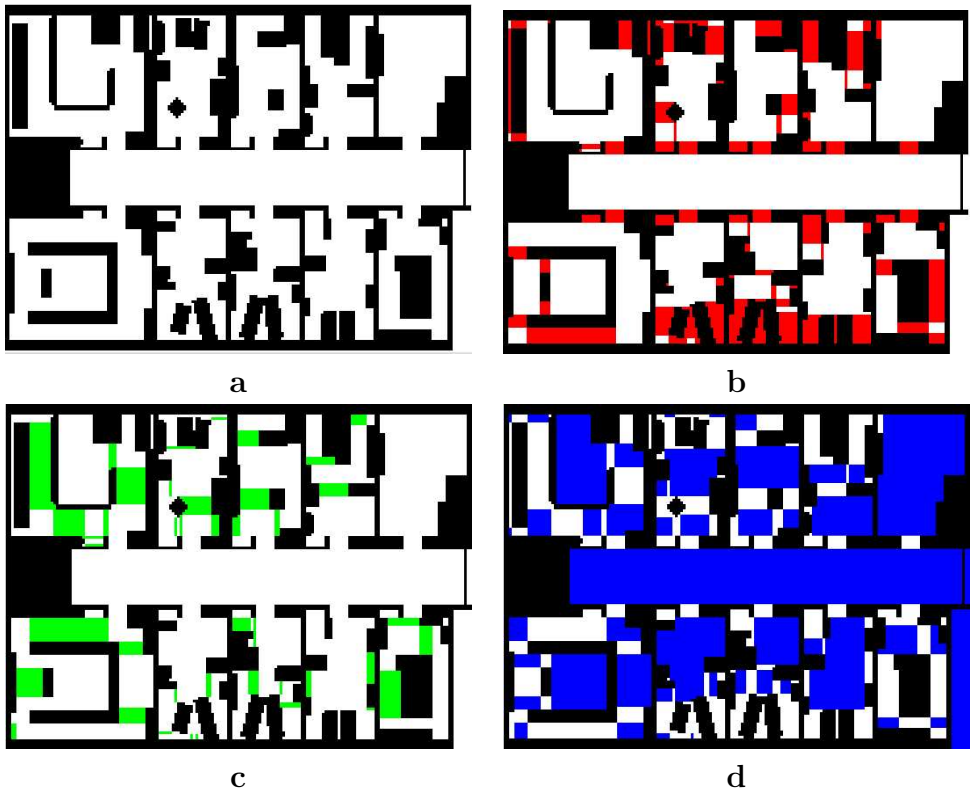

Figure 5.4: An environment segmentation based on the main axis clearance map. (a) The floor plan, (b) narrow passages, (c) passages, and (d) free passages.

in Figure 5.4, we compute the clearance, that is the distance to the closest obstacle, in the four main directions 'up', 'left', 'right' and 'down' for each grid cell s in the map, which yields $c_u(s), c_l(s), c_r(s)$ and $c_d(s)$. The four clearance values can be computed by a simple growing algorithm. The clearances along the x-axis and the y-axis can be defined as follows.

$$c_x(s) := \frac{c_l(s) + c_r(s)}{2} \quad \text{and} \quad c_y(s) := \frac{c_u(s) + c_d(s)}{2}$$

It is straight forward to use c_x and c_y to define a segmentation of the operation environment into walls, narrow passages, passages, and free passages. Figure 5.4 visualizes this segmentation.

The environment segmentation can be used to compute additional features suitable to describe a navigation action and to predict its duration. The three features **narrowPassageCounter**, **passageCounter**, and

feature	explanation
pathLength	length of the path in centimeters
pathCurvature	quotient of path length and distance to target
angleToTarget	angle to the target point
angleToPath	angle to first point on the path
startTvel	initial translational velocity
startRvel	initial rotational velocity
counter	number of grid cells on the path
narrowPassageCounter	number of path cells in a narrow passage
passageCounter	number of path cells in a passage
freePassageCounter	number of path cells in a free passage
numberOfSegments	number of segments on the path
narrowPassageSegments	number of narrow passage segments
passageSegments	number of passage segments
freePassageSegments	number of free passage segments

Table 5.2: Feature language used for the plan projection.

freePassageCounter count the number of grid cells on the path that lie in narrow passages, passages, and free passages, respectively. The features **narrowPassageSegments**, **passageSegments**, and **freePassageSegments** count the number of narrow passage segments, passage segments, and free passage segments along the path. Table 5.2 summarizes the features used in the experiments.

5.5 Scheduling Navigation Tasks

This section introduces the projection-based scheduling system PBS used in APPEAL. PBS schedules asynchronously arriving user requests for navigation tasks in order to maximize a given reward function taking deadlines, expected execution time, and task priorities into account. PBS performs a heuristic search in the space of all possible task orderings to find the schedule that maximizes the robot's expected reward where the expected reward is computed based on its projection.

Scheduling navigation tasks is the problem of deciding on an order in which user-specified navigation tasks are to be carried out in order to maximize the expected robot performance. The robot performance might depend on many factors like missed deadlines, task priorities and expected execution times.

Many interesting mobile robot applications require the robot to schedule its navigation tasks. For example, in a long-run experiment in the *Deutsches Museum* in Bonn where the mobile robot RHINO had to give museum tours, users could specify exhibits they wanted the robot to explain next. This could be done both via a web interface as well as via the on-board touch screen. In this scenario, the robot had to decide in which order the exhibits are visited taking into account (a) that no user should have to wait too long, (b) that the requests of users physically present in the museum should have higher priority than those of web users, and (c) that the time needed for one task is not constant, but varies significantly with respect to where the robot starts to execute the task. Employing the mobile robot for office delivery tasks poses similar problems.

In this section, we introduce the scheduling system PBS, which is specially designed for the scheduling of navigation tasks and addresses this kind of problems. The scheduler is based on the idea to express the above-mentioned constraints as reward function and to search for the schedule that maximizes the expected reward. In this decision-theoretic formalization of the problem, the robot can trade-off different and possibly conflicting goals while taking into account the uncertainty with respect to whether and to which degree these goals are achieved.

PBS schedules asynchronously arriving new navigation tasks as follows. The system considers all possible insertions of the new task into the current schedule. Each resulting schedule is projected to evaluate its expected utility and the one with the highest expected utility is selected. This procedure is clearly not optimal as for the scheduling of n navigation tasks only $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ instead of $n!$ schedules are considered. With the heuristic of inserting the action with the most urgent deadline first, however, this results in a good anytime behavior. This is important as the robot might be forced to commit to the execution of some action before the search for the most promising schedule has been completed. It is straightforward to make the search process optimal. However, as we are more interested in the evaluation of different projection methods and action models rather than different search heuristics, we stick to the heuristic described before .

During the search, each schedule is projected to compute its expected utility. The reward for the navigation task i executed in state s at time t is defined as

$$r(s,t,i) = \delta_{s,t}(i)R(i) - d(s,i) \tag{5.3}$$

where $\delta_{s,t}(i)$ is 0 if the task deadline was missed and 1 otherwise, $R(i)$ is the reward for the successful execution of task i and $d(s,i)$ is the time it takes

the robot to execute the task.

The reward for executing schedule $\pi = [\pi(1), ..., \pi(n)]$ in state s at time t is given by

$$r(s, t, \pi) = \sum_{i=1}^{n} r(s_{i-1}, t_{i-1}, \pi(i)) \tag{5.4}$$

Here, s_i is the state of the robot after executing $\pi(i)$ in state s_{i-1} with $s_0 = s$ and $t_i = t_{i-1} + d(s_{i-1}, \pi(i))$ with $t_0 = t$.

For the projection, the expected duration $E(d(s, i))$ and the probability of a missed deadline $P(\delta_{s,t}(i))$ are used to estimate the expected reward.

$$E(r(s, t, i)) = P(\delta_{s,t}(i))R(i) - E(d(s, i)) \tag{5.5}$$

The prediction of $d(s, i)$ is based on the ideas described in Section 5.4.

5.6 Experimental Results

In this section, we describe two types of experiments. In the first experiment, we compare projection-based scheduling with first-in-first-out scheduling and urgent-first scheduling. In the second experiment, we perform an experimental comparison of different projection methods in two scheduling domains.

Experiment 1

In the first experiment, projection-based scheduling is compared to two other scheduling methods, first-in-first-out scheduling (FIFO scheduling) and urgent-first scheduling. FIFO scheduling executes the tasks in the order they arrived. In urgent-first scheduling, the task with the closest deadline is executed first.

For predicting the duration $d(s, i)$ of task i in state s we use a simple linear prediction function learned from execution data.

$$\textbf{duration} = 0.044615 * \textbf{pathLength} \tag{5.6}$$

$P(\delta_{s,t}(i) = 0)$, the probability that the task deadline will be missed, can be estimated based on the predicted duration of a navigation task i, $E(d(s, i))$ and the deadline of task i, $D(i)$, as follows.

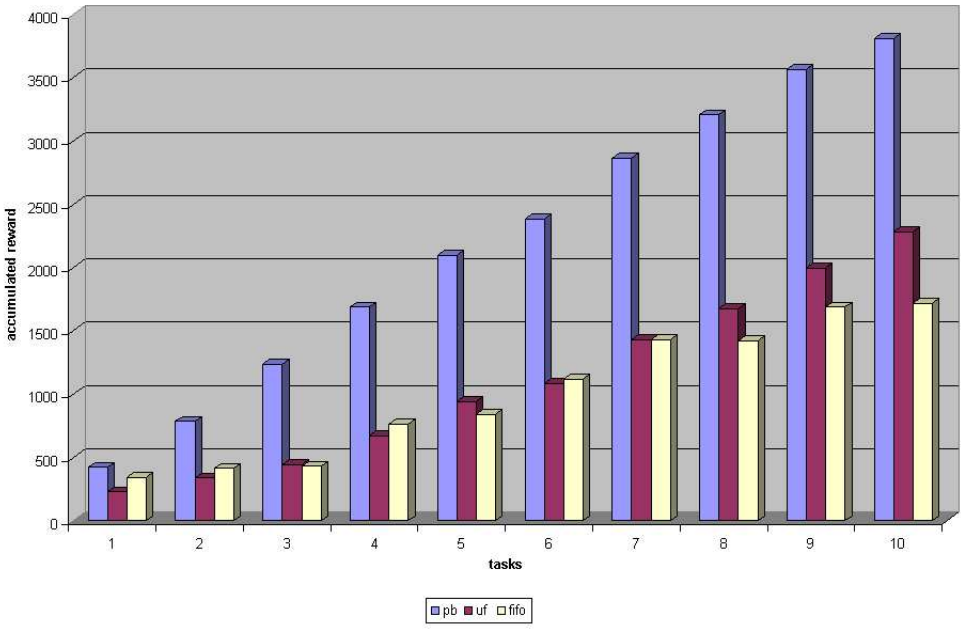

Figure 5.5: The robot performance (accumulated reward) in the first scenario with (a) projection-based scheduling (PB), (b) urgent-first scheduling (UF), and (c) FIFO scheduling (FIFO).

$$P(\delta_{s,t}(i) = 1) = \begin{cases} 0 & \text{if } t + E(d(s,i)) \geq D(i) \\ 1 & \text{else} \end{cases} \tag{5.7}$$

The projection method is deliberately simple in this case. The purpose of this first experiment is to show that even using these primitive projective models, projection-based scheduling outperforms FIFO scheduling as well as urgent-first scheduling. In the following experiments, we will introduce more sophisticated projection methods.

Figure 5.5 shows the accumulated reward of a simulated B21 robot for a sequence of 10 scheduling tasks applying the three scheduling algorithms mentioned above. Each scheduling task consists of a sequence of 10 navigation tasks randomly drawn from a probability distribution. The distribution governs

1. the time between the arrival of two successive queries,

2. the time available for one task (in the current experiment randomly

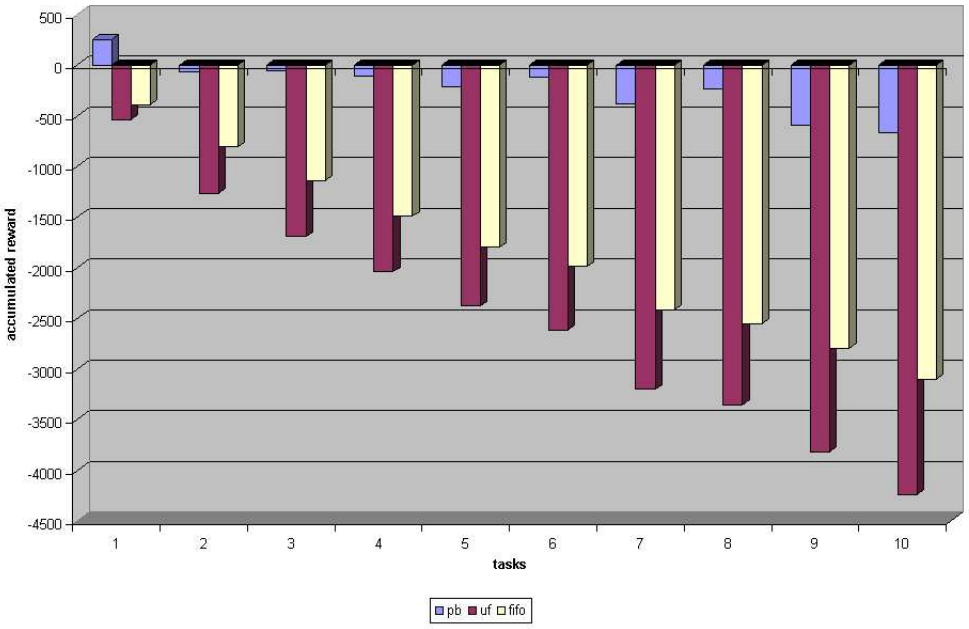

Figure 5.6: The robot's performance in the second scheduling scenario with (a) projection-based scheduling (PB), (b) urgent-first scheduling (UF), and (c) FIFO scheduling (FIFO).

drawn from the interval $[300, 1000]$ with equal probability),

3. and the target coordinates of each task.

Projection-based scheduling dominates the other two scheduling methods. It has an average reward of 380.7 in contrast to 172.0 and 228.2 for FIFO scheduling and urgent-first scheduling. This corresponds to a performance gain of 121% and 66%, respectively. The significance probability of both pairwise comparisons is far below 0.001 and the performance improvement therefore clearly statistically significant. Although urgent-first scheduling has in average a higher average reward than FIFO scheduling in this scenario, the improvement is not statistically significant with respect to strict standards. The significance probability is 0.1242. The significance probabilities have been computed with a randomized paired t test which is applicable in this case as all three scheduling methods have been tested on the same sequence of navigation tasks. Please refer to Appendix A for details on this test.

The second run, depicted in Figure 5.6, differs from the first run only in the time available for one task. It is randomly drawn from the interval $[150, 300]$

with equal probability. In this run, again projection-based scheduling dominates both, FIFO scheduling and urgent-first scheduling, but FIFO scheduling in this case performs also significantly better than urgent-first scheduling. Projection-based scheduling has an average reward of -66.5 in contrast to -424.3 for urgent-first and -310.3 for FIFO scheduling which means a performance improvement by a factor of 6 and 4.5 respectively. The significance probabilities are 0.001 and 0.002. The significance probability of the pairwise comparison of FIFO scheduling and urgent-first scheduling is 0.005.

In both runs, the fact that projection-based scheduling outperforms the other two scheduling methods is clearly statistically significant. In the first run, it is in general not difficult for the robot to meet the deadlines. Instead, the robot arranges tasks in a way that the traveling time is minimized. In the second run, projection-based scheduling predicts that it cannot meet all deadlines and explicitly reasons about which deadlines to miss in order to maximize the probability that other deadlines can be met.

Experiment 2

In the second experiment, we perform an experimental comparison of different projection methods. We are especially interested in the performance gain achieved by more expressive action models and by taking the uncertainty of these models explicitly into account when projecting whether or not a deadline will be missed.

In the previous experiment, the probability of a missed deadline was estimated using Equation 5.7. If we assume that the prediction error is normally distributed around $d(s,i)$ with a variance of $\sigma(s,i)^2$, we compute the probability of meeting the deadline $D(i)$ of task i as follows.

$$P(\delta_{s,t}(i)=1) = \int_{-\infty}^{D(i)} \frac{1}{\sqrt{2\pi\sigma(s,i)^2}} e^{-\frac{(x-d(s,i))^2}{2\sigma(s,i)^2}} dx \tag{5.8}$$

Here, $\sigma(s,i)^2$ is the mean squared training error of the learned model. For a given schedule or schedule prefix $\pi = [\pi(1),...,\pi(n)]$, the error accumulates like follows.

$$\sigma(s,\pi)^2 = \sum_{i=1}^{n} \sigma(s_{i-1},\pi(i))^2 \tag{5.9}$$

In this equation, s_i is the state after executing $\pi(i)$ in state s_{i-1} with $s_0 = s$.

Figure 5.7 gives an example of this computation. The deadline of the current task is at 215, but the expected termination time is 230 with an (accumu-

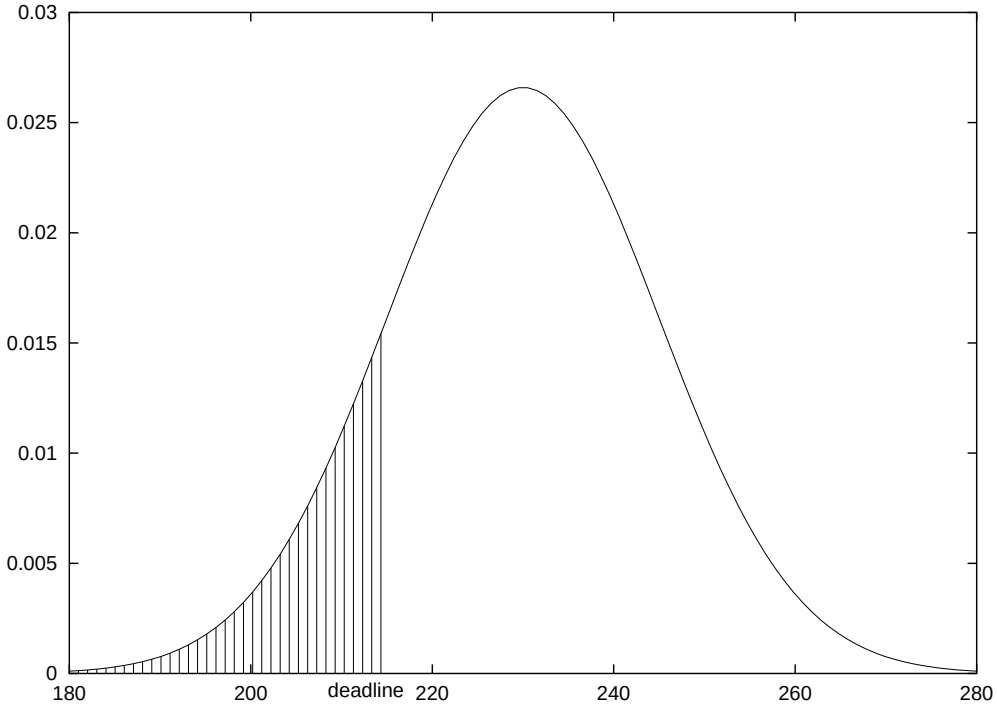

Figure 5.7: Probability that the robot does not miss the deadline at 215.

lated) predicted mean squared error of 15^2. The area of the hatched region is the probability that the deadline is met nevertheless.

Please note that using this model we get a non-zero probability not to miss the deadline even for deadlines in the past. We could escape this problem using the log-normal distribution.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}x}\, e^{\frac{-(\log x-\mu)^2}{2\sigma^2}} \tag{5.10}$$

However, as this anomaly has negligible effect in practice, we stick to the normal distribution.

To improve prediction accuracy, we consider multi-variate linear regression rather than single-variate regression to predict the time the robot needs for executing the task. Equation 5.11 shows the function learned from previously collected execution data. Crossing a narrow passage is almost five times more expensive than crossing a passage and even nine times more expensive than navigating in free space.

$$\begin{aligned} \text{duration} = 5.27 * &\textbf{narrowPassageCounter}+ \\ 1.11 * &\textbf{passageCounter}+ \\ 0.58 * &\textbf{freePassageCounter} \end{aligned} \tag{5.11}$$

In the first run of this experiment, we compare the robot performance using four different projection methods.

