Predicting Real-Time Planner Preformance By Domain Charactorization

Jak Kirman

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-44
October 1994

Predicting Real-Time Planner Performance by Domain Characterization

by
Jak Kirman
B.A., Oxford University, June 1983
M.A., Brown University, May 1992

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

October 1994

Acknowledgements

I would like to thank my advisor, Tom Dean, for the help and guidance he has given me, and for showing me what research is all about; I hope I will be as good an advisor as he has been. Leslie Kaelbling has provided endless support and interesting discussions; thanks also to her and Eugene Charniak for wading through the drafts of my thesis and providing useful comments and advice.

I would like to thank Dominic Prior for first introducing me to computers, and Pascal Nicopolsky and all the others at the university of Marseille for making my first year so rewarding. My thanks also to those at Brandeis who put up with endless questions from a budding hacker.

The computer science department at Brown has been a remarkably rich source of friends and colleagues; I would particularly like to thank Moi Lejter, Ken Basye, Ann Nicholson and Gene Santos for collaborations that made my research here so enjoyable, and for all the good times we have had together. Ken and Moi taught me what fun working with mobile robots can be — thanks also to Huey, Louie and Dewey for suffering silently and patiently while we tried to bring them to life. Working with Leslie, Tom and Ann these past two years has been particularly enjoyable, and it has taught me how much more productive team research can be. Brook Conner, Dan Robbins and Jovanna Ignatowicz patiently provided lots of help with Motif programming and graphical interfaces.

Many discussions and diversions with others at Brown have made my life here more pleasant, thanks in particular to John Bazik, Ted Camus, Glenn Carroll, Tony Davis, John Hughes, Michael Littman, Gail Mitchell, Scott Meyers, Kate Sanders, John Shewchuk and Lynn Stein.

My wife Kathy deserves more thanks than I can give her for putting up with many years of late-night robot-hacking and the trials and tribulations of finishing a thesis; she has been enormously supportive, and shouldered far more than her share of keeping our home and family running smoothly. My daughter Hayley makes it all worthwhile with her smile — thanks to Renee and Karleen for keeping her smiling. I would like to thank my parents for their support and encouragement, and for not asking how much longer *too* many times.

Without the technical staff's competence and kindness I would never have been able to finish these experiments; I only hope my next university has such

wonderful support. The administrative staff has smoothed over many a bump for me; thanks for letting me keep my mind on my research.

This work was supported in part by a National Science Foundation Presidential Young Investigator Award IRI-8957601, by the Air Force and the Advanced Research Projects Agency of the Department of Defense under Contract No. F30602-91-C-0041, and by the National Science foundation in conjunction with the Advanced Research Projects Agency of the Department of Defense under Contract No. IRI-8905436.

Contents

List of Figures

Abstract

There is a large class of problems that involve real-time planning and execution in stochastic domains. Classical solutions either ignore the random element while planning and then modify the plan to deal with failures, or compute the best action to perform in any circumstance. These solutions are often either too brittle or too expensive computationally. I present an approach that interleaves planning and execution and reduces the set of circumstances that need to be considered at a given time. In previous work with other researchers I have shown that this approach is practical for some stochastic problems, but left open the question of which problems the approach can be applied to effectively. I present a classification of real-time stochastic decision problems that allows the prediction of the performance of this planning system (and others) without resorting to exhaustive empirical performance estimation. For a number of such planning problems, I present results showing the correlation between the predicted performance and the actual performance as estimated empirically.

Chapter 1

Introduction

The classical AI approach to planning involves modeling the world as a deterministic automaton and finding a sequence of actions to maximize some performance criterion; the time spent on the computation of this sequence is not taken into consideration. However, many practical problems have elements of non-determinism and time pressure.

In order to address these issues, a number of researchers have designed *reactive* systems, which can determine a good (though not necessarily optimal) action to take in any situation within a short, fixed time. In most work to date in this area, any reasoning about the effect of actions in the world is done off-line, and the control system is then equivalent to a lookup table specifying the appropriate action for each situation. When the number of possible situations is very large, this approach becomes impractical both because it is computationally expensive to find appropriate actions for all situations, and because storing the lookup table or equivalent is expensive in space.

I propose a system for planning and execution, Plexus, that is designed to deal with time pressure, with very large spaces of possible situations and with uncertainty in the effect of actions. The core idea is to restrict the attention of the system to the situations that are likely to occur while accomplishing the overall goal, ignoring improbable situations.

It is natural to ask what types of problem are amenable to this approach. I provide a classification for planning problems that allows the performance of Plexus for a given problem to be predicted with little computational expense. I present an empirical study of a set of planning problems and an analysis of the accuracy of the predictions of performance based on the classification.

The central thesis of this dissertation is that such a classification of planning problems provides an effective way to determine whether a particular planning system is appropriate or not for a given problem, and to estimate the cost of using the planning system.

An important component of the dissertation is a careful statistical analysis of the performance of different planners on different problems, providing a firm basis for understanding the type of problem each planner is best applied to. Although many AI researchers propose strategies for attacking real-time planning problems (*e.g.,* [DB90], [BBS93], [Sut90]), there has been very little comparative work, and few attempts to define the range of problems for which these techniques are most effective.

1.1 Example: traffic control

In order to make the discussion that follows more concrete, consider the following example of a planning problem:

> The Transportation Bureau of a large city is interested in installing an automated system to monitor traffic and control the traffic lights at some of its most congested street intersections. They hire an expert to analyze the problem and design a control system for the lights.
>
> Inputs to the control system are provided by cameras mounted above the traffic lights; each camera monitors one of the incoming streets, and indicates (perhaps inaccurately) the number of vehicles waiting at that intersection. The output of the system is a sequence of signals indicating when to change the lights from red to green and *vice versa.* Traffic light changes cannot be arbitrarily fast, so the expert chooses a small interval, say 10 seconds, as the basic time unit; every ten seconds, signals will be sent to those lights that should change their state.
>
> The expert decides to model the traffic flow as a stochastic process — the state of the world is defined by the number of cars waiting at each street entering the intersection and the current settings of the lights. An action is a specification of the traffic lights that should change. The transition matrix, defining the probability of the world changing from one state to another given an action, defines the dynamics of the system. This matrix can be estimated empirically, or the expert can determine it heuristically.
>
> The stochastic process is parameterized by a number of different variables. Some, like the granularity of the inputs and the basic time interval, are controllable. Others, such as the average rate at which cars arrive in each direction, are not known in advance; they can be experimentally determined, but may change over time; the behaviour of the system at 3 a.m. on a weekday morning will be very different from that at Friday rush-hour.

The goal is to minimize the average, over time, of the number of vehicles waiting at the intersections. To represent this, the expert specifies that the reward obtained for being in a state is the negation of the sum of the number of vehicles waiting at each intersection (thus the best state, where no vehicles are waiting, has reward 0, and all other states have negative rewards). The objective is then to maximize the expected average reward over time.

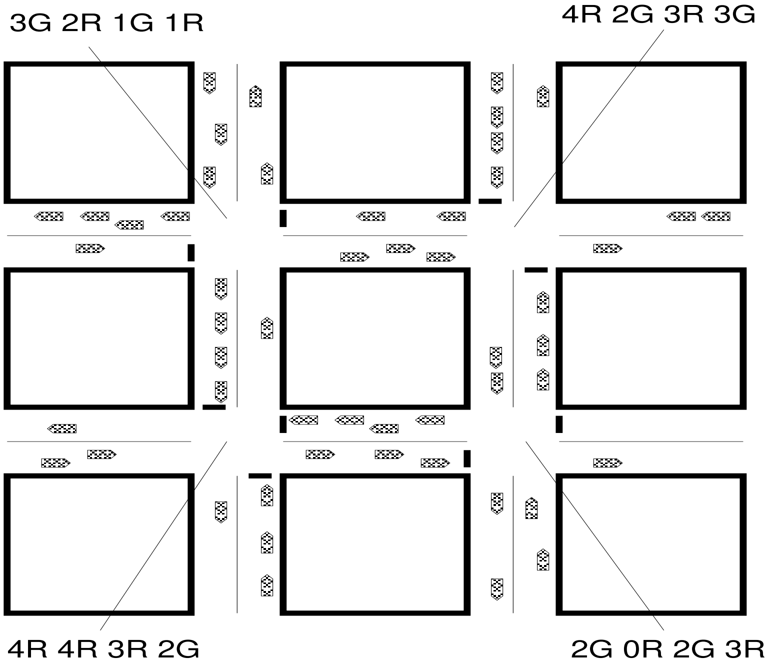


Figure 1.1: Four-intersection traffic world. The state at an intersection is the number of cars waiting to come from each direction, followed by G if the light in that direction is green, or R if it is red. The directions are in the order North, East, South, West.

Figure 1.1 shows an example of a four-intersection system. The thick bars across lanes indicate red lights, and the hatched objects represent cars. The state of the world is the composite of the states at the intersections. The state at an intersection is the number of cars in each of the four incoming directions, and the current light setting. The state in the top left intersection is `3G 2R 1G 1R`: there are three cars coming from the North, with a green light. There are two cars coming from the East, with a red light. There is one car coming from the South, with a green light, and one car coming from the West, with a red light. If the cameras distinguish only up to ten cars in each direction, there are 160,000 states for each intersection, or about 10^{20} states for the four-intersection system.

This traffic world problem has several interesting characteristics. The underlying process is stochastic and difficult to model precisely. The number of states in the stochastic process is large (exponential in the number of intersections). The control task requires a real-time solution; there are bounds on the amount of time that can be taken to decide what action to take next.

These characteristics are common to many problems in planning and control, such as scheduling transportation of troops and materials, job-shop scheduling, robot navigation and air-traffic control.

1.2 An architecture for planning and control

One option for controlling real-time planning systems is to define a *policy* mapping states to actions; a policy indicates what the system should do in any situation. An *optimal policy* is a policy that maximizes some specified measure of the expected reward over time, such as the expected average reward over an infinite horizon. A number of techniques exist to determine an optimal policy for stochastic processes. Given such a policy, a controller for the system is trivial — it looks up the current world state in the policy and executes the indicated action. Unfortunately, it is often computationally infeasible to find an optimal policy, even off-line. This is particularly true if the model is changing dynamically, either to reflect changes in the world, or because the model is being learned on-line.

An alternative approach breaks the controlling task into two parts: an *executive* that uses a fixed policy, allowing it to respond to changes in the world state in a fixed (and very small) amount of time, and a *planner* that determines the policy the executive will use. The executive effectively buffers the planner from the world, allowing the planner to perform lengthy tasks to find better policies, without compromising the reactivity of the system as a whole. Figure 1.2 shows the communications between the components. The executive is given the current state of the world, and must immediately select an action to execute. The result of the action is a new state that is sent to the executive, and the cycle repeats. The executive uses a policy to map from state to action. In parallel with these interactions, the planner watches the sequence of states, and continually tries to improve the policy being used by the executive; each new policy is sent to the executive, and replaces the old one. The two processes are not synchronized in any way; the planner may produce a hundred new policies every step, or one new policy every hundred steps, and the frequency with which it produces new policies may change over time.

Plexus (PLanning and EXecution under Uncertainty System) is a system of this kind that I developed with Tom Dean, Leslie Kaelbling and Ann Nicholson. It is described in detail in Chapter 3. Plexus is designed to operate in domains

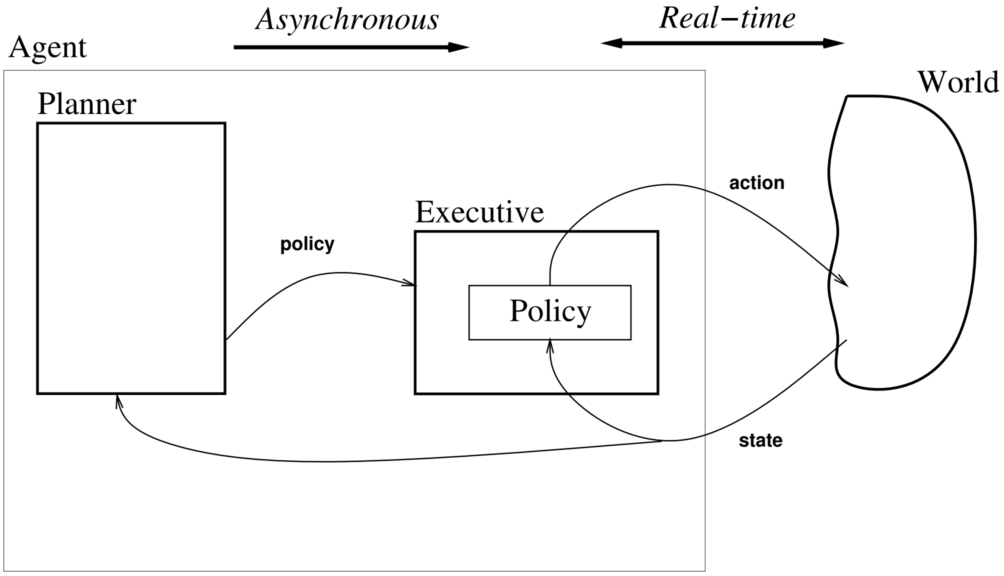


Figure 1.2: Interactions between planner, executive and world

with large numbers of states, of which only a small set will be seen during an execution of the system. Plexus implements an agent consisting of a planner and an executive; the planner reasons about possible futures and best courses of action while the executive follows a policy. Initially, the executive is given a default policy whose intent is to keep the agent out of trouble rather than to accomplish the goal. The executive follows this policy while the planner looks for better policies. The planner does not consider all possible situations, however, but rather those that the system is likely to reach while accomplishing the goal. For these likely states, the planner finds actions that improve the expected value, with respect to the policy the executive is currently using. This association of actions to a subset of the set of states is a *partial policy*. The planner, having found such a policy, passes it to the executive. Until further instruction, the executive will use this partial policy where it is defined, and the default policy elsewhere. This cycle continues until the agent reaches a goal state.

1.3 Contributions

My first contribution is the development of this model of planning and execution. The first part of this development was done jointly with Tom Dean, Leslie Kaelbling and Ann Nicholson. I have since extended the implementation of Plexus by providing algorithms that greatly improve its performance, and I have solidified the theoretical grounding upon which it is based, providing proofs of convergence and optimality for some of the algorithms it uses, and changing some *ad hoc* algorithms to ones with a more solid theoretical basis. Whereas the original implementation of Plexus was hardwired for one particular domain in the interest of efficiency, I have redesigned it to allow the specification of domains with arbitrary representations.

The original implementation of Plexus was tested on one type of domain: robot navigation. In this domain, it out-performed common techniques such as RTDP and search-based techniques. An obvious question was whether the technique was applicable to other domains, or whether some peculiarity of the robot navigation domain made it successful. More generally, what are the characteristics of a problem that make Plexus effective?

I have approached these questions by identifying characterizing aspects (or attributes) of planning problems. By studying the relationship between the performance of Plexus and the attributes of the world it is operating on, I have identified attributes that can be used to predict the performance of Plexus. Many of these attributes are also useful predictors for the performance of an optimal agent, and for the performance of other types of planner, such as RTDP and search-based planners. The attributes quantify notions such as "the amount of uncertainty in the execution of actions" and "the regularity of the reward structure of the problem". They provide a classification of real-time stochastic planning problems. This classification is my second contribution.

My third contribution is an empirical analysis of the performance of the Plexus system on different planning problems, and an analysis of the accuracy of prediction of performance based on the classification presented, using standard statistical techniques such as linear regression. I have selected several different types of planning problems, classified them using a number of attributes, and empirically evaluated their performance by simulation. I have conducted a statistical analysis of the correlation between the domains' attribute values and the planner's performance. This was an iterative process, since the statistical analysis indicated where the classification schema fell short (did not predict performance well), and I used this information to guide the search for more precise classification. To avoid problems of over-fitting the data, I performed tests of predictive accuracy on new sets of domains, different from those used to form the statistical predictive models.

1.4 Outline

In Chapter 2, I present the mathematical notions that underlie the techniques used by the Plexus planning system. I describe the limitations of the system, and how they can be relaxed, and I give an overview of related approaches.

Chapter 3 describes the Plexus system itself, presenting the algorithms for the overall system and the individual policy modification algorithms.

In Chapter 4, I present the notion of domain and task attributes. I define a set of attributes that are useful predictors of performance for Plexus and other planning systems.

Chapter 5 shows how the attributes defined in the previous chapter can be used to predict the performance of Plexus and other planning systems, and provides measures of the effectiveness of the predictions, using statistical techniques.

Chapter 6 summarizes the results and suggests promising avenues for future work.

In the appendices I give some of the detailed proofs and descriptions of the planning problems I studied.

Chapter 2

Background

2.1 Definitions

Overview

In this dissertation I examine control strategies for an agent interacting with its environment in real time. The environment occupies one of a finite number of states at each point in time. At discrete time intervals, the agent observes the current state of the environment and executes one of a finite set of actions. The environment then makes a transition to another state. The transition may be non-deterministic; this makes it possible to model both non-deterministic effects of actions and influences exogenous to the agent. For convenience, I will sometimes speak of the agent, rather than the environment, making a transition from one state to another. In cases where the agent is the only changing part of the environment this phrasing is more intuitive.

Each time a transition is made, the agent receives a reward, determined by the state before the transition and the action taken. The agent's task is to maximize the expected sum of rewards over an infinite horizon. I will assume that transitions from state to state are made at fixed, regular intervals, though I will describe how this requirement may be relaxed.

The remainder of this chapter presents precise definitions of these terms and others that are used throughout the dissertation. Much of the terminology is taken from operations research, where concepts such as Markov decision processes have been studied for decades. In some cases I have adapted the terminology to the AI view of an agent controlling a process, for concordance with other AI researchers. Most of the concepts presented here have been studied extensively in operations research, and applied to a wide range of fields such as fluid mechanics, economics, mechanical engineering and business. The type of problem I am particularly interested in, characterized by very large state spaces, considerable uncertainty, and requirements for on-line adaptability, tend to be very different from the problems studied in those fields, so many of the standard techniques used there cannot be directly applied. Bringing together different approaches used by the very disparate communities working in related areas has been very rewarding, if sometimes frustrating because of the different terminology.

Domain

The *domain* is a model of the agent-environment system, viewed as a stochastic process. It consists of a triple $\{\mathcal{S}, \mathcal{A}, \tau\}$. At discrete time intervals, the environment is observed to occupy one of the states of the finite set $\mathcal{S}$. The agent then executes one of the actions from the finite set $\mathcal{A}$. This causes the environment to make a transition to a new state at the next time interval, and so on.

A more general formulation specifies, for each state, a subset of $\mathcal{A}$ containing the actions that are allowed in that state. For simplicity of presentation, I assume that all actions are allowed in all states, but none of the results presented here depend on this assumption; specifications of algorithms using iteration over all actions may be replaced by iteration over all allowed actions.

The probabilities of transitions are given by the function

$$\tau : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1] \ ;$$

$\tau(s_1, a, s_2)$ is the probability that, given that the environment is in state s_1 and the agent executes action a, the environment will make a transition to state s_2. By definition, $\tau(s, a, \cdot)$ is a probability distribution, *i.e.,*

$$\forall s_1 \in \mathcal{S}, \forall a \in \mathcal{A}, \sum_{s_2 \in \mathcal{S}} \tau(s_1, a, s_2) = 1 \ .$$

This model of the environment has the *Markov property*; the probability of a transition depends only on the current state and action, not on any earlier history of the system. The assumption that the environment can be modelled in this way is restrictive, but Section 2.2 lists means of relaxing this assumption.

Policy

A *policy* for a domain is a mapping from states to actions indicating which action to perform in each state. In general, I consider only deterministic policies, though the notion of a policy can be extended to include non-deterministic policies.[1] The current implementation of Plexus supports only deterministic policies. Formally, a mapping from $\mathcal{S}$ to $\mathcal{A}$ is called a *stationary* policy; a non-stationary policy may also depend on the previous history of the system. Since I consider only stationary policies, I will omit the adjective "stationary".

---

[1] In the non-deterministic case, the policy is a mapping from $\mathcal{S}$ to a probability distribution over the set $\mathcal{A}$.

A *partial policy* is a policy defined on a subset of $\mathcal{S}$. To make the distinction between policies and partial policies clear, I will sometimes refer to a policy as a *complete policy*.

Agent

An *agent* is a process that can influence the environment by performing an action at each time step. Unlike traditional AI planning systems, where the agent is allowed to perform computations for some (often unbounded) time before producing an action, I specify a limit on the time an agent is allowed to produce an action, in order to be able to satisfy the real-time constraints of the overall agent-environment system. Different agent-environment systems will have different real-time requirements; to avoid delving into details of memory access speed and other performance issues, I will consider only agents that respond to state changes in an amount of time that is negligible compared to the time between states.

An agent consists of two parts: an *executive* and a *planner*. The executive is a simple module that has has a complete policy, called the *default policy* and a partial policy, defining actions for some subset of the states in the world. When the agent is required to perform an action, the executive looks at the current state. If it is in the envelope of the partial policy, the executive issues the action specified by the partial policy. Otherwise, it issues the action specified by the default policy. The planner generates the partial policies for use by the executive. Figure 2.1 shows an agent and an environment. States are represented by circles; the shaded state is the current state of the environment.

Tasks and Rewards

The domain encodes information about the operation of the environment, but it does not encode any information describing how to evaluate the performance of the agent. I specify the performance metric for the agent separately, as a *task*. This distinction is natural, since there may be quite different tasks for an agent in the same domain; also, the same task may sometimes be used for different domains (*e.g.,* with different transition functions).

The task is specified by means of four components

1. *instantaneous reward*

   a function ρ, mapping from $\mathcal{S} \times \mathcal{A}$ to $\mathbb{R}$. When the environment is in state s and the agent takes action a, the agent receives a reward of $\rho(s, a)$. The agent tries to maximize the sum of the rewards it receives.

2. *start state*

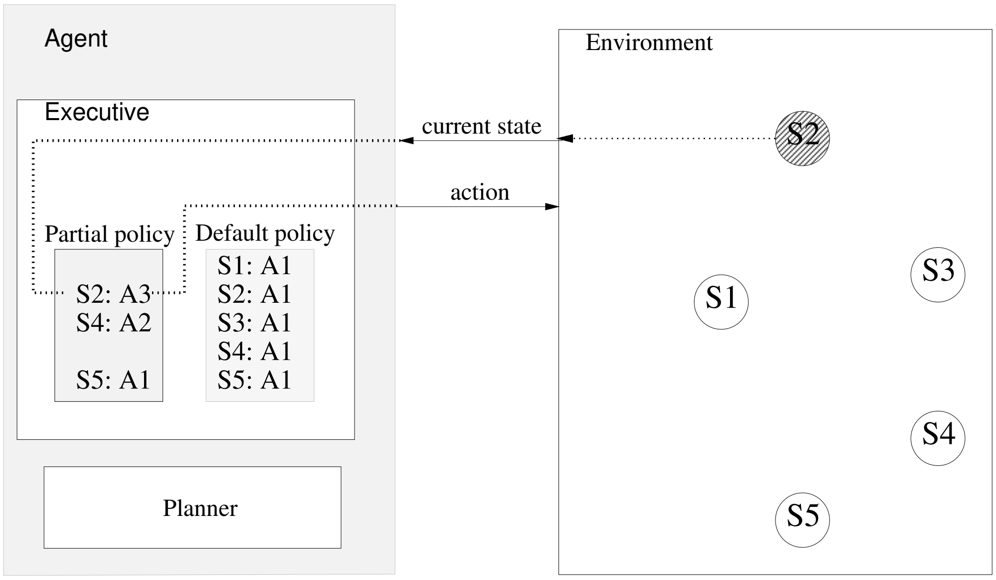


Figure 2.1: Interactions between agent and environment

s_0 is the element of $\mathcal{S}$ that the environment will occupy when the system is started. In some cases, the initial conditions are specified by a probability distribution over states rather than a single state. This can be represented by making a minor change to the domain.

3. *discount factor*

   a real number $\gamma \in [0, 1)$. This is used to discount expected future rewards. A reward r expected after k steps is equivalent to an immediate reward of $\gamma^k r$.

4. *termination states*

   $\mathcal{X}$ is a possibly empty subset of $\mathcal{S}$. The agent-environment system halts as soon as the environment enters one of the states in $\mathcal{X}$. The reward accumulated by the agent up to that point is the performance of the agent on that run.

Markov Decision Process

A more common (and equivalent) representation of this type of problem is in terms of a *Markov decision process*. A Markov decision process consists of a set of states $\mathcal{S}_M$, a set of actions $\mathcal{A}_M$, a transition function τ_M that specifies, for each state-action pair, a probability distribution over states corresponding to outcomes, and a reward function ρ_M that maps states and actions to the reals.

The relationship between the two formulations is straightforward; a domain $\{\mathcal{S}, \mathcal{A}, \tau\}$ and a task $\{\rho, s_0, \gamma, \mathcal{X}\}$ together describe a Markov decision process as follows. We define a new state `terminated`, not in $\mathcal{S}$, and let the Markov process be defined by:

$$
\begin{aligned}
&\mathcal{S}_M = \mathcal{S} \cup \{\texttt{terminated}\} \\
&\mathcal{A}_M = \mathcal{A} \\
&\tau_M(s, a, s') = \begin{cases} \tau(s, a, s') & \text{if } s \in \mathcal{S} - \mathcal{X}, s' \in \mathcal{S} \\ 1 & \text{if } s \in \mathcal{X} \cup \{\texttt{terminated}\}, s' = \texttt{terminated} \\ 0 & \text{if } s \in \mathcal{X} \cup \{\texttt{terminated}\}, s' \neq \texttt{terminated} \end{cases} \\
&\rho_M(s, a) = \begin{cases} \rho(s, a) & \text{if } s \neq \texttt{terminated} \\ 0 & \text{if } s = \texttt{terminated} \end{cases}
\end{aligned}
$$

The advantage of the domain-task representation over the Markov decision process representation is that it is natural in many problem specifications to refer to a set of tasks for a particular domain, and to a set of domains for a particular task. Comparison of domains is a major thread in this thesis, so I will generally describe problems in the domain-task representation, and sometimes refer to the *associated Markov decision process.*

Performance

Given an agent, a task, and a domain, the performance of the agent is defined as the expected discounted sum of the instantaneous rewards obtained at each time step until the system enters one of the termination states, with the reward on the nth step being discounted by γ^n. Expectation is taken over all possible trajectories (sequences of actions/states) that the agent-environment system may follow.

To define this quantity more precisely, let $\{\mathcal{S}_M, \mathcal{A}_M, \tau_M, \rho_M\}$ be the Markov process associated with the domain and the task. Now consider the possible trajectories of the Markov decision process, specifying the state occupied by the process at each step and the action taken by the agent at that step. Such a trajectory is called a history; an n-step history is thus

$$H_n = \{s_0, s_1, \ldots, s_n; a_0, a_1, \ldots, a_{n-1}\} \ .$$

An agent α defines a mapping from histories to probability distributions over actions, specifying the probability that the agent will choose a given action at step $n + 1$ following the n-step history. This is a probability distribution over actions, not simply an action, because the agent may not be using the same policy throughout execution — the policy is repeatedly updated by the planner. Determining this probability distribution theoretically may be infeasible, since it requires reasoning about the policies produced by the planner for a given history, and this itself may be non-deterministic. Note that the agent is operating in an environment that has the Markov property — all the information necessary to (probabilistically) predict future states is contained in the current state. However, the agent may change policies at arbitrary times, so the agent and the environment viewed together as a process does not necessarily possess the Markov property.

I use the notation $\mathrm{P\scriptstyle R}_\alpha(H_n, a)$ for the probability that an agent α will choose action a at step n given the n-step history H_n. Since each step is independent, this probability can be written

$$\mathrm{P\scriptstyle R}_\alpha(H_n) = \prod_{i=0}^{i=n-1} \tau_M(s_i, a_i, s_{i+1})\, \mathrm{P\scriptstyle R}_\alpha(H_{i-1}, a_i) \ ,$$

that is, the product of the probabilities that the system will make each of the transitions given the agent's action, and that the agent will take that action given the prior history of the system.

The value of an n-step history is the discounted sum of the rewards accumulated by the agent:

$$V(H_n) = \sum_{i=0}^{i=n-1} \gamma^i \rho(s_i, a_i) \ .$$

Let $\mathcal{H}_n$ be the set of n-step histories; then the expected performance of an agent over n steps is the discounted value accumulated over each history, weighted by the probability of that history:

$$EP_n(\alpha) = \sum_{H_n \in \mathcal{H}_n} \mathrm{P\scriptstyle R}_\alpha(H_n) V(H_n) \ .$$

In Appendix A.3 I present a proof that $EP_n(\alpha)$ is a Cauchy sequence on the reals, and since the reals are a Banach space, $EP_n(\alpha)$ converges to a finite limit as n tends to infinity. The expected performance of an agent α is this limit.

$$EP(\alpha) = \lim_{n \to \infty} EP_n(\alpha) \ .$$

I show in Section 3.11.1 that this formulation of tasks allows us to represent classical tasks of achievement and maintenance as well as prioritized versions of these tasks.

A domain and a task in fact represent both a Markov decision process and the initial conditions required to compute the utility of the process, which is the term used in operations research for what I have called the performance of the agent.

Value of states

Given a policy π and a reward function ρ, we define the *value* of a state $s \in \mathcal{S}$, $V_{\pi,\rho}(s)$, as the sum of the expected values of the rewards to be received at each future time step, discounted by how far into the future they occur. For convenience, I will abbreviate $V_{\pi,\rho}$ as V when the policy and reward function are clear from the context.

Thus,

$$V(s) = \sum_{t=0}^{\infty} \gamma^t E(R_t) \ ,$$

where R_t is the reward received on the t-th step of executing policy π after starting in state s. Due to the fact that both τ and π are stationary (time-invariant), V can be rewritten as

$$V(s) = \rho(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} \textrm{P}{\scriptstyle\textrm{R}}(s, \pi(s), s') V(s') \ . \tag{2.1}$$

Optimal policy

For a given reward function ρ, we say that policy π *dominates* (is better than) policy π' if, for all $s \in \mathcal{S}$, $V_{\pi,\rho}(s) \geq V_{\pi',\rho}(s)$, and for at least one $s \in \mathcal{S}$, $V_{\pi,\rho}(s) > V_{\pi',\rho}(s)$. A policy is optimal if it is not dominated by any other policy.

An optimal policy is the gold standard for solutions to a domain-task problem; it provides the action to take in every state in order to maximize the discounted sum of future rewards. Optimal policies are not necessarily unique, but the value of a state is the same for all optimal policies.

Markov Chains

A *Markov chain* models a discrete-time stochastic system. It is defined by a set of states $\mathcal{S}$, and a transition function P giving the probability of the system moving from one state to another; the function is defined by

$$P : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R},\ P(i,j) = \textrm{P}{\scriptstyle\textrm{R}}(s_n = j | s_{n-1} = i),$$

and satisfies $\forall i \in \mathcal{S}, \sum_{j \in \mathcal{S}} P(i,j) = 1$. s_k is the state of the system at time k.

A domain $\{\mathcal{S}, \mathcal{A}, \tau\}$ and a policy π induce a Markov chain with the same set of states $\mathcal{S}$, and with the transition function P defined by:

$$\forall s_1, s_2 \in \mathcal{S}, P(s_1, s_2) = \tau(s_1, \pi(s_1), s_2).$$

A state s' is said to be *reachable* from a state s, written $s \twoheadrightarrow s'$ if the system can move from s to s', directly or indirectly:

$$s \twoheadrightarrow s' \equiv \left\{ \begin{array}{ll} P(s,s') \neq 0 & \textrm{or} \\ \exists s'' \in \mathcal{S}, s \twoheadrightarrow s'' \textrm{ and } s'' \twoheadrightarrow s' & \end{array} \right.$$

Two states s and s' can *communicate* if s is reachable from s' and s' is reachable from s.

A *recurrent set* of states in a Markov chain is a set in which all pairs communicate, but no state outside the set is reachable from a state inside the set.[2] Alternatively, we can partition the set of states into equivalence classes using the communicate relation, and use reachability to define a partial ordering: an equivalence class E_1 is greater than E_2 if a state in E_2 can be reached from a state in E_1:

$$E_1 \succeq E_2 \equiv \exists s_1 \in E_1, s_2 \in E_2 | s_1 \twoheadrightarrow s_2 \ .$$

The minimal elements of this partial ordering are exactly the recurrent sets of the chain.

Recurrent sets can be thought of as *sinks*; once the system enters such a set, it can never leave it. The elements of a recurrent set are called *recurrent states.* States that are not in a recurrent set are called *transient states.* A special case of a recurrent set is a single state from which the only transition is to itself. Such a state is called an *absorbing state.* An *absorbing Markov chain* is a Markov chain in which every recurrent set consists of a single absorbing state. Absorbing chains have useful computational properties that I will exploit throughout this thesis. For more extensive definitions of Markov chains and associated notions, see, *e.g.,* Kemeny and Snell [KS76].

2.2 Restrictions

The approach taken by Plexus is not intended to be applicable to all planning problems; it is specifically designed for problems in which uncertainty of the effects of actions, or uncertainty about external events, play a major role in the evolution of the environment. For deterministic environments, standard search techniques are more appropriate. Plexus is also designed for environments that do not settle into small areas of the state space over time, as might the traffic light system at the dead of night. For such environments, computing the optimal policy for the small number of states visited is sufficient. Plexus is principally useful for problems in which the following three conditions hold. The task is non-terminating or repeated, the conditions change over time (because the system moves through the state space over time, or because different goals are posited at different times), there are too many possible conditions to consider them all at once, and some degree of reactivity is required.