DPSVLR Deterministic projection based on single-variate linear regression, the method used in the previous two experiments. The expected time needed to execute the navigation task i in state s is estimated based on Equation 5.6. The probability of a missed deadline is estimated based on Equation 5.7.

PPSVLR Probabilistic projection based on single-variate linear regression. The expected time needed to execute the navigation task i in state s is estimated based on Equation 5.6. The probability of a missed deadline is estimated using Equation 5.8.

DPMVLR Deterministic projection based on multi-variate linear regression. Equation 5.11 is used to estimate $d(s, i)$. The probability of a missed deadline is estimated based on Equation 5.7.

PPMVLR Probabilistic projection based on multi-variate linear regression. Equation 5.11 is used to estimate $d(s, i)$. The probability of a missed deadline is estimated based on Equation 5.8.

For most scheduling scenarios, e.g. for the two scenarios described above, the performance of the projection methods does not have a significant impact on the robot performance. This is especially true if the task deadlines are in general not critical. In this case, it is most important to minimize the overall navigation distance in order to optimize the robot performance.

Only in case that there are many critical deadlines, and tasks vary significantly with respect to the reward the robot can gain by their successful execution, it is important for the robot to precisely project the schedule execution. In this experiment, we therefore consider a scenario where the robot has to schedule ten tasks and where the rewards for successfully executing a task are randomly drawn from the interval $[100, 1000]$ with equal probability. As task deadlines are every 35 seconds, the robot is forced to ignore certain tasks in order to be able to successfully execute more important tasks.

A vs. B	DPSVLR	PPSVLR	DPMVLR	PPMVLR
DPSVLR	-	0.9911	0.9701	0.9993
PPSVLR	0.0089	-	0.2652	0.8622
DPMVLR	0.0299	0.7348	-	0.9765
PPMVLR	0.0007	0.1378	0.0235	-

Table 5.3: Significance probabilities for the first run of Experiment 2. The significance probabilities are computed using the randomized one-tailed paired t test as described in Appendix A. For computing the differences the performance of algorithm B (col) is subtracted from the performance of algorithm A (row).

Figure 5.8 and Figure 5.9 show the reward for a sequence of 20 navigation tasks with the four scheduling methods. A statistical analysis of the data reveals that (a) PPSVLR performs significantly better than DPSVLR, (b) DPMVLR performs significantly better than DPSVLR, and (c) PPMVLR performs significantly better than DPMVLR with respect to a significance level of 0.05. However, (d) PPMVLR does not perform significantly better than PPSVLR with respect to the same significance level. Table 5.3 summarizes the significance probabilities of these pairwise comparisons which were again performed using a randomized paired t test.

This first run demonstrates that there are scheduling scenarios where the projection methods used have a significant impact on the robot performance. It suggests that more precise prediction functions as well as probabilistic plan projection help to improve robot performance. The careful analysis of the experiment data, however, also seems to suggest that switching from deterministic to probabilistic plan projection in this case is more useful than improving the prediction accuracy.

Till now we have limited our comparison to linear models to demonstrate the utility of the task features **narrowPassageSegments**, **passageSegments**, and **freePassageSegments** in combination with a multi-variate linear regression. In the rest of this chapter, we compare linear models to model trees.

Table 5.4 summarizes the prediction accuracies of different model trees depending on the complexity of the tree and the depth limit used during learning. Training and test errors are computed for sets of 486 and 209 examples. The comparison shows that the use of a multi-variate linear regression prediction functions as opposed to single-variate linear regression prediction functions improves the prediction accuracy considerably.

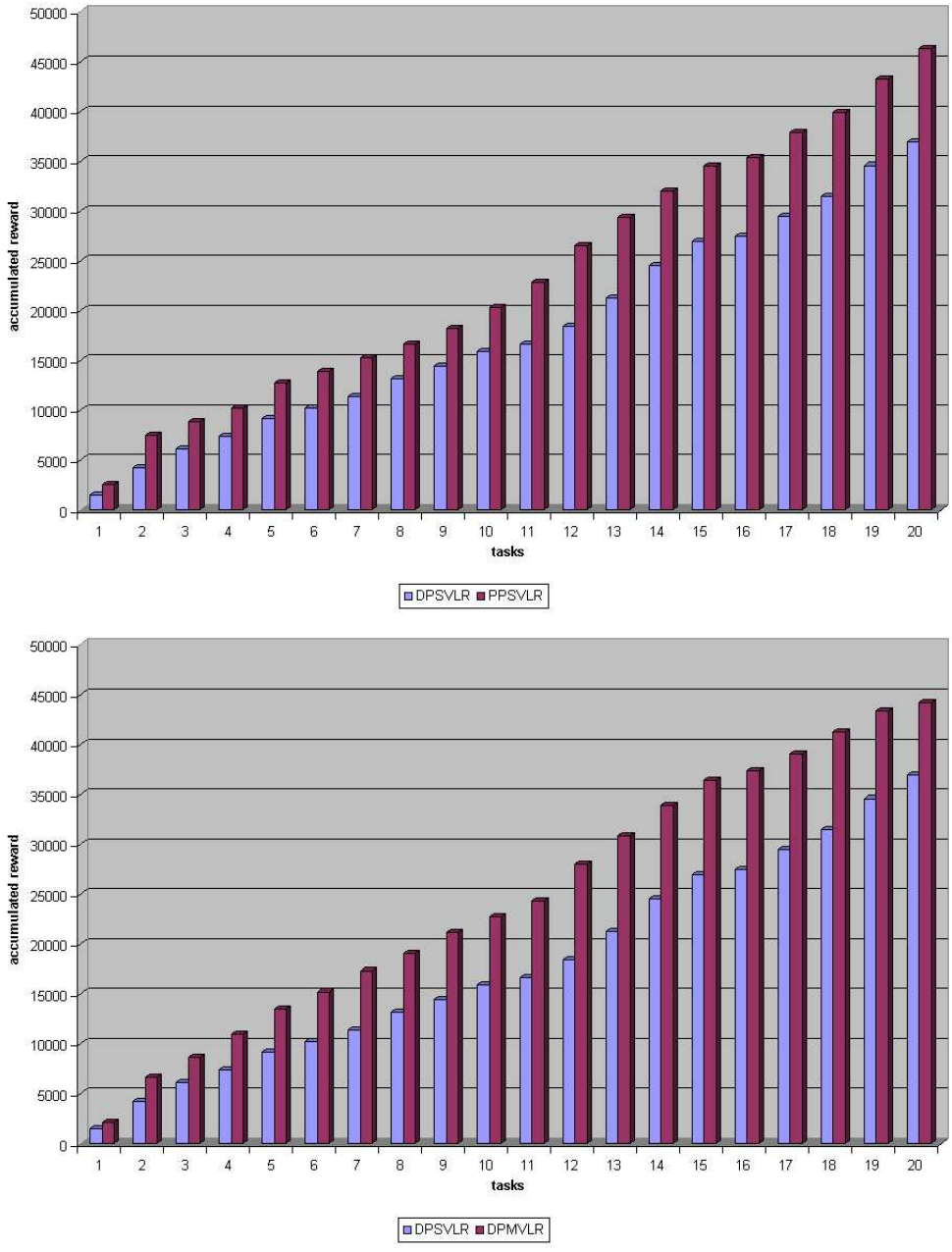

Figure 5.8: Performance comparison between (a) DPSVLR and PPSVLR (top) and (b) DPSVLR and DPMVLR (bottom).

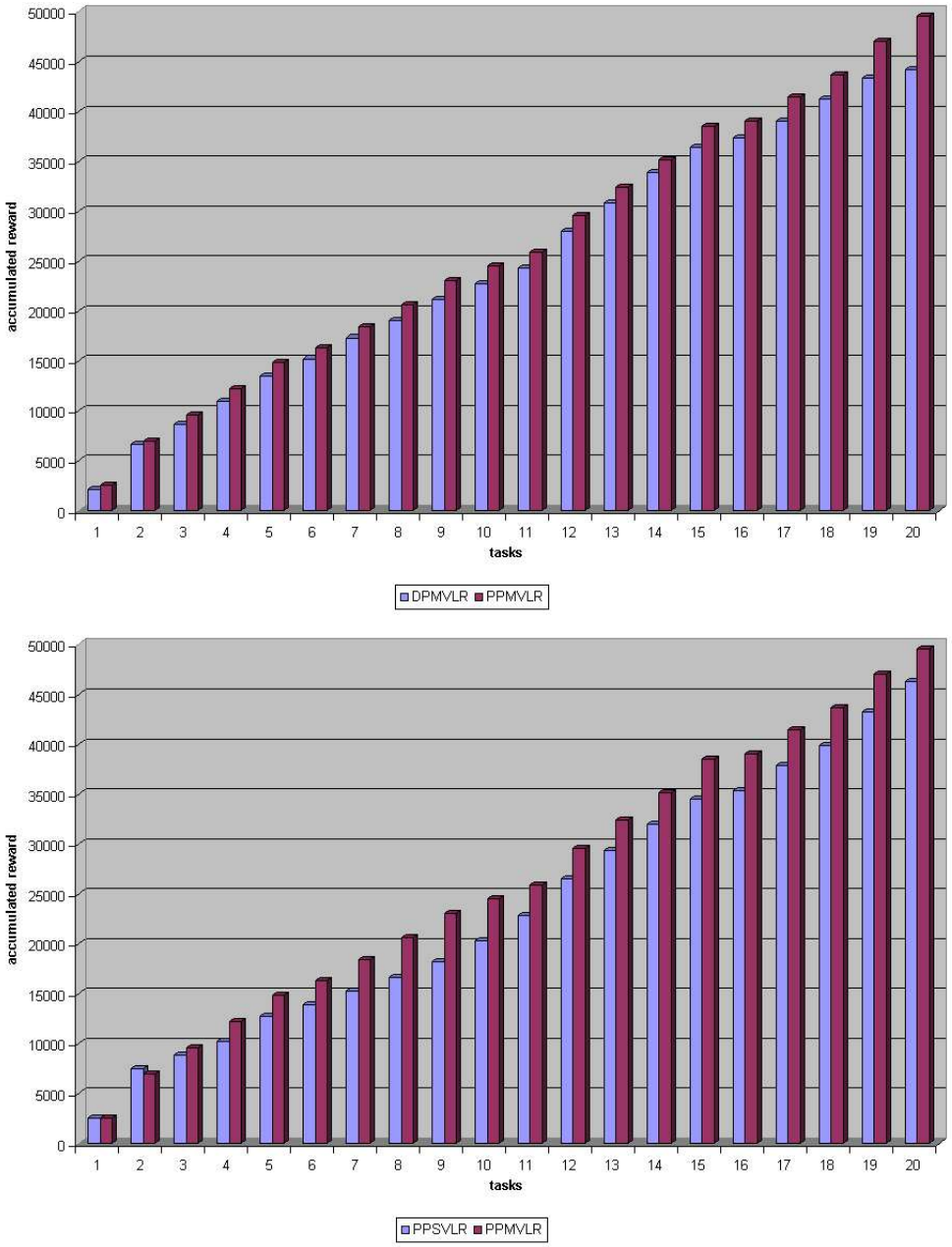

Figure 5.9: Performance comparison between (c) DPMVLR and PPMVLR (top) and (d) PPSVLR and PPMVLR (bottom).

single-variate linear prediction function					
depth limit	mean absolute error		mean squared error		# of rules
	training	test	training	test	
0	10.9199	12.0282	191.011	235.475	1
1	8.58043	9.53059	136.336	173.414	2
2	8.25315	9.2648	122.276	150.979	4
3	8.11783	9.09834	116.276	152.228	5
4	8.11783	9.09834	116.276	152.228	5

multi-variate linear prediction function					
depth limit	mean absolute error		mean squared error		# of rules
	training	test	training	test	
0	6.75188	6.65504	77.1874	80.1471	1
1	6.16721	6.35078	67.6448	77.0089	2
2	5.73725	5.92305	62.2236	70.5484	4
3	5.49640	5.76475	58.5757	67.8078	6
4	5.49640	5.76475	58.5757	67.8078	6

Table 5.4: Prediction accuracy of the model tree learner with single-variate linear regression prediction function (top) and multi-variate linear regression prediction function (bottom). The mean absolute training error, the mean absolute test error, the mean squared training error, the mean squared test error, and the number of rules depending on the depth limit used by the depth-limit stopping criterion.

In a second run of the previously described experiment, we compare the performance of four projection methods based on multi-variate linear regression prediction (MVLR) and model trees with multi-variate linear predictions (MVMT). Both models are applied in combination with deterministic and probabilistic projection methods. In the following, the methods will be abbreviated as DPMVLR, PPMVLR, DPMVMT, and PPMVMT using the same naming convention as before.

Table 5.5 shows the significance probabilities for the pairwise comparison of the four projection methods. They are again computed using a randomized paired t test. DPMVMT performs significantly better than DPMVLR, PPMVMT, however, is not significantly better than PPMVLR (with respect to a significance level of 0.05). The performance gain achieved by using probabilistic instead of deterministic projection is statistically significant again. Like in the previous run, these results seem to suggest that switching from

A vs. B	DPMVLR	PPMVLR	DPMVMT	PPMVMT
DPMVLR	-	0.9756	0.9705	0.9995
PPMVLR	0.0244	-	0.3306	0.8890
DPMVMT	0.0295	0.6694	-	0.9754
PPMVMT	0.0005	0.1110	0.0246	-

Table 5.5: Significance probabilities for the second run of Experiment 2.

deterministic to probabilistic projection is more useful than using a more accurate prediction function.

Appendix B.1 shows the rules used by the MVMT projector. The robot has learned to discriminate tasks where it has to cross one or no door from those where it has to cross at least two doors (**narrowPassageSegments** < 1.5). The path length is also an important feature as tasks where the robot has to travel longer distances, generally involve longer portions of travel in the hallway which tends to be faster.

5.7 Summary

One of the purposes of symbolic plans in hybrid robot control architectures like APPEAL is to increase the robot performance. The focus of this chapter was to examine how plan projection can increase robot performance for the online scheduling of navigation tasks.

We have presented an approach to plan projection which is based on learned models of the robot behavior. We apply model tree learning, a generalization of regression tree learning, to predict the time it takes the robot to execute an action. To ease the learning task, we have developed an expressive feature language to compactly describe navigation tasks. It can be efficiently computed using path planning algorithms and a segmentation of the robot environment, which is based on main axis clearance maps.

The chapter introduces the projection-based scheduling system PBS. PBS schedules asynchronously arriving user requests for navigation tasks in order to maximize a given reward function taking deadlines, expected execution time, and task priorities into account. PBS performs a heuristic search in the space of all possible task orderings to find the schedule that maximizes the robot's expected reward where the expected reward of a schedule is computed based on its projection. It is a transformational planning system which starts

to execute the current best plan immediately and transforms the plan as soon as a new best plan has been determined. Using this technique, sophisticated schedules can be computed without having to wait until the planning process terminates.

In the experimental evaluation, projection-based scheduling was shown to outperform FIFO and urgent-first scheduling by far in two different scheduling scenarios. The scheduling domain is also well suited to investigate which impact the accuracy of the prediction function has on the robot performance. From the experiments performed, we conclude that model trees are well suited for learning action models as they combine the benefits of memory-based learning and general function approximation. Model trees in combination with linear regression are ideally suited to predict the duration of a navigation action based on a compact description of the shortest path to the action's target position.

Chapter 6

Plan Projection for Action Selection

This chapter demonstrates the application of MDP planning techniques for the selection of task expansions in the HTN planning framework introduced in Chapter 4. This requires an estimate of the time the robot needs to execute an action in a given situation. Like in the previous section, this function is learned from execution data using model trees.

The generation of a navigation plan has to be fast as the robot cannot move until it has computed a plan. In consequence, the MDP planning should not be performed when selecting the next expansion. We suggest to perform transformational planning instead. The robot generates a default plan, which it can start to execute immediately. While executing the plan, the robot reasons about alternative courses of action that might result in a better navigation performance and transforms the plan as soon as it has found one.

The experiments demonstrate that MDP execution planning in combination with transformational planning techniques is well suited to improve mobile robot performance substantially and significantly. In extensive simulation experiments, we show that the MDP-based planning of plan transformations performs equally well as a set of carefully designed hand-coded transformation rules. Additional real world experiment shows that models acquired from simulation runs can be used to control a physical mobile robot.

6.1 Introduction

In Chapter 4, we have introduced the HTN planning framework and argued that it is well suited to represent choice points during the plan execution process. The selection of the best operator expansion in a given situation, however, might have a significant impact on the robot performance. This chapter discusses how the action selection can be based on MDP planning.

The selection of the best operator expansion has to be fast as the robot cannot move until it has computed a plan stub. We therefore suggest applying transformational planning techniques to deal with this problem. The robot generates a default plan by applying the default expansion to each operator. This does not involve any reasoning and is therefore very efficient. While starting to execute the plan, the robot can reason about alternative courses of action that might result in a better navigation performance. The plan can be transformed accordingly as soon as a better plan has been determined.

Plan transformation is achieved by replanning, that is, removing all operators from the task tree up to the one that is to be expanded in a different way, re-expanding this one in the desired way, and then applying the default expansions to all other tasks until a new plan stub has been generated. Please note, that the robot does not have to stop plan execution as the previously selected behavior stays activated until a new behavior is selected. This is the reason why we talk of plan transformations rather than replanning.

Besides exception handling and task sequencing, action selection is the third important aspect of plan execution. In the previous chapter, we have demonstrated the application of transformational planning to the online scheduling of navigation tasks. It is based on the online projection of alternative schedules. Besides the selection of the next navigation task to execute, the selection of the next intermediate target point is an important choice point in the plan execution process. We discuss how action selection can be formally described as a *Markov Decision Process*. Action selection is thus the second application of projective planning techniques to the problem of plan execution discussed in this thesis.

Markov Decision Processes allow not only to reason about the costs of actions but also about the expected successor state of the agent after executing an action. We can therefore take the probability of an action failure explicitly into account in this framework as was demonstrated in Chapter 2. However, only very seldom do action failures on higher levels of abstraction occur in the context of Appeal. In this chapter, we are therefore more concerned with *metric uncertainty* rather than *probabilistic uncertainty*. This uncertainty

has to be handled when learning the reward function. We show that model trees are well suited for this learning task.

The experiments show that the MDP-based plan transformations improves the robot performance significantly. Using the MDP-based transformational execution planning, the robot performs equally well as with a set of carefully designed hand-coded transformation rules. We also investigate how the robot's maximal planning horizon limits the possible performance improvement.

The remainder of this chapter is structured as follows. After discussing some related work in the area of reinforcement learning and MDP planning in Section 6.2, we formally describe the problem of selecting the best operator expansion as MDP. Section 6.3 shows how the plan execution problem can be formalized as MDP. In Section 6.4 we discuss how an optimal utility function can be computed for the continuous state MDP using a graph structure of the state space. In Section 6.5, we describe both simulator and real world experiments to evaluate plan execution based on MDP planning. We conclude in Section 6.6.

6.2 Related Work

In this chapter we consider the plan execution problem as a Markov Decision Process (MDP). MDPs have been formally introduced in Chapter 2. They provide a mathematical model of a situation in which an agent acts in a stochastic environment in order to maximize its expected accumulated reward. The effects of its actions might be uncertain, that is, the state of the agent after executing the action might not be predictable with certainty. The environment, however, is completely accessible to the agent and can thus be determined with certainty. A solution of an MDP is a policy, a mapping from states to actions. An optimal policy is a policy that maximizes the agent's expected accumulated future reward. Please refer to Chapter 2 for further references on this topic.

Reinforcement learning is a framework for learning optimal policies for an MDP. In contrast to unsupervised learning (e.g. clustering) and supervised learning (e.g. classification or regression), reinforcement learning is self-supervised. It learns from feedback in the form of incidental reward. Reinforcement learning solves the temporal credit assignment problem, the problem of propagating a reward backwards over time, by either learning a globally consistent utility function or by computing an action-value function.

There are different well-known reinforcement learning algorithms like temporal difference learning [Sut88] and Q-learning [WD92]. While temporal difference learning (TD-learning) updates a utility function defined on the state space, Q-learning directly operates on an action-value function Q that assigns a value $Q(s, a)$ to each state-action pair (s, a). For each experience (s, a, s', r) Q-learning performs the following update of the action-value function

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \max_{a'} Q(s', a') - Q(s, a)) \tag{6.1}$$

Here s' is the observed new state of the agent and r is the reward the agent received. α is the learning factor which should be decreased over time. Please note that Q-learning does neither require an explicit model of the robot's actions nor an explicit representation of the reward function R. It can thus be applied in completely unknown environments.

Another way to solve an MDP using learning techniques is to learn the reward function and the action model from observation in a first step and then compute the optimal policy using some dynamic programming algorithm like value iteration [Bel57] or policy iteration [How60]. This kind of algorithms is especially well suited when computation is considered to be cheap and real-world experience costly. If the model acquisition phase and the planning phase are interleaved, this kind of reinforcement learning is called certainty equivalence [KLM96] or adaptive dynamic programming (ADP) [RN95]. Kaelbling et al. also mention refinements of this idea like Sutton's Dyna architecture [Sut90], prioritized sweeping [MA93] and real-time dynamic programming [BBS95]. We will refer to this class of algorithms as model-based reinforcement learning algorithms as they all learn the action model and the reward function in a learning step and then explore the learned models in a planning step.

The action selection method based on MDP planning to be presented in this chapter can be considered as some kind of model-based reinforcement learning. It is well suited in our case as the same model can be applied to different instances of the plan execution problem. The model can be updated whenever new examples are available e.g. because the robot has been transferred to a new environment.

For large or continuous state and/or action spaces, neither the utility function nor the action-value function can be stored explicitly, that is, in a table. Function approximation algorithms have to be applied to store these functions implicitly to solve the structural credit assignment problem, the problem of how to propagate the reward spatially across states so that similar states cause the agent to take similar actions. Many different approaches have

been considered to perform this kind of input generalization. Incremental learning algorithms like neural network learning have for example been successfully applied in combination with Q-learning. Kaelbling et al. [KLM96] as well as Sutton and Barto [SB98] give a comprehensive overview of different approaches to input generalization in reinforcement learning.

Rather than generalize the utility function or the state-action function, in our approach we apply general action models to compute optimal policies for special instances of the execution problem. Although the problem is continuous in nature, for each instance of the problem only a small number of states has to be considered and the utility function can thus be represented explicitly in a graph structure. As the computation is relatively fast, the optimal execution policy for each new instance can be computed on demand.

There are many applications of reinforcement learning to mobile robot navigation problems [Lin93, Mat94, HP99]. Dorigo and Colombetti [DC94] discuss many practical aspects of mobile robot reinforcement learning. Among other things, they take up the idea first introduced by Mahadevan and Connell [MC92] to split tasks into their constituent tasks and learn policies for each of them separately. In contrast to Mahadevan and Connell they also consider the problem of how to combine the different policies.

For robot learning, it is considered as very costly to generate new experience. To mitigate this problem many different approaches like progress estimators [Mat94] and the combination of planning and learning techniques [Moo93] have been suggested. For the problem of plan execution the path to the goal provides a very useful progress estimator. We will discuss in more detail how path planning in the MDP framework simplifies learning in our application.

With respect to the application domain of plan execution, the work of Morisset and Ghallab [MG02b, MG02a] is most similar to ours. They consider the task of selecting between different ways to perform a navigation task in a known environment. The modalities combine different algorithms for localization, path planning, and local navigation. They propose to select between these modalities using an optimal policy computed from an MDP. The state space consists of all nodes of a topological map taking the precision and confidence of the position estimate, the clutteredness of the local surroundings of the robot and the last selected modality into account. To discretize the state space, the continuous features in the above mentioned feature space are discretized. Given the discrete state space and the small number of navigation modalities, the probabilistic action model as well as the reward function can be estimated using the observed relative frequency of state transitions

as well as the average time needed to execute a modality and there is thus no function approximation involved. In our approach, we apply decision tree and model tree learning in order to automatically discretize the state space.

6.3 Action Selection as an MDP

In Chapter 4, we have argued that HTNs support the optimization of navigation performance by opportunistic plan transformations and fast replanning. However, the detection of opportunities for plan improvements is often difficult and the specification of a good detector requires much insight in the robot's operation. In Chapter 5, opportunities were detected by the combination of heuristic search and the forward projection of the execution of navigation plans. In this section, we will discuss how the detection of opportunities can be based on MDP planning which can be considered as another form of projective planning.

We consider the problem of expanding an **MDPgoto** task into a sequence of **ApproachPoint** tasks. To reduce the time the robot has to wait for the computation of an executable task, the planner starts to compute a default plan, which results from applying the default expansion to each pending task in the task tree. While executing the first elementary task in the task tree, the robot can reason about alternative expansions of tasks in the task tree. During this time, the behavior launched by the default plan stays activated until a new behavior is selected. As soon as an alternative task expansion has been determined that promises to improve the robot performance, the plan is transformed accordingly.

Plan transformations are based on some kind of prediction of the robot's future performance. In this chapter, we will use MDP planning to compute the best action in the current situation. If this action deviates from the action selected by default, it is replaced accordingly and a new plan is computed.

In our application, the selection of the best expansion of a pending **MDPgoto** task is equivalent to the selection of the best next intermediate target point. This selection has to consider the following trade-off: Often the choice of target points that are more distant allows the reactive navigation system to drive smoother trajectories. On the other hand, the target point should be close enough so that it is reachable using the reactive approach point behavior. Please note that the decision between short but difficult versus easy but long paths is already done on initialization of the **MDPgoto** task.

For the selection of the best next way point, we have to compute the optimal

sequence of way points. In the following we will formalize this problem as an MDP. In order to do so, we have to specify the state space, the action space, the probabilistic action model, and the reward function.

The State Space

The state space consists of all triples (x, y, θ) where (x, y) is the robot's position within a global coordinate system of the known environment and θ is the robot's orientation within this coordinate system.

The Action Space

The action space consists of all possible expansions of the **MDPgoto** task, that is, **ApproachPoint**(x,y,d) where (x,y) is a point on the path that is 1 meter, 2 meters, or 4 meters ahead on the optimal path to the goal and d is either 0.5 meters, 1 meter, or 2 meters, respectively. Our approach can be generalized to more than three actions, of course, but depends on a finite set of discrete actions.

The Action Model

After successful execution of a navigation task, we consider the robot to be at the closest point $P = (P_x, P_y)$ on the optimal path to the target position (T_x, T_y) that is at most d centimeters away from the target, facing the target. Here the approach distance d is a parameter of the **ApproachPoint** task. Figure 6.1 visualizes the computation of the projected position P after successfully executing the action.

We consider the optimal path from the robot's current position to its goal state as part of the specification of the MDP. It is computed using a grid map of the environment and is thus given by a finite sequence of way points $p = [p_0, p_1, ..., p_n]$. In our case, the path itself is computed by solving an MDP using value iteration. This, however, is not essential for the solution of the path execution MDP.

In our experiments reported in Section 6.5, we use deterministic action models where we assume that the robot always successfully executes the task. This is justified by the observation that an **ApproachPoint** command fails only very rarely due to the advanced trajectory evaluation mechanism described in Chapter 3 and the sophisticated failure recovery mechanism described in Chapter 4. In the experiments, an unrecoverable failure occurred

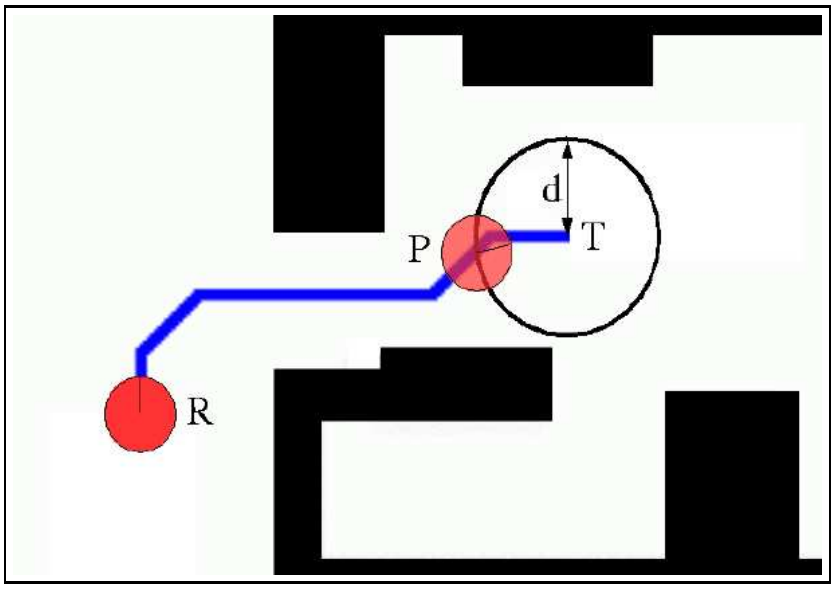

Figure 6.1: Projection of the position of a robot after successful completion of a navigation task.

with a probability of less than 0.001. Due to this fact, it is not only unnecessary to predict the cases where a failure might happen, it is also impossible without having an unrealistically large training set.

The basic idea to consider plan execution as an MDP, however, is not restricted to deterministic action models as demonstrated in Chapter 2. The experiment reported there was performed using the original RHINO control software where failure events occur much more often. However, we discussed the inherent difficulty to predict the new state of the robot after an execution failure occurred.

In the experiment described in Section 2.5, we therefore content ourselves with estimating the probability of an action failure and pessimistically assume in case of an action failure that the robot is where it started to execute the action. With this assumption, the problem of estimating the continuous probability function of the robot's new state is reduced to the problem of predicting with what probability an action failure occurs in a given situation. This, however, is a crude heuristic.

Because of the improvements over the original RHINO system described in Chapter 3 and Chapter 4 this kind of probabilistic uncertainty is transformed into metric uncertainty. This uncertainty has to be handled when learning the reward function from data.

The Reward Function

The reward for executing an action a in state s equals the negative cost of executing action a in state s, that is, $R(s,a) = -C(s,a)$. Here $C(s,a)$ is an estimate for the time the robot needs to execute the action a in situation s.

The time the robot needs to execute an action a in state s has to be estimated based on previous experience of executing this kind of task. As the time the robot needs to execute the navigation task depends on the shape and length of the path to the projected state P, we use a set of features derived from the path. In fact, we use the same feature language as introduced in Chapter 5.

For the learning task we apply model tree learning, both using single-variate and multi-variate linear regression prediction functions. As they cluster different situations according to the expected execution time of an action, model trees are well suited to represent the metric uncertainty inherent in this learning task.

6.4 Computing the Optimal Utility Function

In the setting described above where we only have finitely many actions and each action has only finitely many possible successor states, the MDP can be represented in a graph structure. It is constructed using a queue. Initially it contains only one state, the current state of the robot. It is said to have depth of d as it can be reached from the current state with a sequence of d actions. In each iteration a state s of depth d is removed from the queue. A new state s' of depth $d+1$ is added to the queue if some action a exists that results in state s' with probability $P(s'|s,a) > 0$ and s' is not a terminal state. If the graph already contains a node for state s', no new state has to be added to the queue. State s' is a terminal state if it is sufficiently close to the robot's goal state. The algorithm terminates if the queue is empty.

From the constructed graph an optimal utility function can be computed by one of the standard algorithms like policy iteration or value iteration. In the experiments reported below, we used value iteration. To check whether the constructed state graph already contains a given state s' slightly slows down the construction of the graph structure, but speeds up value iteration considerably. Please note that for the case of a deterministic action model there are more efficient algorithms to compute an optimal policy, e.g. Dijkstra's algorithm.

The standard version of the value iteration algorithm uses an infinite plan-

ning horizon. As replanning is performed continuously in our setting, it might be appropriate to limit the planning horizon to speed up computation. Limiting the planning horizon can be done by limiting the number of iterations performed in the value iteration process. This is not appropriate in our application scenario, however, as this method might not even find a policy leading to the goal state. In contrast, we already take the maximal planning horizon into account when computing the state space graph. For each state s of depth d all possible successor states are computed as described above only if $d \leq h$. If $d > h$ the only successor state of state s is determined by the projected state after successfully executing the default action **ApproachPoint**(2m). In the resulting state space graph, a policy is then computed using full value iteration. Using this method a navigation policy is computed that is guaranteed to lead to a goal state. In the experimental section, we will analyze how the choice of the planning horizon influences the robot performance.

6.5 Experimental Results

This section describes experiments carried out to evaluate the ideas developed throughout this chapter. We experimentally compare hand-coded transformation rules to MDP-based plan transformation, both in simulation and in a real world experiment. We compare the following action selection policies.

DEFAULT The robot always selects the default action for expansion, that is, **ApproachPoint**(2m). In contrast to the other policies, no further plan transformations are performed.

CODED The robot selects the next expansion according to five hand-coded rules. The next best action is computed based on the minimal clearance and the curvature of the path within some region around the robot. These rules have been carefully designed and can be regarded as expert knowledge.

MDP The robot computes the optimal utility function for the plan execution problem in the current situation. It selects the action with the highest utility in the current situation.

MDP(k) The robot computes the optimal utility function for the plan execution problem. The planning horizon in this case is limited to k. Again the robot selects the current best action according to the utility function.

As described above, the MDP planning is based on a model of the execution costs of an action a in a situation s. This model is learned from execution data. The necessary data is selected in an exploration phase. For this purpose, expansions of the **MDPgoto** task are selected randomly and their execution is monitored and logged into a database. From the logged data, a compact description of the actions can be computed using the feature language described in Section 5.4 which provides the training data for the learning problem. The data collected during plan execution comprises for each action start- and end state of the robot, the target state, whether the action was successful and the duration of the action.

For security reasons, the data selection process necessary for learning the model takes place in simulation. In this way, the robot cannot be damaged by any obstacles not visible for its sensors or by a hardware failure. In addition, in simulation, we can abstract away from all aspects of the control software that are not relevant for the selection of the next action, e.g. problems with the localization module. The real world experiment shows that the models learned in simulation are still useful when transferred to the real robot.

6.5.1 Real World Experiment

This section describes an experiment carried out using a PIONEER II mobile robot platform. The robot is to execute the sequence of four navigation tasks shown in Figure 6.3. The above-mentioned transformation policies are compared on basis of the average time the robot needs to complete the sequence of four navigation tasks. The tasks are chosen such that the behavior of the robot in four different situations can be analyzed. Entering an office, leaving an office, navigation in the hallway, and navigation from one office to another.

To generate the data to learn the reward function for solving the plan execution MDP, the robot repeatedly executes a sequence of 12 navigation tasks in simulation by randomly selecting a possible reduction of the current **MDPgoto** task. The tasks contain all possible ways to navigate from one point of Figure 6.3 to another. They thus include the four tasks that have to be carried out in the experiment.

For learning the reward function we apply model tree learning with (a) a greedy error reduction splitting criterion, (b) a linear regression prediction function, (c) a depth-limit stopping criterion, and (d) a reduced-error post-pruning criterion.

single-variate regression prediction					
depth limit	mean absolute error		mean squared error		# of rules
	training	test set	training	test set	
0	15.2789	15.2837	430.923	441.406	1
1	7.52703	7.40560	237.199	230.478	2
2	5.63955	5.52984	170.183	161.316	4
3	5.12713	5.03262	158.559	152.942	7
4	4.80145	4.71544	155.346	149.964	10
5	4.79675	4.70188	154.681	147.922	12
6	4.76554	4.72379	153.643	147.416	17
7	4.76554	4.72379	153.643	147.416	17

multi-variate regression prediction					
depth limit	mean absolute error		mean squared error		# of rules
	training	test set	training	test set	
0	6.65413	6.66462	201.443	201.396	1
1	5.74134	5.55187	162.434	141.487	2
2	5.21999	5.06955	152.820	141.992	4
3	4.78258	4.68903	145.947	138.958	8
4	4.72953	4.65368	144.469	139.642	11
5	4.63652	4.58086	143.199	137.894	14
6	4.60611	4.57695	142.628	138.854	17
7	4.57469	4.54026	141.232	137.747	24
8	4.56540	4.53918	141.175	137.765	27
9	4.56369	4.53271	141.152	137.720	27
10	4.56369	4.53271	141.152	137.720	27

Table 6.1: Prediction accuracy of the model tree learner with single-variate (top) and multi-variate (bottom) linear regression as the prediction function and a depth-limit stopping criterion. The absolute training error, the absolute test error, the mean squared training error, the mean squared test error, and the number of rules depending on the depth limit used for the learning.

Figure 6.2: The Pioneer II platform

Table 6.1 summarizes the prediction accuracies depending on the parameter of the depth-limit stopping criterion and the linear model used as prediction function. Mean absolute and mean squared errors are computed for a training set of 7072 examples and a test set of 3031 examples.

For the single-variate linear regression **pathLength** was used as the only regression variable. In the case of multi-variate regression, the three attributes **narrowPassageSegments**, **passageSegments**, and **freePassageSegments** were used. As you can see from the tables, after a depth limit of 6 and 9, respectively, no further increase of the prediction accuracy could be achieved. Although the model tree with multi-variate prediction has a slightly better prediction accuracy in the experiment, we used the rules derived from the model tree with single-variate regression, because they are more comprehensible. The rule below gives a flavor of the rules learned for this task.

IF	**pathCurvature** < 1.05	AND
	NOT **crossesDoor**	AND
	pathLength ≥ 110.0	AND
	pathLength < 130.0	
THEN	**duration** $= \frac{1}{23.99} *$ **pathLength**	

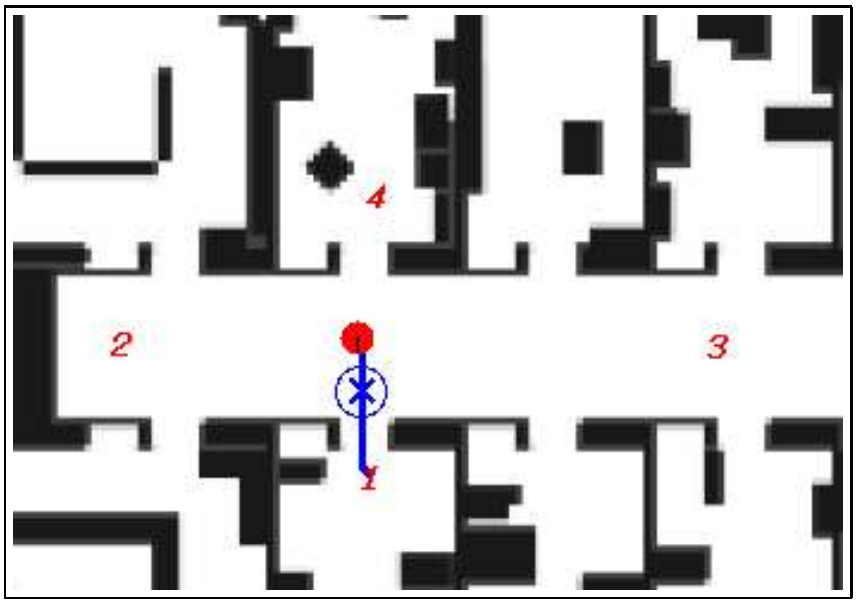

Figure 6.3: The four goal points for the real world experiments.

Appendix B.2 lists the 17 rules learned by the tree-based induction learner. The robot learns to discriminate situations where the robot only navigates in free space (**passageSegments** < 0.50 AND (**narrowPassageSegments** < 0.50), situations where the robot enters a narrow passage from a free passage or leaves a narrow passage and enters a free passage (**passageSegments** < 0.50) AND NOT (**narrowPassageSegments** < 0.50) AND (**freePassageSegments** < 1.5) and situations where the robot crosses a passage (**passageSegments** > 0.50) for all other situations. The learner subdivides these classes of situations using tests on the path length and the path curvature.

To asses the performance of the above-mentioned transformation policies, the sequence is executed 10 times using each of the methods. The application of **MDP(1)** results in a navigation performance of the PIONEER which is 41.88 % better than with **DEFAULT** and still 8.4 % better than with the hand-coded transformation rules of **CODED**. Both performance improvements are statistically significant with respect to a significance level of 0.05. In this example, the performance of **MDP** and **MDP(1)** does not differ significantly. The significance results are computed using a randomized paired t test (please refer to Appendix A for details).