---

[2] Note that some authors, *e.g.,* Kemeny and Snell [KS60], Bertsekas [Ber76], use the term *ergodic set* for this concept. Others, *e.g.,* Karlin and Taylor, [KT74]) use ergodic to mean aperiodic and recurrent, so to avoid confusion I follow Howard [How66] in using the term recurrent.

The current implementation of Plexus imposes some restrictions on the planning problems it operates on. Some of the restrictions are a consequence of the theory underlying the approach, and relaxing them would require substantial modifications to the techniques I describe below. Other restrictions are imposed for simplicity, and can be relaxed with some impact on performance. In this section I describe the restrictions in detail, explaining which of these two categories they fall into, and sketching ways of relaxing them.

- Discreteness of states and actions

  The set of states $\mathcal{S}$ and the set of actions $\mathcal{A}$ are finite sets.

  This is a standard assumption, as techniques for determining optimal policies are substantially more complex to represent and to analyze on continuous, or even denumerably infinite spaces.

- Discrete time steps

  The environment operates in discrete time steps; when the agent specifies an action, the environment makes a transition to a new state, and the agent cannot observe this new state until the next time point.

  Relaxing this assumption would require modeling the problem as a semi-Markov decision process. There is some literature on this subject (see, *e.g.,* [Ros70]), but it is beyond the scope of this dissertation. In practice, semi-Markov decision processes tend to be much harder to solve than Markov decision processes.

- Accurate world model

  I assume that the model of the behaviour of the environment and its interactions with the agent is accurately specified by the domain structure. There is a growing interest in techniques that learn a model of the environment while using an approximate model (see, *e.g.,* [BBS93], [SPS94]); some of these techniques are applicable to the system described here.

- Constant-time actions

  The time taken to make a transition from one state to another is assumed to be fixed and independent both of the original state and the action taken. This assumption makes it much simpler for the planner to reason about the possible future states of the system.

  The assumption can easily be relaxed to allow the time taken to be dependent on the original state and the action taken, provided that it is fixed for each state/action pair, and that the durations are approximated as multiples of a constant. In the worst case, the additional complexity corresponds to multiplying the size of the state space by the largest duration divided by this constant, but in practice the algorithms presented here should not be significantly impacted by this modification. Allowing the duration of an action to be specified as a distribution instead of a constant requires the full power of semi-Markov decision processes, and is beyond the scope of this dissertation.

- Markov property

  The domain satisfies the Markov property: the outcome of an action in a state depends only on the action and the state, not on any previous history of the system.

  At some computational expense, this assumption can be relaxed to allow the behaviour of the environment to depend on the k previous steps, by using a domain in which each state corresponds to a sequence of k states in the original model. This requires $|\mathcal{S}|^k$ states. While this may appear impractical at first sight, note that typically the vast majority of these state sequences are impossible, and therefore need not be explicitly represented.

- Perfect state information

  The agent can determine exactly which state the environment occupies at any time.

  This assumption is unrealistic in many domains, and is made here for simplicity with the hope that future work will allow it to be relaxed. Recent work by Cassandra, Kaelbling and Littman discusses techniques for dealing with this case; see, *e.g.,* [CKL94a], [CKL94b].

- Infinite horizon

  The accumulated reward is to be maximized over an infinite horizon. That is, I do not address problems in which the system will be halted after a fixed number of steps. Problems in which the system is halted after reaching one of a set of states (first-passage problems), however, are amenable to the techniques presented here, as shown in Section 3.11.1.

  Finite horizon problems have been extensively studied in the stochastic control literature (see *e.g.,* [Der70, Ber76, HS82]), and in principle the techniques described in this literature could be applied to the planning system described here. I restrict myself to the infinite horizon case solely for efficiency — in general, policies for the finite horizon case are non-stationary, and require time proportional to the horizon to produce.

  The implementation of the Plexus phases described in Section 3 rely on this assumption.

- Present value and risk-neutrality

  Most AI approaches using Markov decision problems as the underlying formalism use expected reward as the criterion of optimality; in some cases expected discounted sum of rewards over an infinite horizon, in others expected average reward.

  The operations research community has long been aware that expected reward is often not the best criterion to use. Sometimes it is more desirable to have a guaranteed lower bound on accumulated reward, and sometimes one would like to incorporate a notion of risk-aversion or risk-preference. Heger [Heg94] discusses the various options, and provides an algorithm similar to Q-learning [Wat89] that uses worst-case total discounted costs.

  I consider only the risk-neutral case, where the expected accumulated discounted reward is optimized. Expectation of future rewards is discounted by an amount increasing geometrically with the temporal distance from the present. This requirement is made only for the policy iteration mechanism. I present a rationale for this choice and a suggestion of an alternative mechanism to deal with undiscounted rewards in Appendix A.2.

2.3 Overview of other types of solutions

2.3.1 Planning in deterministic domains

Classical planning systems assume an environment in which the agent's actions are the only causes of change, that the outcome of an agent's actions are deterministic, and that the agent can accurately observe its initial state. Various approaches have been taken to relax these assumptions individually, such as the addition of plan recovery described below, but such approaches are generally effective only when the failure in assumptions is fairly benign. For example, plan recovery is useful when the most common outcome of each action is highly probable, and failures are the exception, or when exogenous causes of change are rare.

Most such systems do not consider the time taken by the planner itself in determining performance; performance is characterized by the cost of the execution once the plan has been found. The task specification is usually a set of goal states, and the cost of each action is the same; the problem then becomes to find the shortest sequence of actions which will take the agent from its initial state to a goal state. Specification of the solution to a planning problem as a fixed sequence of actions is called an *open-loop* control policy: the outcome of the action is not fed back into the controller. In contrast, a *closed-loop* control policy is given information about the current state of the system upon which to base the choice of action. In deterministic domains, there is always an open-loop policy that is optimal. However, if the outcomes of actions may be non-deterministic, or if exogenous agents can influence the environment, this is no longer the case; there are closed-loop policies that are strictly better than any open-loop policy. In most of the domains I describe here, open-loop controllers have hopelessly bad performance.

2.3.2 Deterministic planning with plan recovery

Some AI systems, such as SIPE [Wil85] use a deterministic model of the environment in formulating a plan, and then monitor the execution of the plan to detect failures in the actions. This approach is effective when there is a deterministic model that is generally accurate. In domains where the outcomes of actions are inherently uncertain, ignoring the uncertainty can be highly inefficient, since it makes it impossible for the system to reason about contingencies. Systems that model the environment deterministically will usually choose either the worst case outcome for each state and action, or the most likely outcome. Both of these have disadvantages.

As an example, consider a robot navigating around a stairwell that is open on one side. The robot may choose to go around the stairwell in either direction, and if the environment is represented deterministically, the model will not distinguish between the two actions — their most likely outcome is a successful circumnavigation. The planner cannot see that it is better to avoid the side with the opening because there is some small probability that it will fall down the stairwell. If the worst-case outcome is used, the agent cannot distinguish between actions that are likely to be good, but might be bad, and actions that are certain to be bad. This is clearly undesirable; if low-probability unpleasant (resulting in large negative rewards) events are modeled, the agent will behave overly cautiously; if the events are not modeled, the model is inaccurate in an important way.

2.3.3 Probabilistic planning

Draper, Hanks and Weld [DHW94] present an algorithm for probabilistic planning called C-BURIDAN, based on the BURIDAN algorithm by Kushmerick *et al.* [KHW94]. Their approach is more general than ours, in the sense that it does not assume complete observability of state. It is basically equivalent to the partially observable Markov decision process (POMDP) formalism, which has been used for AI planning problems by Cassandra, Kaelbling and Littman [CKL94a]; the main difference is the compact STRIPS-like representation of the problem that C-BURIDAN uses, whereas the classical POMDP representation can be prohibitively large. The principal disadvantage of these techniques is their computational complexity; computing an optimal policy for a POMDP is usually orders of magnitude harder than computing an optimal policy for an MDP. However, recent work on approximately optimal policies shows promise for POMDPs.

2.3.4 Purely reactive systems

A different approach to the problem of controlling agents is taken in *reactive systems*. Here the assumptions are that the environment is non-deterministic, there may not be an accurate model of it, and that there are firm requirements on the response time. The approaches mentioned above are generally not applicable in this type of domain, as they cannot guarantee timely responses.

Purely reactive systems do no on-line reasoning about the effects of actions; any implicit or explicit projections are done off-line. Examples of purely reactive systems include Brooks' subsumption architecture [Bro85] and Agre and Chapman's video-game-playing Pengi system [AC87].

Another type of reactive system is a *universal plan*, a concept introduced by Schoppers [Sch87]. Schoppers' approach requires a specification of the domain in terms of a deterministic process, and assumes that the agent can accurately observe the current state at all times. As with deterministic planning, a set of goal states is specified, and the task is to reach a goal state using as few actions as possible. Planning is done under the assumption that the outcomes of actions will be as specified by the deterministic model, but a complete policy is computed, specifying what action to take in each state. Thus if the process behaves differently from the model, for example because an exogenous influence modifies the state of the agent, the agent can still respond in a timely manner by following its policy. The policy computed is optimal if the model is correct. If exogenous influences disturb the process, the policy may not be optimal, but as long as these disturbances are not systematic, the policy will usually be effective.

2.3.5 Reactive and planning components

A number of more recent approaches to planning relax some of the assumptions made by earlier planners. The assumptions that are relaxed are: complete observability, deterministic outcomes and unlimited reasoning time.

Sanborn [San88] uses an approach he calls dynamic reaction. He assumes that the domain is not fully known in advance, and uses monitors to detect changes in the environment that affect the current plan, replanning if necessary.

CIRCA (Musliner, Durfee, Shin) [MDS93a, MDS93b] is a real-time system connected to a planner in a manner similar to Plexus. However, CIRCA does not explicitly deal with the uncertainty in the model; it assumes the worst-case behaviour whenever there is a doubt. This can lead to very poor performance in uncertain domains, since the agent is utterly pessimistic. As with Plexus, CIRCA requires complete state observability. An additional assumption made in CIRCA that is difficult to justify is that given a state, the best action can be chosen with no lookahead. This is used to avoid searching the space of all possible action sequences.

Drummond and Bresina [DB90] present a technique called *anytime synthetic projection* that is similar in many respects to the ideas behind Plexus; the original idea for Plexus' strengthening came from Drummond and Bresina's work.

2.3.6 Abstraction

There is a growing recognition of the need for abstractions to reduce the size of the problems in planning under uncertainty. Several different approaches have been taken.

Dearden and Boutillier [BD94] consider problems in which the states are represented as vectors of state variables. They cluster states by ignoring certain state variables, and then determine an optimal policy on the decision problem involving the clustered states. As time permits, they cluster states into smaller and smaller groups, giving a progressively more accurate approximation. Nicholson and Kaelbling [NK94] use a method similar to that of Dearden and Boutillier; they form approximate models by ignoring state variables, and provide techniques for automatically determining which variables should be ignored.

Researchers in the field of operations research field have studied aggregation techniques in which states are clustered according to estimates of their value, rather than by state variable. Bertsekas and Castañon [BC89] present one method that is applicable to very large state spaces. Kushner and Kleinman [KK71] present a method based on linear programming that does aggregation on discretized versions of continuous spaces.

Other techniques for abstraction in deterministic domains have been studied by Lin and Dean [LD94] and Knoblock [Kno91].

2.3.7 Learning

Several groups in the learning community have addressed the problem of finding good policies in stochastic domains, using a model of the problem (usually as a Markov Decision Process). In some cases the model is assumed to be known, in others it is learned on-line.

Korf [Kor87] presented an algorithm called RTA$\star$ (Real-Time A$\star$) to do concurrent planning and execution in deterministic domains. RTA$\star$ is a modification of the heuristic search algorithm A$\star$ to do intelligent back-tracking, and to provide *anytime* solutions — successive approximations that improve over time (the term *anytime* is due to Dean and Boddy [DB88]. Under a few reasonable assumptions, RTA$\star$ is guaranteed to eventually find a solution to the problem (a sequence of state-action pairs that leads to a goal state). RTA$\star$ is not applicable to the type of problem I address, since it presumes a deterministic environment, but extensions of it are; *e.g.,* RTDP, described below. Korf also presents an algorithm called LRTA$\star$ (Learning RTA$\star$), a modification of RTA$\star$ for the case when multiple tasks are solved in the same domain.

Sutton [Sut90] presented a class of architectures called Dyna, for solving real-time planning problems. Like Plexus, the Dyna architectures consist of a separate executive component and planning component. The planning component also learns the world model on-line, and generates policies that are designed both to increase the rewards received and to explore parts of the state space that have been visited infrequently. The performance of Dyna degrades with large state spaces, since it examines all states. Peng and Williams [PW93] suggest techniques for improving the performance of Dyna, for example by ascribing priorities to the nodes on the search boundary. Moore and Atkeson [MA93] present a similar technique called prioritized sweeping.

Much of the work in the area of learning Markov decision processes is based on Watkins' Q-learning [Wat89]. Q-learning is a process for updating estimates of the values of each state-action pair as transitions are made. The most interesting aspect of Q-learning is that it simultaneously learns the values associated with state-action pairs and converges to an optimal policy, without ever explicitly formulating a model of the process.

Schwartz [Sch93] presents an extension of Q-learning that uses a more sophisticated optimality criterion than the usual infinite-horizon discounted criterion. He defines the value of a state in terms both of the expected average reward per step and in terms of the *average-discounted* reward: roughly speaking, the part of the expected value not accounted for by the average reward. This allows problems to be specified without discounting, yet distinguishes between policies that have the same value asymptotically, based on whether they accumulate reward quickly or slowly. There are a number of other optimality criteria; Puterman [Put94] gives a good overview of them.

2.3.8 Policy improvement

There has been a great deal of work on various forms of policy improvement for Markov decision processes, both in operations research and in machine learning. The standard techniques of policy iteration and value iteration (described in Section 3.5 and 3.6) are quite inefficient. Broadly speaking, value iteration is inefficient because it updates all states in the domain equally often; policy iteration is inefficient both because it requires solving a set of $\mathcal{S}$ linear equations for each state updated, and because it only updates one state at a time.

Jalali and Ferguson [JF92] give an asynchronous variant of value iteration that updates the states with larger values more frequently than those with lower values (using the average-reward measure of optimality). This leads to substantially quicker convergence to the optimal policy in practice — several orders of magnitude in the examples they report. They show that, under certain fairly weak assumptions, convergence is guaranteed.

Puterman defines a procedure called *modified policy iteration* that incorporates elements of both policy improvement and value iteration. As with policy iteration, the procedure alternates between value computation and policy improvement, but value computation is not performed exactly. A sequence of value iteration steps are performed, giving a closer estimate to the values of the states, but the sequence is halted before complete convergence. Puterman [Put94] proves that modified policy iteration converges strictly faster than value iteration, and gives empirical evidence that, in practice, it is significantly faster than either value iteration or policy iteration.

Chapter 3

Plexus

This chapter describes the Plexus planning and execution system. The first version of Plexus (see [DKKN93b, DKKN94]) was the outcome of work done jointly with Tom Dean, Leslie Kaelbling and Ann Nicholson. I have since made a number of modifications to Plexus as a part of this dissertation; the differences between the original and the current version are described in Section 3.10.

3.1 Overview

Plexus (PLanning and EXecution under Uncertainty System) is an implementation of the agent-environment system described in Chapter 2. Recall that an agent consists of an executive and a planner, and that the executive has a partial policy and a default policy.

The planner is given a domain and a task, as described in Section 2.1. While the executive is acting using the default policy, the planner generates a first partial policy, specifying actions for some subset of the world. It is obviously desirable for the partial policy to be better than the default policy, but it is usually impossible to guarantee this without searching the entire state space.

Once the planner has produced a first partial policy, this policy is sent to the executive. The executive uses this new partial policy when it is applicable, falling back on the default policy when it is not. Meanwhile, the planner is looking for a new (better) partial policy, which it will send to the executive, *etc.*. Figure 3.1 shows the high-level interactions between the different components.

Plexus does not impose any restrictions on the manner in which planners produce sequences of partial policies, but all the planners described in this dissertation repeatedly apply a fixed sequence of procedures that (at least in expectation) improve the policy. Examples of this type of planner are given in Section 3.9.

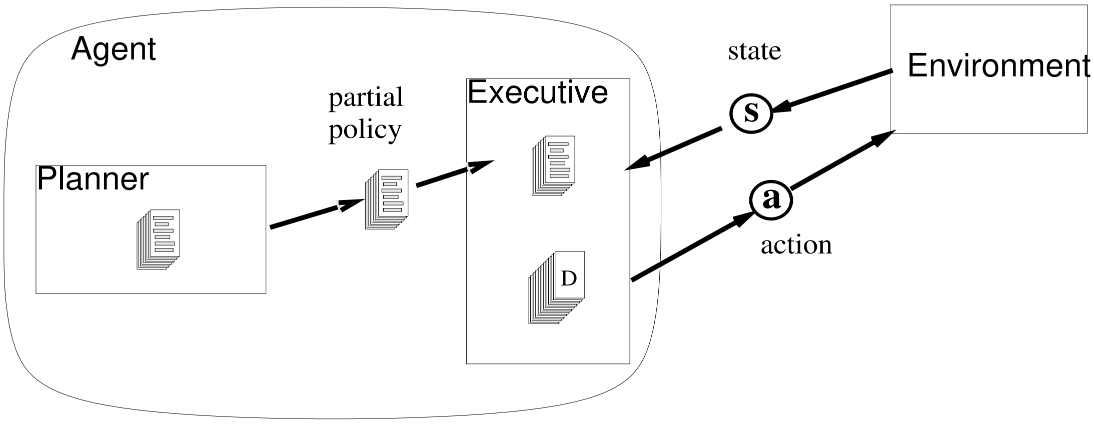


Figure 3.1: Interactions between components of Plexus

3.2 Example domain

The RN1 domain (described in detail in Appendix B.3.1) is a simple robot navigation domain; it is not intended to model any real problem, but is simply an expository tool.

A robot moves around a rectangular grid of cells. It occupies one cell at any given moment. The state of the environment is completely specified by the location of the robot, so states are named L-r-c, where r and c are the row and column of the cell, starting at 1.

The actions available to the robot are to move North, South, East or West, or to Stay: $\mathcal{A} = \{N, E, S, W, Stay\}$.

The outcome of an action is stochastic; with some probability p the action succeeds, and the robot moves one cell in the specified direction. If the action is not successful, it is equally likely to take the robot in one of the diagonals adjacent to the desired direction. Any outcome that would result in the robot moving off the grid instead leaves the robot where it was. Figure 3.2 shows a 5x5 RN1 world. A few of the cells are labeled with the associated state names to show the naming scheme. For compactness, in the diagrams I write simply “rc” for L-r-c. Since the examples all use fewer than ten rows and columns, this abbreviation should not cause confusion. The probability of success in this example is 0.8. The robot's location is represented by a circle with an arrow indicating the direction of the action.

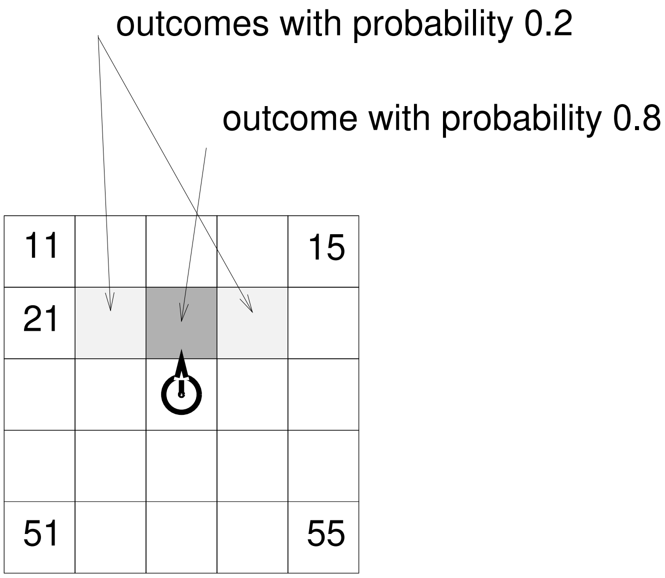


Figure 3.2: 5x5 RN1 world, with outcomes shown for L-3-3 and action North.

3.3 Definitions

In the following, recall that $\{\mathcal{S}, \mathcal{A}, \tau\}$ is a domain and $\{\rho, s_0, \mathcal{X}, \gamma\}$ is a task on that domain.

3.3.1 Envelope

Let π be a partial policy. The set of states for which π is defined is called the *envelope* of the policy, and is denoted $\mathcal{E}(\pi)$, or simply $\mathcal{E}$ if some policy is unambiguously under consideration. A partial policy policy π is thus a mapping from $\mathcal{E}(\pi)$ to $\mathcal{A}$, where $\mathcal{E}(\pi) \subset \mathcal{S}$.

Figure 3.3 shows a policy and its envelope for the 5x5 RN1 domain. Envelope states are shown in black, non-envelope states in white. The arcs indicate the possible outcomes of the action specified by the policy; the policy is not explicitly represented in the figure, but can easily be inferred from the possible outcomes; for example, if the outcomes from a state are all to the south of the state, the action in that state must be S.

3.3.2 State Communication

If two states s and s' are in the envelope of a (partial or complete) policy π, s' is said to be a *successor* of s if there is an action a such that $\tau(s, a, s') \neq 0$. By analogy with the notion of reachability for Markov chains given in Section 2.1,

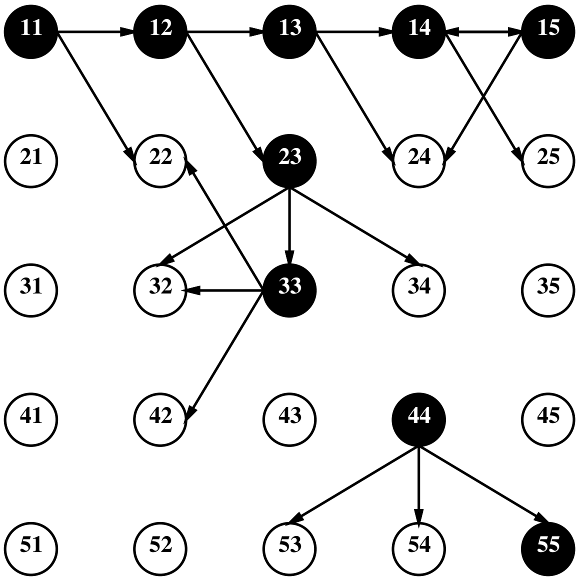


Figure 3.3: A policy $\{\text{L-1-1} : E, \text{L-1-2} : E, \text{L-1-3} : E, \text{L-1-4} : E, \text{L-1-5} : W, \text{L-2-3} : S, \text{L-3-3} : W, \text{L-4-4} : S, \text{L-5-5} : S\}$ and its envelope.

a state s' is said to be *reachable* from state s using a partial policy π, written $s \overset{\pi}{\twoheadrightarrow} s'$, if there is a sequence of states from s to s' that could be followed using the policy. That is, each element of the sequence must be in the envelope of π, and be a successor of the previous element, using the action specified by the policy for the previous element. Formally,

$$s \overset{\pi}{\twoheadrightarrow} s' \equiv \begin{cases} \tau(s, \pi(s), s') \neq 0 & \text{or} \\ \exists s'' \in \mathcal{E}(\pi), s \overset{\pi}{\twoheadrightarrow} s'' \text{ and } s'' \overset{\pi}{\twoheadrightarrow} s' \end{cases}$$

3.3.3 Fringes

Given a partial policy, we are often interested in knowing what the effect of executing it will be. In particular, if the system leaves the envelope of the policy, where will it end up? States outside the envelope that the system is likely to reach are good candidates for further exploration. To make these notions more precise, we define three kinds of fringes of an envelope.

Accessible fringe

We define the *accessible fringe* $\mathcal{AF}(\pi, c)$ of a partial policy π from a current state c, as the set of reachable states that are not in the envelope:

$$\mathcal{AF}(\pi, c) = \{s \in \mathcal{S} - \mathcal{E} \mid c \overset{\pi}{\twoheadrightarrow} s\} \ .$$

Figure 3.4 shows the accessible fringe of the policy described above, given start state L-1-1. Note that states L-5-3 and L-5-4 are not part of the accessible fringe because states L-4-4 and L-5-5 cannot be reached from the start state using the current policy.

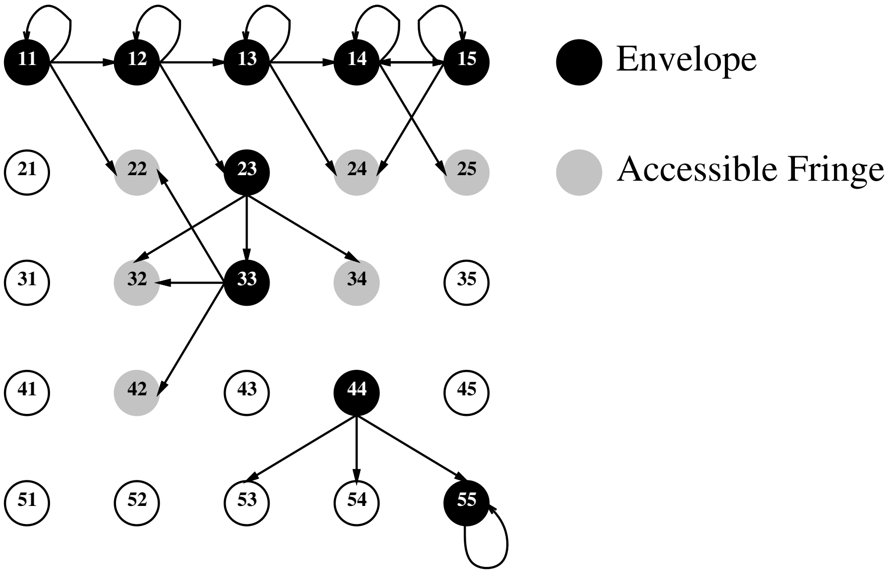


Figure 3.4: Envelope and accessible fringe. $\mathcal{AF} = \{$L-2-2, L-2-4, L-2-5, L-3-2, L-3-4, L-4-2$\}$.

If the agent uses the partial policy π, starting in state c, and at some point falls out of the envelope of π, it will be in one of the states of the accessible fringe.

We define the *extended fringe* $\mathcal{EF}(\pi, c)$ of a partial policy π from a current state c, to be those states that are not in the envelope or accessible fringe, but that are successors of a reachable element of the envelope using some action:

$$\mathcal{EF}(\pi, c) = \{s \in \mathcal{S} - (\mathcal{E} \cup \mathcal{AF}(\pi, c)) \mid \exists\, s' \in \mathcal{E}, c \overset{\pi}{\twoheadrightarrow} s' \text{ and } \exists\, a \in \mathcal{A}, \tau(s', a, s) \neq 0\}\ .$$

Figure 3.5 shows the extended fringe of the policy described above with start state L-1-1. Note that the extended fringe excludes all states in the accessible fringe.

The reason the extended fringe is useful is that a partial policy is executed for some period of time, and then replaced by a new partial policy. The extended fringe contains states the agent could reach after some time executing the original partial policy, and then taking one step of a new partial policy.

We define the *full fringe* $\mathcal{FF}(\pi, c)$ of a partial policy and a state to be those states that are not in the envelope, accessible fringe or extended fringe, but that

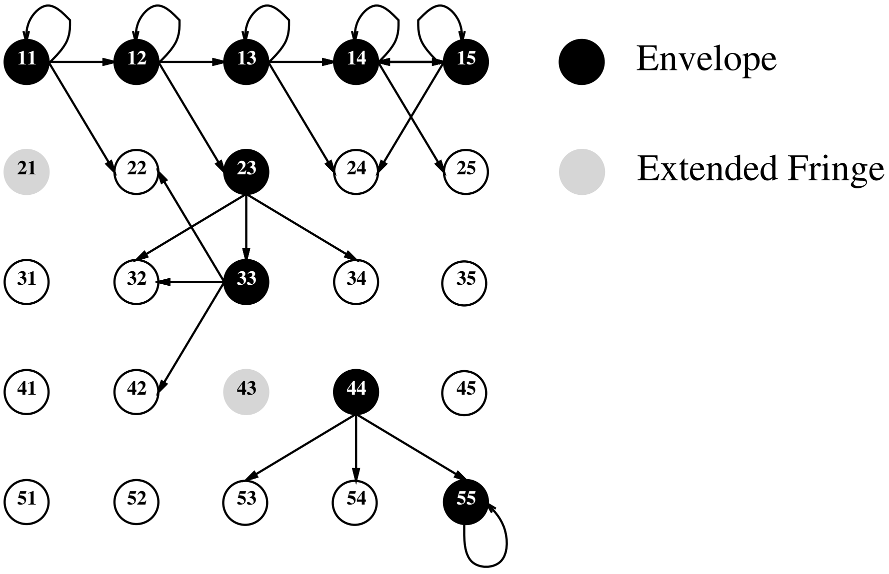


Figure 3.5: Envelope and extended fringe. $\mathcal{EF} = \{\text{L-2-1}, \text{L-4-3}\}$.

are successors of some state (reachable or not) of the envelope, using some action, not necessarily that specified by π:

$$\mathcal{FF}(\pi) = \{s \in \mathcal{S} - (\mathcal{E} \cup \mathcal{AF}(\pi, c) \cup \mathcal{EF}(\pi, c)) \mid \exists\ s' \in \mathcal{E}, a \in \mathcal{A}, \tau(s', a, s) \neq 0\}\ .$$

Note that since there may be inaccessible states in the partial policy's envelope, the actions specified by the policy may lead from some states to the full fringe rather than the extended fringe; this is the case in the example shown, where state L-4-4 is not reachable from the start state, so L-5-4 and L-5-3 are not part of the accessible fringe, but are part of the full fringe.

The full fringe is an interesting area to explore (from the agent's perspective) for several reasons. Firstly, it might encompass states that are on a path to the goal even when the current policy is not very good, leaving the states unreachable. Secondly, it might encompass states that have very low value (the policy will try to avoid them); these states are interesting because they are dangerous, and it is worthwhile looking for ways to avoid getting too close to them.

Figure 3.6 shows the full fringe of the policy described above with start state L-1-1. Note that the full fringe excludes all states in the accessible fringe and all states in the extended fringe. For comparison, Figure 3.7 shows the envelope and the three fringes.[1]

---

[1] In this particular example, the union of the three fringes and the envelope comprise almost all of the state space; this is due to the choice of a small state space for clarity of the example; in general the union of the envelope and fringes is much smaller than the whole state space.

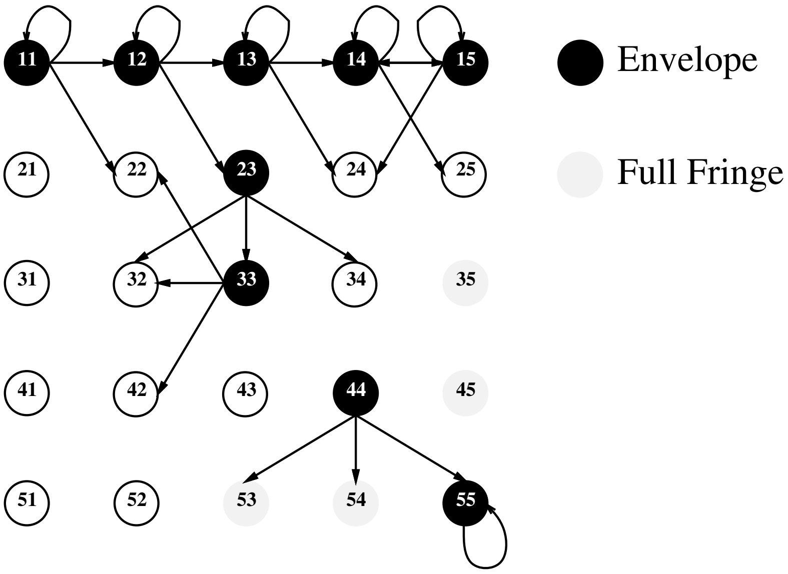


Figure 3.6: Envelope and full fringe. $\mathcal{FF} = \{\text{L-3-5}, \text{L-4-5}, \text{L-5-3}, \text{L-5-4}\}$.

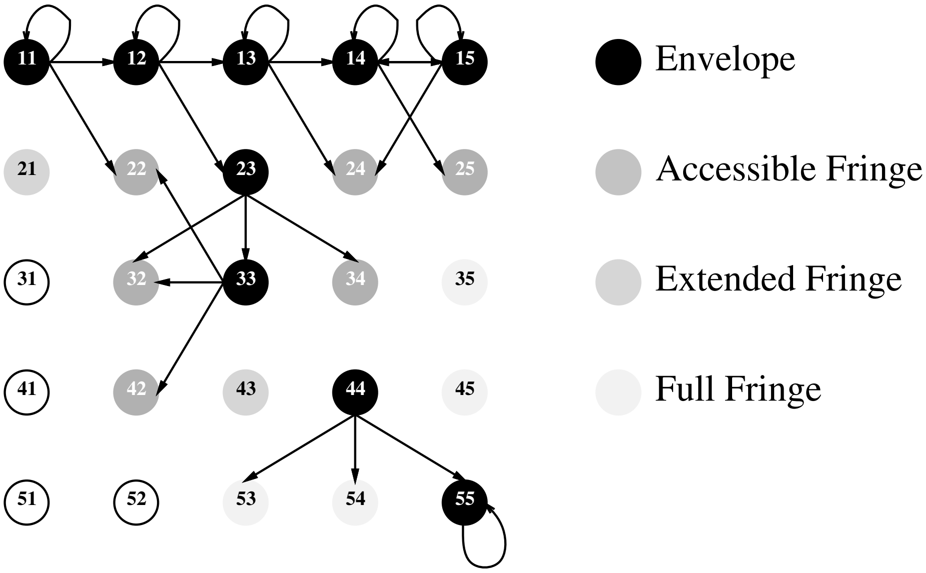


Figure 3.7: Envelope and all fringes.

Intuitively, the accessible fringe contains states that could be reached using the current policy, the extended fringe contains states that could be reached if we executed using the current policy for a time and then changed the policy, and the full fringe contains states that could be reached under a different policy completely (with the same envelope).

3.3.4 Phase

A phase, denoted ϕ, is a procedure that takes a partial policy and a current state and produces a new partial policy. A phase is understood to operate in the context of a domain and a task. We use four basic types of phase:

strengthen Add states to the envelope.

prune Remove states from the envelope.

optimize Improve the expected value of the states in the envelope.

explore Use some type of search to find a path from a state to a goal or to the envelope.

Each of these types of phase is parameterized; for example, the strengthen phase has parameters that indicate the amount by which the domain of the partial policy should be extended. The details of the parameters are described below.

3.3.5 Phase-cycle planner

The planner framework is quite general, allowing for different phases to be applied depending on the current situation (state of the environment and state of the planner). We have used this flexibility to create planners that do *deliberation scheduling* — reasoning about the duration of the planning cycle itself [DKKN93b]. This makes it possible to trade off the advantages of producing a better policy against the disadvantage of executing the current, poorer, policy.

In practice, we found that a class of simple planners performed as well as those that used sophisticated deliberation scheduling.[2] These simple planners are associated with a sequence of phases; they apply each phase in turn, and at the end of the sequence, send the new policy to the executive and go back to the beginning of the sequence. I call this type of planner a phase-cycle planner.

Algorithms for planning often consist of two parts: an initialization part, during which the data structures used by the algorithm are set up, and a recurrent

---

[2] The comparisons were made only for the RN2 robot navigation domain we studied (Appendix B.3.2).

part, that is executed repeatedly. A phase-cycle planner is therefore in fact specified by two sequences of phases, an initial and a recurrent. The top-level algorithm for the planner is straightforward:

```
phase-cycle-planner (initial sequence, recurrent sequence)
    $\{i_1, i_2, \ldots, i_n\} \longleftarrow$ initial seq
    $\{r_1, r_2, \ldots, r_n\} \longleftarrow$ recurrent seq
    $\pi \longleftarrow$ empty partial policy
    for $k$ from 1 to $n$
        cur $\longleftarrow$ current state
        $\pi \longleftarrow i_k(\pi, \texttt{cur})$
    forever
        for $k$ = 1 to $m$
            cur $\longleftarrow$ current state
            $\pi \longleftarrow r_k(\pi, \texttt{cur})$
```

Note that the intermediate policies produced by a subsequence of the phases may not be acceptable for use by the agent, so the agent's current policy remains unchanged throughout the execution of the phase sequence. This is largely a technical issue — it is sometimes more efficient to specify a sequence of phases that together will produce (in expectation) a better policy, although a subsequence may be inferior. For example, the strengthen phase may not use very sophisticated techniques for choosing the best actions for new states, if an optimization phase (*e.g.,* value iteration or policy iteration, defined below) immediately follows it.

Note also that each phase is informed of the state of the environment at the time of its invocation. In the original Plexus implementation, the current state was recorded at the beginning of each cycle, and given to each phase in the cycle, even if the current state had changed in the meantime.

Figure 3.8 shows a phase-cycle planner. The initial phase sequence consists of two phases; the recurrent sequence consists of three phases. Initially, the planner has an empty partial policy. It applies each of the initial phases successively; during this time, the executive is using only the default policy. The state given to each phase is the current state of the system at the start of the phase. After the last initial phase has modified the partial policy, it is sent to the executive. The same partial policy is passed to the first recurrent phase, then to each of the other recurrent phases in turn. During this time the executive is using the first partial policy, generated by the initial sequence. When the last recurrent phase has modified the partial policy, it is sent to the executive, and also returned to the first recurrent phase for further improvement. This cycle continues until the system reaches a termination state.

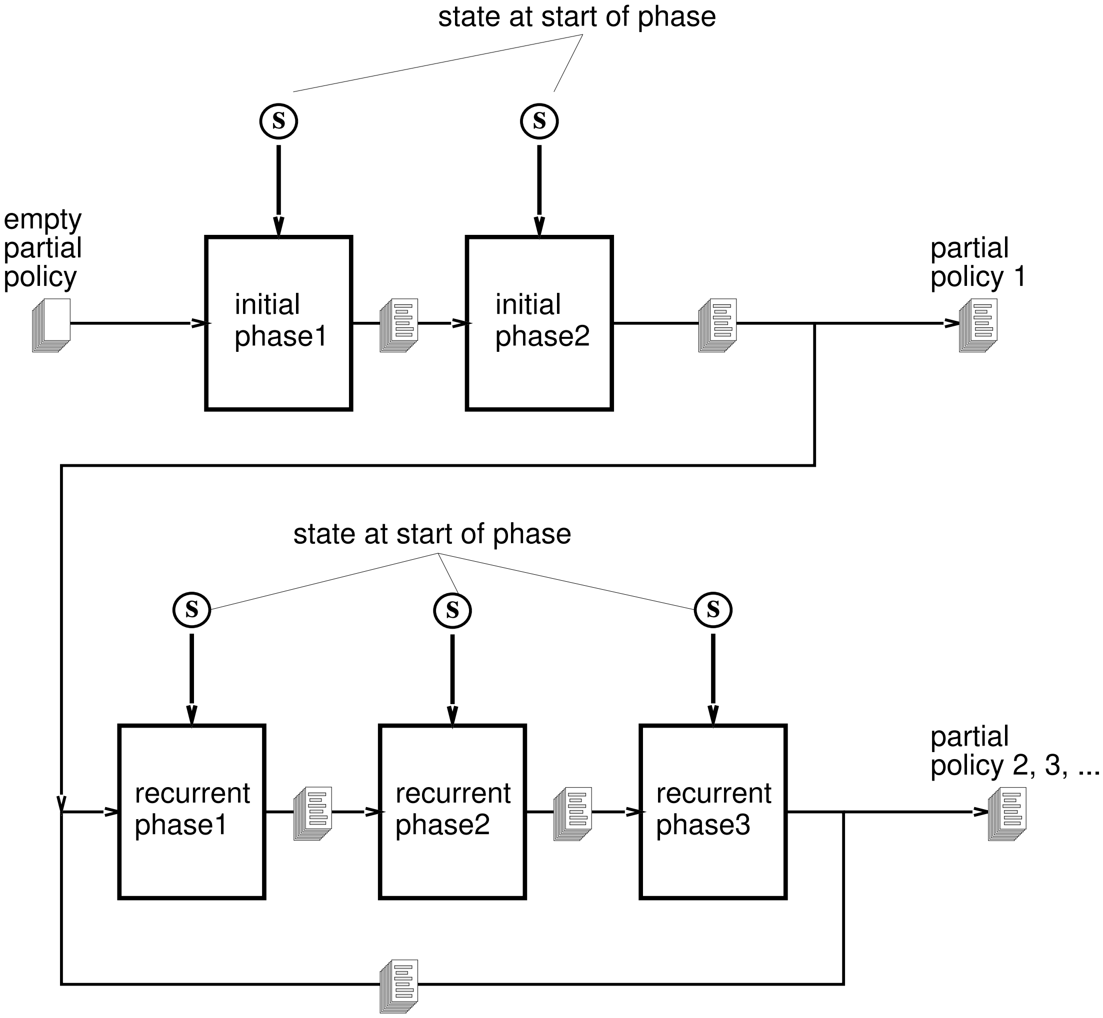


Figure 3.8: Phase-cycle planner

I have chosen to restrict my attention to phase-cycle planners. The main reason for this is that the deliberation scheduling techniques described in Dean *et al* [DKKN93b] require large amounts of off-line computation for each domain, making a survey of a large number of different domains infeasible. The performance of phase-cycle planners on the RN2 domain leads me to believe that an examination of the phase-cycle planners is sufficient to give significant insights into the applicability of more general Plexus planners to different domains and tasks.

3.3.6 Fallout probability and number of passages

While an agent is using a particular partial policy, augmented with a complete default policy, the agent-environment system can be viewed as a Markov chain. There are well-known efficient techniques for obtaining information about Markov chains, such as the probability that a particular state will be entered, or the expected number of times a state will be visited in the future.

The Markov chain *induced by* a policy models the behaviour of the agent-environment system until the system enters a state that is not in the envelope of the policy. We refer to this event informally as the agent "falling out" of the envelope. The set of states of the chain are the states in the envelope of the policy, plus the states in the accessible fringe. The transition probabilities are taken from the world model, using the policy. The states in the accessible fringe are made absorbing: there is a self-transition with probability 1.

More formally, the Markov chain induced by a policy π is the chain (S, P), where $S = \mathcal{E} \cup \mathcal{AF}$, and the transition function P is defined by

$$P(i,j) = \begin{cases} \tau_M(i, \pi(i), j) & \text{if } i \in \mathcal{E} \\ 1 & \text{if } i \in \mathcal{AF} \text{ and } j = i \\ 0 & \text{if } i \in \mathcal{AF} \text{ and } j \neq i \end{cases}$$

Recall that τ_M is the transition function of the Markov process associated with this domain and task (see Section 2.1). Figure 3.9 shows the induced Markov chain for the RN1 example presented in Section B.3.1, with start state L-1-1. Arcs are labeled with transition probabilities; in this example, p is 0.8.

In general, a planner using a phase sequence will want to know where the agent will be at the end of the sequence. The first reason for this is that states the agent might end up in that are not in the envelope are good candidates for exploration; the second reason is that the planner may want to estimate the value of the state it will end in. If we knew exactly how many time steps each phase sequence would take, we could determine this distribution precisely. However, since the algorithms used by the phases do not provide performance guarantees, we cannot in general know what this number of time steps will be. The approach I take is to adaptively estimate this number based on previous phase sequence executions. I denote the expected number of steps until completion of this phase sequence Es (Expected number of steps).

Given the Markov chain induced by a policy, we define the probability distribution η_0, with $\eta_0(c) = 1, \eta_0(s) = 0$ if $s \neq c$, where c is the current state. The quantity η_0 is the expected distribution over states after 0 steps. We recursively define η_i; $\eta_1 = P\eta_0$, the expected distribution after 1 step, and in general, $\eta_i = P^i\eta_0$ is the expected distribution after i steps, following the current policy. The function is defined in the context of a policy and an initial current state; for clarity these are not explicitly specified.

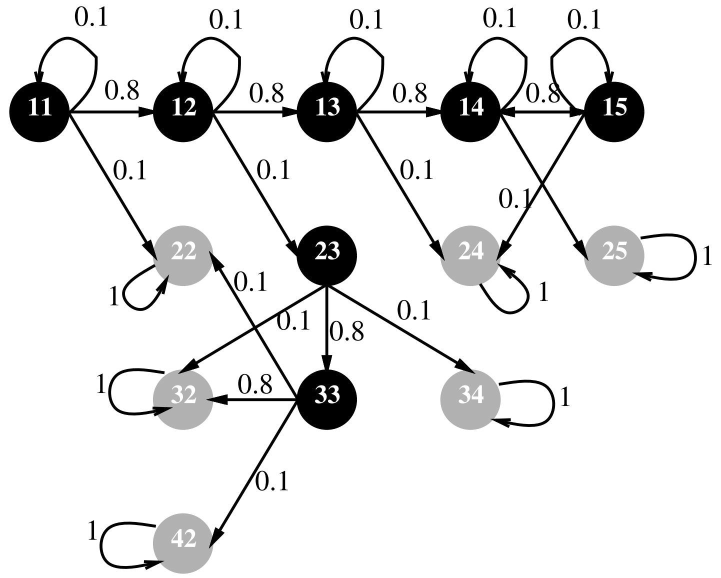


Figure 3.9: Markov chain induced by a policy.

For a state s in the envelope, $\eta_n(s)$ is the probability that the environment will be in state s after n steps, without having fallen out of the envelope. I will refer to this as the occupation probability for state s after n steps.

For a state f in the accessible fringe, $\eta_n(f)$ is the probability that the environment will fall out of the envelope into that state at some time during the next n steps.

Figure 3.10 shows the values of η_4 for the RN1 example with start state L-1-1.

3.4 Strengthen

3.4.1 Description

The purpose of the strengthen procedure is to add states to the envelope in order to reduce the probability of falling out of the envelope, and to improve the policy by considering more possible futures. The states added must be chosen judiciously, since adding too many states will slow down other phases (especially optimization). The states to add come from the three different fringes $\mathcal{AF}, \mathcal{EF} and \mathcal{FF}$.

Adding states from the accessible fringe is strengthening in the true sense, making the partial policy less likely to fail (fall out of the envelope). Adding states from the extended fringe corresponds to local short-term exploration; seeing what

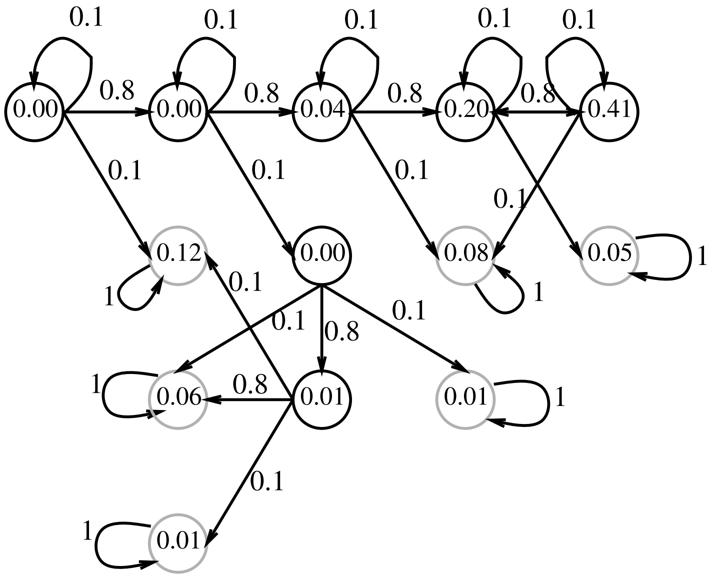


Figure 3.10: Occupation probabilities (rounded) for 4 steps from L-1-1.

could happen if we were to do something different soon. Adding states from the full fringe corresponds to local long-term exploration; seeing what could happen if we did something different later on.

The three are grouped together under the umbrella of strengthening for technical reasons: the computations required by the algorithms are most efficiently combined. Adding states from the accessible fringe is similar to the "robustification" described by Drummond and Bresina [DB90] (In earlier papers describing Plexus we called this phase "robustify"; the name was changed for aesthetic reasons.)

3.4.2 Algorithm

We wish to estimate the probability that the system will be in the various possible states when the policy currently being used is replaced by a new policy (the one being constructed by this phase sequence).

The basic idea is to construct a Markov chain whose states are the states in the envelope and the accessible fringe. The transitions for the states in the envelope are those specified by the policy; fringe states are made absorbing. Next, we estimate the probability of being in the various possible states at the time the next policy starts to be used (specified by the function η). We add the states of the accessible fringe for which the η value is higher than the specified threshold. For the extended fringe, we consider the η values of the states in the envelope. These are the probabilities of still being in the envelope at the end of the sequence. We then consider the possible outcomes resulting from the use of a different action (than that specified by the policy) at each of these states. The outcomes are weighted by the probability of reaching them after the policy is changed; we assume uniform priors for all actions other than the one currently being used (outcomes using the current action are in the accessible fringe, hence not in the extended fringe). Finally, for the full fringe we ignore the η values, giving equal weight to all the elements of the fringe. We assume a uniform probability over the actions other than the one specified by the policy, and look for outcomes that have higher probability than a specified threshold.

There are considerable differences between the version of this algorithm used in the original version of Plexus (described in [DKKN94], [DKKN93a], and the algorithm described here. A detailed list of differences is given in Section 3.10.

The strengthen phase is specified by three parameters between 0 and 1, specifying threshold values for the accessible fringe, `paf`, for the extended fringe, `pef`, and the for the full fringe, `pff`; the meanings of these thresholds will be explained shortly.

As a reminder of the notation, the current state (the state the environment occupied at the beginning of this phase sequence) is denoted c. As before, we will denote the current partial policy π, and the envelope of the policy $\mathcal{E}$. The three fringes are $\mathcal{AF}$ (accessible), $\mathcal{EF}$ (extended) and $\mathcal{FF}$ (full).

We construct the accessible fringe by iterating over all the states of the envelope; for each state s, we look at the outcomes of the action specified by the policy, $\pi(s)$. We add to the accessible fringe each possible outcome that is not in the envelope, that is, each state s' such that $\tau(s, \pi(s), s') \neq 0$ and $s' \notin \mathcal{E}$.

Next, we construct a matrix to represent the Markov chain (S, P) corresponding to the envelope and accessible fringe. The states are defined by

$$S = \mathcal{E} \cup \mathcal{AF} \ ,$$

and the transitions are defined by

$$P(s, s') = \begin{cases} \tau(s, \pi(s), s') & \text{if } s \in \mathcal{E} \\ \delta_{ss'} & \text{if } s \in \mathcal{AF} \end{cases} \ ,$$

where $\delta_{ss'}$ is 1 if $s = s'$ and 0 otherwise.

From this matrix, we wish to determine the distribution over states after Es steps using the current policy, this being the estimate of the duration of a phase sequence[3]. We iteratively construct a vector representing the probability distribution over states after some number of steps. The probability distribution after 0 steps is concentrated entirely at the current state since the system is

---

[3] This ignores the number of steps already taken during this phase sequence, but in the phase cycles I use this is not a bad approximation.

completely observable. To obtain the distribution after 1 step, we multiply the transpose of P by the distribution after 0 steps; to get successive distributions we repeatedly pre-multiply by the transpose of P. For large values of Es, it may be more efficient to raise P to the power Es and then multiply the transpose of this matrix by the vector with probability 1 for the current state, to get the expected distribution after Es states. The relative efficiency of the two approaches is heavily dependent both on the sparseness of the matrix and the manner in which the probability distributions evolve with the number of steps taken. The current implementation of the system uses repeated pre-multiplications, but halts the multiplication when the change in distribution after a step is below a specified threshold.

For each state f of the accessible fringe, we compute the occupation probability $\eta_{Es}(f)$: the probability that the system will be in this state after Es steps using the current policy. We sort the fringe by decreasing occupation probability, and move elements from the fringe to the envelope until the probability mass for the remainder of the fringe is less than the threshold `paf`. Thus the threshold specifies an acceptable probability of (temporary) failure.

Next, we consider the extended fringe. Using the current policy, the agent will never enter the extended fringe, so we consider the case where the current policy is used for Es steps, and then the policy is changed for the state occupied at that time. For a state e in the envelope, $\eta_{Es}(e)$ is the probability that the system will be in state e after Es steps. For each state in the envelope with non-zero η_{Es} value, we consider the successors under actions other than that specified by the current policy. The probability of reaching a state in the extended fringe is

$$pe(s) = (|\mathcal{A}| - 1)^{-1} \sum_{e \in \mathcal{E}} \eta_{Es}(e) P(e, f) \ .$$

Again, we sort the extended fringe by decreasing probability, and move elements from the fringe to the envelope until the remainder of the extended fringe has probability mass less than `pef`.

Finally, we consider the full fringe. We define, for all $s' \in \mathcal{FF}$,

$$pf(s') = \frac{1}{|\mathcal{A}|} \sum_{s \in \mathcal{E}, a \in \mathcal{A}} \tau(s, a, s') \ .$$

$pf(s)$ is the conditional probability of reaching state s after one randomly chosen action, given a uniform distribution over states of the envelope. Those elements of the full fringe with probability higher than `pff` are added to the envelope. We do not use a probability mass, as in the extended fringe case, since the values $pf(s)$ do not correspond to probabilities for a single event, as do the values $pe(s)$. They correspond to probabilities for $|\mathcal{E}|$ different events – the execution of a randomly chosen action in each of the states of the envelope. An alternative approach would be to consider the event of a transition from a randomly chosen state using a randomly chosen action, and to use the parameter `pff` as a probability mass. The effect would be quite different; instead of adding more states to a large envelope, it would add fewer. Although I have not conducted thorough tests comparing the performance of the two approaches, the cases I have examined seem to indicate that the additions from the full fringe should be made locally, independently of the size of the envelope. I have therefore chosen to implement only the approach described first.

As each state is added to the envelope, the action and value associated with it are calculated as follows. Temporarily, we assign a value to the state under the pessimistic assumption that it is absorbing for the default action; this gives a value of

$$\sum_{i=0}^{\infty} \gamma^i \rho(s, \pi_d(s)) = \rho(s, \pi_d(s))/(1-\gamma) \ .$$

We then compute the best action with respect to the current value function, and use this action and value for the state. This has the effect of giving new states pessimistic values, but until the policy has been optimized with this state in the envelope, this is the most reasonable thing to do.

3.4.3 Example

Continuing the RN1 example above, assume the following values for the parameters: `paf` and `pef` are 0.02, `pff` is 0.2, and the current estimate of the number Es of steps before a new policy can be computed is 4. That is, we will add enough states to remain in the envelope in all but 2% of the possible evolutions of the system over the next 5 steps, assuming that after 4 steps we change policies. We also add all states that could be reached in one step from somewhere in the envelope, choosing an action at random, with probability at least 0.1.

There are 6 states in the accessible fringe: L-2-2, L-2-4, L-2-5, L-3-2, L-3-4 and L-4-2. We compute the η_4 values for the envelope and accessible fringe states. These are shown in Figure 3.10. We add these states in order of decreasing η_4 values until the probability mass for the envelope is above $1 -$ `paf`: L-2-2, L-2-4, L-3-2, and L-2-5 are added. Next, we consider the states in the envelope with non-zero η_4 values, and consider the effect of taking an action chosen at random from those other than the one specified by the policy. In this case, the extended fringe consists of the two states L-2-1 and L-4-3, but the sum of their η values is beneath the threshold, so we add neither of them. Finally, we add any state that we can reach with probability at least 0.2 from anywhere else in the envelope under an action other than that specified by the policy. L-4-5 is added from the full fringe. The result of strengthening is shown in Figure 3.11. The actions are not shown here, as they are dependent on the previous values of the states.

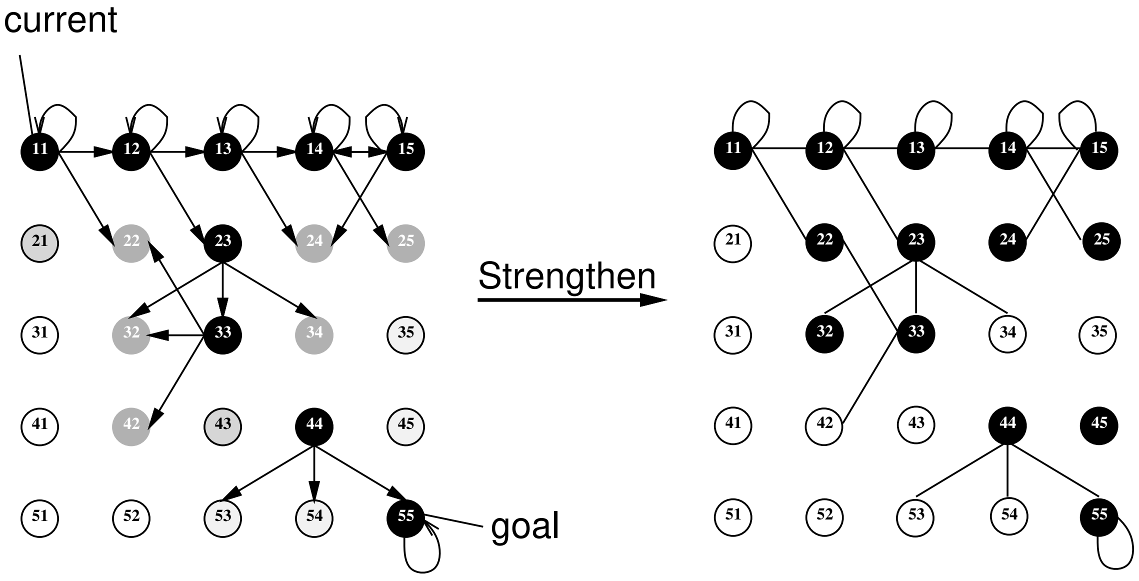


Figure 3.11: Envelope after strengthen phase.

3.5 Policy Iteration

3.5.1 Description

The policy iteration phase uses a variation of Howard's policy iteration algorithm [How66] to improve a policy. First, I will present the algorithm in the case where the envelope is the entire state space; the optimization phase then creates an optimal complete policy. Then I present the approximations I use to deal with an envelope that does not include all the states of the domain. The definition of state value presented earlier is not well-defined in the partial policy case, so I will present a definition of the value of a state in a partial policy.

3.5.2 Policy iteration with $\mathcal{E} = \mathcal{S}$

The input to the algorithm is a policy π defined over the entire state space. The algorithm iteratively modifies π until it is optimal, at which point the algorithm terminates.

The value function V_π, defined in Section 2.1, gives the expected discounted cumulative reward starting in each state, under policy π. Since I refer always to the value function with respect to the policy π in this section, I shall write V_π simply as V.

The algorithm performs two operations on each iteration. The first is value computation: it computes the value of V for all states in the world. The second is policy improvement: for each state, it chooses the action that, taken once and then reverting to the policy π, would maximize cumulative discounted reward. If some action other than the one specified by the policy would improve expectation of reward, then changing the policy to use that action will also improve the expectation of reward, only more so. This fact is not self-evident; a proof is given in Appendix A, as one of the steps in the proof that policy iteration converges to an optimal policy in a finite number of steps.

The algorithm is very similar to the standard policy iteration algorithm due to Howard [How66]; it differs in that it may make several changes to the policy before recomputing the values of the states. Howard showed that with finite state and action sets, policy iteration is guaranteed to converge to an optimal policy.

```
 done ⟵ false
While not done
    Compute values of states $V(s)$
    For each state $s$
        $a' \longleftarrow \arg\max_{a\in\mathcal{A}} Q_\pi(s,a)$
        if $a' \neq \pi(s)$
            $\pi(s) \longleftarrow a'$
    If no changes were made to $\pi$
        done ⟵ true
```

The values are computed by solving the set of linear equations on the $|\mathcal{S}|$ variables $\{V(s), s \in \mathcal{S}\}$:

$$V(s) = \begin{cases} 0 & \text{if } s \in \mathcal{X} \\ \rho(s,\pi(s)) + \gamma\left(\sum_{s'\in\mathcal{S}} \tau(s,\pi(s),s')V(s')\right) & \text{otherwise} \end{cases} .$$

The set of linear equations expresses the fact that the value of being in a state is the immediate reward for taking the corresponding action, plus another term that is discounted by a factor of γ. This second term is the weighted sum of the values of the states this action could lead the agent to. Thus, $V(s)$ is the expected discounted value of starting in state s, and following the policy π forever.

The Q_π function is defined by

$$Q_\pi(s,a) = \begin{cases} 0 & \text{if } s \in \mathcal{X} \\ R(s,a) + \gamma\left(\sum_{s'\in\mathcal{S}} \tau(s,a,s')V_\pi(s')\right) & \text{otherwise} \end{cases} .$$

$Q_\pi(s,a)$ is the expected discounted value of taking action a in state s, and thereafter following the policy π. This function is called Q because it is the basic quantity used by Watkins in his Q-learning algorithm [Wat89].

3.5.3 Policy Iteration with $\mathcal{E} \neq \mathcal{S}$

In general, we are interested in computing an optimal policy over a subset of the state space. Recall that an agent has a default policy π_d, and a current partial policy π with envelope $\mathcal{E}$. To compute the values exactly, we would construct the policy π':

$$\pi'(s) = \begin{cases} \pi(s) & \text{if } s \in \mathcal{E} \\ \pi_d(s) & \text{otherwise} \end{cases} .$$

We would then use a variant of policy iteration that makes changes only to actions of states in $\mathcal{E}$ to find the best partial policy over this envelope.

In practice, however, this is infeasible because it requires solving a set of $|\mathcal{S}|$ linear equations for each policy improvement. Instead, I modify the definition of state value to capture the notion of falling out of the envelope, and optimize using this new definition of value. The basic approach is to divide the possible outcomes of the next action into two categories: those that lead to a state in the envelope, and those that lead to a state outside the envelope. Expectations of value are computed for both of these, and a weighted sum gives the value for the state. The computations are somewhat complex, though the ideas are straightforward. Since the value of a state is defined as a function of the value of other states in the envelope, the definitions below will produce a set of $|\mathcal{E}|$ linear equations whose solutions are the values of the states in the envelope.
Given an envelope $\mathcal{E} \subset \mathcal{S}$, we define the function $P_{in} : \mathcal{E} \to [0, 1]$ by

$$\forall s \in \mathcal{E}, P_{in}(s) = \sum_{s' \in \mathcal{E}} \tau(s, \pi(s), s').$$

This is the <u>p</u>robability that the agent will remain <u>in</u> the envelope after one step of policy π taken from state s.

Next we define a measure of the value of making a transition into another state of the envelope. We define the envelope value function $V_e : \mathcal{E} \longrightarrow \mathbb{R}$ by

$$V_e(s) = \begin{cases} \frac{\rho(s,\pi(s))}{1-\gamma} & \text{if } P_{in}(s) = 0 \\ \frac{1}{P_{in}(s)} \sum_{s' \in \mathcal{E}} \tau(s, \pi(s), s') V(s') & \text{otherwise} \end{cases}$$

If s has successors in the envelope, we compute the expected value of the outcome in those cases where the outcome is in the envelope. $V_e(s)$ is the expected value of the outcome under the assumption that the expected value of the successors that are not in the envelope is the same as the expected value of the successors that are. The factor $\frac{1}{P_{in}(s)}$ performs this normalization.

This notion of value is extended to the case where s has no successors in the envelope, by defining V_e for such states to be the value of an absorbing state whose instantaneous reward is that of the policy's action in the state. That is,

$V_e(s)$ is the expected cumulative reward if the system stays in the state s forever. This is usually a pessimistic approximation, but this has the desirable effect that the planner will usually add states to the envelope rather than moving to states it is not sure can be escaped.

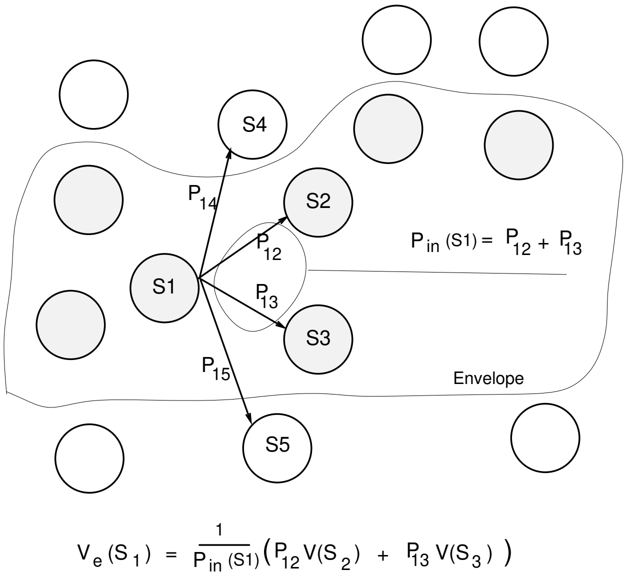


Figure 3.12: Value of a state

For illustration, consider the example shown in Figure 3.12, where states are represented as circles, and transitions under the current policy by arcs, labeled with their probabilities. P_{ij} is the probability that the system will move from state i to state j after taking the action specified by the policy in i. States in the envelope are grey, those outside the envelope are white.

The policy specifies an action to take in state S_1; there are four possible outcomes from this action. P_{in} is the probability that the outcome will be in the envelope; in this case, $P_{12}+P_{13}$. $V_e(S_1)$ is the expected value of the new state after one transition if the new state is still in the envelope, *i.e.,* $1/(P_{in}(1))P_{12}V(S_2)+P_{13}V(S_3)$.

When the environment occupies a state s, there is probability $P_{in}(s)$ that after one step the environment will occupy a state in the envelope. The value associated with this probability mass is $V_e(s)$. With probability $1-P_{in}(s)$, however, the environment will occupy a state outside the envelope. At this point, the agent will execute the default policy π_d until the environment re-enters the envelope. This can happen either because the system makes a transition to a state in the envelope, or because the planner produces a new policy whose envelope contains the current state.

To estimate the cost of falling out of the envelope from a state s, I make several simplifying assumptions.

- The number of time steps during which the current state is not part of the envelope can be estimated globally, *i.e.,* the expected number of time steps does not depend greatly on the particular state from which the agent fell out of the envelope. I use an adaptive estimate, based on previous cases in which the agent fell out of the envelope.
- The instantaneous rewards accumulated during that time can be approximated by assuming they are all equal to the instantaneous reward for the default action in state s.
- The state the environment will occupy once it has re-entered the envelope will have a value close to that expected after a transition from s that did not fall out of the envelope. Note in particular that these restrictions do not require any reference to states not in the envelope; this is desirable for implementations of these algorithms.