Figure 6.4 shows the target points expanded by the two policies in a typical run. While the target points expanded by **DEFAULT** are equally spaced on

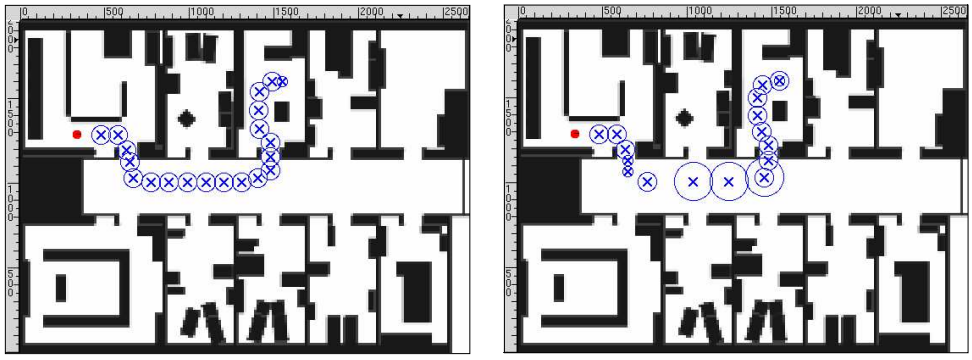

Figure 6.4: Target points selected by **DEFAULT** (top) and **MDP(1)** (bottom). Crosses indicate way points, circle size indicates travel speed.

the path, **MDP(1)** has learned to select a target further ahead on the path while crossing the hallway. When crossing a narrow passage it selects target points that are only 1 m ahead on the path.

The experiment shows that in the context of Appeal, MDP planning outperforms the hand-coded execution policy which can be considered as kind of expert knowledge. It also shows that the model learned from simulation data can be transferred to a physical robot without problems. In this experiment, however, there is no significant difference in the performance achieved by **MDP** and **MDP(1)**. This might be an artefact caused by the relatively small number of tasks in the experiment. In the next experiment, which is carried out in simulation, a larger number of tasks is examined to see whether we find a significant difference in performance in this case.

6.5.2 Simulator Experiments

In the simulator experiment, the robot has to execute the sequence of navigation tasks depicted in Figure 6.5. Each sequence is executed five times using the different policies. Besides comparing the average time the robot needs to accomplish the sequence of navigation tasks, we compute the statistical significance of an observed performance using a randomized paired t test.

We use a model learned from 7139 examples. In contrast to the first experiment, we apply multi-variate linear regression as the prediction function. From these examples, the robot learns a set of 11 rules to predict the time it needs to complete an **ApproachPoint** task.

Figure 6.6 visualizes the time the robot needs to accomplish the sequence

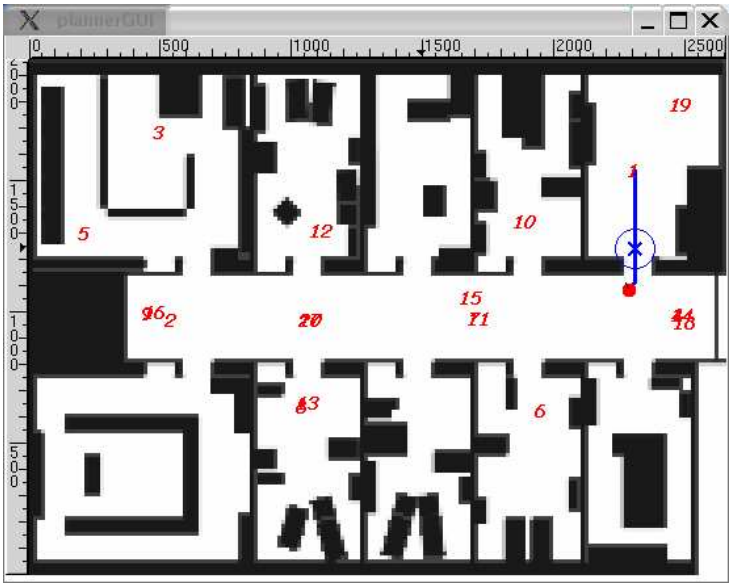

Figure 6.5: The twenty goal points used in the simulator experiment.

of navigation tasks using the different policies. As you can see, all planned execution policies as well as **CODED** outperform **DEFAULT**. The graphic also shows that a lookahead of more than one improves the performance considerably in this experiment. In figures, **CODED** performs 15.30 %, **MDP(1)** 2.22 %, **MDP(3)** 11.05 %, **MDP(5)** 10.22 % and **MDP** 12.70 % better than **default**.

Table 6.2 shows the significance probabilities for the pairwise comparison of the different methods, that is, the probabilities to obtain a mean pairwise difference higher than the observed mean pairwise differences under the assumption that algorithm A and algorithm B in fact perform equally well. The probabilities have been computed using a randomized paired t test which is discussed in some more detail in Appendix A. In contrast to the parametric t test it does not rely on the assumptions that each sample is drawn from a normal distribution and that the distributions have equal variance. We thus follow Noreen's advice [Nor89] to use it instead of the parametric t test because that frees us from worrying about the above mentioned assumptions.

With respect to the well-established significance level of α=0.05 **DEFAULT** is outperformed by **CODED**, **MDP(3)**, **MDP(5)** and **MDP**. At this level, the latter algorithms do not differ in performance. They all outperform **MDP(1)**, however, at a significance level of α=0.1. We conclude from these observations that MDP-based plan transformation can improve the robot

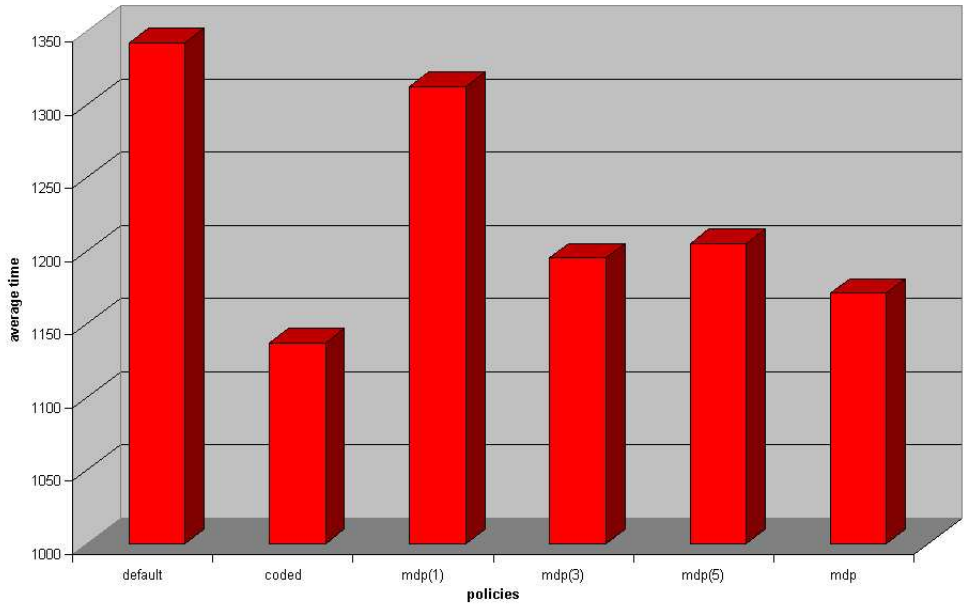

Figure 6.6: The average time needed to execute the sequence of 20 navigation tasks in the experiment computed from 5 passes.

performance significantly. A larger planning horizon significantly improves the plan quality only to a certain level k, in this experiment k=3.

6.6 Summary

This chapter has introduced the application of MDP planning to select expansions in the HTN planning framework discussed in Chapter 4. The reward function is learned using traces of previous plan executions. As MDP planning is a time consuming task, we combine MDP planning with an idea from transformational planning. Rather than performing the reasoning at the time of plan generation, the robot starts to execute a default plan which can be computed very efficiently. While executing the plan, the robot reasons about alternative courses of action that might result in a better navigation performance and transforms the plan as soon as it has found an alternative plan with a better expected plan quality.

The ideas presented in this chapter expand the ideas presented in Chapter 2. They broaden the ideas in at least two respects. First, by applying full value

A vs. B	DEF	CODED	MDP(1)	MDP(3)	MDP(5)	MDP
DEF	-	0.9998	0.6392	0.9917	0.9853	0.9963
CODED	0.0006	-	0.0068	0.0795	0.0378	0.2690
MDP(1)	0.3487	0.9909	-	0.9551	0.9012	0.9693
MDP(3)	0.0073	0.9193	0.0476	-	0.4189	0.6849
MDP(5)	0.0131	0.9618	0.0941	0.5840	-	0.6907
MDP	0.0031	0.7266	0.0313	0.3126	0.3009	-

Table 6.2: The significance probabilities for the pairwise comparison of the different policies. The significance probabilities are computed using the randomized one-tailed paired t test as described in Appendix A. For computing the differences the performance of algorithm B (col) is subtracted from the performance of algorithm A (row). The observed fact that corresponding entries do not sum to 1 is an effect caused by the randomization.

iteration in a state space graph rather than a lookahead of one. Second, by using model trees to learn the reward function rather than regression trees.

In contrast to this early experiment performed with the original RHINO control software, we apply deterministic action models. This is motivated by the fact that in APPEAL action failures on higher levels of abstraction occur only very seldom so that it is hardly possible to acquire accurate predictions for when these events occur. In principle, however, the same learning techniques can be applied to acquire probabilistic action models.

As argued in Chapter 2, it is an inherently difficult problem to obtain good predictions of the robot state in case of action failures even if failures can be predicted with reasonable accuracy. The enhancements over the original RHINO control software presented in Chapters 3 and 4 transform this kind of probabilistic uncertainty into a metric uncertainty and therefore simplify the learning task considerably. This was one of the major motivations for these enhancements. Model trees are well suited to handle metric uncertainty.

The experiments demonstrate that MDP execution planning in combination with transformational planning techniques is well suited to improve mobile robot performance substantially and significantly. The real world experiment shows that models acquired from simulation runs can be used to control a physical mobile robot.

Chapter 7

Learning Action Selection Rules

The generation of a navigation plan has to be fast as the robot cannot move until it has computed a plan. In the previous chapter, we have applied transformational planning techniques to deal with this problem. The robot immediately starts to execute a default plan, which can be computed very efficiently using default expansions. During execution, the robot projects alternative plans in order to detect opportunities for plan improvements. As soon as a promising alternative plan has been detected, the default plan is transformed accordingly.

In this chapter, we will explore a different approach to handle this problem: to learn action selection rules from offline projections. In order to learn them, we apply the same projection techniques as described before. In contrast to the transformational planning approach, they are applied offline to randomly sampled navigation tasks. The union of all the situation-action mappings computed this way forms the input of a rule learning algorithm. The learning algorithm generalizes from task-specific execution policies to general situation-action mappings.

The learned action selection rules are not only fast to process, but also well accessible for human inspection. Our experiments show that the learned action selection rules perform slightly worse than the projection-based plan transformation, although not significantly. The transparency of the learned action selection rules, however, probably outweighs the slightly worse performance.

7.1 Introduction

The experimental results of Chapter 6 have shown that the number of actions the robot looks ahead makes a difference only to some degree. For the environment used in the experiments, this seems to be a look-ahead of three. This result suggests that instead of performing expensive plan projections online, action selection rules can be learned from offline plan projection. This general idea is concretized in this chapter.

This generalization process can be regarded as speed-up learning. Experience from a planning process is memorized to avoid repeated computation of the same kind and to speed up the decision process in time critical situations.

In order to learn action selection rules from offline projection, we apply the same projection mechanisms as introduced in the previous chapter. They are applied to randomly sampled navigation tasks. The projection computes the optimal action a for each tuple (x, y, θ, x_G, y_G) where (x, y, θ) denotes the current pose of the robot and (x_G, y_G) denotes the goal of the robot. Each pair $((x, y, \theta, x_G, y_G), a)$ forms an input for the learning algorithm.

The training examples have to be given in a suitable feature language. Like in the previous chapters, we compute the feature language from the shortest path to the goal and the environment segmentation depicted in Figure 5.4. In contrast to the regression task of learning the expected reward when executing an action, the feature language has to be more expressive in this case. We use the length and type of the k first segments of the optimal path to the goal as additional features. In the experiments, we chose k=3. This is justified by the observation that in the experiments reported in a previous chapter a planning horizon of more than 3 did not result in a significant further performance improvement.

For the learning task, we apply two different learning algorithms: Decision tree learning which has been discussed in some detail in Chapter 5 and sequential covering. While decision tree learners partition the input space by a divide-and-conquer algorithm, sequential covering algorithms in each step separate a set of training examples from the set of all training examples for which the class can be predicted perfectly or with sufficient accuracy by a single rule. Therefore sequential covering algorithms are called separate-and-conquer algorithms. We introduce a sequential covering algorithm which computes a decision list, i.e. a list of rules that have to be processed in order. In the experiments, sequential covering algorithms perform slightly better in comparison with decision tree algorithms, but are less comprehensible. Section 7.3 will introduce this kind of algorithms in more detail.

Our experiments show that the learned action selection rules perform slightly worse than the projection-based plan transformation, although not significantly. The transparency of the learned action selection rules, however, probably outweighs the slightly worse performance as the action selection rules help to understand the robot's behavior.

The remainder of this chapter is structured as follows. After discussing some related work in the areas of explanation-based learning in general and speed-up learning in particular in Section 7.2, we introduce the class of sequential covering algorithms in Section 7.3. In Section 7.4, we describe how symbolic learning algorithms like decision tree learning and sequential covering algorithms are applied to the task of learning general action selection rules. A detailed experimental comparison of the action selection rules learned by different methods is given in Section 7.5. We conclude in Section 7.6.

7.2 Related Work

In this chapter, we propose to learn general action selection policies from examples. Each example is composed of a compact description of a navigation task as well as the action to be executed by the robot in the current situation. Although in terms of the learning techniques applied this is a classical supervised learning task , it is also closely related to *explanation-based learning* (EBL). In the following, we will examine the relation between learning explanations and explanation-based learning more closely and discuss related work in this field.

EBL [MKKC86, Thr95, RN95, Mit97] is motivated by the attempt to speed up an inference process by trying to learn from past experience. For first order learning, this principle can be explained considering PROLOG-EBG [KCM87] as an example. PROLOG-EBG works by proving first-order classifications from first-order descriptions of the examples and some relational background knowledge. From the proof tree a rule can be learned where the clause body is formed from the predicates in the tree leafs and the clause head is formed from the predicate in the root of the leaf. The rules learned in this way can themselves be used in the proof process and thus help to speed up inference.

If B denotes the background knowledge, D denotes the first-order description of the examples, C the description of the classifications, and H the learned hypothesis, PROLOG-EBG satisfies the following entailment constraints:

$$\text{(a) } H \wedge D \models C \text{ and (b) } B \wedge D \models H \tag{7.1}$$

and thus the entailment constraint

$$B \wedge D \models C \tag{7.2}$$

In general, however, EBL algorithms only satisfy the two constraints

$$\text{(a) } H \wedge D \models C \text{ and (b) } B \wedge D \wedge C \models H \tag{7.3}$$

from which the same conclusion cannot be drawn. In this more general formulation, the classifications contain information necessary to form the hypothesis H. As you can see from constraint 7.3 (b), the learner does not acquire any knowledge that would allow completely new inferences to be performed, but knowledge of a new quality. The new quality is that the knowledge helps to speed up the inference process and is more explicit. EBL like the learning reported in this chapter is a knowledge compilation process.

There are mainly three manifestations of EBL.

1. learning of inference rules [MKKC86, KCM87]
2. learning of plan libraries [Sus77, Sus90, Ham89]
3. learning of search control knowledge [LRN86, VCP+95]

EBG [MKKC86] and PROLOG-EBG [KCM87] are examples of algorithms for the first class of problems. Mitchell et al. have also applied explanation-based learning algorithms to learn stimulus-response rules [Mit90]. In the THEO architecture, a set of stimulus-response rules is applied to control a robot collecting cans in an office environment. Whenever there is no rule applicable to the current situation, the robot uses a general purpose planning system to compute a plan for this situation. Given the plan, a PROLOG system is used to explain the choice of action in the current situation. The generalization of the computed explanation forms a new stimulus-response rule.

The learning in THEO and APPEAL is closely related. Both systems are intended to speed up action selection and make the action selection mechanisms more explicit. In both systems, the planning and the generation of explanations are two different processes. While in THEO this is a deductive inference process, APPEAL uses an inductive inference process. This approach differs from ours in at least two important respects. First, the learning process in THEO depends on hand-coded action models, both for the planning as well

as the explanation process. Second, the planning system as well as the explanation system of THEO are fairly limited in that they cannot deal with uncertain action effects and actions with situation-dependent costs.

Action selection rules are the search control knowledge in the HTN planning framework. They are, however, not intended to speed up the computation of a correct plan, but to direct the search towards a plan with high quality, i.e. with a high expected performance. As argued before, it is quite simple to compute a working navigation plan, although much more difficult to compute a plan with a high expected performance. As action selection rules help to avoid costly online plan projections, they help to speed up the search for a performant plan. In addition, these rules transform declarative knowledge in a procedural form which is much simpler to understand.

Some of the learning algorithms applied in the PRODIGY architecture [VCP+95] have a very similar motivation. Veloso et al. consider learning algorithms designed to extend a strips-like planner which combines forward projection and backward chaining. They identify three types of learning tasks in this context: 1.) reducing the planning time, 2.) improving the quality of plans and 3.) refining the domain knowledge. QUALITY and HAMLET are two learning algorithms intended to improve plan quality.

In the QUALITY algorithm, a user-supplied plan and a plan generated by the current control rules are compared. The algorithm explains why the second plan is better than the original one (if it actually is). The algorithm identifies the choices that might have led to the better performance and makes them operational in form of a control rule. As the rule is induced from a single example the resulting rule tends to over-generalize. An incremental process adds more specific rules to override the too general rule in case of plan failures.

The HAMLET algorithm produces a set of rules by explaining the differences in quality and success of different plans in the space of all plans for some simple planning problem. In contrast to QUALITY, HAMLET therefore does not need any user guidance. The resulting rules again can be too general or overly specific. They are refined or generalized when false positive or false negative examples are encountered in new episodes. HAMLET thus acquires increasingly correct control knowledge.

Both algorithms are incremental learning algorithms. The same is true for PROLOG-EBG. The learning of action selection rules in APPEAL is not incremental. All action selection rules are generated at once. Although in general this might be a disadvantage, we avoid the danger to add more and more rules which at the end might even slow down the action selection process. In addition, batch learning algorithms like decision tree learning and sequential

covering are less vulnerable to overfitting in the case of noisy data.

Laird et. al [LRN86] apply explanation-based learning techniques to improve the general problem solving architecture SOAR [LNR87]. SOAR is based on a forward chaining production system. Conflict resolution rules can be learned to improve the problem solving behavior. The explanation-based learning in SOAR is intended as a cognitive model of human *chunking*, the association of chunks (expressions or symbols) into a new, single chunk. It aims at explaining a human's ability to move from problematic to routine behavior.

The learning approach presented in this chapter is intended in improving plan execution. While we are mainly interested to improve plan quality in terms of the average robot performance, Bennett and DeJong [BD96] are more interested in improving the robustness of a plan. They combine classical explanation-based learning with an approach to make robot plans more permissive. The technique, which they call permissive planning, applies knowledge-based diagnosis of execution failures. It is used to add and refine preference constraints on the parameters of the operator schemes. In the GRASPER system they apply permissive planning to improve robot manipulation tasks.

7.3 Sequential Covering Algorithms

While decision tree learners partition the input space by a divide-and-conquer algorithm, sequential covering algorithms in each step separate a set of training examples from the set of all training examples for which the class can be predicted perfectly or with sufficient accuracy by a single rule. Therefore sequential covering algorithms are called separate-and-conquer algorithms. The AQ system [Mic93] and the PRISM system [Cen87] are early implementations of this idea. The systems CN2 [CN89] and INDUCT [GC95] generalize the approach to noisy training data. FOIL [Qui90] and ICL [uRuL95] are systems for learning first-order concepts from relational data using sequential covering algorithms.

In classification tasks where one class is much more frequent than the other classes, rule sets generated by sequential covering algorithms can be much more compact than the ones generated by decision tree learning algorithms.