Figure 3.13 shows the algorithm pictorially, with a sequence of pseudo-states "Out" representing the fringe states accessible from S_1. Eo is the adaptive estimate of the number of steps the agent spends outside the envelope when it falls out. More formally, I define a function "value of falling out" to represent

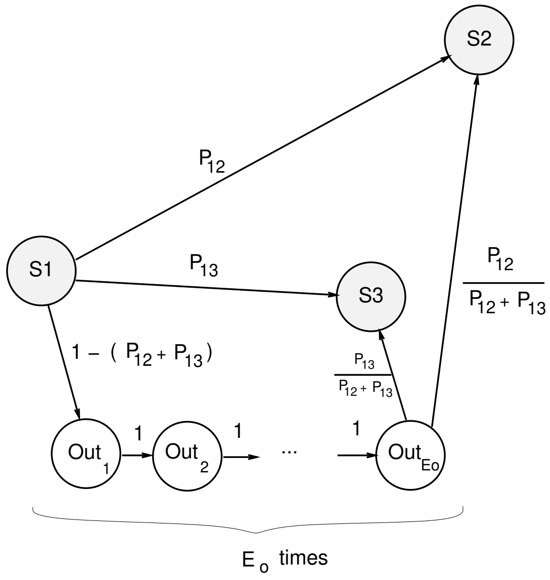


Figure 3.13: Value of a state adjoining the fringe

this estimate of value accumulated while not in the envelope, $V_o : \mathcal{E} \to \mathbb{R}$:

$$V_o(s) = \rho(s, \pi_d(s)) \left(\sum_{i=1}^{Eo} \gamma^i \right) = \rho(s, \pi_d(s)) \left(\gamma \frac{1 - \gamma^{Eo+1}}{1 - \gamma} \right) ,$$

where π_d is the agent's default policy and Eo (envelope out) is the expected number of time-steps during which the agent is out of the envelope. Like Es, this is adaptively estimated. $V_o(s)$ is the sum of the instantaneous rewards for the default policy over the next Eo steps, assuming that the instantaneous rewards for all the states visited during that time are the same.

Given these estimates of the value for the two possible cases, we can define the value of a state s that has successors in the fringe by

$$V(s) = \rho(s, \pi(s)) + \gamma \left(P_{in}(s) V_e(s) + (1 - P_{in}(s))(V_o(s) + \gamma^{Eo} V_e(s) \right) .$$

This estimate says that in all cases, the cost $\rho(s, \pi(s))$ is incurred. With probability $P_{in}(s)$, the agent remains in the envelope, and the new value is just $V_e(s)$. With probability $1 - P_{in}(s)$, a cost of $V_o(s)$ is incurred while the agent is out of the envelope, and upon re-entry, the value is estimated as $V_e(s)$, the expected value of the state following s if that state is in the envelope. These values are discounted by their distance in time from the present.

The policy iteration algorithm is the same as that presented in Section 3.5.2, but the V and Q functions are computed differently. The value function V is defined by solving the set of $|\mathcal{E}|$ linear equations

$$V(s) = \begin{cases} \rho(s, \pi(s)) + \gamma \left(P_{in}(s) V_e(s) + (1 - P_{in}(s))(V_o(s) + \gamma^{Eo} V_e(s) \right) & \text{if } s \in \mathcal{E} - \mathcal{X} \\ 0 & \text{if } s \in \mathcal{X} \end{cases}$$

$$V_e(s) = \begin{cases} \frac{\rho(s, \pi(s)}{1-\gamma} & \text{if } P_{in}(s) = 0 \\ \frac{1}{P_{in}(s)} \sum_{s' \in \mathcal{E}} \tau(s, \pi(s), s') V(s') & \text{otherwise} \end{cases}$$

The Q-values are defined analogously, using the specified action instead of the policy action throughout.

3.5.4 Comments

Policy iteration is guaranteed to converge to an optimal policy, but it may do so very slowly. In particular, it tends to converge to a solution that is close to optimal in a few iterations[4], and then spend many iterations finding an optimal

---

[4] By close to optimal I mean that the expected reward using the policy is close to the expected reward using the optimal policy.

policy. For embedded systems, approximate solutions obtained quickly are often preferable to exact solutions obtained slowly, so I allow premature termination of the optimization algorithm in the following way. Each time the action for a state is changed, I record the improvement in value for that state. When the sum of all the improvements made during one iteration is less than a certain threshold, the procedure terminates.

3.5.5 Example

After strengthening the envelope as described above, the envelope has 14 states. Value computation yields the values shown in Figure 3.14. The discount factor γ here is 0.99. Three more rounds of value computation and policy improvement are needed before the procedure converges on the optimal policy for this envelope, shown in Figure 3.15. The convergence rate and the quality of intermediate solutions are dependent on the order in which the states are visited. For comparison, the values for the optimal policy over the entire state space are given in Figure 3.16. Note that in states from which execution of the policy cannot lead to the goal, the value is $-1/(1-\gamma)$.

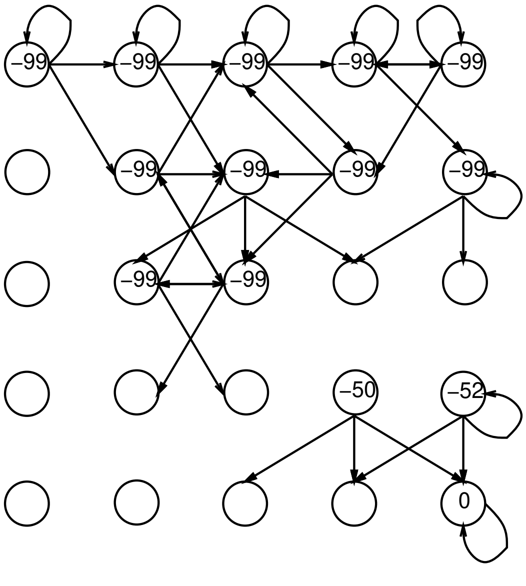


Figure 3.14: Result of value computation

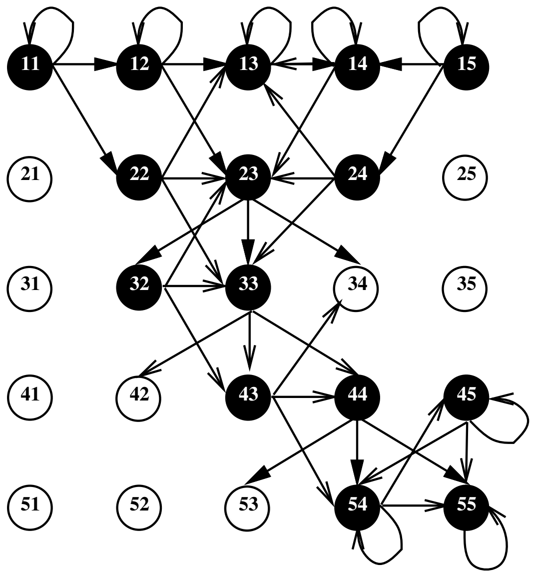


Figure 3.15: Result of policy iteration

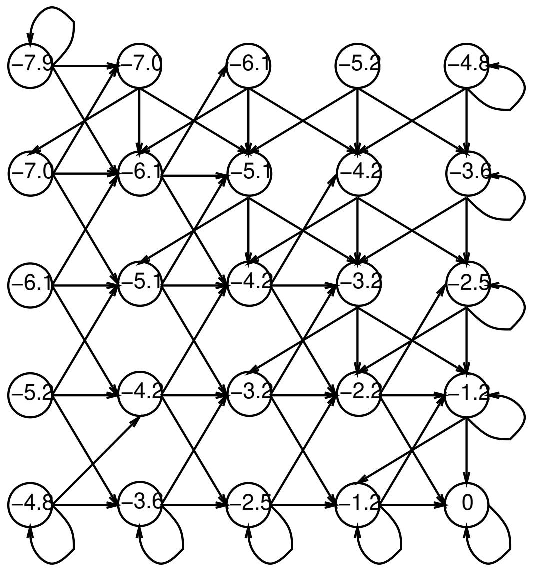


Figure 3.16: Result of policy iteration on entire state space

3.6 Value Iteration

Value iteration [Bel57] is one of the three standard techniques for computing an optimal policy for a Markov decision problem, linear programming and policy iteration (described in Section 3.5) being the others. Value iteration is particularly well-adapted to the case where a precise model of the world is unavailable, and must be learned while computing a policy; it has therefore been extensively used by the machine learning community (see Sections 2.2 and 2.3 for references).

The conditions under which value iteration performs better than policy iteration, or *vice versa*, are understood theoretically only in very limited circumstances; empirically, combinations of value iteration and policy iteration perform better than either basic technique. Puterman [Put94] has shown that a particular combination, called modified policy iteration, will converge more quickly than simple value iteration. A natural extension to the set of phases for Plexus would implement this combination. Other interesting improvements to value iteration and policy iteration algorithms have been presented by, *e.g.,* Dearden and Boutillier [BD94], Bertsekas and Castañon [BC89], Kushner and Kleinman [KK71].

3.6.1 Description

Value iteration is a technique for determining an optimal policy for a set of states. We start with arbitrary values associated with each state, and an arbitrary policy π. On each iteration, we compute $Q_\pi(s, a)$ values for each state s and action a; these are estimates of the total discounted reward that would be accumulated if we were to take action a from state s and then use policy π forever. We choose the action with highest Q value, and update the value of the state and the policy. This process converges to an optimal policy, though the number of steps it takes may be very large. One of the problems with value iteration is that there is no natural halting criterion — the values of states may continue to change even once the optimal policy has been obtained. There are two basic approaches to solving this problem. One is to perform value iteration until the absolute value of the difference between the value functions at successive iterations is less than some constant (for all components). The constant may be chosen so as to ensure the optimality of the resulting policy (see, *e.g.,* [WB93]). A second alternative is to periodically perform one round of policy iteration; if this does not change any actions, then the policy is optimal. If it does change any actions, the new values it produces will usually be better approximations than those the value iteration had reached, and they can therefore be re-used by the value iteration when it is continued.

3.6.2 Algorithm

The algorithm takes as input a partial policy π defined over an envelope $\mathcal{E}$. It makes modifications to the policy (without changing the envelope). There are no guarantees about improvement, but the new policy will in general be better than the old policy (it improves in expectation).

For each state s we define a value $V(s)$; if some value measure already exists (as a result of an earlier phase) we use that, otherwise we ascribe a value of 0 to all states.

For each state s we define the value of falling out from that state $V_o(s)$, as in Section 3.5.3. This value does not depend on the action taken in state s.

We define a function $Q(s, a)$, for every state s and every action a such that $P_{in}(s, a) \neq 0$ (some of the possible outcomes of this combination are in the envelope),

$$Q(s,a) = \rho(s,a) + \gamma \left(P_{in}(s,a) V_e(s) + (1 - P_{in}(s,a))(V_o(s) + \gamma^{Eo} V(s)) \right) \ .$$

In the case where none of the possible outcomes of action a in state s are in the envelope, we use a pessimistic estimate of value, which would be obtained if the agent stayed in this state forever:

$$\rho(s,a)/(1-\gamma) \ .$$

The value iteration algorithm is parameterized by the number of iterations, `viters`:

```
 done ⟵ false
viters times
    For each state $s$
        $V'(s) \leftarrow \max_{a' \in \mathcal{A}} Q(s, a')$
        $\pi'(s) \leftarrow \arg\max_{a' \in \mathcal{A}} Q(s, a')$
    For each state $s$
        $V(s) \leftarrow V'(s)$
        $\pi(s) \leftarrow \pi'(s)$
```

Value iteration is a very useful complement to policy iteration. Policy iteration operates globally, and is therefore quick to propagate value changes throughout the envelope. Precisely because it is global, however, it is expensive computationally. Value iteration is computationally very cheap, but the changes it makes are local, so changes propagate slowly.

3.7 Prune

3.7.1 Description

The prune operation removes states from the envelope. In general, this will make the policy worse, or at least no better, so the purpose of pruning is solely to decrease the cost of executing the other phases. Quantifying the relationship between the value of a better policy and the cost of executing phases is difficult except in very specific cases, so the approach I take is pragmatic: the algorithm described below works well in the cases I have studied.

The prune phase has one parameter `pp`: a threshold governing how many states will be removed. For each state in the envelope, we determine the probability that the state will be reached within Es steps using the current policy. If this probability is less than `pp`, and if the state has a worse value than the current state, then the state is removed. The reason for the restriction to states that have worse values than the current state is to ensure that if the goal is reachable from the current state before pruning, it will be after pruning.

3.7.2 Algorithm

The prune phase does not modify the policy function; only its domain.

A state s is a candidate for deletion if its value $V(s)$ is less than the value of the current state $V(c)$. Candidate states are ranked according to their η_{Es} values, the probability of being in that state in Es steps, as described in Section 3.3.6, and states for which this probability is below `pp` are removed. That is, we remove states that are unlikely to be visited.

3.7.3 Example

Continuing the RN1 example, suppose the prune parameter `pp`, is 10^{-6}, and consider the situation shown in Figure 3.17a. The current state at the point when the pruning is done is L-2-3. States L-1-1, L-1-2, L-2-2, L-3-2, L-1-3 and L-1-4 have occupation probabilities (η_4) smaller than 10^{-6}, so they are removed, resulting in the envelope shown in Figure 3.17b.

3.8 Explore

3.8.1 Description

Explore modifies a policy by adding sets of states to the envelope. Each set is a *path* from one state to another, that is, a sequence of states $s_1 \ldots s_n$ such that

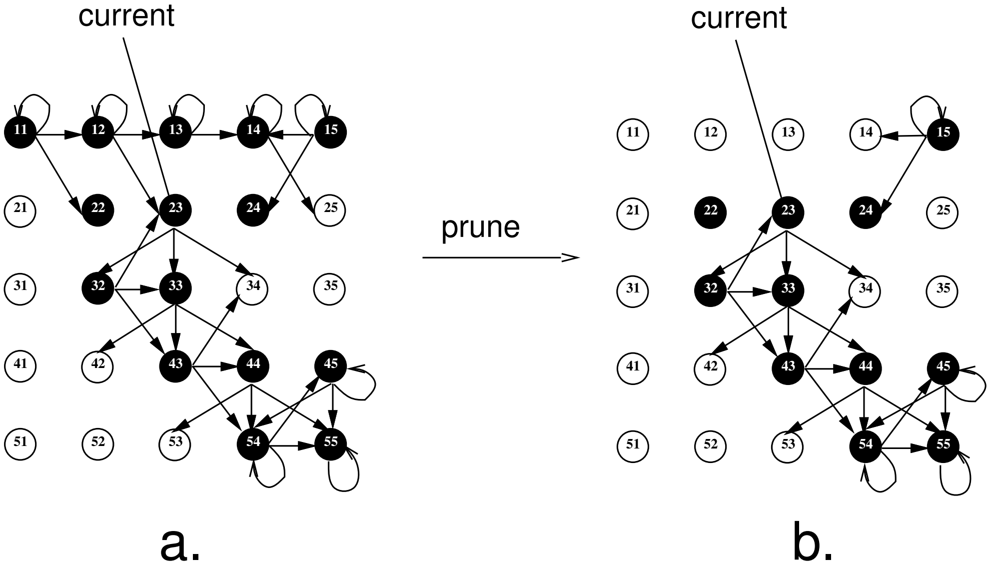


Figure 3.17: Prune phase

for all $i < n$, s_{i+1} is a successor of s_i for some action.

The explore phase is given the current partial policy, the current state, a set of target states in the envelope, and a set of states to avoid.

If the current state is not one of the target states, explore finds a path from it to one of the target states. In the current implementation, if the current state is in the target set, this phase does nothing. The path will not contain any of the states in the set of states to avoid. This allows one to search from a state to a goal state while avoiding the states already in the envelope.

In general, the explore phase is used to find a path from a starting state to one of a set of states satisfying a particular requirement. For tasks of achievement, this requirement is typically that the state be terminating. For tasks of maximization with no terminating states, the requirement is usually that the instantaneous reward in that state be above some threshold. Exploration of this type is not appropriate for tasks of maintenance; local improvements of the type made by the strengthen phase are more effective.

The algorithm given below makes no assumptions about the state space; in particular, it does not assume the existence of any heuristic estimating the distance to the goal of the search. If such a heuristic is available, adding it would greatly improve the performance of the search. The reason I have avoided defining such heuristics for the example domains I use is that it would confound the comparison of different domains. While the best-first algorithm here is not efficient, it is equally inefficient for all domains, so that differences in performance can be ascribed to fundamental differences in the domains, rather than differences in the value of the search heuristic.

3.8.2 Algorithm

Explore is a procedure that takes as input a partial policy π on envelope $\mathcal{E}$, a current state c, a set of target states $\tau \subset \mathcal{E}$, and a set of states to avoid $\alpha \subset \mathcal{S} - \tau - \{c\}$. It finds a path $\mathcal{P}$ such that $\mathcal{P} \cap \mathcal{A} = \emptyset$, the first element of $\mathcal{P}$ is c, and the last element is in τ. The new policy π' produced by the phase has envelope $\mathcal{E} \cup \mathcal{P}$, and

$$\forall s \in \mathcal{E}, \pi'(s) = \pi(s).$$

For each state s in the path, the action π' associates with s an action that has a non-zero probability of leading to the next state in the path. These may not be optimal actions, but in practice they are much better than random.

This algorithm assumes that the rewards are all negative or zero. It can be modified trivially to handle the case where rewards are all positive or zero.

The algorithm is a type of best-first-search through the state space; we rate a path by its probability (the product of the probabilities of each transition along it) divided by the negated sum of the rewards along the path. This measure intuitively corresponds to the desiderata for a path: a high-probability path is better than a low-probability one, and a small negative reward is better than a large negative reward. Each state is rated by the best rating of a path that leads to it. Search proceeds from the state with the best rating of those that still have unvisited successors.

The state space is divided into states that have been visited (or should be avoided), states that are in the frontier[5] and unseen states. Initially, the states in the set to be avoided, α, are added to the visited set. The current state is made the only state in the frontier, and the remainder of the states are added to the unseen set.

Given a path $s_1, s_2 \ldots s_n; a_1, a_2 \ldots a_n$, we define the value of the path as the probability of the path divided by its negated reward sum:

$$\nu(s_1, s_2, \ldots, s_n; a_1, a_2, \ldots, a_n) = \frac{\prod_{i=1}^{i=n-1} \tau(s_i, a_i, s_{i+1})}{-\sum_{i=1}^{i=n-1} \rho(s_i, a_i)}$$

We associate with each state s in the frontier a path to s from the starting state. We next choose the state s from the frontier with the highest path value. s is removed from the frontier, and all the states that are successors of s under any action are added to the frontier unless they are in the visited set. If s' is a successor of s that was already in the frontier, the path associated with s' is

---

[5] This is often called the fringe; I use the term frontier instead, to avoid confusion with the fringes of the envelope.

compared to the path constructed by following the best path to s and then taking one step to s'. If this latter path has a higher value, it is associated with s'.

This procedure guarantees that when a state is added to the visited set, it has associated with it the path with highest value. When a goal state is visited, the path associated with it is added to the envelope of the policy, using the actions specified in the path.

The path value function ν is monotonic; the value of a path is lower than the value of any initial subsequence. To see this, consider a path

$$\mathcal{P} = (s_1, s_2, \ldots, s_n; a_1, a_2, \ldots, a_n) \ .$$

Let P be the probability of the path:

$$P = \prod_{i=1}^{i=n-1} \tau(s_i, a_i, s_{i+1}) \ ,$$

and let R be the negated reward sum of the path:

$$R = -\sum_{i=1}^{i=n-1} \rho(s_i, a_i) \ .$$

The value of this path $\mathcal{P}$ is then P/R. Now consider a path $\mathcal{P}'$, created by adding one state to the end of $\mathcal{P}$.

$$\mathcal{P}' = (s_1, s_2, \ldots, s_{n+1}; a_1, a_2, \ldots, a_{n+1}) \ .$$

The probability of $\mathcal{P}'$ is $P \times \tau(s_n, a_n, s_{n+1})$, and its negated reward sum is $R - \rho(s_n, a_n)$. The value of $\mathcal{P}'$ is therefore

$$\nu(\mathcal{P}') = \frac{P \times \tau(s_n, a_n, s_{n+1})}{R - \rho(s_n, a_n)} \ .$$

Since transition probabilities are at most 1, and rewards are negative or 0,

$$\nu(\mathcal{P}') = \frac{P \times \tau(s_n, a_n, s_{n+1})}{R - \rho(s_n, a_n)} \leq \frac{P}{R} = \nu(\mathcal{P}) \ .$$

The monotonicity of ν guarantees that the states will be visited in decreasing order of their path value. The path values do not translate directly into state values $V(s)$ as defined above, but in general if an optimal policy is found for the set of states constituting a path, the state value of the starting state of the path is correlated with the path value, so this heuristic guides the search well. When the search is not to a goal, but instead to the envelope (for example, when the agent falls out of the envelope, it may search for a path back to the envelope), the values of the states on the path are offset by the value of the final state in the path.

3.9 Phase-cycle planner

The particular phase-cycle planner I used for all of the domains consisted of an explore phase in the initial sequence, and a recurrent sequence consisting of four phases: explore, prune, strengthen, optimization. The particular choice of phases was motivated by a set of experiments I performed with Ann Nicholson on an earlier version of Plexus. Working on a small set of representative RN2 domains, we ran several dozen planners and compared the results. We chose only planners with an optimization phase as the last phase, since modification without optimizing can lead to very poor policies. We made the first phase an exploration phase to handle the case where the agent fell out of the envelope during the previous cycle. With these constraints, we constructed planners using intuitively reasonable sequences of phases (intuitions developed after a great deal of experimentation).

The planners with long sequences of phases did not perform very well in general; this was probably due to the fact that (in that implementation) phases continued until completion even if the agent fell out of the envelope. Empirically, pruning worked better before strengthening, rather than after, leading us to the choice of explore, prune, strengthen, optimize.

The parameters were specified differently for the different classes of domain, but were the same for all domains in a class. Specifically, the parameters for the RN1 domains and the RN2 domains were: $\texttt{paf} = 1, \texttt{pef} = 1, \texttt{pff} = 4, \texttt{pp} = 0.01$. For the racetrack domains, I used: $\texttt{paf} = 1, \texttt{pef} = 1, \texttt{pff} = 0, \texttt{pp} = 0.00$. In both cases, policy iteration was stopped when the total change in value over one iteration was less than 0.01.

These values were chosen empirically, by running experiments on representative domains with different values for the parameters, and choosing the ones with the best performance. The performance is not very sensitive to the values of the parameters; similar results were obtained for nearby values. For the racetrack domains, pruning was turned off completely, since the envelope tends not to grow very quickly (most paths fall off the track and return to the starting state). The full fringe parameter was turned off also, since it tended to expand parts of the envelope that were too distant to be of great use.

3.10 Differences from original version

There are significant differences between Plexus as described here and the versions described in earlier papers referring to it [DKKN93a, DKKN93b, DKKN94].

3.10.1 Strengthen

In the original implementation of Plexus, the strengthen phase was given a single parameter indicating the number of states to add to the envelope of the policy. It first looked at the accessible fringe, ordering the states by decreasing order of the system reaching the state using the current policy in a fixed number of steps (this fixed number was chosen by hand), and adding the most likely states. If the parameter specified more states than were available in the accessible fringe, it added the entire accessible fringe, then used the extended fringe, and, if necessary, the full fringe for the remainder of the states. In the latter two cases, the notion of "most likely to be reached" is fuzzy; the heuristic we used was to assume a uniform likelihood for all actions in any state. If the parameter specified was larger than the number of states in all three of these sets, no further attempt to add states was made.

There are three basic problems with the original strengthen algorithm. The first is that while clearly what actually affects the performance of the system is the number of states *actually* added, the parameter is not directly related to these numbers; above a certain number, increasing the strengthen parameters does not affect the way the policy is changed. This threshold is dependent on the current policy, so it is difficult to evaluate the effect of the parameter on the algorithm. While for a particular domain it is possible to determine a reasonable range of values for the strengthen parameter, it is difficult to automate this choice. The new version specifies how conservative the algorithm should be in terms of the probability that states will be reached. This is much better adapted to domains of different sizes.

The second problem with the original strengthen phase is that it relied on a computation of the expected state after some number of steps. This number is the expected number of steps the executive will use this policy, *i.e.,* it corresponds to the time the planner takes to generate the next policy. In the original implementation, this number was determined manually for each domain, in a way that was difficult to automate. The new implementation avoids the need for determining the number, by estimating it adaptively.

The third problem is one of representation. The three different sources of states to be added to the envelope correspond to different high-level notions. Adding states from the fringe strengthens the policy; that is, it makes it less likely that this policy will fail. Adding states from the extended fringe corresponds to near-term local exploration: what might happen if we did something different in the near future? Adding states from the inaccessible fringe corresponds to long-term local exploration: assuming that our overall plan works well, what would happen if we tried something different later on? Since these three are conceptually quite different, it seemed more natural to have separate parameters controlling them. The three fringes thresholds are now specified separately.

3.10.2 Prune

The original implementation of prune took a single parameter: a number of states to remove from the envelope. Only states with values lower than the current state were candidates for removal. Those with lowest expected number of transitions (the function η described in Section 3.3.6) were removed. The first two problems described in the strengthen section above apply equally to the prune phase; the solution is the same: prune now takes a meaningful parameter, and uses an adaptive estimate of the number of steps to the end of the phase sequence.

3.10.3 Policy iteration

If the envelope comprises the entire state space, the original algorithm performed the same operations as the current one. When the envelope is not the whole state space, the evaluation of states is done slightly differently. Specifically, in the original implementation the quantity corresponding to $V_o(s)$ was a parameter chosen by hand; the same value was used for all states. The efficiency of the algorithm was highly dependent on this parameter. The current implementation does not suffer from this limitation, since it computes $V_o(s)$ individually for each state.

3.10.4 Explore

The original implementation of the explore phase used a depth-first-search that considered only the most likely outcome of each state-action pair. One consequence of this was that it was not always able to find paths in domains where the highest-probability outcomes were self-transitions. The original implementation chose the action at each state in the search uniformly, rather than based on the probability of the most likely outcome, as now. This resulted in paths of lower probability than those generated by the current system.

3.11 Representation issues

3.11.1 Goals

In this section I show how the domain-task problem formulation can be used to approximate several common types of goal specification.

As soon as possible

Given a domain, a starting state s_0 and a set of target states $\mathcal{T}$, the goal is to reach one of the target states in the fewest transitions.
Define the task $\{{s_0}', \rho', \gamma', \mathcal{X}'\}$ by

$$\begin{aligned} {s_0}' &= s_0 \\ \rho'(s,a) &= \begin{cases} 0 & \text{if } s \in \mathcal{T} \\ -1 & \text{otherwise} \end{cases} \\ \gamma' &= \gamma \\ \mathcal{X}' &= \mathcal{T} \end{aligned}$$

With $\gamma = 1$, the performance measure described in Section 2.1 is exactly the one desired; the performance of the agent is the expected number of steps before reaching the goal. As pointed out above, this measure can be made arbitrarily close to the desired measure if we are willing to ignore the possibility of the agent taking more than some fixed number of steps to reach the goal. While the optimal policy generated under the domain-task representation may not be identical to the optimal policy under the desired performance measure, in practice they are identical if γ is chosen close to 1 ($1 - 10^{-5}$ is sufficient for all the domains I have studied so far).

As long as possible

Given a domain, a starting state s_0, and a set of target states $\mathcal{T}$, satisfying $s_0 \in \mathcal{T}$, the goal is for the agent to remain in the set of states $\mathcal{T}$ for as many steps as possible.
Define the task $\{{s_0}', \rho', \gamma', \mathcal{X}'\}$ by

$$\begin{aligned} {s_0}' &= s_0 \\ \rho'(s,a) &= \begin{cases} 0 & \text{if } s \in \mathcal{T} \\ -1 & \text{otherwise} \end{cases} \\ \gamma' &= \gamma \\ \mathcal{X} = \mathcal{S} - \mathcal{T} & \end{aligned}$$

Sequential goals

Given a domain, a starting state i, a set of states P_1 and a set of states P_2, the goal is for the agent to reach one of the states in P_1 and then one of the states in P_2, minimizing the total number of steps taken.

This can be represented by adding a boolean state variable corresponding to the proposition "P_1 has already been achieved"; this simply doubles the number of states. The transitions are defined in the obvious way: a transition from a state in which the boolean is false, to a state in P_1, will change the boolean to true deterministically. No transition changes the boolean from true to false. All other transitions leave the boolean unchanged, and have the same probabilities as in the original problem.

$$
\begin{array}{lcl}
\mathcal{S}' & = & \mathcal{S} \times \{F, T\} \\
\mathcal{A}' & = & \mathcal{A} \\
\tau'((s,F),a,(s',F)) & = & \left\{ \begin{array}{ll} \tau(s,a,s') & \text{if } s' \notin P_1 \\ 0 & \text{if } s' \in P_1 \end{array} \right. \\
\tau'((s,F),a,(s',T)) & = & \left\{ \begin{array}{ll} \tau(s,a,s') & \text{if } s' \in P_1 \\ 0 & \text{if } s' \notin P_1 \end{array} \right. \\
\tau'((s,T),a,(s',T)) & = & \tau(s,a,s') \\
\tau'((s,T),a,(s',F)) & = & 0
\end{array}
$$

The task is defined by

$$
\begin{array}{lcl}
s_0{}' & = & (s_0, T) \\
\rho'((s,T),a) & = & \left\{ \begin{array}{ll} 0 & \text{if } s \in \mathcal{T} \\ -1 & \text{otherwise} \end{array} \right. \\
\rho'((s,F),a) & = & -1 \\
\mathcal{X}' = \mathcal{S} - \mathcal{T} & &
\end{array}
$$

Clearly, this can be extended to a sequence of tasks of any length; the size of the state space is multiplied by the number of tasks. Furthermore, the same approach can be used for sequences of achieve and maintain goals, such as “pick up the part, then paint the part while holding the part, then put the part in the bin”.

3.12 Summary

In this chapter I have described a framework for concurrent planning and execution, and an implementation of this framework, Plexus. The approach is based on the theory of stochastic processes, specifically Markov process theory. In Section 2.2 I present some requirements for this approach to be applicable; it is natural to ask in what cases the approach is effective, in the sense of producing better results than other approaches. In order to address this question, I have conducted empirical studies on a number of different planning problems, comparing the performance of the Plexus planner with other approaches, such as real-time dynamic programming and several search-based techniques. The goal of this study is to determine classes of problems for which one planning approach can be shown empirically to perform better than another, and to formulate rules or heuristics to predict this, given new planning problems. In order to give quantitative comparisons of planning problems, I define a number of metrics on the space of planning problems (satisfying the conditions presented in Section 2.2). I present these metrics in Chapter 4, and then discuss the results of the empirical evaluation in Chapter 5.

Chapter 4

Characterization of domains and tasks

A domain and a task, as I have defined them, specify both the mechanics of the world and a measure of performance: maximize discounted expected reward over an infinite horizon in the associated Markov decision process (see Section 2.1). In general there might be other requirements made on the system, such as a minimum performance that must be guaranteed, or a specification of attitude to risk, but I do not address such issues here, as maximizing expected performance is a good criterion for the types of domain I have studied.

For a particular domain-task pair, the expected performance of a planner can be estimated empirically simply by simulating its interaction with the environment a large number of times and measuring the performance in each run. However, this process can be very time-consuming. In addition, it is often the case that we are interested in the performance over a large set, or even a continuum, of different domain-task pairs, and it may be infeasible to estimate the performance for every pair empirically.

As an example of the need to examine a large set of planning problems, consider the traffic light problem posed in the introduction. A number of parameters that define the problem are under the control of the designer. One such parameter is the granularity of the sensors: do they count the number of cars accurately, or do they report readings such as “one”, “two”, “many”? Another parameter the designer controls is the time between actions. Should the system be allowed to make lights change every second? Every minute? The effect of these parameters may be quite unobvious; intuitively, accurate sensors are a good thing, yet in fact a large number of possible readings may bog the system down with irrelevant detail. It is useful therefore to be able to predict the performance of a particular planning system on different problems of this type, without resorting to lengthy simulations.

To make these predictions, I classify domain-task pairs by the values of a number of *attributes* such as size, connectivity, *etc.* that are correlated with the performance of the planner on the domain and task. An attribute is a function mapping a domain-task pair to a real number, quantifying the attribute. The size of the problem, for example, can be defined in different ways: the number of states; the number of non-zero transitions; the information-theoretic content of the transition matrix, and so on. Each of these is an attribute. Other attributes measure aspects of the connectivity of the states in the markov decision process, aspects of the reward structure, and so on.

4.1 Performance comparison

In order to situate the performance of the Plexus planning system, I compare its performance to several other planners.

- Optimal planner

  A planner that immediately provides the optimal policy for the domain-task pair and never changes it. The optimal policy is defined in Section 2.1. Since finding the optimal policy can be time-consuming, this is done off-line.

  While it may be possible for relatively small problems to find the optimal policy for comparison with the performance of other planners, finding an optimal policy is not a practical approach for the real-time planning problems I address here. The principal reason for this is that computing the optimal policy takes too long to be done on-line by an embedded system that must respond to requests quickly. Optimal policies are therefore only useful if they can be generated off-line. This is possible if there is one well-defined problem to solve over and over again, or for a long period of time, but if the task can change dynamically, one would have to compute optimal policies for all possible tasks off-line. This is generally completely infeasible, both because of the time it would take to compute them and the space it would take to store them.

- Cautious planner

  The cautious planner initially provides a random policy to the agent. It then applies Howard's policy iteration algorithm on the entire state-space until convergence, *i.e.,* until the optimal policy has been found. It gives this policy to the agent, and thereafter does nothing. It is called conservative because it does not commit to anything until it is sure it has the optimal policy.

- RTDP

  RTDP is an implementation of the real-time dynamic programming system suggested by Barto *et al* [BBS93]; it uses a modified version of value iteration combined with an iterative-deepening search to approximate the values of states (the expected discounted reward starting from each state), and uses a greedy policy (choosing the action that maximizes the expected value).