Suppose the boolean concept $f : \mathbb{N}^2 \to \mathbb{B}$ given by

$$f(x,y) = \begin{cases} \text{false} & x > 2 \wedge x < 5 \wedge y > 2 \wedge y < 5 \\ \text{true} & \text{else} \end{cases}$$

and the training set depicted in Figure 7.1.

1	+	+	+	+	+	+
2	+	+	+	+	+	+
3	+	+	-	-	+	+
4	+	+	-	-	+	+
5	+	+	+	+	+	+
6	+	+	+	+	+	+
	1	2	3	4	5	6

Figure 7.1: Training data for the function f.

Using a decision tree learner the training set is partitioned as shown in Figure 7.2 (a) which results in 5 partitions and consequently in five rules. Using the knowledge that *true* is the most frequent value of f, a sequential covering algorithm generates two rules which correspond to the definition of f. Figure 7.2 (b) shows how the generated rules partition the training set. The example shows that sequential covering algorithms are well suited to find and explain deviations from expectations.

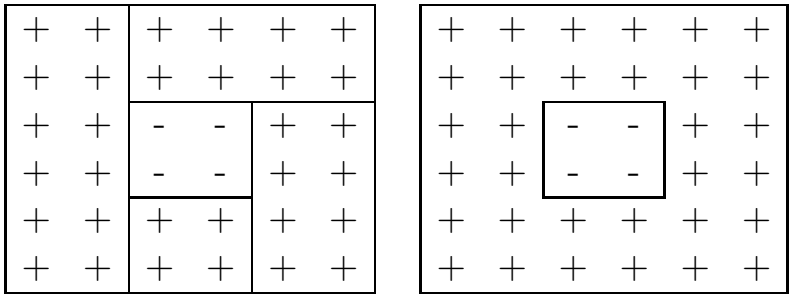

Figure 7.2: (a) The partitions of the input space generated by a decision tree learning algorithm. (b) The partitions generated by a sequential covering algorithm.

Algorithm 4 describes a basic sequential covering algorithm for learning decision lists (lists of rules that have to be processed in order). The algorithm is parameterized by two choices, the choice of the stopping criterion and the choice of the cutting criterion.

Algorithm 4 Sequential covering decision list growing

Require: a set of data points $E = \{(\vec{x}, y)\}$
  for all possible values y_i of y **do**
    while E contains instances of class y_i **do**
      create a rule R with an empty left hand side that predicts class y_i
      while the stopping criterion is not met **do**
        select the best test t with respect to the selection criterion
        add the test to the left hand side of R
        remove the instances covered by R from E
      end while
    end while
  end for

7.3.1 Stopping Criteria

For noise free training sets a good stopping criterion might be to stop if the instances in the set of examples E' covered by the current rule all belong to the same class, that is, the current rule is perfect. In the presence of noise, however, it might happen that no more attribute-value tests exist that have not been tested, although the rule is not perfect. An alternative criterion is to stop when the coverage of the rule (the number of examples to which it applies) falls below some threshold value θ.

7.3.2 Cutting Criteria

A simple but often used cutting criterion is to always select a test such that the accuracy of the new rule is maximal. Here, the accuracy of a rule is defined as $\frac{p}{n}$ where n is the number of examples covered by the rule and p is the number of examples that are classified correctly, that is, support the rule. This criterion, however, cannot discriminate between rules with the same accuracy but different coverage. A criterion, which can do this discrimination is the information gain criterion. It selects a test t such that the information gain of test t

$$I(t) = p[\log\frac{p}{n} - log\frac{P}{N}]$$

is maximal. Here $\frac{P}{N}$ is the accuracy of the rule before test t was added to the preconditions and $\frac{p}{n}$ the accuracy of the rule after t was added.

A third alternative is to always select the cut that maximizes the quality of the resulting rule. In the INDUCT system [GC95] the value of a rule r that

predicts class c with accuracy $\frac{p}{n}$ is computed as 1 minus the probability that a rule r' predicting the same class, has the same coverage n and a support $p' \geq p$ is generated randomly. This probability can be computed based on the hypergeometric probability distribution.

The main idea behind this probabilistic measure of rule quality is to define a rule extensionally, that is, by the coverage and support of the rule for a given set of examples. Assume a set of examples S of size N where P examples are of class c. Given a rule r that covers n examples of S and classifies p of these examples correctly, the probability to generate r by drawing n examples randomly from S (without replacement) is given by the hypergeometric distribution

$$H(N, P, n, p) = \frac{\binom{P}{p}\binom{N-P}{n-p}}{\binom{N}{n}}. \tag{7.4}$$

The probability to draw a rule r' with coverage $n' = n$ and support $p' \geq p$ by random sampling without replacement from S is therefore

$$M(r') = \sum_{p'=p}^{P} H(N, P, n, p') = \sum_{p'=p}^{P} \frac{\binom{P}{p'}\binom{N-P}{n-p'}}{\binom{N}{n}} \tag{7.5}$$

The computation of $H(N, P, n, p)$ is expensive. For large N, $H(N, P, n, p)$ can be approximated by the binomial distribution

$$B(N, P, n, p) = \binom{n}{p}\left(\frac{P}{N}\right)^{p}\left(\frac{N-P}{N}\right)^{(n-p)}. \tag{7.6}$$

This distribution can be computed more efficiently than the hypergeometric distribution which speeds up learning considerably. Witten and Frank [WF00] describe how the computation can be further accelerated.

7.3.3 Incremental Post Pruning

An obvious problem for sequential covering algorithms is that they tend to overfit the training data, especially when the stopping criterion requires perfect prediction. The solution as in the case of decision trees is to do post-pruning. The main difference here is that for decision trees, the pruning can be done after the growing is completed. Although in principle this is possible as well for sequential covering algorithms, when learning decision lists (i.e. lists of rules that are to be processed in a given order) it makes more sense to interleave the growing and the pruning process, which is known as incremental pruning [FW94]. To interleave growing and pruning has the

Algorithm 5 Sequential covering decision list growing and pruning

Require: a set of data points $E = \{(\vec{x}, y)\}$
  for all possible values y_i of target y **do**
    while E contains instances of class y_i **do**
      create a rule R with an empty left hand side that predicts class y_i
      while the stopping criterion is not met **do**
        select the best test t with respect to the cutting criterion
        add the test to the left hand side of R
      end while
      while the stopping criterion for pruning is not met **do**
        select a test from R with respect to the pruning method
        and remove it
      end while
      append rule R to the decision list
      remove the instances covered by R from E
    end while
  end for

advantage that rules are immediately adjusted to the right level of generality and the learning of subsequent clauses cannot be disturbed by the influence of an overly specific first rule. Algorithm 5 describes how this is done.

One possible choice for the pruning method is reduced-error pruning. Before starting the learning process, a validation set is split from the training set. As already mentioned, for learning decision lists the growing and pruning has to be interleaved. After a rule has been built, tests are removed from the preconditions of the rule until the accuracy of the rule with respect to the validation set cannot be further improved. The pruning starts from the last test added and works backwards along the rule. Please note that this greedy procedure will not necessarily select the best test to remove from the rule. Using the probabilistic measure for rule quality for pruning results in another pruning criterion. In our experiments, however, the pruning algorithm performed slightly worse using this measure rather than rule accuracy.

Algorithm 5 generates a decision list. To classify a given example, the rules in the decision list have to be processed in order. If the rule base is processed this way, every possible example is classified by exactly one rule.

7.3.4 Speeding Up Learning

If Algorithm 5 is used to build the decision list, the quality of the generated rules depends on the order in which the classes are processed. It is often a good choice to process the rarest classes first, more common ones later. Besides the accuracy of the learned rules, this improves the learning speed, especially if deviations from a default prediction are infrequent.

It is often useful to stop building rules for one class before all instances of this rule have been covered by some rule. In our implementation, we stop growing rules for a class as soon as a rule has been found that predicts class c but would perform better when predicting another class c'. In this case we skip the learned rule and continue learning for the next class in the class ordering. A more complicated criterion of when to stop learning rules for one class is based on the MDL principle [Coh95b].

7.4 Learning Action Selection Rules

In the previous chapter, it has been described how the computation of optimal plan execution policies can be understood as a Markov Decision Process. We have demonstrated how optimal plan execution policies can be computed for instances of the **MDPgoto** task. The computed policies map poses (x, y, θ) (on the optimal path to the goal position) to the optimal action in this situation which is an instantiation of an **ApproachPoint** task. The policy is therefore specific to the problem instance which is given by the goal state. In this section, we will describe how *general* action selection rules can be computed from a set of problem specific situation-action mappings.

A general action selection policy is given by a mapping from general state descriptions to instances of the **ApproachPoint** task where a general state description is independent both of the robot's current position and its goal position. It is given by a vector of features computed from the path to the goal. In the experiments, we have used the twelve features described in Table 7.1.

The first six features are computed based on the environment segmentation depicted in Figure 5.4 and the optimal path to the goal. They summarize the class as well as the length of the first three passages on the path. The feature **numberOfPassages** is defined as the number of different passages on the path and gives some measure of how difficult the navigation task is.

feature	description
firstPassage	class of the first passage
secondPassage	class of the second passage
thirdPassage	class of the third passage
lengthOfFirstPassage	length of the first passage
lengthOfSecondPassage	length of the second passage
lengthOfThirdPassage	length of the third passage
numberOfPassages	number of passages on the path
straightLinePath	distance the robot can translate forward without having to leave the path
angleToGoal	angle to the goal
angleToPath	angle the robot has to turn to face the first point on the path to the goal
pathLength	length of the path
pathCurvature	curvature of the path

Table 7.1: Features for a general state space description

The feature **straightLinePath** is defined as follows.

$$\mathrm{s} = \begin{cases} 0 & \text{if } \neg\exists\ 1 \leq i \leq n\ (\angle(\theta, R, P_i) < d) \\ |P_0, P_i|_p & \text{if } \exists\ 1 \leq i \leq n\ (\forall\ k \leq i(\angle(\theta, R, P_k) < d) \wedge \\ & \qquad \angle(\theta, R, P_{i+1}) \geq d) \\ |P_0, P_n|_p & \text{else} \end{cases} \tag{7.7}$$

where $P_0, P_1, .., P_n$ is the sequence of points on the optimal path to the goal $G = P_n$, $R = P_0$ is the current position of the robot, θ is the current orientation of the robot, and $\angle(\theta, R, P)$ is the minimum distance between point P and the line given by the robot position R and its orientation θ. $|P_i, P_j|_p$ denotes the path distance between two points P_i and P_j on the path and is defined as $\sum_{k=i}^{k=j-1} |P_k, P_{k+1}|$. In the experiments, we used a threshold of $d = 50.0$.

As the feature language is propositional, we have to decide on the maximal number of segments that are to be considered. An alternative approach is to characterize the path by a conjunction of logical predicates like

$$\text{path}(x) \wedge \text{length}(x, 1130) \wedge \text{curvature}(x, 1.2) \wedge$$
$$\text{pathSegment}(x1, 1, x) \wedge \text{isFreePassage}(x1) \wedge \text{length}(x1, 450) \wedge$$
$$\text{pathSegment}(x2, 2, x) \wedge \text{isNarrowPassage}(x2) \wedge \text{length}(x2, 70) \wedge$$
$$...$$

Using this first-order language, it is possible to describe paths of arbitrary length and complexity. To learn action selection rules from examples specified in a relational feature language, ILP algorithms have to be applied. Blockeel and deRaedt [BuR98] describe how to upgrade decision tree learning to first-order theories, Kramer [Kra96] describes a first-order version of regression tree learning. FOIL [Qui90] and ICL [uRuL95] are examples of first-order sequential covering algorithms.

As the experiments suggest, in our application the propositional language introduced above suffices. To learn a general action selection policy, we proceed as follows. In each iteration a navigation task, which is given by a start- and a goal state, is sampled randomly and the optimal policy for this task is computed using the algorithm described above. From each policy k state-action mappings are sampled and for each such mapping, the state description is computed based on the optimal path to the goal. The resulting feature vector together with the recommended action is then added to the training data. After collecting a sufficient number of examples in this way, any propositional classification algorithm like decision tree learning or sequential covering can be applied to the data to predict the best action in a given situation.

For the learned action selection rules, the number of the rules as well as the prediction quality of the rules depends on various factors. Besides the number of examples used for training, the maximal lookahead in the planning process as well as the maximal rule depth determine the performance of the learning algorithms. The numbers given in Table 7.2 show this interrelation. The first column lists the number of examples used for training and testing. 70 % of the examples form the training set and the remaining 30 % form the test set. 30 % of the training examples are used for the pruning set, the rest is used for the growing set.

Not surprisingly, the policies computed with a lookahead of one are easier to learn which can be read from both training- and test error. Sequential covering outperforms decision tree learning in all cases. The higher prediction accuracy, however, is bought by a much longer time required for learning. In addition, the sequential covering algorithm described in Section 7.3 generates

# of examples	planning horizon	maximal depth	# of rules	training error	test error	time (sec)
decision tree learning						
2000	-	5	16	0.311	0.396	5
2000	1	5	9	0.222	0.219	5
2000	-	10	62	0.244	0.321	6
2000	1	10	64	0.157	0.184	6
10000	-	5	17	0.310	0.320	119
10000	1	5	15	0.198	0.209	118
10000	-	10	111	0.230	0.267	133
10000	1	10	59	0.165	0.178	136
50000	-	10	113	0.240	0.246	2055
50000	1	10	85	0.162	0.170	2188
sequential covering algorithm						
2000	-	5	17	0.268	0.317	61
2000	1	5	10	0.210	0.229	46
2000	-	10	29	0.235	0.296	114
2000	1	10	26	0.134	0.209	142
10000	-	5	22	0.259	0.287	1733
10000	1	5	9	0.188	0.201	1239
10000	-	10	72	0.211	0.244	11167
10000	1	10	34	0.152	0.l69	2674
50000	-	10	166	0.178	0.193	262406
50000	1	10	58	0.137	0.141	104703

Table 7.2: Comparison of the two learning algorithms

IF	**lengthOfFirstPassage** ≥ 400.0	AND
	firstPassage=freePassage	
THEN	**expansion**=approachPointFar	
IF	**lengthOfFirstPassage** < 200	AND
	secondPassage=narrowPassage	
THEN	**expansion**=approachPointNear	
IF	TRUE	
THEN	**expansion**=approachPointMid	

Table 7.3: Four hand-coded rules that specify opportunities for plan improvements. The rules are arranged in a decision list, that is, have to be processed in order until the first applicable rule is found.

a decision list, that is, rules have to be processed in a given order. The resulting rule set is thus more difficult to understand.

Appendix B.3.1 and Appendix B.3.2 show sets of action selection rules learned from 10000 examples. The examples were generated applying value iteration with infinite planning horizon to the action selection MDP. In both cases, a depth limit of 5 was used. In the experiments described below, we use the rules learned from the data set containing 50000 examples using a depth limit of 10.

7.5 Experimental Results

In this section, we experimentally compare the performance of the action selection policies listed below. Each one is given by a set of action selection rules. As the set of rules can be processed efficiently, they can be applied already for plan generation and not only for plan transformation.

DEFAULT The robot always selects the default action for expansion, i.e. **ApproachPoint**(2m).

CODED The robot selects the next expansion according to five hand-coded rules. The next best action is computed based on the minimal clearance

and the curvature of the path within some region around the robot. These rules have been carefully designed and can be regarded as expert knowledge. The rules are the same rules used in the previous section. Throughout this section, however, they are used for task expansion rather than plan transformation.

NAIVE The robot selects the next expansion according to the three hand-coded rules shown in Table 7.3. The rules are arranged in a decision list, i.e. rules that have to be processed in order until the first rule is found that is applicable. In contrast to **CODED**, the rules are expressed in the feature language which is also used for the learning. They constitute a first guess of how the robot's performance might be improved.

DT The action selection rules learned form a set of n independently sampled plan execution problems. The best action is computed using **MDP** and the rules are learned by decision tree learning.

DT(k) The action selection rules learned form a set of n independently sampled plan execution problems. The best action is computed using **MDP(k)** and the rules are learned by decision tree learning.

SC The action selection rules learned form a set of n independently sampled plan execution problems. The best action is computed using **MDP** and the rules are learned by sequential covering.

SC(k) The action selection rules learned form a set of n independently sampled plan execution problems. The best action is computed using **MDP(k)** and the rules are learned by sequential covering.

To compare the above listed action selection policies, the robot has to execute the sequence of 20 navigation tasks shown in Figure 6.5, each sequence five times. We thus have the same conditions as in the experimental comparison of the plan transformation policies in Chapter 6.

Figure 7.3 shows the average time needed by the robot for completing the sequence of 20 navigation tasks. Compared to **DEFAULT**, **NAIVE** performs 3.74 %, **DT(1)** 1.79 %, and **SC(1)** 3.82 % better. The performance gain achieved when learning from the execution policies generated by full MDP planning is considerably higher: 10.75 % with decision tree learning and even 12.56 % with sequential covering. However, the highest performance gain is achieved when applying the hand coded rules of **CODED**. In this case the performance can be improved by 16.75 %.

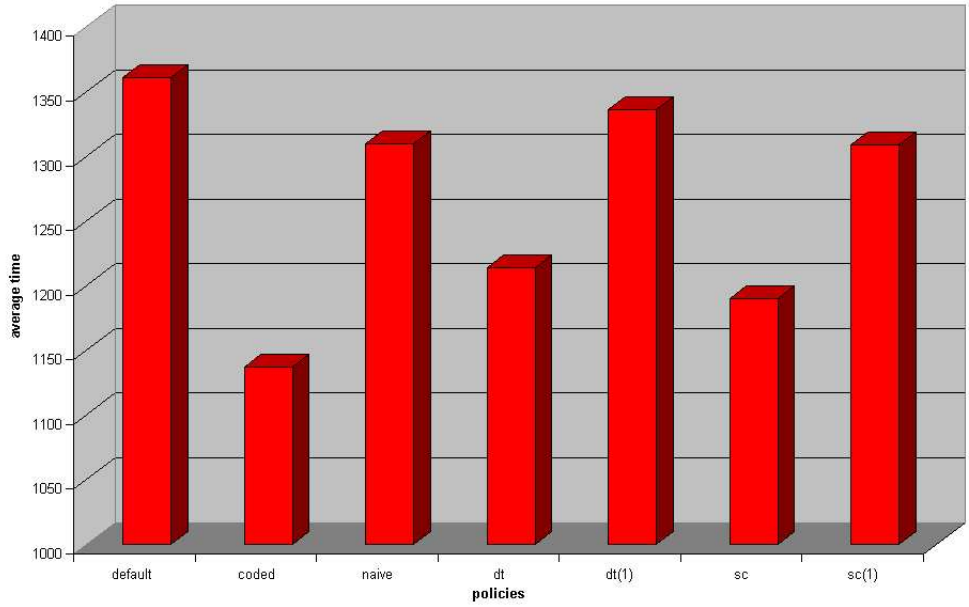

Figure 7.3: Comparison of the hand-coded and the learned action selection policies.

Table 7.4 shows the significance probabilities for the pairwise comparisons of the action selection policies. With respect to a significance level of α=0.05, **DT**, **SC**, and **CODED** outperform **DEFAULT**. The other action selection policies do not. With respect to the same significance level, **SC**, **DT**, and **CODED** perform equally well. The same is true for the three policies **DT(1)**, **SC(1)**, and **NAIVE**. **SC** and **SC(1)** perform better than **DT** and **DT(1)** respectively, although not significantly.

Figure 7.4 compares the learned action selection policies with the MDP-based transformation policies introduced in the previous chapter. The differences are not considerable and, as you can see from Table 7.5, not significant.

This observation can be interpreted as follows. Although the MDP-based action selection is more accurate, this advantage is compensated by the fact that the action selection rules can be processed much faster. In consequence, they can be used for plan generation as well as for plan transformation. Although the online plan projection results in a slightly better performance compared to the learned action selection rules, the rules are more intelligible and the behavior specification therefore more transparent. This advantage might in general outweigh the slightly worse performance.