- Recover, replan

  Recover and replan are two strategies based on search. They use a best-first search algorithm to find a path from the initial state to a goal, then execute the corresponding policy until the agent enters a state not on the path. At this point the replan planner discards the old path and searches for a new one, while the recover planner looks for a path from the current state back to the old path. These planners are described in more detail in B.2.

4.2 Examples

To ground the following discussion, I will take examples from several different types of planning problem. The problems are described fully in Section B.2.

The first example set is part of the RN1 class of planning problems described earlier. The size of the grid is fixed at 5 by 5, and the task under consideration is one of achievement; the goal state is L-5-5, and the start state is L-1-1. I chose a discount factor $\gamma = 0.999$; with this value (for these small domains), discounting plays a minimal role in performance. The domains are therefore completely determined by the success parameter, specifying the probability that an action will result in the nominal outcome (a movement in the specified direction). The domain is described in detail in Section B.3.1; the task is defined by

$$\begin{array}{lcl} \mathcal{X} & = & \{\text{L-5-5}\} \\ s_0 & = & \text{L-1-1} \\ \rho(s,a) & = & \left\{ \begin{array}{ll} -1 & \text{if } s \neq \text{L-5-5} \\ 0 & \text{if } s = \text{L-5-5} \end{array} \right. \end{array}$$

4.3 Attributes

In this section I present a list of the attributes I use to predict the performance of planners. These attributes were chosen based on experience with the three classes of domains described in Appendix B.3.2. In Section 5 I present the experimental results of prediction of performance using these attributes. I do not expect these attributes to be sufficient for any large class of domains; in general I have found that new types of domains require new characterizations of some aspect of complexity that was not present in the other domains. However, since these attributes are by nature domain-independent, one might expect that a fairly small set of attributes could be developed for a given class of problems. That has been my experience with the robot navigation and racetrack domains.

4.3.1 Planning cost *vs* execution cost

There are two basic types of attributes; those that influence the expected value accumulated during execution using the optimal policy, and those that influence the difficulty of determining a good policy.

For example, Figure 4.1 illustrates a problem with $n+2$ states, two actions (a and b), and a uniform cost function. The annotations on the arcs indicate the action, and the probability of making the transition if the action is taken. In this problem, the task is inexpensive for the optimal policy: the policy that always chooses the action b will succeed with an expected cost of 2. However, a planner cannot determine the optimal policy without examining each of the intermediate states, and this requires at least $O(n)$ operations.

On the other hand, Figure 4.2 illustrates a problem where the planner's job is trivial; in constant time (independent of n) it can compute the optimal policy: always execute action a, but the expected cost of execution is proportional to n.

In the discussion that follows, I distinguish these two types of attributes, though all comments at this level of generality are of necessity very broad. At this stage, I am not concerned with the computational complexity of determining the attributes themselves; the first goal is to determine a set of attributes that predict the performance of Plexus. When determining the value of attributes is prohibitively expensive (as with the distance asymmetry, for example), I use sampling techniques to obtain an approximation of the value.

I will attempt to provide the reader with some intuitions about the meanings of the various attributes I use, by showing how the attributes vary as a function of model parameters, which usually have obvious interpretations.

In the remainder of this section, I consider a domain $\{\mathcal{S}, \mathcal{A}, \tau\}$, and a task $\{\rho, s_0, \mathcal{X}, \gamma\}$, and I assume that the assumptions listed in 2.2 hold.

4.3.2 Size

I define the *size* of the domain as the number of states in the domain, $\mid \mathcal{S} \mid$. To show the order of magnitude of the sizes of the different problems, here is a log-scale plot of some of the example problems I discuss.

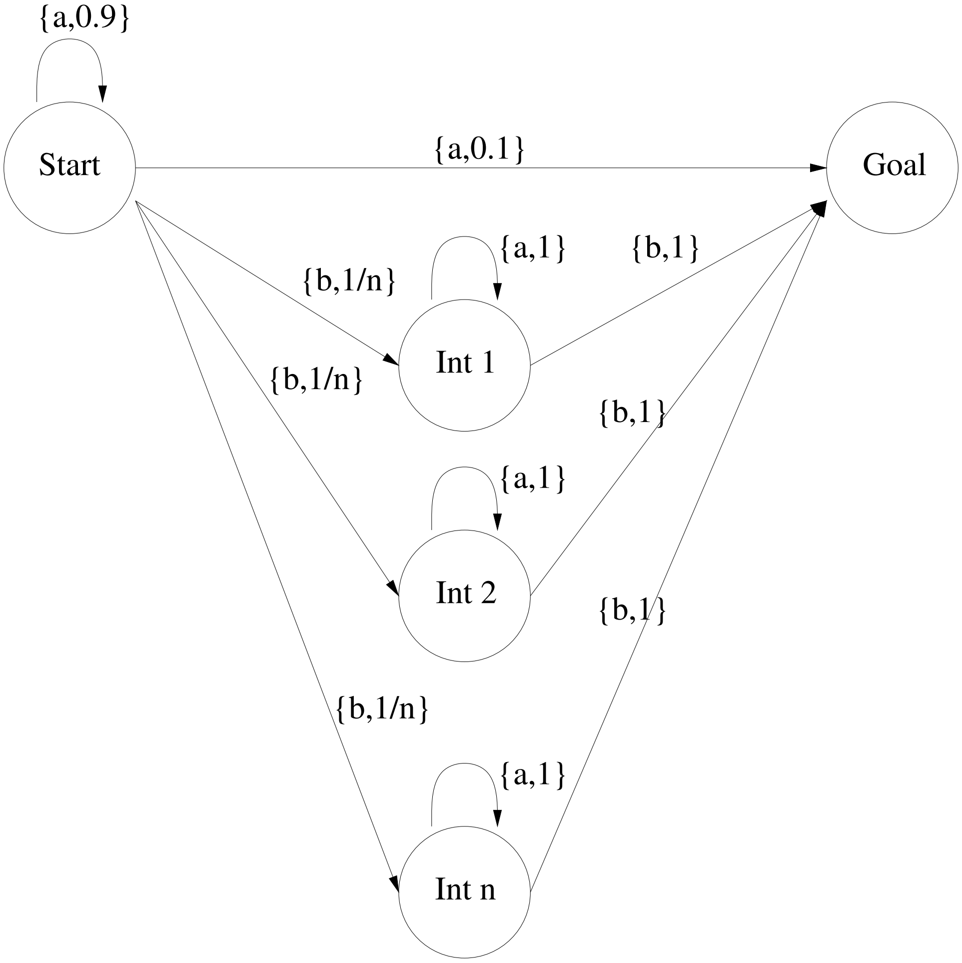


Figure 4.1: Expensive planning, cheap execution

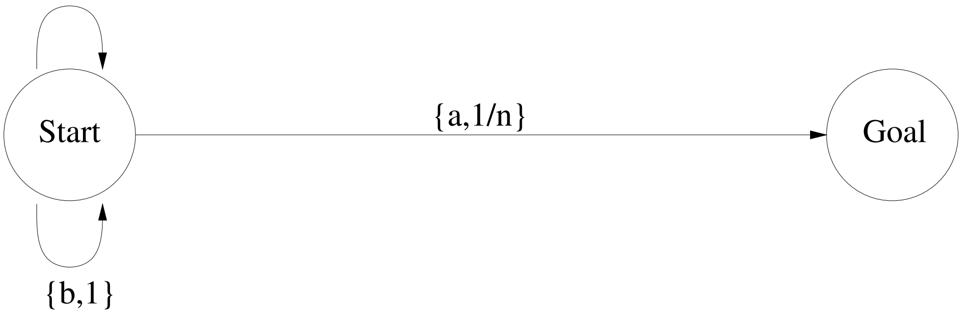


Figure 4.2: Expensive execution, cheap planning

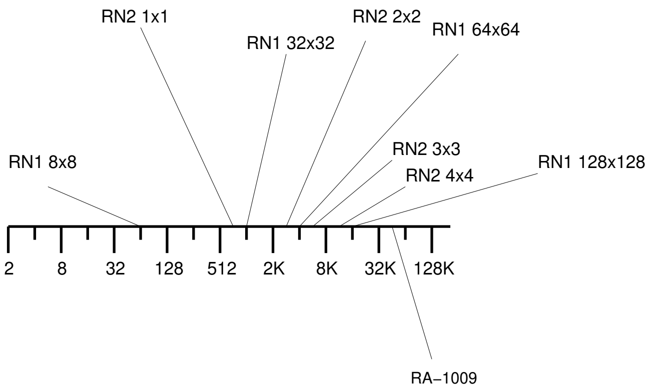


Figure 4.3: Range of sizes of example problems

4.3.3 Entropy

The entropy (see, for example, Renyi [R84]) of a random variable x ranging over a discrete state space S is defined as

$$H(x) = -\sum_{s \in S} \textsc{Pr}(x = s) \log(\textsc{Pr}(x = s)) \ .$$

Entropy is a characterization of the uncertainty in a probability distribution. Consider, for a state s and an action a, the random variable $O_{s,a}$ defined over $\mathcal{S}$ by

$$\textsc{Pr}(O_{s,a} = s') = \tau(s, a, s') \ .$$

The entropy of this variable is a measure of the uncertainty of the outcome of the action a in state s. I define the entropy of a state/action pair as

$$H(s, a) = H(O_{s,a}) \ .$$

The *average entropy* of the domain is defined in the obvious way, as the average, over all states and all actions, of the entropy of the state-action pair

$$\frac{\sum_{s \in \mathcal{S}, a \in \mathcal{A}} H(\tau(s, a))}{|\mathcal{S}| \times |\mathcal{A}|} \ .$$

Figure 4.4 shows the average entropy in the RN1 domains, as p varies from 0 to 1. As one would expect, the entropy is lowest when $p = 1$ and the outcome is deterministic. Less obviously, the entropy is lower for very small values of p than for medium values; when $p = 0$, for example, there are two equally likely outcomes, whereas when $p = 1/3$, there are three equally likely outcomes. The

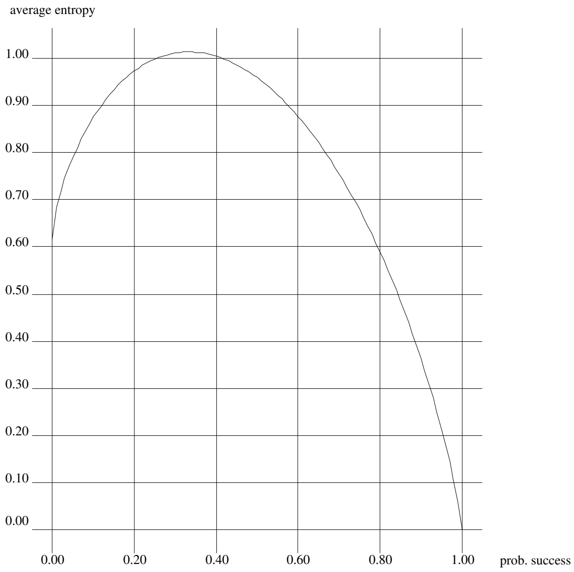


Figure 4.4: Entropies for different values of p in the RN1 domain

fact that some actions are deterministic (`stay` always is, and some others are for locations at the edge of the grid) lowers the average entropy. With a larger grid, the average entropy for $p = 0$ tends to $4/5$: an entropy of 1 for the four movement actions, and an entropy of 0 for the stay action.

The RN2 domains, described in detail in Appendix B.3.2 domains are similar to the RN1 domains, but much richer; some locations are inaccessible and some are difficult to exit, for example. Also, the representation of state and action is slightly different; the state encodes the orientation of the robot, and the actions are in the co-ordinate frame of the robot. Despite these differences, entropy as a function of the success probability is very similar to that of the RN1 domains, though the entropy is substantially higher. Figure 4.5 shows the average entropy in the RN2 domains , as the probability of success varies from 0 to 1. In some cases, as in these two examples, the entropy corresponds very closely to a model parameter chosen by the designer. In other cases there may not be any obvious relationship. Figure 4.6 shows the entropy of a set of RN2 domains as the number of sinks increases from 0 to 8 (at which point three quarters of the states are sinks). Sinks are used to model areas the robot should avoid because they may confuse its low-level sensing equipment. As the number of sinks increases, the entropy decreases approximately linearly. This is a case where low entropy is not desirable.

Domains with high entropy generally make planning more expensive, since

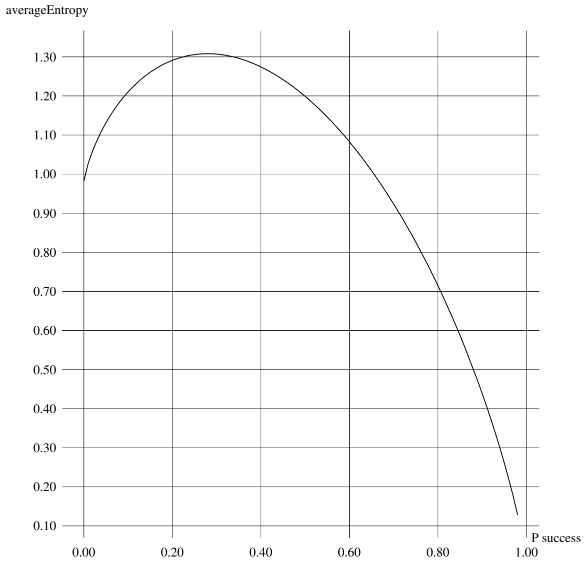


Figure 4.5: Entropies for different probabilities of success in the RN2 domain

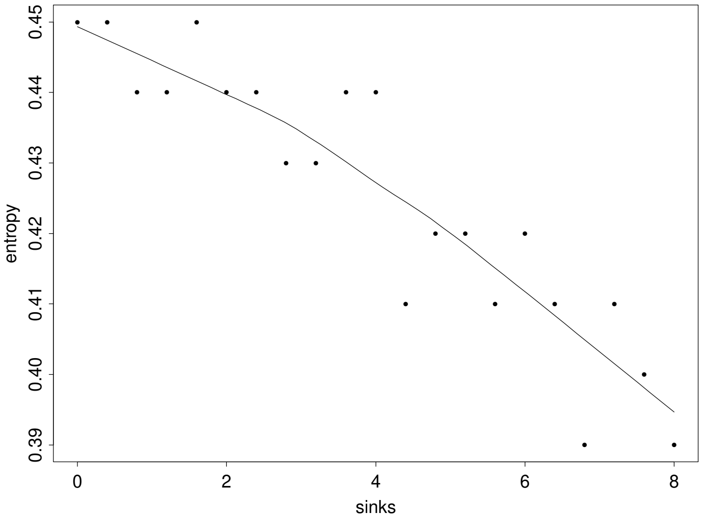


Figure 4.6: Entropies for different numbers of sinks in the RN2 domain.

more possible outcomes must be taken into consideration. The influence of entropy on execution cost is very dependent on other attributes such as the controllability.

4.3.4 Controllability

I define the *controllability*[1] of a stochastic process as a measure of the effect of the agent's choice of actions on the outcome of the action. If the choice of action does not influence the outcome, the controllability is 0. If the choice of action completely determines the outcome, and all outcomes are different, the controllability is 1.

I define the controllability of a domain as the average over states of the controllability in that state. The controllability in a state is defined as follows. Given a state s, let A be the random variable corresponding to the action taken, and let O be the random variable corresponding to the outcome. For a given action a, let O_a be the random variable corresponding to the outcome of action a in state s. Let O_r be the random variable corresponding to the outcome of a random action in state s, *i.e.*, $O_r = 1/\left|\mathcal{A}\right| \sum_{a \in \mathcal{A}} O_a$.

The *mutual information* of O and A is the measure of controllability of a state. It is the amount of information about the variable O that we obtain when we determine what value A takes:

$$\mathcal{I}(O, A) = H(O) - H_A(O) \ .$$

I assume that the prior probability for each action is equal, so the mutual information becomes

$$\mathcal{I}(O, A) = H(O_r) - \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} H(O_a) \ .$$

Intuitively, the notion of controllability in a state s corresponds to the amount of influence the agent has on the successor of s by its choice of action. Figure 4.7 shows the average controllability in the RN1 domains, as p varies from 0 to 1. Notice that the controllability is monotonic; when p is very small, the outcome of any given action (moving along one of two diagonals with equal probability) is similar to the outcome of the random action (moving along any of the diagonals with equal probability). The outcome of the action is more certain (the entropy is lower), but the agent has less control over the outcome by choosing amongst the different actions. This is a fairly subtle point, and it highlights the difficulty of interpreting the attributes.

---

[1] The term controllability is used for a different (though vaguely related) concept in control theory.

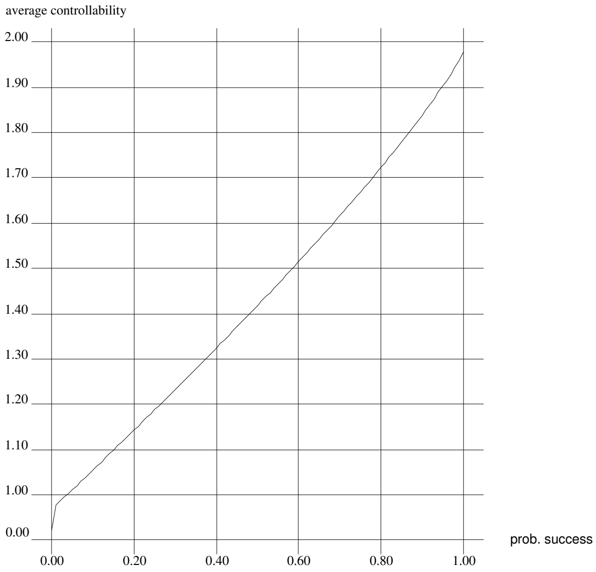


Figure 4.7: Entropies for different values of p in the RN1 domain

The RN2 domains exhibit a slightly different pattern of controllability as the probability of success is varied; this is illustrated in Figure 4.8. On average, the RN2 domains have less controllability than the RN1 domains. In the set of RN2 domains that vary by number of sinks, the controllability decreases both as the number of sinks increases, as shown in Figure 4.9. This is because the controllability in the sink states is very low (the state will remain unchanged most of the time, regardless of the action used).

4.3.5 Openness

I define the openness of a domain as a measure of the ease with which the process moves through the state space over time when the agent takes random actions. In an open domain prediction is difficult, because there are many possible futures; in a closed domain the system is constrained to move through the state space slowly. Problems involving robots in a manufacturing assembly line would be less open than ones involving mobile robots, as the possible sequences of events are much more closely choreographed in the static case.

This is a difficult notion to measure, since it depends on long-term sequences of actions. As a first approximation, I use a local measure, the difference of the entropy of two random actions and that of one random action. That is, it specifies how much more uncertain the state is after two steps than after one step. In an open domain, this difference will be large; the uncertainty grows continually as one looks further into the future.

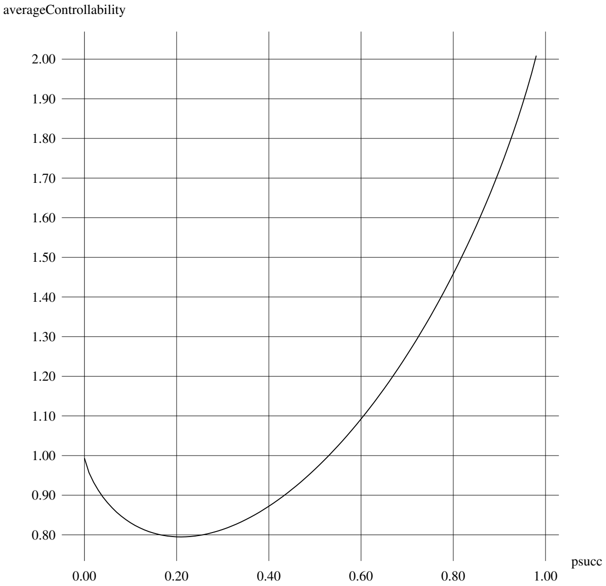


Figure 4.8: Controllabilities for different probabilities of success in the RN2 domain

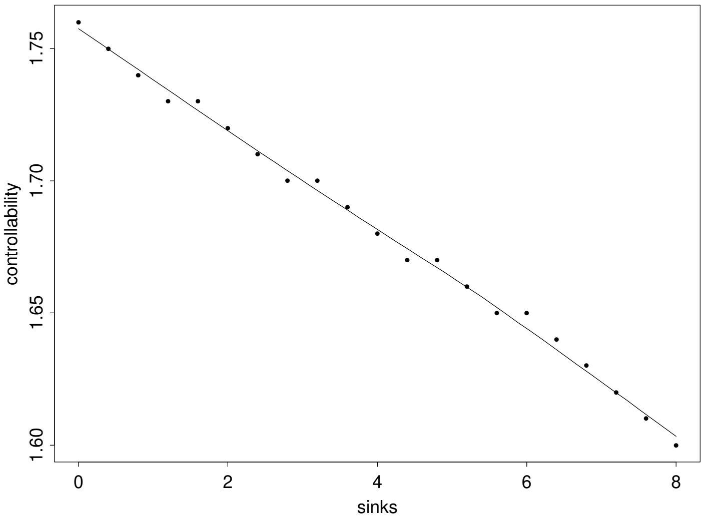


Figure 4.9: Controllabilities for different numbers of sinks in the RN2 domain

I define the openness of a state as

$$H(O_{O_{s,r},r}) - H(O_{s,r})$$

where r is the (uniformly) random action.

4.3.6 Volatility

The volatility of a domain is the average time taken to execute one action; this characterizes the amount of time the planner has to generate new policies as the agent moves through the state space. Although the optimal values are independent of volatility, in practice I have found this attribute to be very important for all planners.

4.3.7 Distance asymmetry

The ability to retrace one steps — to undo previously done actions — is valuable for planners in stochastic domains. To quantify this ability, I will first define a notion of distance between states in a Markov decision problem, quantifying the cost of getting from one state to another. The measure of the reversibility of transitions is then based on the asymmetry of this distance measure.

For any two states s and s' in the state space of the problem, the distance measure from s to s' is intuitively defined as the discounted expected reward of getting from state s to state s' under the policy that maximizes this reward. Alternatively, one can think of it as the minimum discounted cost of getting from s to s'. In order to formalize this notion, I define, for each state s', a new Markov decision problem in which s' is an absorbing state with reward 0. Otherwise the problem remains the same. The distance from s to s', $d(s, s')$, is the value of the state s under the optimal policy using the discounted infinite horizon model of cost.

The distance asymmetry for a pair of states s, s' is the ratio of the difference in the distances and the sum of the differences:

$$da(s, s') = \frac{|\, d(s, s') - d(s', s)\, |}{d(s, s') + d(s', s)} .$$

The distance asymmetry for the domain is the average distance asymmetry over all pairs of states. The reason for using the ratio of the difference and the sum, rather than simply the difference, has to do with the fact that computing the distance between all pairs of states is prohibitively expensive. To address this, I use Monte-Carlo sampling. I computed both measures, and looked at the confidence intervals for the values in domains of all three types, as a function of sample size.

In each case the convergence of the normalized measure was substantially faster than that of the absolute rate.

In large domains, even a single policy optimization is computationally infeasible; in this case, approximations of the cost of getting from one state to another can be gathered for subsets of the domain. Since the distance asymmetry is averaged over many pairs of states, obtaining an exact measure of the distance between each pair is not necessary. Approximate techniques for policy improvement converge much more quickly than exact ones, making this attribute somewhat easier to compute, though it is still infeasible for large domains. Generally, planners do less well on domains with high distance asymmetry, but more precise analysis is domain-specific.

4.3.8 Irregularity of rewards

The reward irregularity is a measure of the lack of smoothness in the distribution of rewards over space. A domain with regular rewards is generally "safer" than one that is irregular, in the sense that the agent need not be as preoccupied with the precise path it will follow.

I first define the average reward of a state as

$$\forall s \in \mathcal{S}, \mathcal{AR}(s) = \sum_{a \in \mathcal{A}} \rho(s, a)$$

then, I define the reward irregularity of the domain by

$$\frac{1}{|\mathcal{S}| \times |\mathcal{A}|} \sum_{s \in \mathcal{S}, a \in \mathcal{A}} (\rho(s, a) - \sum s' \in \mathcal{S}(\textsc{Pr}(s, a, s') \times \mathcal{AR}(s')))$$

That is, it is the average difference between the reward in a state and the reward in the state the system is in after one time step.

For problems of achievement, the reward irregularity is 0, since the reward in each state other than the goal is -1. For problems with more complex reward structures, the reward irregularity is usually non-zero. I have not yet performed extensive tests on problems with irregular rewards.

4.3.9 Additional attributes

I have implemented the computation of the above attributes, and measured their correlation with performance in the RN1, RN2 and racetrack domains, as described in the next section. A number of other attributes still require examination to determine whether they will be useful in increasing the accuracy of performance prediction in these and other domains. I simply list them here with a brief explanation.

- Multi-step entropy

  The entropy resulting from a sequence of actions (as opposed to a single action) will provide a measure of the tendency of a random walk to spread the probability distribution. In domains where movement through the state space is restricted, the increase in entropy as a function of action sequence length will not be as quick as in domains where movement through the state space is not restricted. Therefore both the average entropy for fixed length sequences and the tendency of the entropy to increase as a function of the length of the sequence should be of use in predicting performance, used in conjunction with attributes such as the controllability.

  The n-step entropy of a state s is the average over all action sequences σ of length n of the entropy of the random variable corresponding to the outcome of following action sequence σ from state s. The n-step entropy of the domain is the average of the n-step entropies of the states in the domain.

- Multi-step controllability

  For similar reasons to those described for multiple-step entropy, the notion of controllability can be extended to consider multiple actions. The n-step controllability at state s is the mutual information of a variable corresponding to an n-step action sequence and a variable corresponding to the outcome of executing that sequence from state s. The n-step controllability of the domain is the average of the n-step controllabilities of the states in the domain.

- Conductance, merging conductance

  Jerrum and Sinclair [JS88] and Mihail ([Mih89]) discuss related concepts for Markov chains, called the *conductance* and *merging conductance*; however, these do not have obvious analogs in the general Markov decision problem case, and are very expensive to compute.

Chapter 5

Experimental results

In this chapter I present experimental results of the performance of several planners on a large number of different domains. The primary goal is to formulate models of the performance of the planners as a function of attributes of the domains.

Because of computational resource limitations, I have not examined all the planners on all the domains; I present results for the Plexus phase planner (described in Section 3.9) for all domains, and selected results for the other planners. For comparison, I will present the optimal performance (the expected discounted sum of rewards for an optimal policy) and the performance of the planner called "cautious" in previous work [DKKN94]; it first computes the optimal policy using policy iteration and then executes that policy. The optimal performance gives an upper bound on the performance one can expect from a planner, while the cautious planner illustrates the impracticability of optimizing a policy over the entire state space for large spaces.

There are many different views of the data that are of interest for each of the three domains sets I examine. Planner performance can be represented as a function of the domain parameters or the attributes; the "performance" may be the absolute performance of Plexus (accumulated reward), or that of the optimal policy, or that of another planner, or yet the ratio of one to another. One can examine the performance averaged over all domains of a particular type (*e.g.,* all those of a particular size), or take a one-dimensional slice in which all the parameters or attributes remain constant except for one. In order to keep the presentation down to a reasonable length, I have eliminated many of the possible views; for the first few analyses of each type I go into some detail, but in subsequent analyses I will generally skip those views that do not offer any interesting points of discussion.

5.1 Statistical models

A statistical model is a description of a relationship between a set of *predictors* and a *response*. For example, one might model the height of a group of people in terms of their age and sex. `height` is the response (the variable we are interested in modeling); `age` and `sex` are the predictors.

The model can be *linear*, *e.g.,* $\texttt{height} = \texttt{age} + 2 * \texttt{sex}$, with the convention that `sex` is 1 for females and 0 for males. Linear models are not restricted to weighted linear combinations. As long as the response can be represented as the sum of the predictors, (possibly after some transformation), the model is linear. A wide range of statistical techniques are then available to determine the best set of weights and the accuracy of the prediction. Thus, the model $\texttt{height} \sim \texttt{age}^3 - 20 * \texttt{age} + 3 * \texttt{age}^2$ is also linear.

Linear models are a kind of *parametric* model: the response is specified as a function of the predictors. Parametric models are useful in many circumstances, but they depend on assumptions about the relationship being modeled. For example, that the variance of the response for each set of predictors be the same. When these assumptions do not hold, *generalized linear models* can be used; these models allow the variance to depend on the mean value of the response. More general still are non-parametric models, such as local regression models, which use local estimates of the relationship. For the most part we will be interested in parametric models, since, empirically, non-parametric models tend to over-fit our data.

5.1.1 Model notation

When a precise description of a statistical linear model is necessary, I will follow the notation of the S language [Sci93]. The basic form of a simple linear model is

$$\texttt{response} \sim \texttt{predictor}\ .$$

This means that `response` is a linear function of `predictor`. More generally, the response may be a linear function of several predictors, written

$$\texttt{response} \sim \texttt{predictor1} + \texttt{predictor2}\ldots\ .$$

This kind of model is called *additive*; it is a sum of components that are computed independently, one for each predictor. When the response is to be modeled as a polynomial function of a predictor, the model is written, *e.g.,*

$$\texttt{response} \sim \texttt{poly(predictor, 3)}\ ,$$

to mean that a third-degree polynomial should be used — the coefficients are chosen to minimize error in the same way that the linear coefficients are chosen.

In the type of study I present here, the coefficients of the polynomial are rarely of interest, since the number of points in any dimension is insufficient to determine them accurately. The general form of the curve is useful, however, to form an idea of the relationship.

5.1.2 Response

There are several different responses that we may be interested in:

Absolute performance. If the designer has constraints on the acceptable performance of the planner, predicting the absolute performance can help determine acceptable ranges of predictors. For example, an analysis of the traffic world problem described in the introduction might reveal that the planner under consideration can only perform acceptably when the number of intersections is less than 20.

Performance relative to optimal. This type of model is particularly useful for determining areas in which a planner needs improvement. When the planner's performance is close to optimal, there is no point in expending energy trying to improve that performance.

Performance relative to other planners. When several planners can be used to solve a particular problem, it is useful to look at the performance of one relative to another. The problem space can then be divided into regions, with the best planner associated to each region. This process can take place off-line; alternatively, if many different problems are to be solved consecutively, the comparison can be done on-line.

5.1.3 Predictors

There are two candidates for sets of predictors. The most obvious choice is to use the parameters that specify the domain. In the RN1 examples, the width of the grid, the duration of actions and the success probability are the three domain parameters. Domain parameters often provide good models. However, domain parameters are inadequate for several reasons:

- For real problems, the world does not behave like a mathematical model – although the designer may specify parameters that affect the behaviour of the world, there are often hidden parameters. The values of the parameters chosen by the designer may therefore not be sufficient to specify the world, or to predict the performance of a planner operating in that world.

- Parameters are sometimes discrete. There are statistical techniques for modelling discrete parameters, but they do not allow any generalization to new values for the parameters.

The second possibility is to use attributes such as those described in the previous chapter. attributes have both advantages and disadvantages as predictors, compared with domain parameters. To their advantage, attributes are, by definition, quantitative, whereas domain parameters may not have any useful quantitative interpretation. They are also independent of the domain, making it possible to compare models across domains (though in practice I have found this of limited utility). On the other hand, the choice of particular attributes may be hard, whereas with domain parameters the only question is which ones are significantly correlated with performance. Domain parameters also typically have more obvious interpretations than attributes. In this chapter I will examine both types of predictors. The choice of domain parameters or attributes will depend on the particular problem at hand, and the purpose of the statistical model.

5.1.4 Predictive accuracy of models

In order to compare the predictive accuracy of different models, we must specify a metric by which to evaluate the accuracy. Various measures of this accuracy have been proposed (see, *e.g.,* [Gre93]). One is the root mean-squared error, defined by

$$RMSE = \sqrt{\sum_i (y_i - \hat{y}_i)^2},$$

where the y_i are the actual measurements of the response, and $\hat{y}_i$ are the predicted responses. Another measure is the mean absolute error, defined by

$$MAE = \sum_i |y_i - \hat{y}_i| \, .$$

Both of these suffer from a dependence on the scale of the response. A third measure, suggested by Theil [The61], is called the U-statistic, and is defined by

$$U = \sqrt{\frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i y_i^2}}.$$

In each case, the smaller the value, the better the predictive accuracy. None of these is ideal for our purpose; the first two are scale-dependent, and the U-statistic is too dependent on those data with large absolute response — in cases where a few of the data have much larger absolute response than the others, the U-statistic will report a low value for models that accurately fit those data, while being proportionally quite poor for the majority of the data.

The measure I use is the average of the ratio of the error to the response; this measure is similar to the Theil U-statistic in that it normalizes the error by the magnitude of the response, but the normalization is done on each datum, not globally. The measure is therefore sensitive to small absolute differences when the response is small. Formally, I define the mean normalized error of a model predicting the values $\hat{y}$ by

$$MNE = \text{mean}(|\,(y_i - \hat{y}_i)/y_i\,|).$$

This measure can be interpreted as the average proportion by which the prediction is in error; a MNE of 0.1 means that on average the prediction is accurate to within 10% of the performance. I will generally refer to the mean normalized error of a model as the error of the model.

5.2 RN1

The RN1 domains are described in detail in Section B.3.1; briefly, a robot navigates through a 2-dimensional grid. The state of the world is just the grid cell the robot occupies, and the actions the agent may take are to move in any of the four compass directions. The size refers to the size of the grid; the success is the probability that the robot will move in the specified direction; if the action "fails", the robot moves diagonally. The robot's moves are made at discrete time intervals; the duration specifies the length of the intervals, *i.e.,* how much time the agent has to think during each move.