A vs. B	DEF	NAIVE	CODED	DT	DT(1)	SC	SC(1)
DEF	-	0.7603	0.9992	0.9676	0.6232	0.9816	0.7454
NAIVE	0.2422	-	0.9939	0.8952	0.3477	0.9532	0.5025
CODED	0.0007	0.0046	-	0.1059	0.0055	0.1923	0.0099
DT	0.0295	0.1052	0.8883	-	0.0580	0.6694	0.0924
DT(1)	0.3881	0.6516	0.9952	0.9430	-	0.9741	0.6460
SC	0.0188	0.0440	0.7992	0.3283	0.0271	-	0.0480
SC(1)	0.2538	0.4960	0.9910	0.9108	0.3562	0.9547	-

Table 7.4: The significance probabilities for the pairwise comparison of the different policies. The significance probabilities are computed using a randomized paired t-test as described in Appendix A. For computing the differences, the performance of algorithm B (col) is subtracted from the performance of algorithm A (row). The observed fact that corresponding entries do not sum to 1 is an effect caused by the randomization.

A vs. B	DT	DT(1)	SC	SC(1)	MDP	MDP(1)
DT	-	0.0582	0.6736	0.0894	0.7234	0.0666
DT(1)	0.9404	-	0.9726	0.6530	0.9868	0.6126
SC	0.3334	0.0247	-	0.0450	0.6018	0.0284
SC(1)	0.9068	0.3515	0.9578	-	0.9735	0.4826
MDP	0.2730	0.0121	0.3974	0.0229	-	0.0298
MDP(1)	0.9244	0.3843	0.9731	0.5176	0.9676	-

Table 7.5: The significance probabilities for the pairwise comparison of the learned action selection policies and the transformation policies introduced in Chapter 6. The significance probabilities are computed using a randomized paired t-test as described in Appendix A. For computing the differences, the performance of algorithm B (col) is subtracted from the performance of algorithm A (row). The observed fact that corresponding entries do not sum to 1 is an effect caused by the randomization.

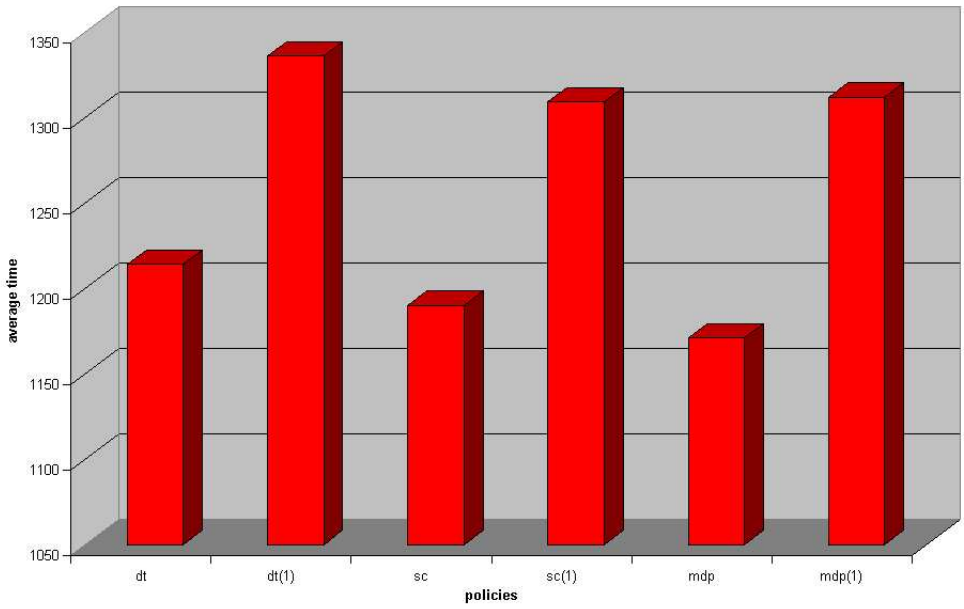

Figure 7.4: Comparison of the action selection policies and the transformation policies.

7.6 Summary

The execution planning discussed in the previous chapter is computationally demanding. This is the reason why we suggested using online planning of plan execution policies only for plan transformation rather than plan generation. In this chapter, we propose to learn action selection policies from offline execution planning. The action selection policies consist of lists or sets of rules and can be processed very efficiently. They are thus well suited to be used for plan generation.

The rules are learned by randomly sampling navigation tasks and applying the MDP-based execution planning introduced in the previous chapter to compute an optimal plan execution policy for each of them. With a suitable feature language, a learning algorithm generalizes from task specific execution policies to general situation-action mappings. We propose to use symbolic learning algorithms like decision tree learning and sequential covering algorithms which produce symbolic rules well accessible for human inspection.

The experiments show that when using these learned action selection rules, the robot performs almost as well as using online execution planning in

combination with plan transformations. The slightly worse performance is, however, probably outweighed by the advantage of more transparent behavior specifications. The rules learned with the sequential covering algorithm perform slightly better than those produced by decision tree learning, although this observation is not statistically significant with respect to the well-established significance level of $\alpha = 0.05$.

Due to the higher intelligibility, we suggest using action selection rules learned from offline planning rather than transformational online execution planning. However, this recommendation is restricted to the task of selecting intermediate target points. Other execution planning tasks – like the scheduling of navigation tasks – are more complicated planning tasks. In this case, comparably good execution policies cannot be specified as a set of rules.

Chapter 8

Conclusions

Throughout the thesis, we have addressed different aspects of the problem of plan execution in the context of mobile robot control. This problem results from the need to compose a mobile robot control system of reactive and deliberative components. Deliberative components tend to be slow and reason about complex, abstract, and mainly static models of the world. They are required for the robot to deal with situations it was not explicitly programmed for. Reactive components, on the other hand, are needed to guarantee a safe and reliable operation even in the case of unexpected situations that have not been foreseen by the deliberative system. Deliberative and reactive components often form two separate layers of a mobile robot control system. The problem of plan execution is the problem to arbitrate between these two layers.

We have examined this problem considering the RHINO control system as an example. The RHINO control system is well known for the consistent application of probabilistic algorithms to both map learning and localization. The high robustness of the system has been demonstrated in two early tour-guide projects in the Deutsches Museum in Bonn and the Smithsonian Institute in Washington D.C. Despite this fact, we could identify a large potential for performance improvement by introducing an intermediate layer to the RHINO control system that mediates between the reactive and the deliberative layer. The resulting three-tiered architecture is called APPEAL, Architecture for Plan Projection, Execution And Learning.

Like in the original RHINO software, the reactive layer is composed of a single module, which is mainly responsible for approaching local target points while avoiding obstacles and taking the robot's dynamic constraints explicitly into account. We have proposed to enhance the Dynamic Window Approach to

collision avoidance by using heuristic functions that are computed using local path planning. These heuristic functions are applied in order to guide the search for the next motor control commands. Besides improving the robot's performance considerably, this enhancement makes the robot's behavior more reliable, which simplifies its prediction.

The deliberation layer consists of a single module as well. A user interface allows a human user to specify navigation tasks with deadlines and priorities. The deliberation layer may also contain a symbolic planner to compute a partially ordered plan from any user specified goal. In the current system, however, a symbolic planner is not required as all reasoning tasks necessary for navigation planning can be performed by the execution planner which is part of the execution layer.

The core of the execution layer is an HTN plan representation. It represents the state of plan execution using Hierarchical Task Networks. The hierarchical representation of tasks and their potential refinements allow for an explicit representation of choice points in the plan execution process. The HTN representation allows for the efficient computation of default plans, the handling of execution failures on the right level of abstraction, and the reasoning about alternative courses of action.

The reasoning is based on learned models of the robot's possible actions. To compute an expressive feature language suitable to predict the execution costs of the navigation actions with sufficient accuracy, we use a feature language computed from an optimal path to the goal or target point of the robot together with its segmentation, which is based on a clearance map of the environment. The execution costs for this kind of actions depend on the path length and shape. To learn them, we apply model tree learning, which combines the advantages of memory-based learning and general function approximation. The learned models can be expressed in the form of symbolic rules and are thus well intelligible for human inspection.

The learned models are applied in two different reasoning tasks. First, the sequencing of a partially ordered set of tasks. Second, the selection between different possible expansions of a given task. The sequencing is demonstrated for the example of scheduling a set of user-specified navigation tasks that all have a deadline and some utility value. The action selection is demonstrated for the problem of selecting an intermediate target point when navigating along a computed path to the goal.

For the scheduling we have suggested a heuristic search in the space of all possible schedules. The schedules are evaluated based on a forward projection. Plan projection is a powerful tool for execution planning as it can

take different factors that influence the robot's performance into account, i.e. missed deadlines, execution costs, and task priorities.

The action selection problem can be considered as a Markov Decision Problem. The problem of selecting the next intermediate target point in APPEAL has been considered as a deterministic problem, as complete failures on this level of abstraction are very rare. On lower levels of the task decomposition hierarchy, however, it might be more appropriate to consider uncertain action effects and to model the possibility of execution failures.

For mobile robot navigation, it is extremely important that planning and plan execution is interleaved. In both cases, this is achieved using transformational planning techniques. The scheduling is limited to tasks that have not already been started and is therefore straightforward. For the action selection problem, a new plan has to be generated whenever a more promising task expansion has been determined.

Extensive robot and simulator experiments have demonstrated that online execution planning improves the mobile robot's performance considerably. The action selection, however, is not very transparent to human observers. For simple planning problems, i.e. problems that require only limited lookahead, action selection rules can be learned form offline planning. Our experiments demonstrate that using these rules, the robot performs almost equally well for the task of selecting intermediate target points as with using online planning. Similar results, however, cannot be expected for the sequencing task as this task in general requires a full lookahead.

The application of online and offline plan projection in the APPEAL architecture proved to improve the performance of the mobile robot RHINO considerably. This is remarkable as RHINO can be regarded as a state-of-the art mobile robot control system, which has been proven performant and reliable in two long-term experiments. However, we expect the techniques to be useful in other domains as well. These domains comprise mobile robot manipulation as well as action selection in robot soccer.

The first central idea of this thesis is to represent the state of plan execution explicitly using Hierarchical Task Networks. This representation allows to handle execution failures on the right level of abstraction by failing upwards and forms the basis for planning, reasoning, and learning in APPEAL. Planning takes the form of transformational planning, which is based on the possibility to generate working default plans fast and to improve them by plan transformations during execution.

We have demonstrated the application of plan transformations for two tasks:

The selection of the best expansion for a given task and the computation of the optimal order for a set of partially ordered tasks, i.e. task sequencing and action selection. In both cases, plan transformations are selected on the basis of learned models of execution costs where the models are acquired from execution traces logged during plan execution. For the task of mobile robot navigation, it is central to predict the robot performance for actions that are intended to change the robot's position or pose. This has been achieved using a feature language based on a path planning process. These features in combination with model tree learning provide an accurate prediction of the robot performance.

This combination of planning and learning techniques can be regarded as a self-supervised learning framework. It can be generalized to the task of learning action selection rules. We have demonstrated how a task-independent action selection policy can be computed from a set of task-specific action selection rules using inductive learning. This learning technique, however, is applicable only to problems that require a limited lookahead like the problem of selecting intermediate target points. For this task, the robot performs equally well using the learned action selection rules as with online execution planning. The action selection policy, however, is much more intelligible for human inspection.

In summary, the techniques presented throughout this thesis provide a general framework for the problem of mobile robot plan execution. The ideas have been implemented in APPEAL and demonstrated to bear a large potential for performance improvement.

Appendix A

Statistical Testing

To compare the average performance of two systems e.g. two control programs or learned models is often difficult if the systems show a high variance in their performance. This problem is unavoidable in robotics and perhaps even more so in the field of robot learning. With statistical significance tests the probability that a performance gain was observed by chance, that is, although the system performance has not changed, can be estimated. If this probability, the significance probability, is small, say less than α, one can speak of a significant performance with respect to the significance level of α. This means that the probability that the observed performance gain was not caused by an improvement of the system but only measured by chance, is sufficiently small.

This chapter recapitulates the basics of statistical testing. After introducing a general method for hypothesis testing, methods for testing hypotheses about the mean are introduced: the Z test, the one-sample t test, the two-sample t test, the paired sample t test. They all make critical assumptions about the underlying population distribution, although they tend to be robust against their violation.

An alternative approach is to use computer-intensive tests which are based on resampling techniques. We discuss two such methods. The bootstrap two-sample t test and the randomization paired-sample t test. These methods do not rely on the assumptions made by parametric tests and are therefore more robust. They are also not limited to testing hypothesis about the mean,

but can be applied to testing hypothesis about any statistic.

A.1 Parametric Tests

Hypotheses testing is a procedure that consists of five steps.

1. Make an assumption about the population distribution.
2. Formulate a null hypothesis. For the comparison of two sample means $\bar{x}_1$ and $\bar{x}_2$ this is for example the assumption that both means are equal: $H_0 : \bar{x}_1 = \bar{x}_2$.
3. Compute the test statistic and its distribution under the assumption that the null hypothesis is true.
4. Compute the critical range of values given significance level α.
5. If the observed value of the test statistic lies within the critical range, reject H_0 and keep it otherwise.

The only difficult part of this procedure is estimating the sampling distribution in step 3. In general, this is only possible when the assumptions made in step 1 are sufficiently strong. For the comparison of means, the sampling distribution is the t distribution which gives the t test its name. Given the sampling distribution the critical range for a significance level α can be computed from the $\frac{\alpha}{2}$-quantile and the $(1 - \frac{\alpha}{2})$-quantile of the sampling distribution in case of a two-tailed test or from the $(1 - \alpha)$-quantile of the sampling distribution in case of a one-tailed test.

A.2 Testing Hypotheses about the Mean

For the comparison of the performance of two systems, we need algorithms for testing hypotheses about the mean, median, mode or some other expectation measure. However, only for hypotheses about the mean, a sampling distribution can be computed analytically. Hypotheses about e.g. the median can only be tested with so called computer-intensive procedures as described in Section A.3.

This section describes four statistical tests for hypotheses about the mean: the Z test, the one-sample t test, the two-sample t test and the paired-sample

t test. It discusses in which test situation these tests can be applied and give an algorithm for each of them.

A.2.1 Z Test

Applicability

1. To test the null hypothesis $\mu = \mu_0$ for a sample x
2. if x is drawn from $N(\mu, \sigma^2)$ and
3. σ^2 is known.

Test Procedure

Let $\bar{x}$ be the mean of sample x with sample size n.

Compute $Z = \frac{\bar{x}-\mu_0}{\sigma_{\bar{x}}}$ with $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$

Choose a significance level α, e.g. $\alpha = 0.05$.

Let $q_{1-\frac{\alpha}{2}}$ be the $(1-\frac{\alpha}{2})$-quantile of the $N(0,1)$ distribution and $q_{\frac{\alpha}{2}}$ be the $\frac{\alpha}{2}$-quantile of the $N(0,1)$ distribution. Reject the null hypotheses $H_0 : \mu = \mu_0$ and confirm the hypothesis $H_1 : \mu \neq \mu_0$ with significance level α if $Z \leq q_{\frac{\alpha}{2}}$ or $Z \geq q_{1-\frac{\alpha}{2}}$.

Let $q_{1-\alpha}$ be the $(1-\alpha)$-quantile of the Z distribution. Reject the null hypotheses $H_0 : \mu = \mu_0$ and confirm the hypothesis $H_1 : \mu > \mu_0$ with significance level α if $Z \geq q_{1-\alpha}$.

Remarks

If the null hypothesis is true, Z is $N(0,1)$ distributed[1]. Due to the central limit theorem the same is approximately true for large n (say $n > 30$). In this case, σ can be approximated by the sample variance s of x. $(1-\alpha)100$ percent of the mass of the $N(0,1)$ distribution lie within the interval $[q_{\frac{\alpha}{2}}, q_{1-\frac{\alpha}{2}}]$. That means in the case of the two-tailed test that if $Z \leq q_{\frac{\alpha}{2}}$ or $Z \geq q_{1-\frac{\alpha}{2}}$ the null hypothesis is true with a probablilty of at most α. Only $\alpha 100$ percent of the mass of the distribution lie in the interval $[q_{1-\alpha}, \infty]$. Therefore in the case of a one-tailed test, if $Z \geq q_{1-\alpha}$ the probability that the null hypothesis is true is at most α and the null hypothesis can therefore be rejected.

[1]The $N(0,1)$ distribution is also called Z distribution which gives the Z test its name. Another common name for this test is Gauß test.

A.2.2 One-sample t Test

Applicability

1. To test the null hypothesis $H_0 : \mu = \mu_0$
2. if the sample x is drawn from $N(\mu, \sigma^2)$
3. but the variance σ^2 is not known.

Test Procedure

Let $\bar{x}$ be the mean of sample x with sample size n and let s be the sample standard deviation.

Compute $t = \frac{\bar{x}-\mu}{\sigma_{\bar{x}}}$ with $\sigma_{\bar{x}} = \frac{s}{\sqrt{n}}$

Choose a significance level α, e.g. $\alpha = 0.05$.

Let $q_{1-\frac{\alpha}{2}}$ be the $(1-\frac{\alpha}{2})$-quantile of the $t(n-1)$ distribution and $q_{\frac{\alpha}{2}}$ be the $\frac{\alpha}{2}$-quantile of the $t(n-1)$ distribution. Reject the null hypotheses $H_0 : \mu = \mu_0$ and confirm the hypothesis $H_1 : \mu \neq \mu_0$ with significance level α if $t \leq q_{\frac{\alpha}{2}}$ or $t \geq q_{1-\frac{\alpha}{2}}$.

Let $q_{1-\alpha}$ be the $(1-\alpha)$-quantile of the $t(n-1)$ distribution. Reject the null hypotheses $H_0 : \mu = \mu_0$ and confirm the hypothesis $H_1 : \mu > \mu_0$ with significance level α if $t \geq q_{1-\alpha}$.

Remarks

The t test in contrast to the Z test is also applicable if the population variance σ^2 is not known. Instead of testing against the $N(0,1)$ distribution the t test tests against the $t(k)$ distribution. The $t(k)$ distribution looks like the normal distribution but has heavier tails than the normal distribution. More of the mass of the distribution is in the tails and consequently it is harder to pass the test. That the shape of the $t(k)$ distribution depends on k where $k = n-1$ for the test, takes the uncertainty into account that results from small sample sizes.

A.2.3 Two-sample t Test

Applicability

1. To test the null hypothesis $H_0: \mu_1 = \mu_2$ about two samples x_1 and x_2
2. if the samples are drawn from $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$ and
3. $\sigma^2 = \sigma_1^2 = \sigma_2^2$ and
4. σ^2 is not known.

Test Procedure

Let $\bar{x}_1$ and $\bar{x}_2$ be the means of the samples x_1 and x_2, let s_1 and s_2 be the sample standard deviations and n_1 and n_2 their sample sizes.

Compute the pooled variance

$$\hat{\sigma}_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2} \tag{A.1}$$

and the standard deviation of the sampling distribution

$$\hat{\sigma}_{\hat{x}_1 - \hat{x}_2} = \sqrt{\hat{\sigma}_p^2 \left(\frac{1}{n_1} + \frac{1}{n_2} \right)} \tag{A.2}$$

Compute $t = \frac{\bar{x}_1 - \bar{x}_2}{\hat{\sigma}_{\hat{x}_1 - \hat{x}_2}}$

Choose a significance level α, e.g. $\alpha = 0.05$.

Let $q_{1-\frac{\alpha}{2}}$ be the $(1 - \frac{\alpha}{2})$-quantile of the $t(n_1 + n_2 - 2)$ distribution and $q_{\frac{\alpha}{2}}$ be the $\frac{\alpha}{2}$-quantile of the $t(n_1 + n_2 - 2)$ distribution. Reject the null hypotheses $H_0 : \mu_1 = \mu_2$ and confirm the hypothesis $H_1 : \mu_1 \neq \mu_2$ with significance level α if $t \leq q_{\frac{\alpha}{2}}$ or $t \geq q_{1-\frac{\alpha}{2}}$.

Let $q_{1-\alpha}$ be the $(1-\alpha)$-quantile of the $t(n_1 + n_2 - 2)$ distribution. Reject the null hypotheses $H_0 : \mu_1 = \mu_2$ and confirm the hypothesis $H_1 : \mu_1 > \mu_2$ with significance level α if $t \geq q_{1-\alpha}$.