5.2.1 Domains

There are 72 different domains, varying in three parameters: the size of the grid, the probability of success of an action (the time between state transitions), and the duration of an action. The different values were:

- Size: 8x8, 32x32, 64x64, 128x128
- Success: 1.0, 0.9, 0.8, 0.7, 0.5, 0.2
- Duration (seconds): 5, 1, 0.1

The task in each case was to reach the lower right corner, starting from the upper left corner. To encode this task of achievement, the set of terminating states contained just the lower right corner state, and the rewards were -1 for all states and actions except for the terminating state, where they were 0. The discount factor was 0.9999. Since this set of domains gives a fairly coarse coverage of the space of all possible values for these three parameters, I have added various other sets of domains for a more detailed look at some relationships. I describe these sets as they are used.

Since this RN1 domain set is not a characterization of any real problem, I use it mainly as a simple case to illustrate the form of the statistical analyses and to show how the data is presented. The first analyses will be detailed, but in subsequent ones I will frequently refer back to earlier cases, and concentrate on the more interesting differences.

First, we will examine the performance of the Plexus planner as a function of the domain parameters `width`, `success` and `duration`. Then we will consider the performance as a function of the attributes described in the previous chapter.

5.2.2 Influence of domain parameters on absolute Plexus performance

Principal influences

Figure 5.1 shows the ranges of the means for each parameter. For example, the four ticks on the leftmost line show the mean performances when `width` is 8, 32, 64 and 128. The analysis of variance of the dependence of performance on `width`, `duration` and `success`, gives a more conventional summary:

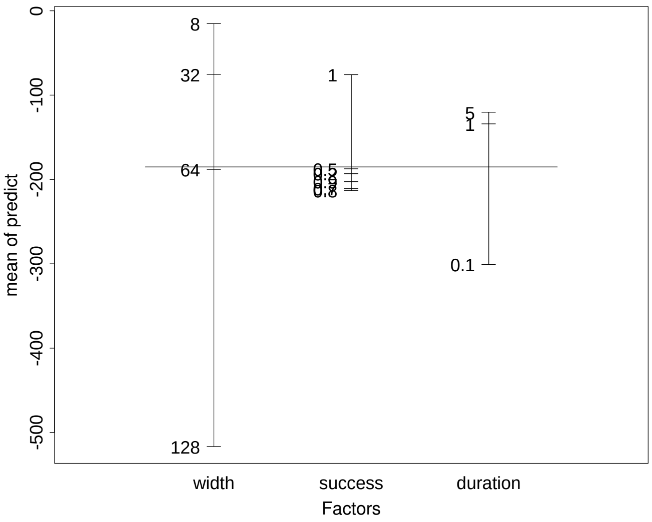


Figure 5.1: Variation in performance as a function of RN1 parameters

```
Df Sum of Sq  Mean Sq
```

```
width                   3   2281615 760538.3
success                 5     20720   4144.0
duration                2    400705 200352.5
width:success          15     48111   3207.4
width:duration          6    725925 120987.5
success:duration       10     42176   4217.6
width:success:duration 29     69920   2411.0
```

The analysis of variance table lists, for each parameter, the *sum of squares between classes.* To find these, we divide the data into classes, one for each value the parameter can take. We compute the mean performance for each class, and then take the difference of the square of the mean for the class and the square of the overall mean of the data set. If the parameter has a big effect on performance, the different classes will have very different means. This difference is weighted by the number of elements in the class, and the sum of the weighted differences is listed in the ANOVA table. The larger the sum of the squares, the more the parameter contributed to the variability in performance. The degrees of freedom column (Df) specifies the number of different values the parameter can take. Entries such as "`width:duration`" refer to the interaction of `width` and `duration`.

From this, we see that `width` and `duration` are the two important parameters; there is little dependence on `success`. Also, there is substantial inter-dependence between `width` and `duration`. A simple additive model will not capture this inter-dependence well.

General form

To get a general idea of the form of the performance as a function of the three parameters, we will first collapse the data to two dimensions by averaging over all values of `success`; to see the effect of `success`, we will fix `width` to each of the values it can take, and show how performance varies as a function of `duration` and `success`.

Figure 5.2 shows the performance as a function of `width` and `duration`. I usually let each parameter take on only four or five values; the number of domains is the product of the number of values each parameter can take. Tens or hundreds of runs are required for each planner and each domain; more values for parameters would have been prohibitively time-consuming. To get a better notion of the precise effect of each parameter, we will later look at one-dimensional slices through this three-dimensional parameter space.

The parameter that contributes the most to the variation in performance is `width`, so we next look at the performance as a function of `duration` and `success` for each of the four values of `width`. Figure 5.3 shows the four cases (only the

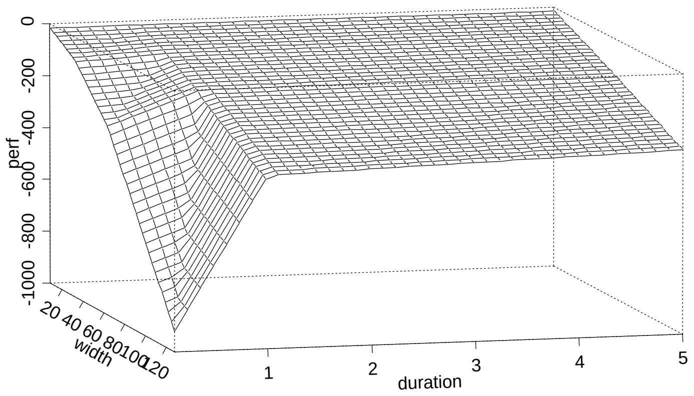


Figure 5.2: Performance as a function of `width` and `duration`

general form of the surface is of interest here, hence the low resolution.) A first look at the data thus shows that the performance decreases more rapidly than linearly in `width`, and decreases more rapidly than linearly as `duration` tends to 0. There is a positive interaction between `width` and `duration` — a small duration and a large width lead to lower performances than one would expect with a purely additive relationship. For small sizes, there is very little effect of `duration` or `success` on performance. For larger sizes, performance decreases faster than linearly as `duration` tends to 0; it is highest when `success` is 0.9, slightly lower when `success` is 1, and steadily lower as `success` decreases. This somewhat surprising result is due to the fact that with low success rates the robot tends to move diagonally, so each step takes it further than a straight step would. High success rates lead to fewer lost steps adjusting for failures, so either extreme has advantages; the intermediate values lose on both counts.

Detailed relationships

Before setting up a statistical model of the relationship to get a quantitative view, let us look at three one-dimensional slices through the parameter space. This will give us a better idea of the precise form of the relationship between performance and each of the parameters. I ran three experiments, fixing two parameters and varying the third, with much finer granularity than in the previous experiment.

The results for `duration` are shown in Figure 5.4. Performance is plotted

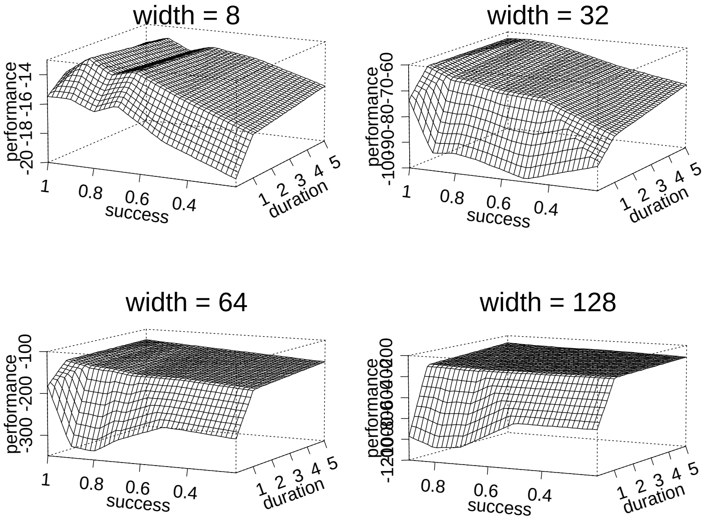


Figure 5.3: Performance as a function of `duration` and `success`

against `duration` raised to the power -1.5. The relationship is close to linear[1].

Next, we look at performance as a function of the width of the grid; performance is clearly proportional to the square of `width`, as Figure 5.5 shows.

Finally, the relationship between performance and `success` is naturally less well-defined (success rate is the parameter that least affects the performance). Performance is very bad when the success probability is very small (less than 0.1). Performance gradually increases as `success` tends to 0.4, then decreases to a minimum at 0.6 and finally rises steadily as `success` tends to 1. Figure 5.6 illustrates this.

This set of views of the data give us a good idea of the form of the relationship between performance and the three parameters: `width` and `duration` both have a strong influence, with performance decreasing as the duration decreases and the width increases. The effect of combined small durations and large widths is greater than a simple additive relationship would lead us to expect. For small durations (the case that accounts for the most variation in performance), performance is approximately linear in the square of the `width`. For large sizes, the performance is roughly proportional to the `duration` raised to the power -1.5 — it decreases more quickly than the inverse of `duration`. The effect of `success` is fairly small and definitely non-linear; there is one local (and global) maximum

---

[1] The two points corresponding to the smallest durations are anomalous, due to the overhead of inter-process communication with small durations (below a hundredth of a second).

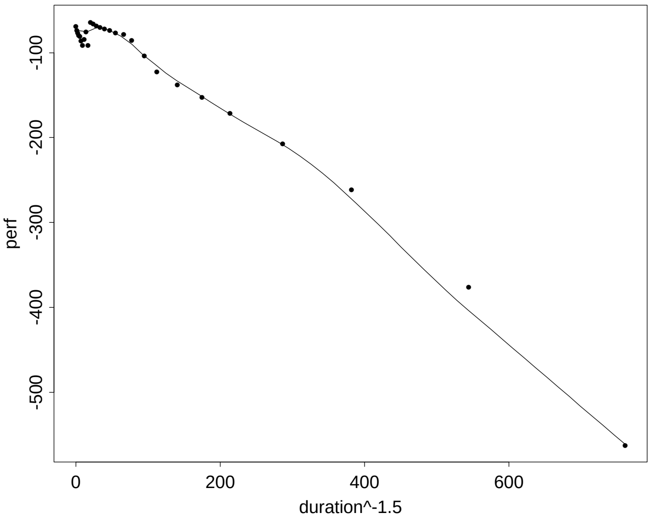


Figure 5.4: Plexus performance as a function of `duration`

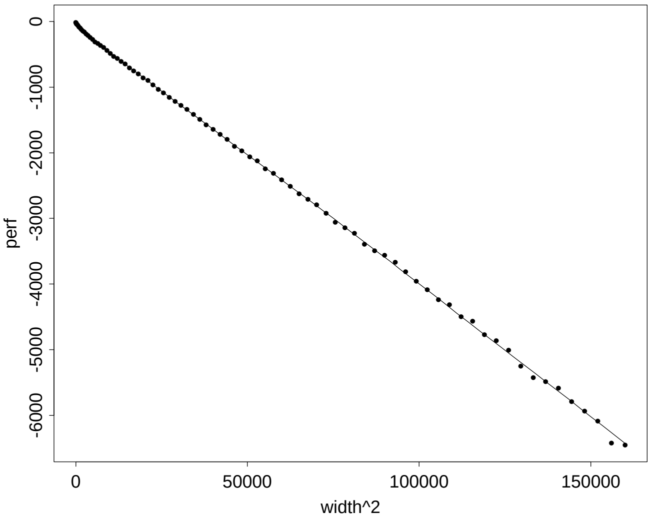


Figure 5.5: Plexus performance as a function of `width`

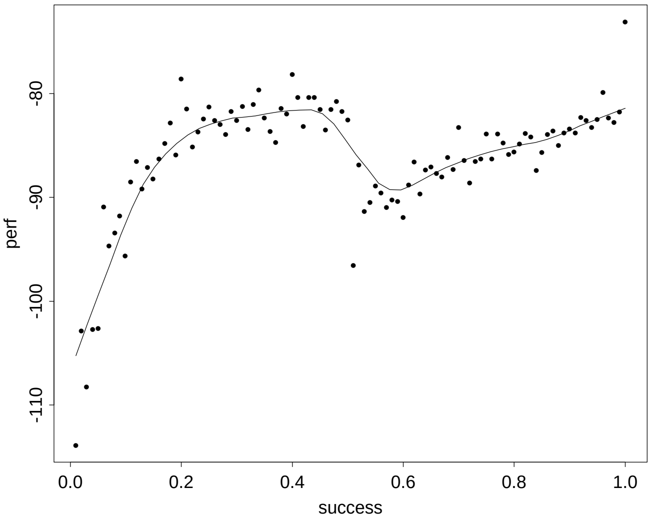


Figure 5.6: Plexus performance as a function of success probability

for `success` equal to 1, a local minimum around 0.6, a local maximum around 0.4, and the minimum is obtained for 0. These observations will form the basis of our statistical model of performance as a function of these three parameters.

Statistical models

Because there is a great deal of variance in the data, we construct the initial models using a version from which outliers have been removed. For each domain, we compute the mean performance and the standard deviation of the performance, and eliminate all data that is more than one standard deviation from the mean. The measurement of precision of the model is done with a second dataset, of similar size to the dataset from which the model was constructed, and from which outliers are *not* removed.

Some trial and error with different forms of models led me to a few reasonable models. The first is a simple linear model in `width` squared and `duration` to the power -1.5 (the powers are taken from the one-dimensional slices):

$$\texttt{perf} \sim \texttt{width}^2 * \texttt{duration}^{-1.5} \ .$$

The form of the relationship, as shown in Figure 5.7(b), is quite close to the actual data, shown in part (a) of the figure. On the second dataset, this model has an error of 0.35. That is to say, on average, the predictions of performance are off by about 35% of the performance. This is quite a large error, but the model is very simple, and would probably be sufficient for many applications, such as choosing between different planners, where order of magnitude differences are common. Numerically, the model is

$$\texttt{perf} = -43.1 - 0.017\,\texttt{width}^2 + 0.4\,\texttt{duration}^{-1.5} - 0.001\,\texttt{width}\,\texttt{duration}^{1.5}$$

If more accurate numeric predictions are required, rather than a simple description of the relationship, various transformations can be tried on the predictors or response. Again by trial and error, I found that the following model gives predictions that are considerably more accurate:

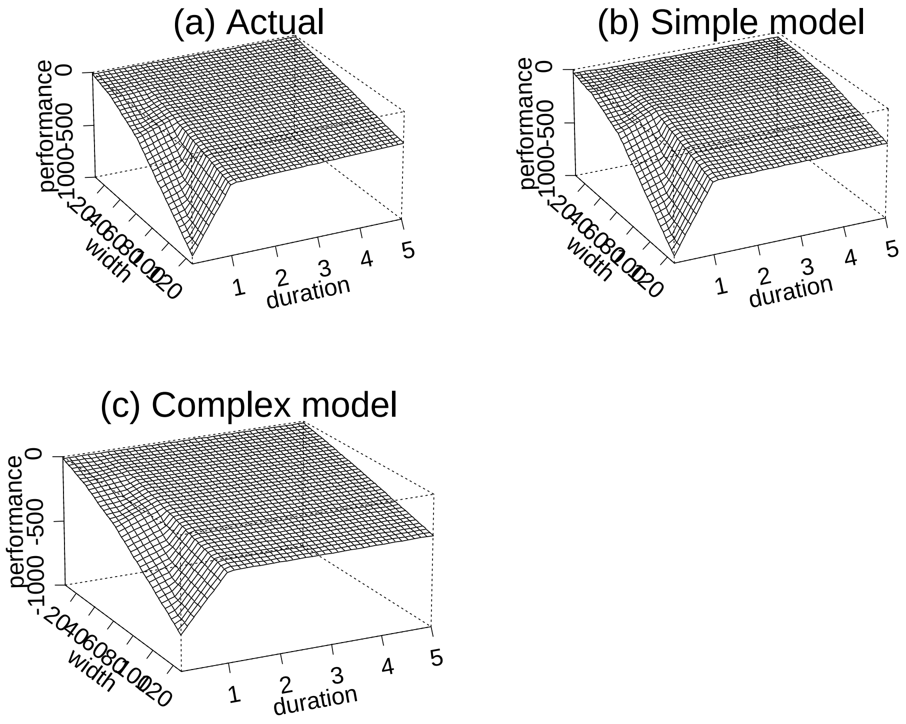


Figure 5.7: Actual performance (a), prediction from simple model (b), prediction from complex model (c) of Plexus on the RN1 problems as a function of `width` and `duration`

$$\log(-\texttt{perf}) \sim \log(\texttt{width}) * \texttt{duration}^{-1.5} * \texttt{poly}(\texttt{success}, 3).$$

The error of this model on the second dataset is 0.18. The model is shown in Figure 5.7(c).

Prediction

We have seen that in this case the model is reasonably accurate at predicting performance. We must also consider how well the model fits new domains. I consider the three one-dimensional slices specified earlier, and show what the various models predict. Each figure shows the mean performance over a set of experiments in which just one parameter varies. The duration is shown on a log-scale, since the interesting portion of the graph is otherwise too small. The actual mean performance in each case is represented by a point on the solid line. The light dotted line shows the predictions of the simple linear model of `width` squared and inverse `duration`; the darker dotted line shows the prediction of the more complex model, relating the log of the performance to the log of `width`, the inverse of `duration` and a polynomial of third degree in `success`. The models were constructed using only the coarse set of domains shown in earlier figures; they are being tested against sets of domains that require interpolation, and, for `duration` and `width`, also require extrapolation. There are several interesting points to note here. Both the simple and the more complex model are ample to see the general form of the relationship between the two principal parameters (width of the grid and duration of actions). The complex model captures the form of the relationship between success probability and performance accurately, though there is a substantial constant error. The complex model has considerable

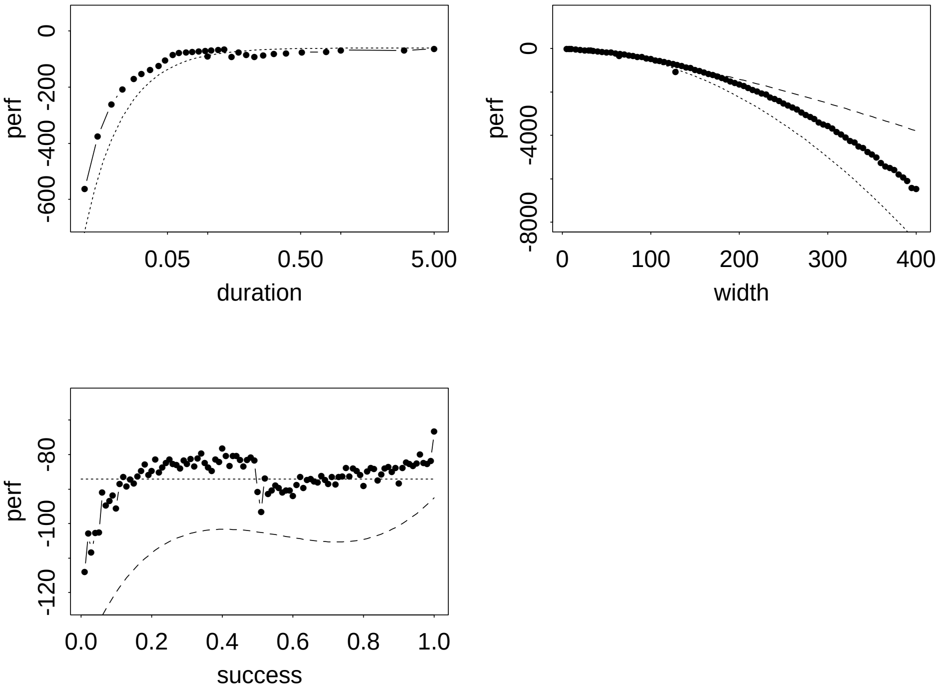


Figure 5.8: Actual performance (a), prediction from simple model (b), prediction from complex model (c) of Plexus on the RN1 problems

constant errors in several cases, and performs poorly in an accuracy test against the data on the duration-varying experiments.

Summary

The goal of this analysis was to determine a statistical model of the performance of the Plexus planner as a function of the three domain parameters. The first use for such a model is to visualize the general form of the influence of the parameters on performance; this was accomplished using a decomposition by successively less important parameters, with 2-D and 3-D graphs showing the relationships to performance.

The second use of a model is to predict the performance in cases that were not explicitly tested. This was shown for three different cases, in which one parameter was varied both more finely and more broadly, testing interpolation and extrapolation. A numerical test of accuracy of prediction was given, based on standard measures such as the Theil U-statistic.

The third use of a model, not explicitly demonstrated here, is to find algorithmic or implementation bugs. The first graphs of `duration` versus performance, in the one-dimensional slice, showed *improved* performance when the duration of actions was 0.005 seconds, compared to the performance when it was 0.01. This anomalous result lead me to try a number of different durations between 0.01 and 0.001. I found that the performance leveled off below about 0.005, but was very variable, as illustrated by confidence limits added to the plots. On looking at the output of the runs, I discovered that the communication delays between the processes implementing the agent and the environment were providing the agent with more time than it should have received. This type of problem is often immediately obvious when the results of an experiment are shown graphically, but it can be very hard to foresee and test for every such case. There is something to be said, therefore, for some use of these modeling techniques as a development tool.

5.2.3 Influence of domain parameters on Plexus performance relative to optimal

The above analysis provided a way of predicting the absolute performance of the Plexus planner for the RN1 problems, but the absolute performance is not a very good indicator of "how well" the planner is doing — is it worth looking at other planners, or does Plexus do as well as one can hope for? Is it worth trying to improve the performance of Plexus on these domains, for example by considering modifications to some of the parameters of the planner itself? To answer these questions, we look at the ratio of the performance of Plexus to the expected value of using an optimal policy. This is equivalent to the performance of a planner that uses an optimal policy computed off-line.

Principal influences

An analysis of variance shows that `duration` has the most significant effect on performance, followed closely by `width` and distantly by `success`. This is to be expected, since the optimal performance is not dependent on `duration`, but as we saw above, Plexus' performance is strongly affected by `duration`.

General form

Figure 5.9 shows the ratio of Plexus' performance to the optimal performance as a function of `width` and `duration`. For all but the shortest durations, Plexus performs very close to optimally; on average 16% worse than optimal. For a duration of 0.1, the ratio climbs quickly; the relationship is not clear from so few data points. The only interesting case here is the short duration; Figure 5.10 shows the ratio of Plexus' performance to the optimal performance as a function of `success` and `width` when `duration` is 0.1. The form is basically the same as that of the absolute performance; Plexus works best with extreme values of the success probability, and better with a probability of 1 than 0.

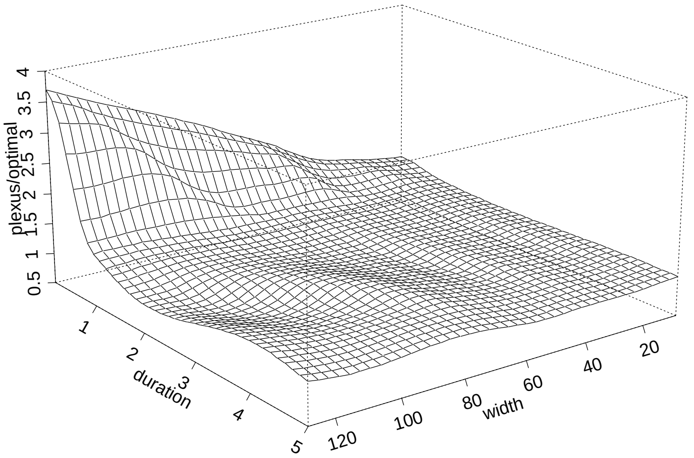


Figure 5.9: Ratio of Plexus performance to optimal, as a function of `width` and `duration`

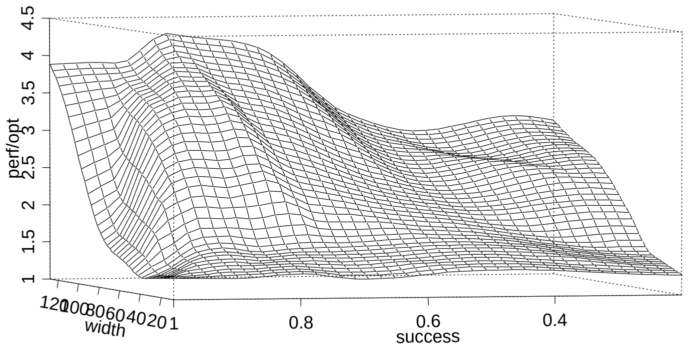


Figure 5.10: Ratio of Plexus performance to optimal, as a function of `width` and `success`

Detailed relationships

To look more closely at the relationships between `width` and `success` and performance, we consider the one-dimensional datasets described above. Figure 5.11 shows the ratio of the performance of the Plexus planner to the optimal performance as a function of the three domain parameters. The optimal performance is not dependent on the duration of the actions, so Plexus over optimal is also linear in `duration` raised to the power -1.5. This is clearly the greatest contributor to a decrease in the relative performance. The relationship to `width` (for short durations), however, is linear; the performance of Plexus on these domains was quadratic in `size`, while optimal performance is linear in `size`. Finally, `success` does not affect the ratio very greatly (though the variation is statistically highly significant) compared to the other two parameters, and it is definitely not monotonic; the best performance occurs at the extremes of this probability value — this is to be expected, since the domain is then deterministic, and computing the optimal policy is relatively easy.

Summary

The ratio to optimal performance is mainly useful for looking at general trends of a planner's performance; here we see that the performance of Plexus degrades rapidly as the duration of the actions gets shorter. Plexus performs progressively worse (relative to the optimal performance) as `size` increases; this almost

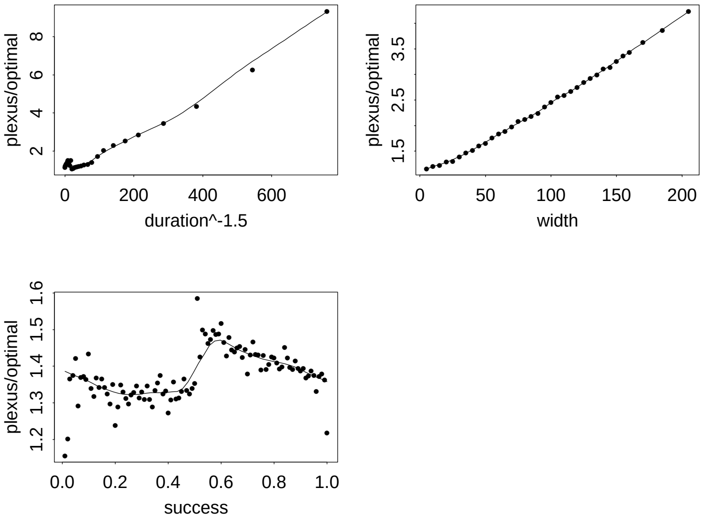


Figure 5.11: Ratio of Plexus performance to optimal, as each of the three parameters is varied, keeping the other two fixed.

certainly indicates that either the planner parameters are poorly adjusted, or `duration` was so short that most of the execution time was spent doing a random walk. Further analyses of the kind shown here would determine which case holds.

5.2.4 Influence of domain parameters on Plexus performance relative to other planners

If several planners are available, it is useful to know which ones perform best under any given circumstances. In this section I compare the Plexus planner (plexus), a planner using real-time dynamic programming based on Barto *et al* [BBS93] (rtdp), the expected performance of a pre-computed optimal policy (optimal), and the expected performance of a planner that executes random actions while it computes an optimal policy, and then executes the optimal policy (cautious).

Figure 5.12 shows the performance of the four different planners. In the first graph, domains were grouped by `width`, and mean performances plotted within each group. The second and third graphs show performance as a function of `duration` and `success` respectively. As the graphs show, the Plexus planner is the best one to use for most of the RN1 domains. However, for some small domains, RTDP does perform better (statistically significantly so, but not by a very large amount — 25% at most in this experiment). Figure 5.13 shows the 70 principal RN1 experiments described earlier. For each one, the performances of

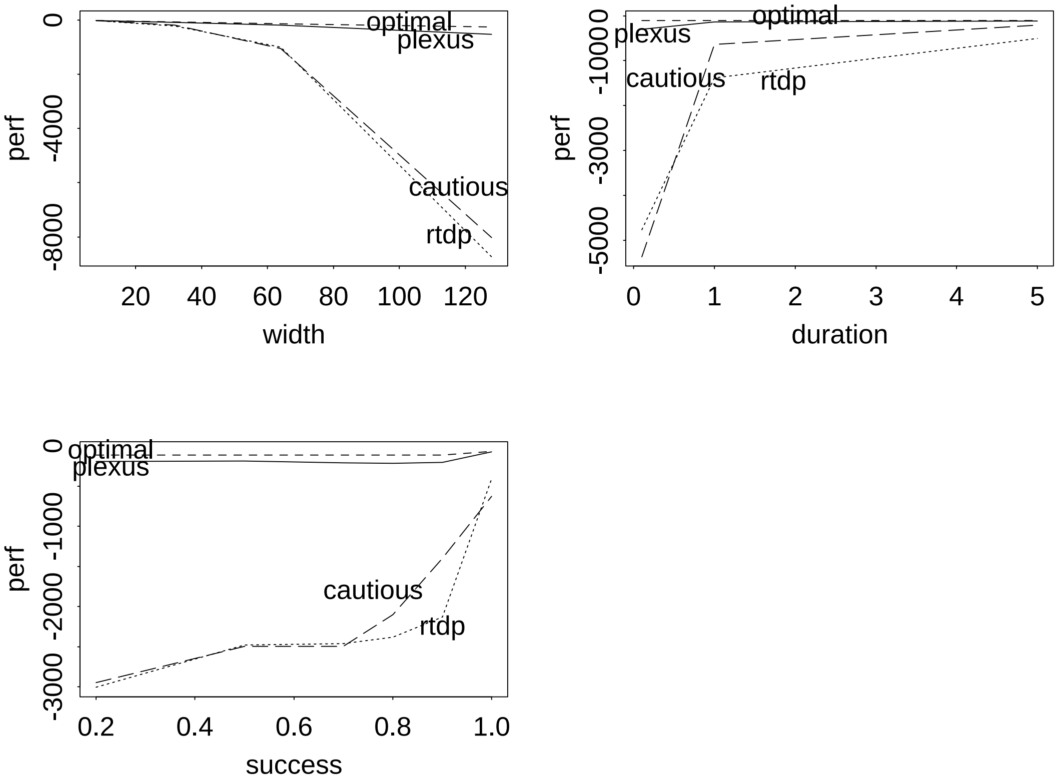


Figure 5.12: Comparison of the performance of four different planners, averaged over each of the three domain parameters

Plexus and those of RTDP were compared using a Welch modified two-sample t-test [Asp49], at the 0.05 significance level. Those for which RTDP performed significantly better are marked with a star; those for which Plexus performed significantly better are marked with a –; the three domains for which neither was significantly better than the other have been omitted. The points have been jittered to show the six different points for each `width`/`duration` combination, corresponding to the six values of `success`. As we will see in the other two domain types, different planners work well under different conditions; for example, RTDP works quite well in deterministic domains; Plexus does not. A statistical comparison of performance of the type shown above can serve to determine which conditions favour a particular planner, or to associate a collection of problems with the planner the most appropriate for each. Exactly the same techniques can be used to compare the same planner running with different parameters. This provides a disciplined method of choosing parameters, rather than the usual

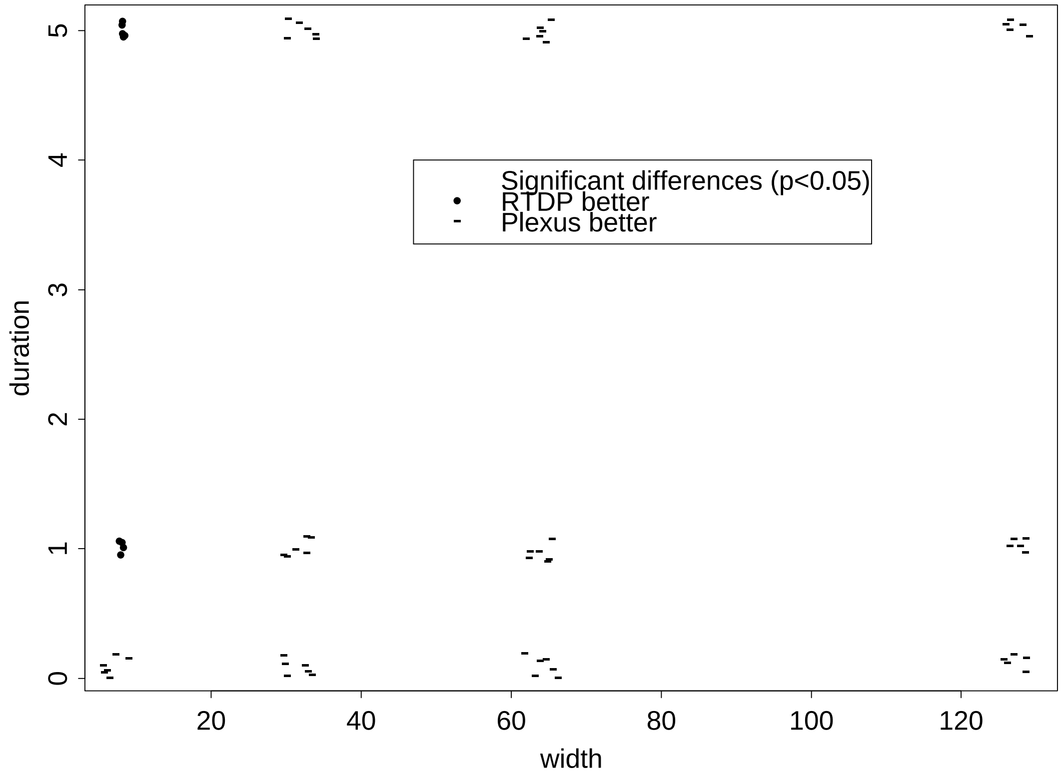


Figure 5.13: Domains for which Plexus performs significantly better than RTDP, and *vice versa*.

approach of trying a few settings and choosing the best.

5.2.5 Influence of attributes on absolute Plexus performance

In the RN1 domains, two of the domain parameters are directly related to attributes of obvious importance: the size of the problem and the rate of change of the stochastic process. The results for the parameters therefore carry over directly to these attributes; `size` of the domain (the number of states) is the square of `width` parameter, and `volatility` of the domain (the number of steps per second) is the inverse of `duration` parameter. Our earlier observation that the performance of Plexus was approximately linear in the square of the `width` and in `duration` raised to the power -1.5 can be rephrased: the performance is approximately linear in `size` of the problem and in `volatility` raised to the power 1.5. Obviously the difference between attributes and parameters is simply one of representation in this case, but attributes are less idiosyncratic, and provide a common denominator for aspects of a problem such as `size` and rate of change. Figure 5.14 shows this alternative view of the performance.

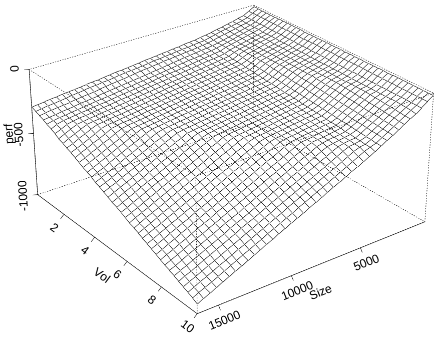


Figure 5.14: Performance as a function of `size` and `volatility`

`controllability` is a measure of the influence the agent has on the transitions by choosing different actions. As it happens, in this set of domains, `controllability` is almost perfectly co-linear with the `success` parameter, so it affords no significant advantage. Other attributes, such as the entropy, provide alternative perspectives, but no additional modeling power. Figure 5.15 shows the performance as a function of `volatility` and `controllability` for each of the four `sizes`; the shape of the graphs is different from Figure 5.3 because `volatility` axis is inverted with respect to `duration` axis in that figure.

Summary

In this particular case, the choice of domain parameters versus attributes is purely a matter of taste — the two are almost exactly equivalent. Attributes are in general a better choice, since they are more easily comparable between different types of domains.

5.3 Robot Navigation Domains (RN2)

The RN2 domains are described in detail in Appendix B.3.2. Briefly, a robot navigates through an office-like environment consisting of corridors and offices. The smallest worlds represent one floor of the Brown Computer Science department; the larger worlds correspond to several smaller worlds pieced together. The

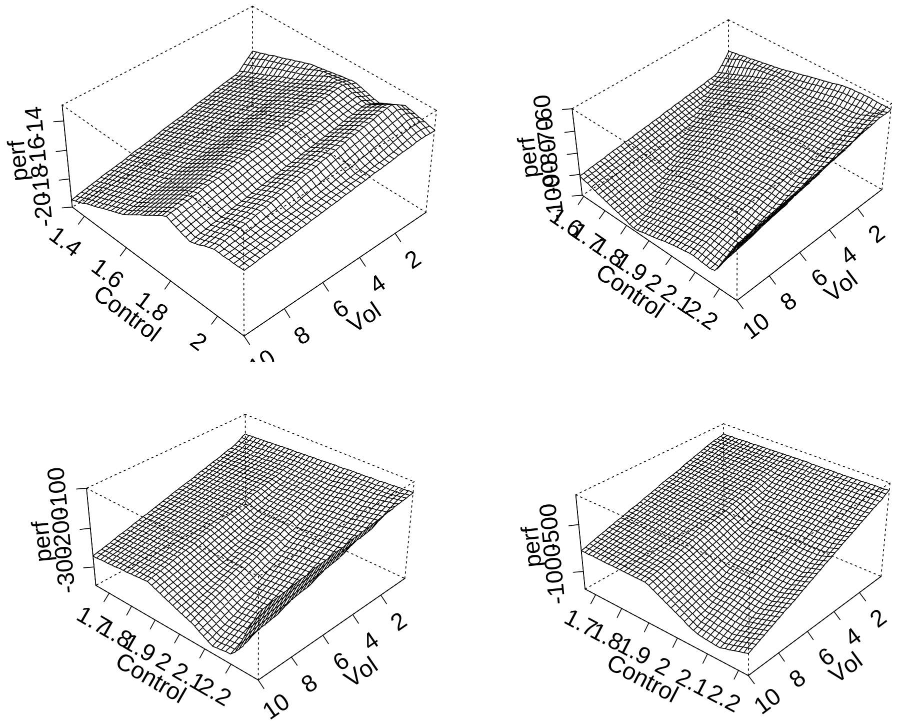


Figure 5.15: Performance as a function of `volatility` and `controllability`, for each of the four `sizes`

robot's movements are not deterministic; the success rate determines the likelihood that it will execute an action correctly. The environment may also contain "sinks", states that the robot can easily get into, but has difficulty getting out of.

5.3.1 Domains

There were 192 different domains, varying in four parameters.

- Width. The smallest world is the one described in Section B.3.2, consisting of 632 states. The other three widths were 2x2 (2528 states), 3x3 (5688 states) and 4x4 (10112 states).
- Action duration. There were three durations: 0.1, 0.3 and 1 second.
- Success. At one extreme of this dimension the outcome of an action is always deterministic; at the other extreme every movement is likely to overshoot or fail or twist the robot sideways. There were four levels of determinism, denoted 1, 0.9, 0.8 and 0.5.

- Sinks. At one extreme, a majority of the states were turned into "sinks", states that the robot can get into easily, but out of with great difficulty. At the other extreme, there were no sinks, and transition probabilities are determined as described in Section B.3.2. There were four levels of regularity, 0 (no sinks) through 3 (many sinks).

The task in each case was to reach a specified location (middle bottom) from an initial location (top left). The start and goal locations were chosen at opposite ends of the floor plan; in larger worlds they were always placed as far apart as possible.

5.3.2 Influence of domain parameters on absolute Plexus performance

Principal influences

Figure 5.16 shows the ranges of the means by each attribute. The analysis of variance of the dependence of performance on the parameters is given below.

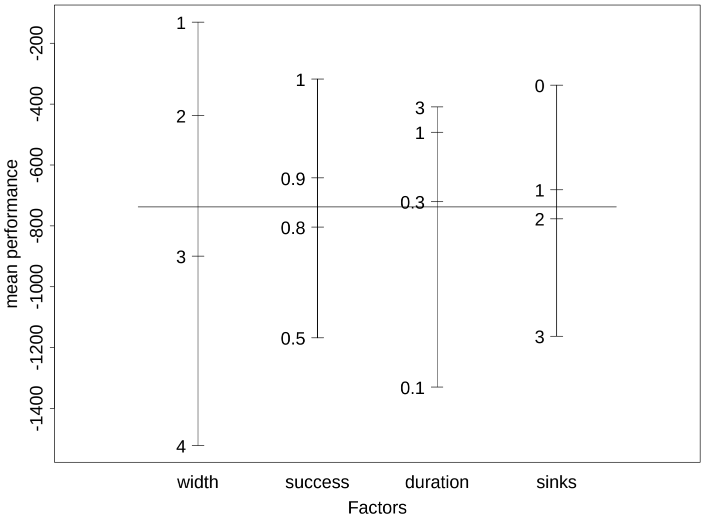


Figure 5.16: Variation in performance as a function of RN2 parameters

```
                              Df Sum of Sq Mean Sq  F Value     Pr(F)
width                         1  63709830 63709830 294.1581 0.0000000
sinks                         1  22619250 22619250 104.4366 0.0000000
success                       1  20092518 20092518  92.7703 0.0000000
duration                      1  14854752 14854752  68.5867 0.0000000
```

```
width:duration                 1  10200495 10200495  47.0973 0.0000000
width:sinks                    1   7851681  7851681  36.2524 0.0000000
width:success                  1   7686304  7686304  35.4888 0.0000000
success:sinks                  1   4144051  4144051  19.1337 0.0000182
duration:sinks                 1   2177747  2177747  10.0550 0.0017189
success:duration               1   1949494  1949494   9.0011 0.0029850
width:success:sinks            1   1516239  1516239   7.0007 0.0086909
width:success:duration         1   1258630  1258630   5.8113 0.0166827
width:duration:sinks           1   1067225  1067225   4.9275 0.0273749
success:duration:sinks         1    570761   570761   2.6353 0.1058366
width:success:duration:sinks   1    117518   117518   0.5426 0.4620835
```

The F value, or variance ratio, gives an indication of the strength of the influence of a variable (or interaction between variables) on the performance. The last column gives the probability that the effect is *not* statistically significant. Since our data usually consists of many repetitions of each experiment, almost all effects are statistically significant — our interest is in the relative effects. Clearly, `width` has the largest effect on performance; `success` and `sinks` both have strong effects, and `duration` has considerably less. There are some interactions with `width`. One other point to note is that there is not very much interaction between `duration` and the other parameters; this type of observation is useful for reducing the dimensionality of the space to be examined.

General form

Figure 5.17 shows the performance of the Plexus planner as a function of `width` and `success`. `Width` influences the performance more than linearly, and, `success` influences it somewhat more sharply as it approaches 1. This is quite different from the behaviour in the RN1 domains, where `success` had little effect on performance, and the effect was decidedly non-monotonic. There are several reasons for this: Many of the RN2 worlds have sinks, and the probability of falling into sinks goes up dramatically as the robot's movement becomes less predictable. The failures in the RN1 domains were often useful to the agent, whereas failures in the RN2 domains can turn the robot sideways, or move it backwards, which penalizes the agent. If one looks at the smaller domains, the relationship between `success` and performance appears quite similar to the one in the RN1 domains. Figure 5.18 shows, for each of the four `widths`, performance as a function of `success` and `sinks`. The general form is the same for all `widths`, except that the dimple at `sinks` = 1 and `success` = 0.9 seems more prominent with larger widths. Sinks clearly make the performance worse, but it is not clear whether a value of 2 or 3 is worse for this parameter. In terms of `success`, the deterministic case is always the best, but the form otherwise varies somewhat

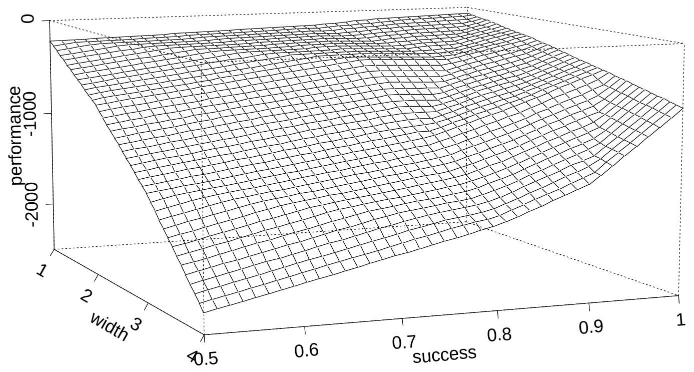


Figure 5.17: Plexus performance on RN2 worlds as a function of `width` and `success`

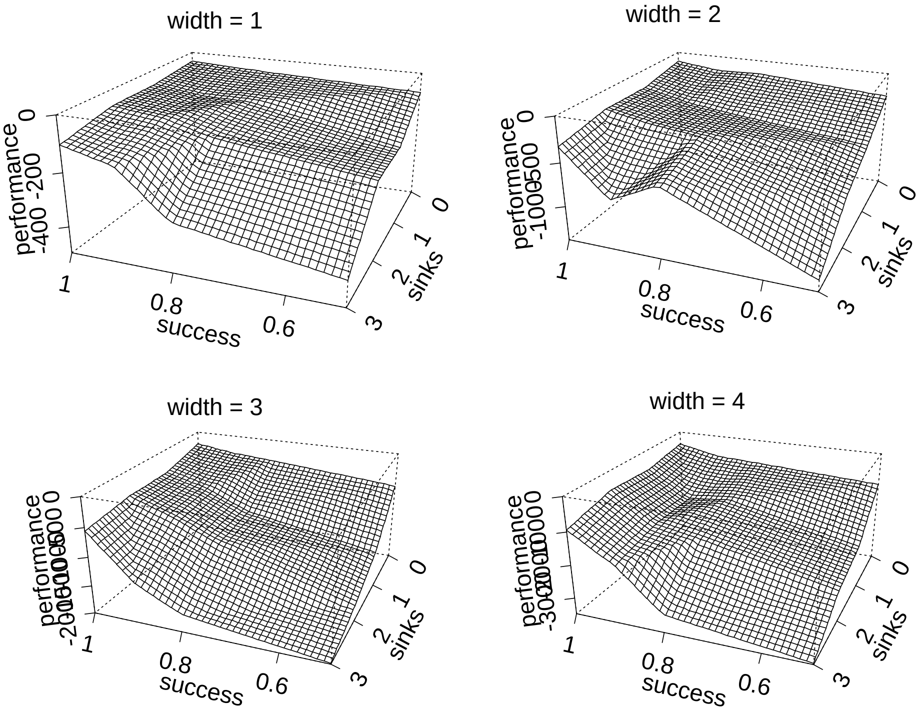


Figure 5.18: Plexus performance as a function of `success` and `sinks`, for each of the four different `widths`.

from case to case. Generally, the performance is approximately linear in `success` and `sinks`, linear in `width` raised to the power 1.5, and linear in `duration` raised to the power -0.75.

Since the width for both the RN2 and RN1 domains correspond to the same notion (diameter of a physical space), and the durations are measured in seconds, it is reasonable to compare the two. We see that the RN2 domains suffer much less from decreased duration; closer examination shows this to be especially true for low values of `success` and high values of `sinks`. The reason for this difference in the effect of `duration` is that with low success probability and many sinks, the agent can only move through the state space slowly, giving it time to evaluate better policies. In the deterministic, no-sink case, the agent moves through the space quickly, giving it less time to look ahead.

Statistical model

I consider two models of the performance. The first is a simple linear model incorporating `width` to the power 1.5, `duration` to the power -0.75, `success` and `sinks`:

$$\texttt{performance} \sim \texttt{width}^{1.5} + \texttt{duration}^{-0.75} + \texttt{success} + \texttt{sinks} \ .$$

The second incorporates a polynomial of degree 3 for both `success` and `sinks`, and models interactions between the variables:

$$\texttt{performance} \sim \texttt{width}^{1.5} * \texttt{duration}^{-0.75} * \texttt{poly}(\texttt{success}, 3) * \texttt{poly}(\texttt{sinks}, 3) \ .$$

The coefficients of the polynomials are chosen to minimize the error between the predicted and actual values, in the same way as the coefficients of the linear relationships.

Figure 5.19 shows the actual performance, simple model and complex model predictions for the average performance over domains of a given `width` and `success`. Figure 5.20 shows the actual and predicted average performances as a function of `success` and `sinks`, for the domains of `width` 2. Figure 5.21 shows the actual and predicted average performances as a function of `sinks` and `duration`, for the domains of `width` 2 and `success` 0.8. The three figures show data that is averaged over progressively fewer domains; in the first figure, each point in the 3-D plot corresponds to the average of 16 domains; in the second figure, the average of 4 domains, and in the third figure each point corresponds to just one domain. This makes the general form of the models closest to the actual data in the first figure, and least close in the third figure, for two reasons. Firstly, there are fewer experiments for each point in the last figure (the number of times the planner was run) and so deviations due to noise are more evident, distorting the

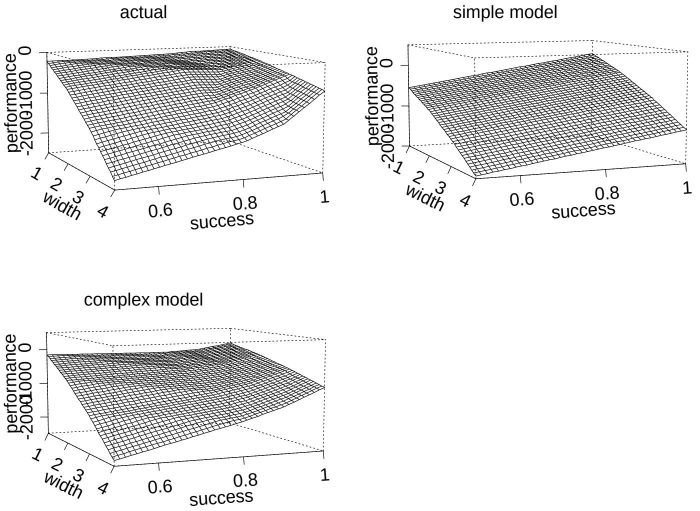


Figure 5.19: Performance of Plexus as a function of `width` and `success`.

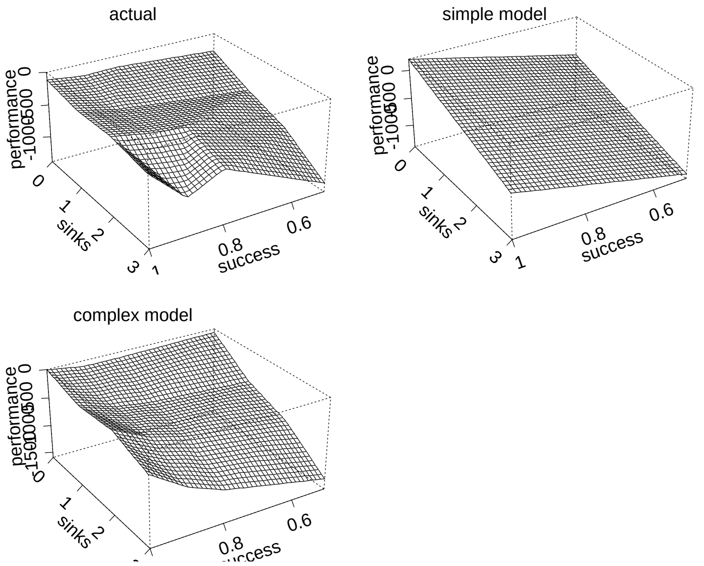
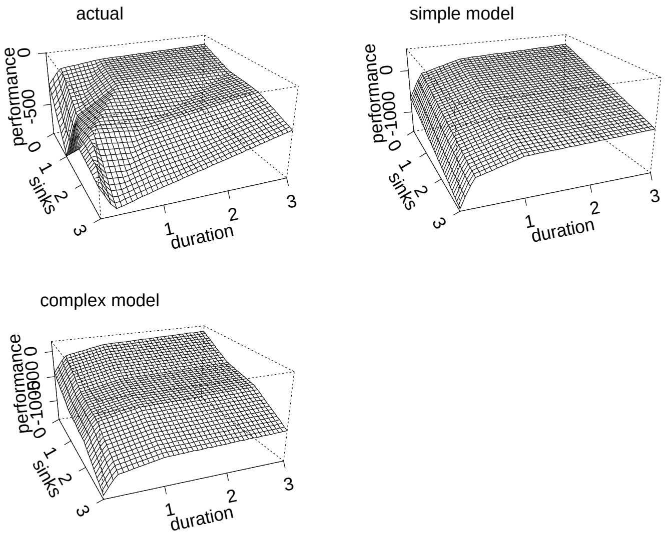


Figure 5.20: Performance of Plexus as a function of `success` and `sinks`, for worlds of `width` 2.


actual

simple model

complex model

Figure 5.21: Performance of Plexus as a function of `sinks` and `duration`, for worlds of `width` 2 with `success` 0.8.

"actual" graph. Secondly, the model minimizes error over the entire set of domains, so generally provides a less good fit for any particular small set of domains than for any particular large set.

Generally speaking, the simple model is considerably less effective at capturing the overall form of the performance, but the numerical errors of the two are comparable, when tested on a second data set with the same domains. The error from the simple model was 0.95 (errors nearly as large as the values being estimated); that of the second model was 0.79. The error here is much worse than for the RN1 domains; this is to be expected, since the number of variables is larger and the problem is much less regular.

Detailed relationships

The influence on performance of `success` and `sinks` parameters was not very clear from the initial data, so I ran two experiments, one with a set of domains varying by success, and one with a set of domains varying by sink level. Figure 5.22 shows the performance of Plexus on twenty domains with `duration` 0.1 seconds, `width` 2 (four copies of the floor plan joined in a 2 by 2 square), sink level 1 (5% of the states are deep sinks, taking an average of 100 actions to escape from them, and 10% of the states are shallow sinks, taking an average of 10 actions to escape from them). The empirical mean performances are plotted as points on the solid line; 95% confidence intervals for the means of these performances

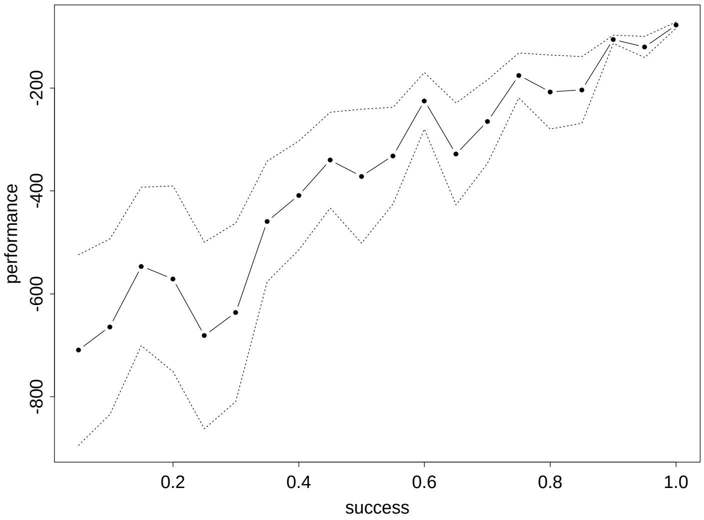


Figure 5.22: Plexus performance as a function of `success` parameter

are shown in dotted lines. The number of runs for each different domain was the same (40, in this case); the confidence intervals are wider for low probabilities because the variability of the performance is much greater for low probabilities. The Pearson correlation coefficient for the best linear fit is 0.92, indicating that a linear approximation is quite good.

Figure 5.23 shows the performance as a function of the number of sinks. The figure on the right is just an enlarged version of the plot corresponding to the range of values used in the original (4-dimensional) experiments.

The sink parameter determines the number of sinks in the domain. At level 0, there are no sinks at all. At level 1, about one in 30 of the locations[2] in the world are "deep" sinks, taking an average of 100 steps to move out of the location, and twice as many locations are "shallow" sinks, taking an average of 10 steps to escape. The number of locations that are sinks is linear in the parameter. Sinks are chosen from a randomized list of locations, but the same randomized list is used for all domains, so increasing the sink level simply adds more sinks. The optimal performance therefore decreases steadily as the sinks increase.

The performance of Plexus is plotted in Figure 5.23 as a function of `sinks` parameter; the relationship is roughly cubic overall, but for the range of sink levels used in the initial experiments (0 to 3), shown on the right, the performance degrades more slowly, between linear and quadratic in the number of sinks. In absolute terms, the sinks have quite a mild effect on performance for low values

---

[2] In the RN2 worlds, a state consists of a location and a direction (N, E, S or W).

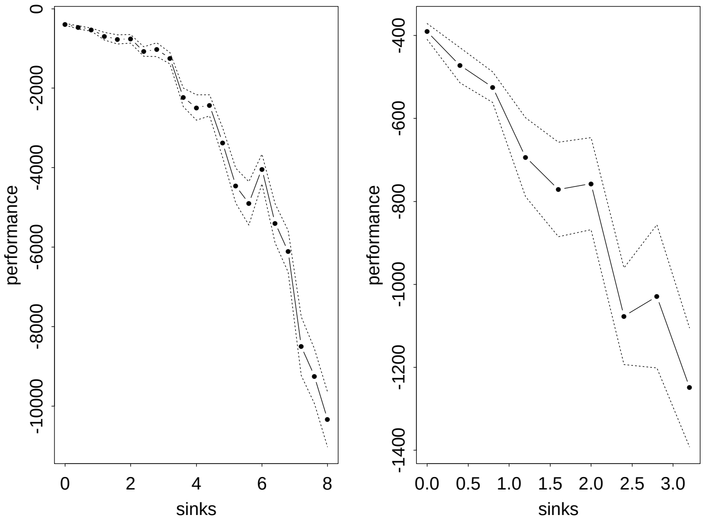


Figure 5.23: Plexus performance as a function of `sinks` parameter.

of `sinks` parameter, but as the density of sinks gets higher, the performance degrades more and more rapidly. The connectivity of the location map is not very high, particularly between replicas in the larger domains — there is one passage leading from each replica in each direction, so sinks can easily block connections between replicas, requiring the agent to make long detours or to go through sinks (which decrease the performance by an average of 10 or 100 for shallow and deep sinks). Interestingly, the improvement in performance when the number of sinks increases, around a value of 6, is highly statistically significant, indicating that the local maximum is not due to noise. Examination of some of the post-mortems of planner execution did not reveal any obvious reason for this reversal in trend, though the fact that the optimal performance slows its decrease at this point (see Figure 5.24) contributes to it.

The two models of performance suggested earlier (Section 5.3.2) can be used to predict the performance over these new sets of domains. In the case of `success` parameter, the model is being used to interpolate new points that lie between ones in the original data. In the case of `sinks` parameter, the model must both interpolate and extrapolate to points outside the convex hull of the region formed by the original data.

Recall that both models (simple and complex) are linear in $\texttt{width}^{1.5}$ and $\texttt{duration}^{-0.75}$; the complex model also includes an interaction term for these two. The simple model is linear in `success` and `sinks` parameter; the complex model is linear in a third-degree polynomial of `success` parameter, and a third-degree polynomial of `sinks` parameter. Figure 5.25 shows the actual and

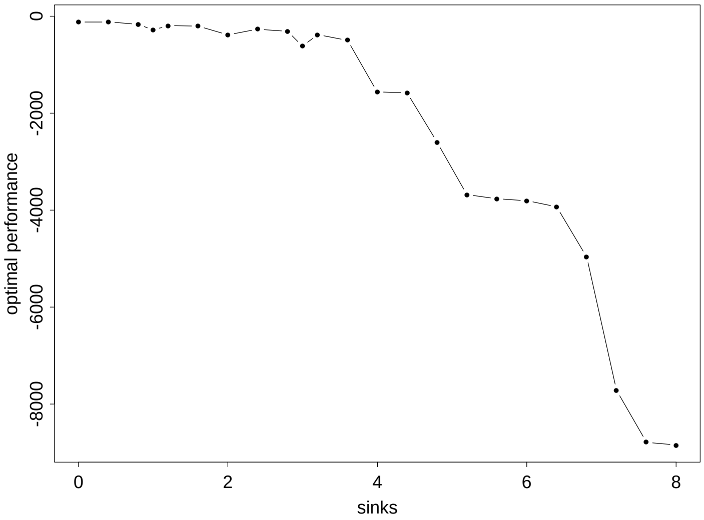


Figure 5.24: Optimal performance as a function of `sinks` parameter.

predicted performance of Plexus on the new domains, varying `success` parameter (the revised model is presented below). The complex model has clearly wildly over-fitted the data, whereas the simple model has considerable constant error. This is borne out by the quantitative predictive error measure described earlier; it is 0.74 for the simple model (on average the error is 74% of the value) and 0.77 for the complex model. It is interesting to note that Figure 5.25 and the error measure give quite different impressions of the usefulness of the models: from the graph, it would appear that the simple model is much better than the complex model, whereas the error measures show little difference. This is because the error measure emphasizes big relative differences, while the graph highlights big absolute differences; these can be measured numerically by the mean absolute error (see Section 5.1.4).

Figure 5.26 shows the actual and predicted performance as a function of `sinks` parameter. Again, the complex model is wildly off in extrapolating; the simple model is also quite far from the actual values.

I used the observations of relationship between performance and `success` and `sinks` to make a third model, labeled "revised" in the figure. The model uses the same exponents for `width` and `duration` as the other two models, but is linear in `success` parameter and in the cube of the `width` parameter. All interactions are included. The model coefficients were determined using the original data — the same data used to determine the coefficients of the simple and complex models. The error of the revised model on the new `success` domains is 0.65 better than either of the other models, but not strikingly so. On the new `sinks`

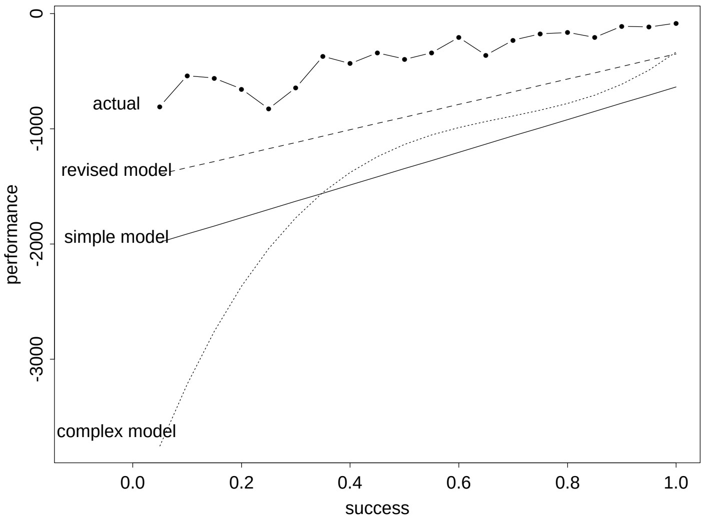


Figure 5.25: Actual and predicted Plexus performance as a function of `success` parameter.

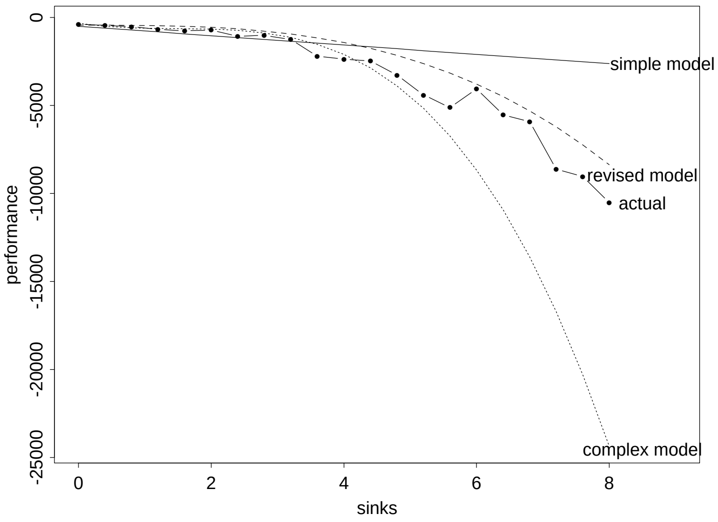


Figure 5.26: Actual and predicted Plexus performance as a function of `sinks` parameter.

domains, the error of the revised model is 0.5, much better than the simple model (1.0), but actually worse than the complex model (0.43). Here the best choice of model will depend on whether absolute or relative error is more important.

Summary

Plexus' performance is, on average, between linear and quadratic in the `width` of the domain and somewhat less than linear (*i.e.,* better than linear) in the inverse of `duration`. In this set of domains, there are significant interactions between `success` and `sinks` parameters and the two principal parameters, `width` and `duration`. Such interactions can result in misleading models if one is not careful to examine sets of domains in which parameters vary one by one, rather than averaging over entire dimensions. This will be all the more true for domains with many different parameters. We found that different models were more appropriate depending on whether it is important to minimize the absolute error or the relative error (ratio of predicted to actual values). I show graphs that illustrate absolute error and quantitative measures of the relative error, but of course one can plot the ratio of predicted to observed performance, and the mean absolute error (see Section 5.1.4) can be used as a quantitative measure of absolute predictive ability.

5.3.3 Influence of domain parameters on Plexus performance relative to optimal

Principal influences

An analysis of variance shows that the principal influence is due to the `duration`; as in the RN1 case, `duration` has more influence on the ratio of Plexus performance to optimal than on the absolute performance, since `duration` does not affect the optimal performance. `width` is the next most important factor, followed by `sinks` and `success`. The `width` interacts quite strongly with `duration` and `sinks`.

General form

Figure 5.27 shows the ratio of Plexus' performance to the optimal performance, as a function of `duration` and `width`. For all but the shortest durations, the performance is close to optimal; the ratio increases faster than linearly in `width`, and the relationship to `width` is not qualitatively affected by `duration`.

Looking at the shortest duration case only, Figure 5.28 shows the ratio as a function of `sinks` and `width`. The performance is roughly linear in the `width`; this is similar to the RN1 case. The performance is worst for extreme values of

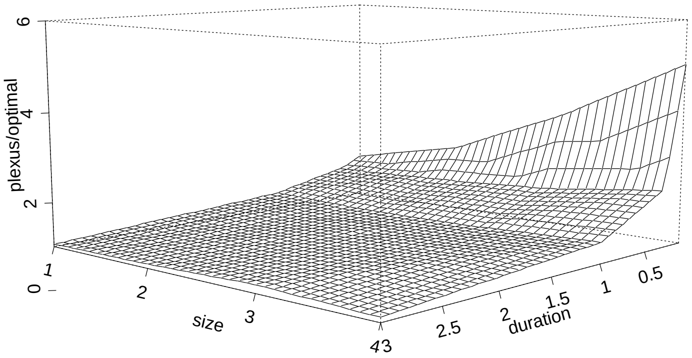


Figure 5.27: Ratio of Plexus performance to optimal performance, as a function of `duration` and `width`.

`sinks` — when there are no sinks, this is to be expected, since the optimal policy reaches the goal very quickly, before the Plexus planner has been able to amortize its initial search cost. However, the performance relative to optimal should not be high when there are many sinks. This is probably due to a problem I have observed with the planning strategy that makes it overly adventurous when the cost of a bad move may be very high.