Remarks

In the case of a two-sample t test we do not know the standard deviation of the underlying distributions. We can approximate it by the pooled variance given by equation A.1. For a more detailed discussion of the two-sample t test please refer to [Coh95a, pp.127-129].

A.2.4 Paired-Sample t Test

Applicability

1. To test the null hypothesis $H_0 : \mu_{1_j} = \mu_{2_j} \; \forall j \in \{1, ..., n\}$
2. given two samples $x_1 = (x_{1_1}, .., x_{1_n})$ and $x_2 = (x_{2_1}, ..., x_{2_n})$ of equal size
3. where each x_{i_j} is drawn from $N(\mu_j, \sigma_j^2)$ and
4. $\sigma_1^2 = \sigma_2^2 = ... = \sigma_n^2 = \sigma^2$.

Test Procedure

Compute $x = (x_1, ..., x_n)$ where $x_i = x_{1_i} - x_{2_i}$. Let $\bar{x}$ be the sample mean and s be the sample standard deviation. If the null hypothesis is true x can be generated by sampling from an $N(0, \sigma^2)$ distribution. We can test this hypothesis using a one-sample t test.

Compute $t = \frac{\bar{x}}{\sigma_{\bar{x}}}$ with $\sigma_{\bar{x}} = \frac{s}{\sqrt{n}}$.

Choose a significance level α, e.g. $\alpha = 0.05$.

Let $q_{1-\frac{\alpha}{2}}$ be the $(1 - \frac{\alpha}{2})$-quantile of the $t(n-1)$ distribution and $q_{\frac{\alpha}{2}}$ be the $\frac{\alpha}{2}$-quantile of the $t(n-1)$ distribution. Reject the null hypotheses $H_0 : \mu = 0$ and confirm the hypothesis $H_1 : \mu \neq 0$ with significance level α if $t \leq q_{\frac{\alpha}{2}}$ or $t \geq q_{1-\frac{\alpha}{2}}$.

Let $q_{1-\alpha}$ be the $(1 - \alpha)$-quantile of the $t(n-1)$ distribution. Reject the null hypotheses $H_0 : \mu = 0$ and confirm the hypothesis $H_1 : \mu > 0$ with significance level α if $t \geq q_{1-\alpha}$.

Remarks

The paired t test is easier to pass than the corresponding two-sample t test. However, it should only be applied if the variance in the experiment has been controlled carefully in the sense described above. Please note that the variance is assumed to be equal for all pairs which in practice might be a very strong assumption. For further details on the paired sample t test please refer to [Coh95a, pp.129-130].

A.3 Computer-Intensive Statistical Methods

As outlined before, the classical approach to statistical testing is to derive the sampling distribution for a statistic and then calculate the critical range of the sample statistic from this distribution. This method has two major drawbacks.

- For many interesting statistics like the trimmed mean or the interquartile range the sampling distribution of the statistic cannot be derived analytically.
- Parametric tests make non trivial assumptions about the distribution of the population distribution. The family of t tests for example assumes that the samples are drawn from a normal distribution and – in the case of the two-sample t test – that both samples are drawn from distributions with the same variance.

These problems can be avoided using computer-intensive sampling techniques. For the purpose of this thesis, we are mainly concerned with the second point. Please refer to Cohen [Coh95a] for an example of the application of a computer-intensive test procedure to the comparison of censored means. We describe the application of computer-intensive techniques to comparing means using a bootstrap two-sample t test and a randomization version of a paired t test.

A.3.1 Bootstrap Two-Sample t Test

Applicability

1. To test the null hypothesis $H_0 : \bar{x}_1 = \bar{x}_2$ about two samples x_1 and x_2
2. if the two samples are representative for the underlying distributions.

Test Procedure

Algorithm 6 describes the test procedure. The algorithm computes an estimate of the probability that the difference of the two means is observed under the assumption that the null hypothesis is true, i.e. the significance probability.

Algorithm 6 Bootstrap Two-Sample t Test

Require: Two samples x_1 and x_2 with the size n_1 and n_2 respectively.
Be $\bar{x}_1$ and $\bar{x}_2$ be the sample means with $\bar{x}_1 > \bar{x}_2$ and be $\theta = \bar{x}_1 - \bar{x}_2$.

Ensure: Returns an estimate of the probability that the difference θ is observed although x_1 and x_2 are drawn from distributions with equal expectation values

```
c = 0
Combine x_1 and x_2 in a single sample x_p
for all i = 1 to k do
  Draw a pseudosample x_1^* of size n_1 from x_p by sampling with replacement
  Draw a pseudosample x_2^* of size n_2 from x_p by sampling with replacement
  Compute θ* = x̄_1^* − x̄_2^*
  if θ* ≥ θ then
    c=c+1
  end if
end for
return c/k
```

Remarks

We call the method bootstrap two-sample t test as it tests the same type of hypothesis as the t test, although it does not make use of the t distribution to compute the critical range of the test statistic. Bootstrap methods do not require to know the population distribution, but assume that the sample is representative of the population. In contrast to randomization methods, bootstrap methods sample with replacement.

The null hypothesis that both samples have the same mean is reflected by the fact that both samples are combined into a new sample which is used to draw pseudosamples of the same size as the original samples.

Bootstrap methods can also be applied to test hypothesis about any other test statistic like the median, the interquartile range or a censored mean. For those statistics no sampling distribution can be computed analytically. Unlike the parametric t test, the bootstrap t test does not make any assumptions about the underlying distributions and does not require the two samples to have the same variance. Please refer to [ET93] and [Coh95a] for a more detailed discussion of bootstrap methods.

Algorithm 7 Randomization Paired-Sample t Test

Require: Two paired samples x_1 and x_2 of equal size n.
Be $x_d = (x_{d_1}, ..., x_{d_n})$ where $x_{d_i} = x_{1_i} - x_{2_i}$. Be $\bar{x}_d$ the mean difference.
Ensure: Returns an estimate of the probability that $\bar{x}_d$ is observed although x_1 and x_2 are generated by distributions with pairwise equal expected values.

```
c = 0
Create an empty list x*_d
for all i = 1 to k do
  for all j = 1 to n do
    Draw a random number p from the interval [0,1]
    if p > 0.5 then
      Add x_dj to x*_d
    else
      Add −x_dj to x*_d
    end if
  end for
  Calculate x̄*_d, the mean of the elements in x_d
  if x̄*_d > x̄_d then
    c=c+1
  end if
end for
return c/k
```

A.3.2 Randomization Paired-Sample t Test

Applicability

1. To test the null hypothesis $H_0 : \mu_{1_j} = \mu_{2_j} \; \forall j \in \{1, ..., n\}$
2. given two samples $x_1 = (x_{1_1}, .., x_{1_n})$ and $x_2 = (x_{2_1}, ..., x_{2_n})$ and
3. each x_{i_j} is sampled from some distribution with expected value μ_{i_j}.

Test Procedure

Algorithm 7 describes the test procedure. Again the algorithm returns an estimate of the significance probability and does not assume a significance level as input.

Remarks

Like for the bootstrap method, the null hypothesis is reflected by the way pseudosamples are generated. Under the null hypothesis that the two samples x_1 and x_2 are generated by the same probability distribution, randomly changing the signs of the elements of x_d should have no influence on the expected value of $\bar{x}_d$ as each element might be drawn with equal probability in one sample or the other.

The main difference between bootstrap methods and randomization methods is the way they draw pseudosamples. As the bootstrap methods draw with replacement they have to assume that the samples are representative for the underlying distribution. The randomization methods on the other hand perform sampling without replacement. The algorithm described above for example could equally well be implemented as follows. In each step draw a pseudosample of size n from a sample x' where x' contains for each element x_{d_i} in x_d both x_{d_i} and $-x_{d_i}$. Please refer to [Nor89] and [Coh95a] for a more detailed discussion of statistical testing based on randomization.

Appendix B

Learned Rules

B.1 Plan Projection for Online Scheduling

IF	pathLength < 1530.0	AND
	narrowPassageSegments < 1.5	AND
	pathLength < 890.0	
THEN	duration = 5.92*narrowPassageCounter	+
	1.22*passageCounter	+
	0.76*freePassageCounter	
		(55 examples)

IF	pathLength < 1530.0	AND
	narrowPassageSegments < 1.5	AND
	NOT pathLength < 890.0	
THEN	duration = 6.27*narrowPassageCounter	+
	1.44*passageCounter	+
	0.59*freePassageCounter	
		(53 examples)

IF	pathLength < 1530.0	AND
	NOT narrowPassageSegments < 1.5	
THEN	duration = 4.29*narrowPassageCounter	+
	0.95*passageCounter	+
	0.74*freePassageCounter	
		(60 examples)

IF	NOT pathLength < 1530.0	AND
	freePassageSegements < 3.5	
THEN	duration = 4.83*narrowPassageCounter	+
	1.0*passageCounter	+
	0.57*freePassageCounter	
		(70 examples)
IF	NOT pathLength < 1530.0	AND
	NOT freePassageSegements < 3.5	AND
	pathLength < 2110.0	
THEN	duration = 3.42*narrowPassageCounter	+
	1.08*passageCounter	+
	0.85*freePassageCounter	
		(50 examples)
IF	NOT pathLength < 1530.0	AND
	NOT freePassageSegements < 3.5	AND
	NOT pathLength < 2110.0	
THEN	duration = 4.83*narrowPassageCounter	+
	1.01*passageCounter	+
	0.62*freePassageCounter+0.0	
		(52 examples)

B.2 Plan Projection for Action Selection

IF	passageSegments < 0.5	AND
	narrowPassageSegments < 0.5	AND
	pathLength < 150.0	
THEN	duration = 0.058929 * pathLength	
		(1491 examples)

IF	passageSegments < 0.5	AND
	narrowPassageSegments < 0.5	AND
	NOT pathLength < 150.0	AND
	pathLength < 350.0	AND
	pathCurvature < 1.05	
THEN	duration = 0.035200 * pathLength	
		(59 examples)

IF	passageSegments < 0.5	AND
	narrowPassageSegments < 0.5	AND
	NOT pathLength < 150.0	AND
	pathLength < 350.0	AND
	NOT pathCurvature < 1.05	AND
	pathLength < 270.0	
THEN	duration = 0.030952 * pathLength	
		(1115 examples)

IF	passageSegments < 0.5	AND
	narrowPassageSegments < 0.5	AND
	NOT pathLength < 150.0	AND
	pathLength < 350.0	AND
	NOT pathCurvature < 1.05	AND
	NOT pathLength < 270.0	
THEN	duration = 0.031452 * pathLength	
		(83 examples)

IF	passageSegments < 0.5	AND
	narrowPassageSegments < 0.5	AND
	NOT pathLength < 150.0	AND
	NOT pathLength < 350.0	
THEN	duration = 0.015116 * pathLength	
		(671 examples)

IF	passageSegments < 0.5	AND
	NOT narrowPassageSegments < 0.5	AND
	freePassageSegments < 1.5	AND
	pathCurvature < 1.55	AND
	pathLength < 130.0	
THEN	duration = 0.109375 * pathLength	
		(224 examples)

IF	passageSegments < 0.5	AND
	NOT narrowPassageSegments < 0.5	AND
	freePassageSegments < 1.5	AND
	pathCurvature < 1.55	AND
	NOT pathLength < 130.0	AND
	pathLength < 260.0	
THEN	duration = 0.041228 * pathLength	
		(154 examples)

IF	passageSegments < 0.5	AND
	NOT narrowPassageSegments < 0.5	AND
	freePassageSegments < 1.5	AND
	pathCurvature < 1.55	AND
	NOT pathLength < 130.0	AND
	NOT pathLength < 260.0	
THEN	duration = 0.071622 * pathLength	
		(50 examples)

IF	passageSegments < 0.5	AND
	NOT narrowPassageSegments < 0.5	AND
	freePassageSegments < 1.5	AND
	NOT pathCurvature < 1.55	
THEN	duration = 0.037061 * pathLength	
		(63 examples)

IF	passageSegments < 0.5	AND
	NOT narrowPassageSegments < 0.5	AND
	NOT freePassageSegments < 1.5	AND
	pathLength < 510.0	AND
	pathLength < 210.0	
THEN	duration = 0.076389 * pathLength	
		(214 examples)

IF	passageSegments < 0.5	AND
	NOT narrowPassageSegments < 0.5	AND
	NOT freePassageSegments < 1.5	AND
	pathLength < 510.0	AND
	NOT pathLength < 210.0	
THEN	duration $= 0.066111$ * pathLength	
		(355 examples)

IF	passageSegments < 0.5	AND
	NOT narrowPassageSegments < 0.5	AND
	NOT freePassageSegments < 1.5	AND
	NOT pathLength < 510.0	
THEN	duration $= 0.047044$ * pathLength	
		(103 examples)

IF	NOT passageSegments < 0.5	AND
	pathCurvature < 1.55	AND
	pathLength < 390.0	AND
	pathCurvature < 1.25	
THEN	duration $= 0.072222$ * pathLength	
		(78 examples

IF	NOT passageSegments < 0.5	AND
	pathCurvature < 1.55	AND
	pathLength < 390.0	AND
	NOT pathCurvature < 1.25	AND
	pathCurvature < 1.35	
THEN	duration $= 0.051333$ * pathLength	
		(62 examples

IF	NOT passageSegments < 0.5	AND
	pathCurvature < 1.55	AND
	pathLength < 390.0	AND
	NOT pathCurvature < 1.25	AND
	NOT pathCurvature < 1.35	
THEN	duration $= 0.060920$ * pathLength	
		(54 examples)

IF	NOT passageSegments < 0.5	AND
	pathCurvature < 1.55	AND
	NOT pathLength < 390.0	
THEN	duration $= 0.034884$ * pathLength	
		(66 examples)

IF	NOT passageSegments < 0.5	AND
	NOT pathCurvature < 1.55	
THEN	duration $= 0.180791$ * pathLength	
		(108 examples)

B.3 Learning Action Selection Rules

Action selection rules learned from a set of 10000 examples with (a) decision tree learning and (b) sequential covering. In both cases, we used a depth limit of 5. The examples were generated from the action selection MDP by value iteration with infinite planning horizon.

B.3.1 Using Decision Trees

IF	(lengthOfFirstPassage < 375.00)	AND
	(firstPassage = narrowPassage)	AND
	(lengthOfSecondPassage < 375.00)	
THEN	expansion=approachNear	

IF	(lengthOfFirstPassage < 375.00)	AND
	(lengthOfSecondPassage < 525.00)	AND
	(firstPassage = narrowPassage)	AND
	NOT (lengthOfSecondPassage < 375.00)	
THEN	expansion=approachMid	

IF	(lengthOfFirstPassage < 375.00)	AND
	(lengthOfSecondPassage < 525.00)	AND
	NOT (firstPassage = narrowPassage)	AND
	(secondPassage = narrowPassage)	
THEN	expansion=approachNear	

IF	(lengthOfFirstPassage < 375.00)	AND
	(lengthOfSecondPassage < 525.00)	AND
	NOT (firstPassage = narrowPassage)	AND
	NOT (secondPassage = narrowPassage)	
THEN	expansion=approachMid	

IF	NOT (lengthOfSecondPassage < 525.00)	AND
	(lengthOfFirstPassage < 325.00)	
THEN	expansion=approachNear	

IF	(lengthOfFirstPassage < 375.00)	AND
	NOT (lengthOfSecondPassage < 525.00)	AND
	NOT (lengthOfFirstPassage < 325.00)	AND
	(firstPassage = passage)	
THEN	expansion=approachMid	

IF	(lengthOfFirstPassage < 375.00)	AND
	NOT (lengthOfSecondPassage < 525.00)	AND
	NOT (lengthOfFirstPassage < 325.00)	AND
	NOT (firstPassage = passage)	
THEN	expansion=approachNear	

IF	NOT (lengthOfFirstPassage < 375.00)	AND
	(lengthOfFirstPassage < 600.00)	AND
	(straightLineDistance < 375.00)	AND
	(lengthOfSecondPassage < 75.00)	AND
	(numberOfPassages < 4.50)	AND
THEN	expansion=approachNear	

IF	NOT (lengthOfFirstPassage < 375.00)	AND
	(lengthOfFirstPassage < 600.00)	AND
	(straightLineDistance < 375.00)	AND
	(lengthOfSecondPassage < 75.00)	AND
	NOT (numberOfPassages < 4.50)	
THEN	expansion=approachFar	

IF	NOT (lengthOfFirstPassage < 375.00)	AND
	(lengthOfFirstPassage < 600.00)	AND
	(straightLineDistance < 375.00)	AND
	NOT (lengthOfSecondPassage < 75.00)	AND
THEN	expansion=approachFar	

IF	NOT (lengthOfFirstPassage < 375.00)	AND
	(lengthOfFirstPassage < 600.00)	AND
	NOT (straightLineDistance < 375.00)	AND
	(firstPassage = passage)	
THEN	expansion=approachMid	

```
IF      NOT (lengthOfFirstPassage < 375.00)    AND
        (lengthOfFirstPassage < 600.00)        AND
        NOT (straightLineDistance < 375.00)    AND
        NOT (firstPassage = passage)
THEN    expansion=approachFar

IF      NOT (lengthOfFirstPassage < 600.00)    AND
        (straightLineDistance < 600.00)        AND
        (lengthOfThirdPassage < 225.00)
THEN    expansion=approachFar

IF      NOT (lengthOfFirstPassage < 600.00)    AND
        (straightLineDistance < 600.00)        AND
        (lengthOfThirdPassage < 375.00)        AND
        NOT (lengthOfThirdPassage < 225.00)
THEN    expansion=approachMid

IF      NOT (lengthOfFirstPassage < 600.00)    AND
        NOT (lengthOfThirdPassage < 375.00)    AND
        (straightLineDistance < 300.00)
THEN    expansion=approachFar

IF      NOT (lengthOfFirstPassage < 600.00)    AND
        (straightLineDistance < 600.00)        AND
        NOT (lengthOfThirdPassage < 375.00)    AND
        NOT (straightLineDistance < 300.00)
THEN    expansion=approachNear

IF      NOT (lengthOfFirstPassage < 600.00)    AND
        NOT (straightLineDistance < 600.00)
THEN    expansion=approachFar
```

B.3.2 Using Sequential Covering

IF (secondPassage = narrowPassage) AND
(lengthOfFirstPassage $\geq$ 375.00) AND
(lengthOfFirstPassage $<$ 600.00)
THEN expansion=approachFar

IF (firstPassage = freePassage) AND
(lengthOfFirstPassage $\geq$ 775.00)
THEN expansion=approachFar

IF (secondPassage = undefined) AND
(lengthOfFirstPassage $<$ 675.00) AND
(straightLineDistance $\geq$ 375.00)
THEN expansion=approachFar

IF (lengthOfFirstPassage $\geq$ 525.00) AND
(lengthOfSecondPassage $<$ 75.00) AND
(straightLineDistance $<$ 300.00)
THEN expansion=approachFar

IF (lengthOfFirstPassage $\geq$ 525.00) AND
(lengthOfThirdPassage $<$ 225.00)
THEN expansion=approachFar

IF (lengthOfFirstPassage $\geq$ 375.00) AND
(lengthOfFirstPassage $<$ 675.00) AND
(secondPassage = undefined) AND
(straightLineDistance $<$ 325.00)
THEN expansion=approachFar

IF (lengthOfFirstPassage $\geq$ 375.00) AND
(lengthOfSecondPassage $<$ 75.00) AND
(lengthOfFirstPassage $<$ 625.00) AND
(secondPassage = narrowPassage)
THEN expansion=approachFar

IF (secondPassage = passage) AND
(lengthOfFirstPassage $\geq$ 75.00) AND
(lengthOfFirstPassage $<$ 225.00)
THEN expansion=approachMid

IF (lengthOfSecondPassage < 225.00) AND
(lengthOfFirstPassage < 150.00) AND
(numberOfPassages ≥ 4.50) AND
(straightLineDistance ≥ 475.00)
THEN expansion=approachMid

IF (lengthOfSecondPassage < 300.00) AND
(straightLineDistance ≥ 175.00) AND
(straightLineDistance < 300.00) AND
(lengthOfFirstPassage ≥ 225.00) AND
(lengthOfSecondPassage ≥ 75.00)
THEN expansion=approachMid

IF (lengthOfSecondPassage < 300.00) AND
(lengthOfFirstPassage < 150.00) AND
(straightLineDistance < 375.00) AND
(straightLineDistance ≥ 300.00)
THEN expansion=approachMid

IF (firstPassage = passage) AND
(lengthOfThirdPassage < 75.00) AND
(straightLineDistance < 300.00) AND
(numberOfPassages < 6.00)
THEN expansion=approachMid

IF (straightLineDistance ≥ 175.00) AND
(firstPassage = passage) AND
(lengthOfFirstPassage < 75.00) AND
(lengthOfThirdPassage ≥ 225.00)
THEN expansion=approachMid

IF (firstPassage = passage) AND
(lengthOfThirdPassage < 75.00) AND
(pathCurvature < 2.10) AND
(pathCurvature ≥ 1.65)
THEN expansion=approachMid

IF (straightLineDistance ≥ 175.00) AND
(lengthOfSecondPassage < 75.00) AND
(lengthOfFirstPassage ≥ 225.00) AND
(straightLineDistance < 300.00) AND
(lengthOfThirdPassage ≥ 300.00)
THEN expansion=approachMid

IF (straightLineDistance $\geq$ 175.00) AND
(firstPassage = passage) AND
(pathCurvature $<$ 3.15) AND
(straightLineDistance $<$ 525.00)
THEN expansion=approachMid

IF (lengthOfSecondPassage $<$ 75.00) AND
(lengthOfFirstPassage $\geq$ 225.00) AND
(straightLineDistance $<$ 300.00)
THEN expansion=approachMid

IF (straightLineDistance $\geq$ 450.00) AND
(lengthOfFirstPassage $<$ 750.00) AND
(lengthOfFirstPassage $\geq$ 225.00)
THEN expansion=approachMid

IF (straightLineDistance $\geq$ 175.00) AND
(secondPassage = passage) AND
(straightLineDistance $<$ 300.00) AND
(lengthOfFirstPassage $<$ 25.00)
THEN expansion=approachMid

IF (straightLineDistance $\geq$ 175.00) AND
(secondPassage = passage) AND
(pathCurvature $<$ 1.55) AND
(numberOfPassages $<$ 7.50) AND
(lengthOfSecondPassage $<$ 175.00)
THEN expansion=approachMid

IF (straightLineDistance $\geq$ 175.00) AND
(lengthOfSecondPassage $<$ 625.00) AND
(lengthOfSecondPassage $\geq$ 375.00) AND
(straightLineDistance $\geq$ 375.00) AND
(firstPassage = narrowPassage)
THEN expansion=approachMid

IF TRUE
THEN expansion=approachNear

Bibliography

[AFH+98] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi robot cooperation in the Martha project. *IEEE Robotics and Automation Magazine*, 5(1):36–47, 1998.

[Ark98] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.

[BAB+00] M. Beetz, T. Arbuckle, T. Belker, A. B. Cremers, D. Hähnel, and D. Schulz. Enabling autonomous robots to perform complex tasks. *Künstliche Intelligenz*, 14(4):5–10, 2000.

[BAB+01] M. Beetz, T. Arbuckle, T. Belker, M. Bennewitz, W. Burgard, A. B. Cremers, D. Fox, H. Grosskreutz, D. Hähnel, and D. Schulz. Integrated plan-based control of autonomous service robots. *IEEE Intelligent Systems*, 16(5):56–65, 2001.

[Bal02] N. Balac. *ERA: Learning Planner Knowledge in Complex, Continuous and Noisy Environments*. PhD thesis, Vanderbilt University, 2002.

[BB99] M. Beetz and T. Belker. Experience- and model-based transformational learning of symbolic behavior specifications – prelimary report. Technical Report IAI-TR-99-3, University of Bonn, 1999.

[BB00a] M. Beetz and T. Belker. Environment and task adaptation for robotic agents. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence*, pages 648–652, 2000.

[BB00b] M. Beetz and T. Belker. Learning structured reactive navigation plans from executing MDP policies. In *Proceedings of the Second International Cognitive Robotics Workshop*, pages 3–11, 2000.

[BB00c] M. Beetz and T. Belker. XRFRMLearn - a system for learning structured reactive navigation plans. In *Proceedings of the Eighth International Symposium on Intelligent Robotic Systems*, pages 61–73, 2000.

[BB01a] M. Beetz and T. Belker. Learning robot action plans for controlling continuous, percept-driven behavior. In *Proceedings of the Sixth European Conference on Planning*, 2001.

[BB01b] M. Beetz and T. Belker. Learning structured reactive navigation plans from executing MDP policies. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 19–20, 2001.

[BB01c] T. Belker and M. Beetz. Learning to execute navigation plans. In *KI-2001: Advances in Artificial Intelligence*, pages 425–439, 2001.

[BBC+95] J. Buhmann, W. Burgard, A. B. Cremers, D. Fox, T. Hofmann, F. Schneider, J. Strikos, and S. Thrun. The mobile robot Rhino. *AI Magazine*, 16(2):31–38, 1995.

[BBC02a] T. Belker, M. Beetz, and A. B. Cremers. Learning action models for the improved execution of navigation plans. *Robotics and Autonomous Systems*, 38(3-4):137–148, 2002.

[BBC02b] T. Belker, M. Beetz, and A. B. Cremers. Learning of plan execution policies for indoor navigation. *AI Communications*, 15(1):3–16, 2002.

[BBFC98] M. Beetz, W. Burgard, D. Fox, and A. B. Cremers. Integrating active localization into high-level control systems. *Robotics and Autonomous Systems*, 23:205–220, 1998.

[BBG99] M. Beetz, M. Bennewitz, and H. Grosskreutz. Probabilistic, prediction-based schedule debugging for autonomous robot office couriers. In *Proceedings of the 23rd German Conference on Artificial Intelligence*, 1999.

[BBS95] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[BCF+98] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 11–18, 1998.

[BD96] S. Bennett and G. F. DeJong. Real-world robotics: Learning to plan for robust execution. *Machine Learning*, 23(2-3):121–161, 1996.

[BDFC98] W. Burgard, A. Derr, D. Fox, and A. B. Cremers. Integrating global position estimation and position tracking for mobile robots: the dynamic markov localization approach. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 730–735, 1998.

[BDH99] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[Bee99] M. Beetz. Structured reactive controllers — a computational model of everyday activity. In *Proceedings of the Third International Conference on Autonomous Agents*, pages 228–235, 1999.

[Bee00] M. Beetz. *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*. LNAI 1772. Springer Publishers, 2000.

[Bee02] M. Beetz. *Plan-Based Control of Robotic Agents: Improving the Capabilities of Autonomous Robots*. LNCS 2554. Springer Publishers, 2002. Habilitationsschrift, Dept. of Computer Science, University of Bonn.

[Bel57] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[BFG+97] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2-3):237–256, 1997.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Inc., Belmont, CA, 1984.

[BFT97] W. Burgard, D. Fox, and S. Thrun. Active mobile robot localization. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[BHGP02] M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack. Preface. In *Advances in Plan-Based Control of Robotic Agents*, 2002.

[BHH03] T. Belker, M. Hammel, and J. Hertzberg. Learning to optimize mobile robot navigation based on HTN plans. In *Proceedings of the International Conference on Robotics and Automation*, pages 4136–4141, 2003.

[BHH04] T. Belker, M. Hammel, and J. Hertzberg. Plan projection under the APPEAL robot control architecture. In *Proceedings of the Eigtth Conference on Intelligent Autonomous Systems*, 2004. to appear.

[Bis96] C. M. Bishop. *Neural networks for pattern recognition*. Oxford University Press, Oxford, England, 1996.

[BK91] J. Borenstein and Y. Koren. The vector field histogram – fast obstacle avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 7(3):278–288, 1991.

[BK99] O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 341–346, 1999.

[Bon91] R. P. Bonasso. Integrating reaction plans and layered competences through synchronous control. In *Proceedings of the Twelfth International Conference on Artificial Intelligence*, pages 1225–1231, 1991.

[Bro86] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(2):14–23, 1986.

[Bro99] R. A. Brooks. *Cambrian Intelligence*. MIT Press, 1999.

[BS00] T. Belker and D. Schulz. Robot soccer - a second year practical course on mobile robotics. In *Proceedings of the First Workshop on Edutainment Robotic*, 2000. http://www.edutainment-robotics.com/Workshop-Papers/.

[BU95] C. E. Brodley and P. E. Utgoff. Multivariate decision trees. *Machine Learning*, 19(1):45–77, 1995.

[BuR98] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

[Cen87] J. Cendrowska. PRISM: an algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987.

[CKK96] A. R. Cassandra, L. P. Kaelbling, and J. A. Kurien. Acting under uncertainty: Discrete bayesian models for mobile-robot navigation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1996.

[CN89] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.

[Coh95a] P. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA, 1995.

[Coh95b] W. W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123, 1995.

[Con92] J. Connell. SSS: A hybrid architecture applied to robot navigation. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 2719–2724, 1992.

[DC94] M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents though learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[Elf89] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *IEEE Computer*, 22(6):46–57, 1989.

[ET93] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall, New York, NY, 1993.

[FBDT99] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 343–349, Orlando, FL, 1999.

[FBT97] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4(1):23–33, 1997.

[FBT99] D. Fox, W. Burgard, and S. Thrun. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11:391–427, 1999.

[FBTC98] D. Fox, W. Burgard, S. Thrun, and A. B. Cremers. Position estimation for mobile robots in dynamic environments. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.

[Fed93] C. Fedor. An interprocess communication system for building robotic architectures. programmer's guide to version 10.xx. Technical Report PA 15213, Carnegie Mellon University, 1993.

[Fir87] J. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 809–815, 1987.

[Fir89] J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* Technical report 672, Yale University, Department of Computer Science, January 1989.

[Fir95] J. Firby. The RAP language manual. Animate Agent Project Working Note AAP-6, University of Chicago, 1995.

[FKPS95] J. Firby, R. Kahn, P. Prokopowitz, and M. Swain. An architecture for vision and action. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 72–79, 1995.

[Fox98] D. Fox. *Markov Localization: A Probabilistic Framework for Mobile Robot Localization and Navigation.* PhD thesis, Dept. of Computer Science, University of Bonn, Germany, December 1998.

[FW94] J. Fürnkranz and G. Widmer. Incremental reduced error pruning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 70–77, 1994.

[Gat92] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 809–815, 1992.

[Gat97] E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the IEEE Aerospace Conference*, 1997.

[Gat98] E. Gat. Three-layer architectures. In D. Kortenkamp, P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case studies of successful robot systems*. MIT Press, 1998.

[GC95] B. R. Gaines and P. Compton. Induction of ripple-down rules applied to modeling large data bases. *Journal of Intelligent Information Systems*, 5(3):211–228, 1995.

[GL87] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682, 1987.

[Hai98] K. Z. Haigh. *Situation-Dependent Learning for Interleaved Planning and Robot Execution*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, 1998.

[Ham89] K. Hammond. *Case-Based Planning*. Academic Press, Inc., 1989.

[Ham03] M. Hammel. Planbasierte Robotersteuerung mit HTNs. Master's thesis, Department of Applied Computer Science, University of Bonn, 2003.

[HBL98] D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX - bridging the gap between logic (GOLOG) and a real robot. In *Proceedings of the 22nd German Conference on Artificial Intelligence*, 1998.

[How60] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.

[HP99] A. E. Howe and L. D. Pyeatt. Integrating POMDP and reinforcement learning for a two layer simulated robot architecture. In *Proceedings of the Third International Conference on Autonomous Agents*, pages 168–174, 1999.

[HR76] L. Hyale and R. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5:15–17, 1976.

[Jor98] M. Jordan, editor. *Learning in Graphical Models*. Kluwer Acadamic Publishers, 1998.

[Kae87] L. P. Kaelbling. REX: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of AIAA Conference on Computers in Aerospace*, pages 143–150, 1987.

[Kal60] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME - Journal of Basic Engineering*, 8:35–45, 1960.

[Kar92] A. Karalic. Employing linear regression in regression tree leaves. In *Proceedings of the Tenth European Conference of Artificial Intelligence*, pages 440–441, 1992.

[KB91] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1398–1404, 1991.

[KBM98] D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors. *Artificial Intelligence and Mobile Robots: Case studies of successful robot systems*. MIT Press, Cambridge, MA, 1998.

[KCM87] S. Kedar-Cabelli and T. McCarthy. Explanation-based generalization as resolution threorem proving. In *Proceedings of The Fourth International Workshop on Machine Learning*, pages 383–389, 1987.

[KLC98] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[KLM96] L. P. Kaelbling, M. L. Littman, and A. P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Kon00] K. Konolige. A gradient method for realtime robot control. In *Proceedings of IEEE/RSJ Conference on Intelligent Robots and Systems*, 2000.

[Kra96] S. Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 812–819, 1996.

[KS96] S. Koenig and R. G. Simmons. Unsupervised learning of probabilistic models for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2301–2308, 1996.

[KS98] S. Koenig and R. G. Simmons. Xavier: A robot navigation architecture based on partially observable markov decision process models. In D. Kortenkamp, P. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, 1998.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[LCK95] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370, 1995.

[Lin93] L. J. Lin. Hierarchical learning of robot skills by reinforcement. In *Proceedings of the International Conference on Neural Networks*, pages 181–186, 1993.

[LNR87] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[LRL+97] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal on Logic Programming*, 31(1-3):59–84, 1997.

[LRN86] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in SOAR: the anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[MA93] A. W. Moore and C. G. Atkeson. Prioritized sweeping – reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[Mat94] M. J. Mataric. Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 181–189, 1994.

[MC92] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2):311–365, 1992.

[McD92a] D. McDermott. Robot planning. *AI Magazine*, 13(2):55–79, 1992.

[McD92b] D. McDermott. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University, 1992.

[McD94] D. McDermott. An algorithm for probabilistic, totally-ordered temporal projection. Research Report YALEU/DCS/RR-1014, Yale University, 1994.

[Mey90] A. Meystel. Knowldge based nested hierarchical control. *Advances in Automation and Robotics*, 2(2):63–152, 1990.

[MG02a] B. Morisset and M. Ghallab. Learning how to combine sensory-motor modalities for a robust behavior. In *Advances in Plan-Based Control of Robotic Agents*, pages 157–178, 2002.

[MG02b] B. Morisset and M. Ghallab. Synthesis of supervision policies for robust sensory-motor behaviors. In *Proceedings of the Seventh International Conference on Intelligent Autonomous Systems*, pages 236–243, 2002.

[MHC99] O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 541–548, 1999.

[Mic93] R. S. Michalski. A theory and methodology of inductive learning. In R. S. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83–134. Morgan Kaufmann, San Mateo, CA, 1993.

[Mit90] T. Mitchell. Becoming increasingly reactive. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1051–1058, 1990.

[Mit97] T. Mitchell. *Machine Learning*. Mc Graw-Hill, New York, 1997.

[MKKC86] T. Mitchell, R. Kellar, and S. Keddar-Cabelli. Explanation based learning: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[MKSB93] S. Murphy, S. Kasif, S. Salzberg, and R. Beigel. OC1: Randomized induction of oblique decision trees. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 322–327, 1993.

[Moo93] A. Moore. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. In *Advances in Neural Information Processings Systems*, pages 711–718, 1993.

[Mor89] H. P. Moravec. Sensor fusion in certainty grids for mobile robots. *AI Magazine*, 9(2):61–74, 1989.

[Mur00] R. Murphy. *Introduction to AI Robotics*. MIT Press, 2000.

[NCLMA99] D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968–973, 1999.

[Nil84] N. J. Nilsson. Shakey the robot. Technical Report 323, SRI International, Menlo Park, California, 1984.

[Nor89] E. W. Noreen. *Computer-intensive Methods for Testing Hypotheses: An Introduction*. John Wiley & Sons, New York, NY, 1989.

[Nou98] I. Nourbakhah. Dervish: An office-navigating robot. In D. Kortenkamp, P. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, 1998.

[NPB95] I. Nourbakhsh, R. Powers, and S. Birchfield. Dervish: an office-navigating robot. *AI Magazine*, 16(2):53–60, 1995.

[PT87] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of markov chain decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[QR89] J. R. Quinlan and R. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80(3):227–248, 1989.

[Qui87] J. R. Quinlan. Rule induction with statistical data – a comparison with multiple regression. *Journal of the Operational Research Society*, 38:347–352, 1987.

[Qui90] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Qui92] J. R. Quinlan. Learning with continuous classes. In *Proceedings of the Fifth Australien Joint Conference on Artificial Intelligence*, pages 343–348, 1992.

[Qui93a] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, San Mateo, California, 1993.

[Qui93b] J. R. Quinlan. Combining instance-based and model-based learning. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 236–243, 1993.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Ros00] J. Rosenblatt. Maximizing expected utility for optimal action selection under uncertainty. *Autonomous Robots*, 9(1):17–25, 2000.

[RT99] N. Roy and S. Thrun. Coastal navigation with mobile robots. In *Advances in Neural Information Processing Systems*, 1999.

[SA98] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robotics and Systems*, 1998.

[Sac77] E. D. Sacerdoti. *A Structure for Plans and Behavior*. Elsevier, North-Holland, 1977.

[SB98] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[SB02] C. Stachniss and W. Burgard. An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[SBC99] D. Schulz, W. Burgard, and A. B. Cremers. Robust visualization of navigation experiments with mobile robots over the internet. In *Proceedings of IEEE/RSJ Conference on Intelligent Robots and Systems*, 1999.

[SBF+00] D. Schulz, W. Burgard, D. Fox, S. Thrun, and A. B. Cremers. Web interfaces for mobile robots in public places. *IEEE Robotics and Automation Magazine*, 7(1):49–56, 2000.

[Sch98] C. Schlegel. Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot. In *Proceedings of IEEE/RSJ Conference on Intelligent Robots and Systems*, 1998.

[Sch02] D. Schulz. *Internet-Based Robotic Tele-Presence*. PhD thesis, Dept. of Computer Science, University of Bonn, Germany, May 2002.

[Sim94] R. Simmons. Structured control for autonomous robots. *Transactions on Robotics and Automation*, 10(1):34–43, 1994.

[Sim96] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *IEEE International Conference on Robotics and Automation*, 1996.

[SK95] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1080–1087, 1995.

[SM02] H. Surman and A. Morales. Scheduling tasks to a team of autonomous mobile service robots in indoor environments. *Journal of Universal Computer Science*, 8(8):809–833, 2002.

[SNT98] S. J. J. Smith, D. S. Nau, and T. Throop. Success in spades: Using AI planning techniques to win the world championship of computer bridge. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1079–1086, 1998.

[ST00] M. Sridharan and G. J. Tesauro. Multi-agent Q-learning and regression trees for automated pricing decisions. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 927–934, 2000.

[Sus77] G. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier, New York, NY, 1977.

[Sus90] G. Sussman. The virtuous nature of bugs. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 111–117. Kaufmann, San Mateo, CA, 1990.

[Sut88] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(3):9–44, 1988.

[Sut90] R. S. Sutton. Integrated architectures for learning, planning and reacting based on approximate dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[TBB+98] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schmidt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, P. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, 1998.

[TBB+99] S. Thrun, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Minerva: A second generation mobile tour-guide robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1999.

[Thr95] S. Thrun. *Explanation-Based Network Learning: A Lifelong Learning Approach*. PhD thesis, Dept. of Computer Science, University of Bonn, Germany, July 1995.

[Tor99] L. Torgo. Predicting the density of algae communities using local regression trees. In *Proceedings of the European Congress on Intelligent Techniques and Soft Computing (EUFIT)*, 1999.

[UB00] I. Ulrich and J. Borenstein. VFH*: Local obstacle avoidance with look-ahead verification. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2505–2511, 2000.

[uRuL95] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, pages 80–94, 1995.

[VCP+95] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[VNE+01] R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *Proceedings of the IEEE Aerospace Conference*, 2001.

[WD92] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[WD99] X. Wang and T. Dietterich. Efficient value function approximation using regression trees. In *Proceedings of the IJCAI-99 Workshop on Statistical Machine Learning for Large-Scale Optimization*, 1999.

[Wel94] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[WF00] I. H. Witten and E. Frank. *Data Mining*. Morgan Kaufman, 2000.

[Wil88] D. Wilkins. *Practical Planning. Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

[WL93] R. J. Williams and C. B. Leemon. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Northeastern University, College of Computer Science, 1993.