When moving from value computation over the entire state space to value computation over an envelope, I base the approximation on several assumptions. One of these assumptions is that the values of states near the fringe of the envelope change fairly continuously, in particular that one can estimate the value of a state that is not in the envelope by using the values of its neighbours that are in the envelope. When there are many sinks, the value function becomes more chaotic; neighbouring states often have values that differ by 100 or more. Furthermore, since states with low value are pruned from the envelope, states in the envelope tend to have higher value than those in the fringe. The net consequence of this is that the planner over-estimates the values of states in the fringe, making it too adventurous.

Finally, Figure 5.29 shows, for each of the four `widths`, the ratio as a function of `success` and `sinks`. Here there are obviously some unusual interactions; the form of the relationship is quite different for the different `widths`. Careful examination of the execution traces revealed that in the small domains, certain configurations of sinks forced the agent to take paths through deep sinks,

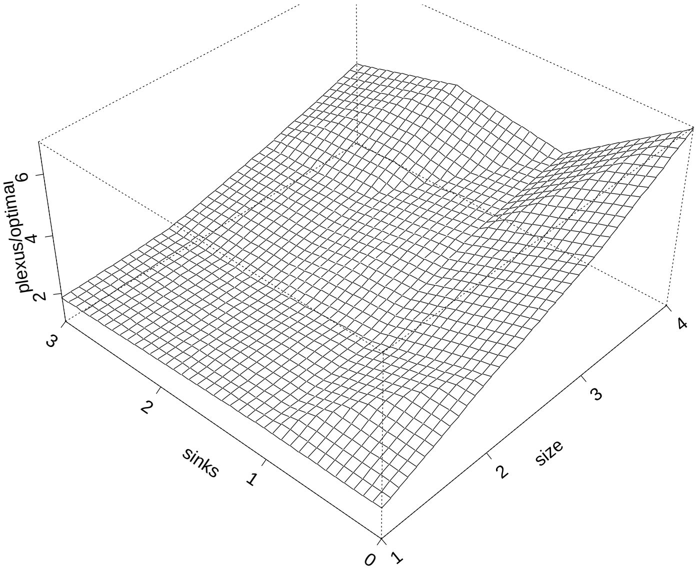


Figure 5.28: Ratio of Plexus performance to optimal performance, as a function of `width` and `sinks`.

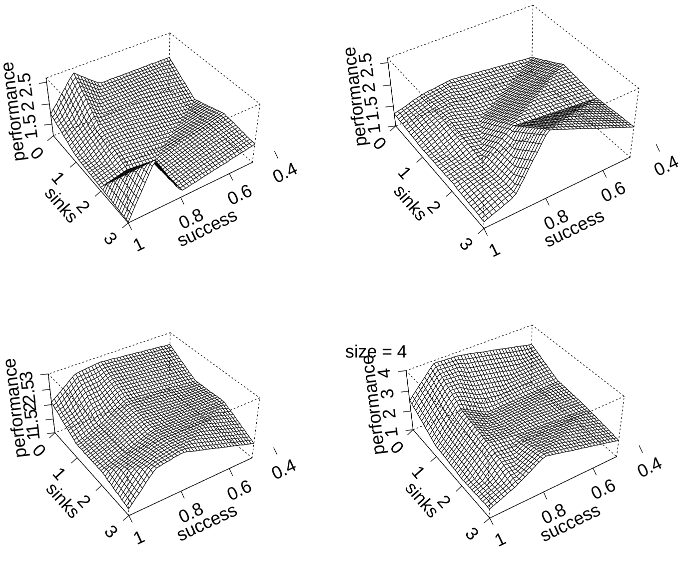


Figure 5.29: Ratio of Plexus performance to optimal performance, as a function of `sinks` and `success`, for each of the four `widths`.

whereas with the larger domains, there were enough alternative routes that it could avoid the worst ones. This kind of interaction, largely invisible when examining data averaged over more domains, shows the importance of graphically viewing different subsets of the data, as well as different projections (averaging over dimensions). This is quite tedious with high-dimensional spaces, even with sophisticated statistical analysis packages, but it can be automated to a large extent. Of course, this level of detail is only required when one is interested in a high degree of accuracy in the prediction; for approximate prediction, which is often sufficient (*e.g.,* for comparing planners), averaging over a range of domains is adequate.

Statistical model

The statistical model of the ratio of performance does not raise any particularly important points, so I omit it for brevity. A simple model (linear in `duration` to the power -0.75, `width` cubed, `success` and `sinks`) has an error of 0.45 when tested on a second data from the same domains; more complex models, including interactions and polynomials in `success` and `sinks`, do not add greatly to the accuracy.

Summary

The performance of Plexus is close to optimal for `durations` greater than 0.3 seconds, except on the largest domains, where the performance (cost) degrades to twice the optimal, even for long `durations`. A simple model is sufficient to predict the performance relative to optimal quite accurately. The performance of Plexus relative to the optimal is quite difficult to model for this set of domains; the main reason for this is that there are subtle interactions between `width` and `sinks`, related to the topology of the physical space the problem models. This points out the necessity of detailed analysis if accurate predictions or models are required. Models based on averages over several dimensions of parameters are sufficient to give predictions accurate to within about 50%; this would be sufficient for estimates of feasibility — given bounds on acceptable performance, determine ranges of parameters for which the planner might perform acceptably. To obtain more precise models, it is necessary to restrict the model to particular subsets of the space of problems, *e.g.,* by fixing some of the parameters. The exploratory analysis of the relative performance of Plexus proved a useful tool for finding obscure bugs in the implementation and in the algorithms used in Plexus.

5.3.4 Influence of domain parameters on Plexus performance relative to other planners

General form

Figures 5.30-5.33 show the performances of five planners (Plexus, recover, replan, RTDP and cautious), and the expected performance under the optimal policy. The first figure shows the average over all domains of `width` 1, 2, 3 and 4; the second shows the average over all domains with `duration` 0.1, 0.3, 1, and 3, and so forth. The left half of the figure shows all five planners, but because the cautious planner and RTDP are so much worse than the others, I show the four other planners alone on the right, so that differences are clearer. RTDP takes longer to reach the goal than even the cautious planner, though its asymptotic behaviour with respect to `size` appears to be better. The poor performance of RTDP in these domains is mainly caused by `sinks`. Because RTDP simulates actions to explore the state space, it spends inordinate amounts of time simulating the self-transitions that occur in sinks, leaving much less time to explore the rest of the space. This is aggravated by small action durations, as can be seen from Figures 5.31 and 5.33.

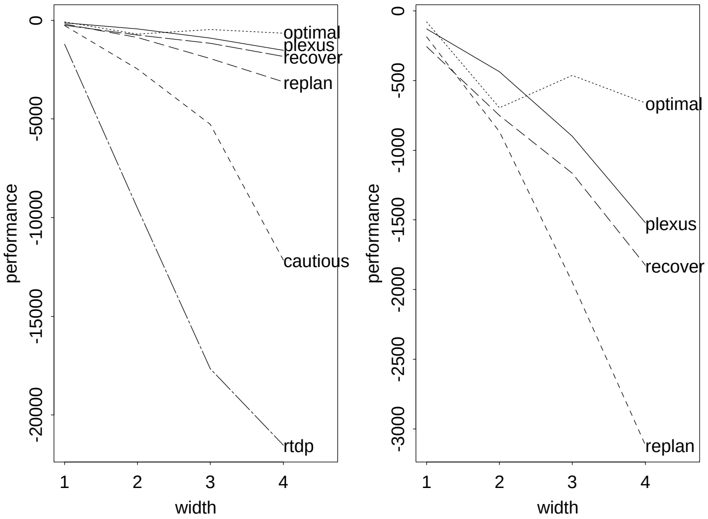


Figure 5.30: Average performances over domains of `width` 1, 2, 3 and 4, for five planners.

Cautious and RTDP are always by far the worst in absolute performance, though, interestingly, cautious improves as the success rate falls and as the sinks

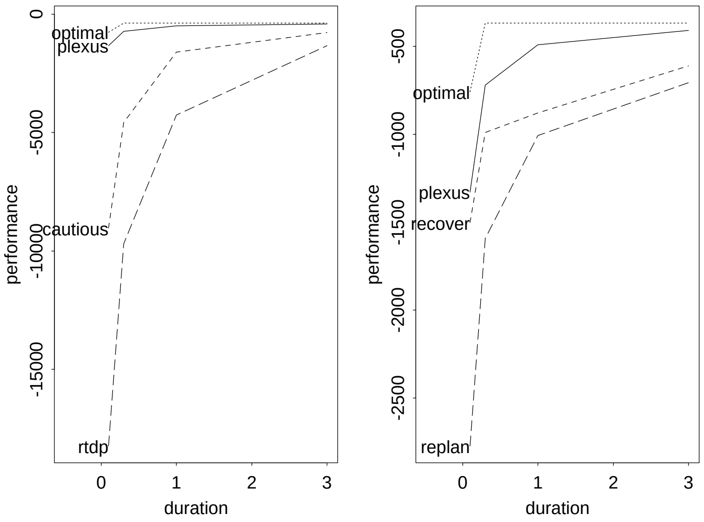


Figure 5.31: Average performances over domains with `duration` 0.1, 0.3, 1 and 3, for five planners.

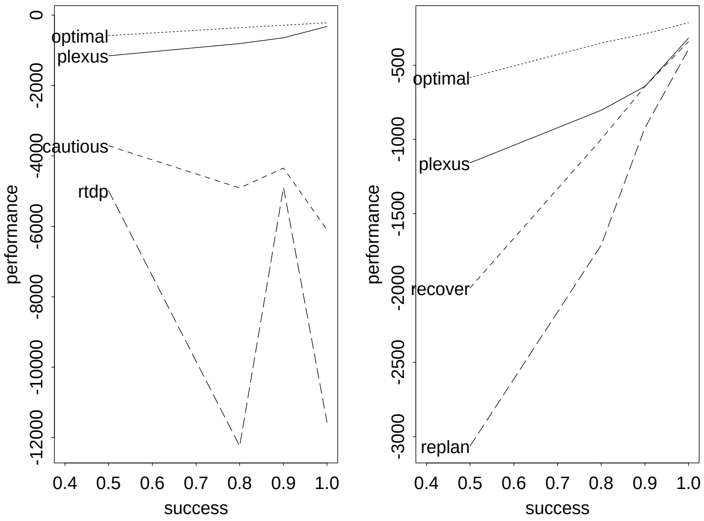


Figure 5.32: Average performances over domains with `success` parameter 0.5, 0.8, 0.9 and 1, for five planners.

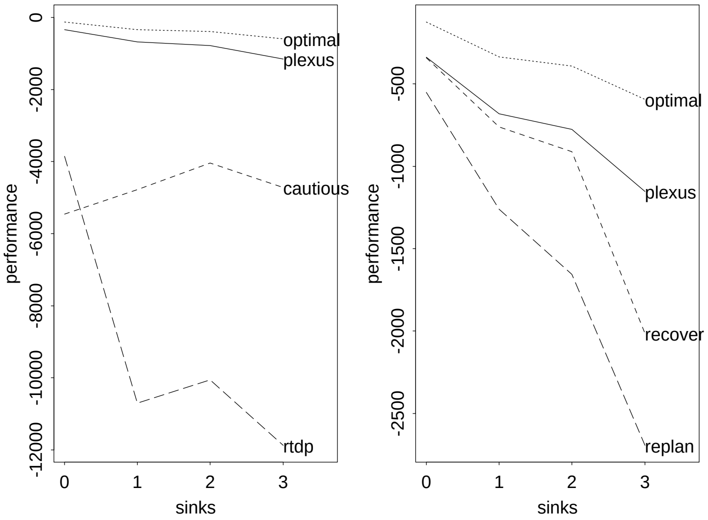


Figure 5.33: Average performances over domains with sink levels 0, 1, 2 and 3, for five planners.

get worse. This is because policy iteration can converge very slowly when transitions are deterministic, or close to deterministic. Each round of policy improvement may update as few as one action, and although value computation is quicker with a deterministic matrix, the overall cost is usually higher.

In all cases replan is worse than recover, which is to be expected, since starting a best-first search from scratch can be very expensive. Especially for the domains with small durations, the initial search time dominates the execution time for all the planners. The only case in which replan has an advantage over recover is if it falls out of the envelope, and then finds a shorter path to the goal than the original path. If we used a heuristic search technique, this would be possible, but since the best-first search finds the best path, it is always preferable to return to the path than to find a new path to the goal.

Recover works remarkably well in the RN2 domains; it is close to the performance of Plexus in many cases, and better in a few. In fact, this presentation is unfair to the recover planner; it performs better than Plexus in about a third of the domains, but not by as much. A better way of seeing this is to look at a histogram of the ratio of performances. Figure 5.3.4 shows, above, the ratio of Plexus to recover, and, below, the ratio of recover to Plexus. Recall that since performances are negative (because all instantaneous rewards are negative), the planner in the numerator of the ratio performs better when the ratio is less than 1. From this figure, one can see that in most cases, there is not much difference between Plexus and recover, but Plexus does better somewhat more often. However, recover sometimes does much worse; taking 6 times as long to reach the goal, whereas at worst, Plexus only takes 3 times as long as recover.

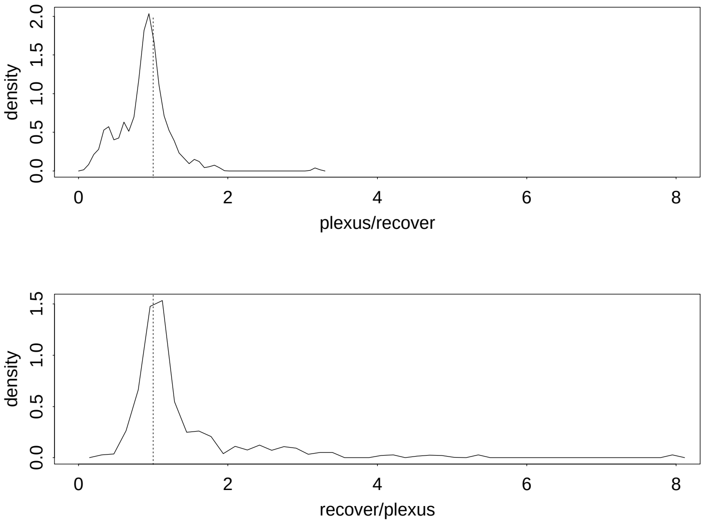


Figure 5.34: Density plots for the performance ratios plexus/recover and recover/plexus.

Another useful view of the comparative performances is shown in Figure 5.3.4, where each domain is represented by one point. The X co-ordinates correspond to the ratio of Plexus to recover performance (above) and recover to Plexus performance (below). The Y co-ordinates correspond to the optimal performance for that domain. From this figure, one can see that the cases where recover does better than Plexus are those where the optimal performance is low, *i.e.,* the "easy" cases.

Summary

In the RN2 domains, the cautious planner performs very poorly; on average about 6 times worse than Plexus, and at worst nearly a hundred times worse. Re-using an existing "plan", or envelope (recover), is almost always better than restarting the search each time the agent falls out of the envelope (replan). Recover is almost as good as Plexus, though it tends to be worse on "harder" domains, *i.e.,* those for which the optimal performance is low. The difference is not, for the most part, statistically significant; there are 3 domains on which Plexus performs significantly better at the 0.95 confidence level, and 2 domains on which recover

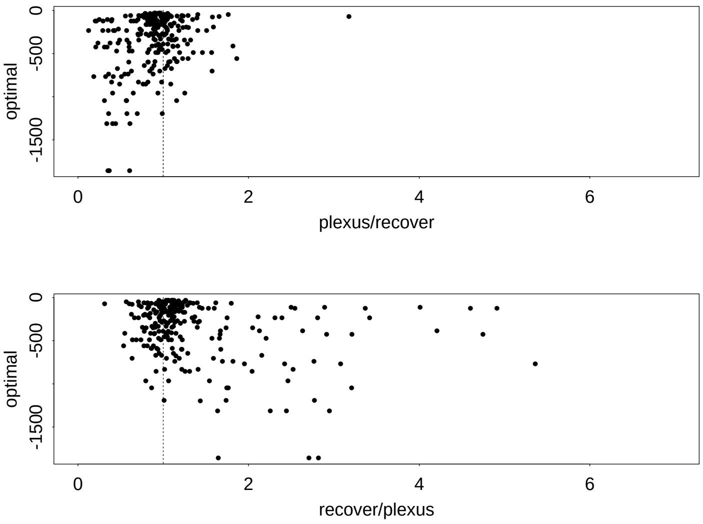


Figure 5.35: Performance ratios plexus/recover and recover/plexus, plotted against optimal performance.

performs better at that level. At the 0.9 confidence level, Plexus does better on 7 domains, recover on 6.

The important aspect of these domains that allows replan to perform so well compared to Plexus is the reversibility of actions. If the recover agent falls out of the envelope, there is always a single transition that can take it back into the envelope, so the best-first search finds a solution immediately. Actually making this transition may take some time; in domains with more sinks, it is more likely that it will — this explains why Plexus has more of an edge on recover in domains with many sinks.

This kind of reversibility is very common in navigation domains, but there are also many domains for which it does not hold. The racetrack problem presented next is an example where a single transition may require many steps to undo (to return to the state from which the transition was made). Scheduling problems and the traffic light problem discussed in the Introduction also do not have the reversibility property.

5.3.5 Influence of attributes on Plexus absolute performance

In this section, I consider the correlation between a number of attributes and the performance of Plexus on the RN2 domains; the attributes are defined in detail in Chapter 4 — I provide a brief summary here.

- Size

  The number of states in the domain.

- Volatility

  The number of transitions per second. Recall that the agent-environment system operates on a fixed-interval, discrete-time basis.

- Entropy

  The entropy of a domain is the average of the entropy of each state-action pair. The entropy of a state-action pair is a measure of the uncertainty in the outcome of the action in the state. If the entropy is high, there is much uncertainty about the state that will result from the transition. If the entropy is 0, the domain is deterministic.

- Controllability

  The controllability of a domain is the average of the controllability in each state. The controllability in a state is a measure of the difference between the expected outcomes of different actions. If the controllability is high, different actions lead to very different outcomes; if the controllability is 0, the choice of action has no effect on the outcome of the transition.

- Openness

  The openness of a domain is a measure of the ease with which the agent can move through the state space. The lowest openness occurs when every transition is a self-transition; the agent cannot move through the space at all. The highest openness would be obtained if every possible sequence of actions led to a distinct set of states.

- Distance asymmetry

  The distance between a state s_1 and a state s_2, denoted $d(s_1, s_2)$, is the expected reward accumulated by starting in s_1 until s_2 is reached, under the policy that maximizes the reward. This is generally not a symmetric measure. The distance asymmetry between s_1 and s_2 is $|\, d(s_1, s_2) - d(s_2, s_1)\,|$. The distance asymmetry of a domain is the average of the distance asymmetry over all pairs of states. A low distance asymmetry makes it easy for the agent to recover from undesirable transitions. If the distance asymmetry is high, this may be very expensive.

- Normalized distance asymmetry

  This is the distance asymmetry of each pair of states divided by the sum of the distances between them. This provides a measure that can be used to compare domains of very different sizes; the distance asymmetry is strongly affected by size.

Principal influences

An analysis of variance reveals that `size`, `volatility`, `controllability` and normalized distance asymmetry are strongly correlated with the performance; `openness` is to a much lesser degree, and the distance asymmetry very little.

General form

Figure 5.36 shows the performance of Plexus as a function of `size` and `volatility`. Of course, this graph is equivalent to the graph of performance as a function of `width` and `duration`, but in this form it is easier to compare to other domains, since the values of the attributes are comparable, whereas the values of parameters are generally not. The effect of `size` and `volatility` is very similar to that for the RN1 domains, though increased `size` here has a less deleterious effect on performance than in the RN1 domains.

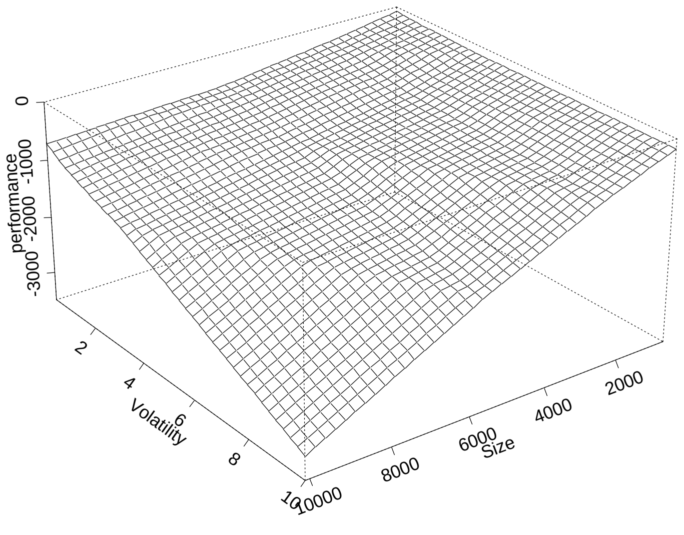


Figure 5.36: Plexus performance as a function of `size` and `volatility`

Figure 5.37 shows, for each of the four `sizes` of domain, the performance as a function of `volatility` and `controllability`. Although there are some odd interactions in the small sizes, the general shape is the same for all sizes; volatility does not affect performance very greatly unless the controllability is low, and in all cases the performance is linear in the volatility. There is a sharp decrease in performance as the controllability falls below about 1.5.

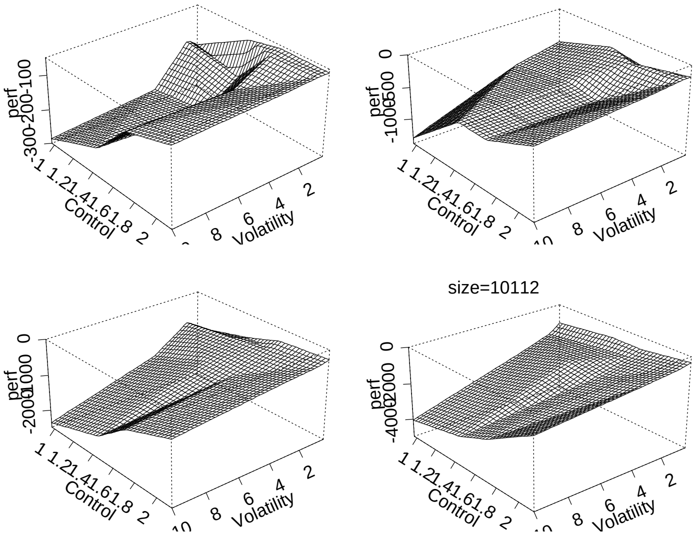


Figure 5.37: Plexus performance as a function of `volatility` and `controllability`, for four different `sizes`

Statistical models

A simple model, using `size`, `volatility` and `controllability`, with no interactions, has an error of about 1 on the second dataset (the difference between the predicted performance and the actual performance is about the same as the absolute value of the performance). A more complex model, adding the interactions, reduces the error to 0.5; this is better than a model using all four domain parameters.

5.3.6 Influence of attributes on Plexus performance relative to other planners

General form

Figures 5.3.6-5.3.6 show the performances of six planners as a function of `size`, `volatility`, `openness` and `entropy`. As `size` increases, the performance of Plexus and recover degrades linearly; optimal, replan and RTDP degrade slightly less quickly than linearly, and cautious degrades more quickly. Plexus performs better on average for all the sizes I examined.

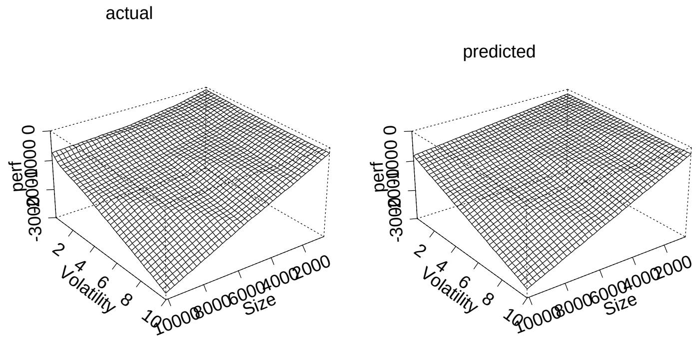


Figure 5.38: Actual and predicted Plexus performance as a function of `size` and `volatility`

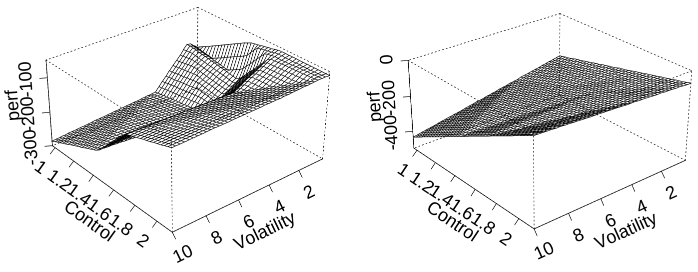


Figure 5.39: Actual and predicted Plexus performance as a function of `volatility` and `controllability`, for worlds of `size` 632

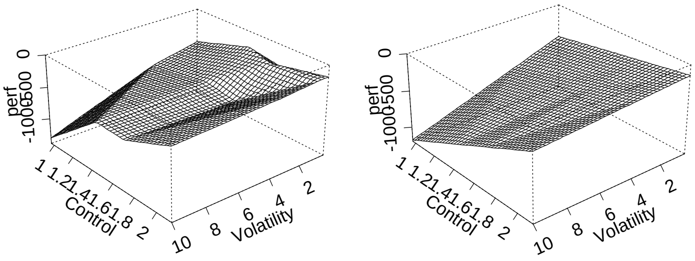


Figure 5.40: Actual and predicted Plexus performance as a function of `volatility` and `controllability`, for worlds of `size` 2528

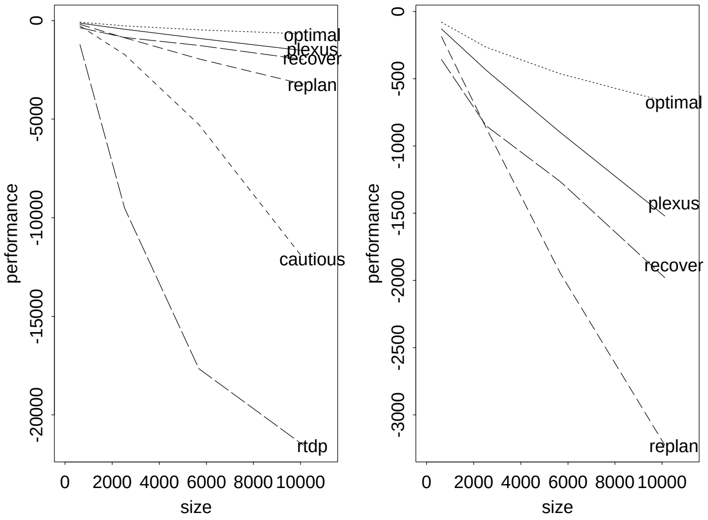


Figure 5.41: Performances of different planners as a function of `size`.

As a function of `volatility`, a similar picture emerges: the performance of Plexus and replan degrades linearly, while that of recover degrades less quickly. Again, Plexus out-performs all the other planners.

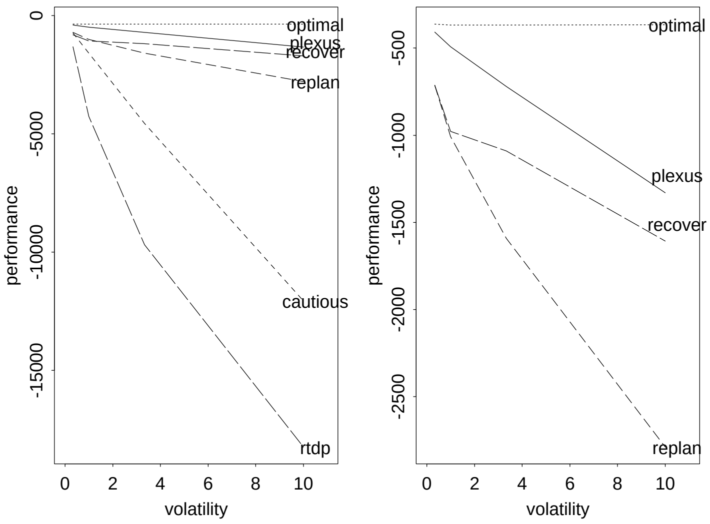


Figure 5.42: Performances of different planners as a function of `volatility`.

As `openness` increases, the performance of all the planners becomes rapidly worse; the forms are similar for all the planners. Plexus still out-performs all the other planners.

Finally, as `entropy` increases, the performance of the planners is not monotonic, but the general trend is towards worse performance. Plexus performs significantly better than the other planners for domains with high `entropy`.

We see from these graphs that as the problems become more difficult (increased `size`, decreased thinking time, increased uncertainty and more rapidly spreading uncertainty), the performance of Plexus degrades less quickly than that of recover, replan and RTDP.

In terms of individual domains, of the 252 domains, Plexus performs significantly (at the 0.05 level) better than the others in 58 cases, recover performs significantly better than Plexus in 2 cases, and replan performs significantly better than Plexus in 6 cases. The cases where the search-based planners do better have either the smallest size or the lowest success probabilities.

Summary

Looking at the relative performances of the different planners as a function of attributes gives an alternative perspective on their behaviours. As `entropy` and

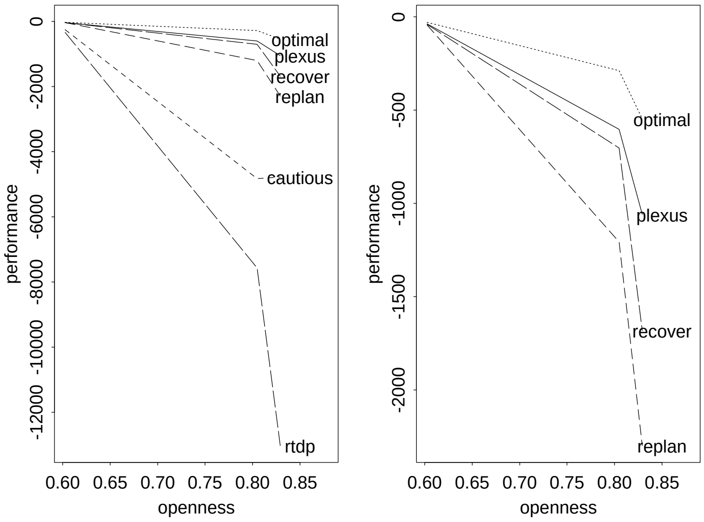


Figure 5.43: Performances of different planners as a function of `openness`.

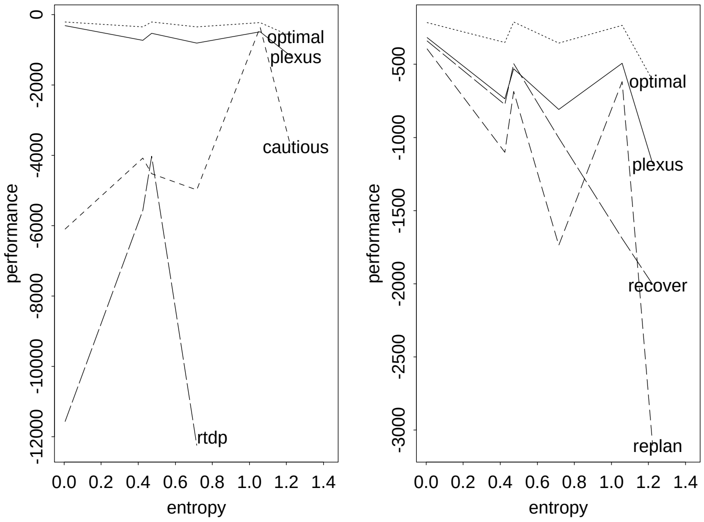


Figure 5.44: Performances of different planners as a function of `entropy`.

`openness` increase, replan and recover start to catch up with Plexus, though they do not reach Plexus' level of performance on any but a very few of the domains. More extensive analysis is necessary to determine whether this trend continues.

5.4 Racetrack

Because the racetrack problems are relatively difficult to solve (often requiring tens of thousands of transitions), I did not create additional sets of domains as I did with the RN1 and RN2 domains. Instead, to test the adequacy of the models in the face of new domains, I constructed the models based on one subset of the domains, and then tested them against the remainder of the domains.

5.4.1 Domains

I defined 181 different racetrack problems, described in detail in Section B.3.3. They varied by three parameters: the duration of an action, a success probability (the probability that the acceleration specified by the agent takes effect), and the track. There were 11 different tracks: 5 geometric (simple shapes), 5 Grand Prix (curved, single path to goal), one Rally (multiple paths to the goal, with speed/accuracy tradeoffs) and three random (50% density of track and non-track cells).

5.4.2 Influence of domain parameters on absolute Plexus performance

An analysis of variance shows that `success` is the parameter that most influences performance; `track` and `duration` are of roughly equal importance. There are significant interactions between `track` and `success`, and between `success` and `duration`.

General form

Figure 5.4.2 shows the performance of Plexus in the racetrack domains as a function of `duration` and `success` parameters, averaged over all tracks. Plexus tolerates either a low success rate or a small duration, but the two combined cause its performance to degrade seriously. Figure 5.4.2 shows the same graph, but for each individual track. Clearly, the effect of `duration` and `success` are highly dependent on the track in question. The grouping of tracks into geometric, grand prix, *etc.* in some cases correspond well to particular forms of interaction, but the match is not very good: the four geometric domains seem to correspond to two different types of interactions; `gp-21` and `gp-69` seem to be qualitatively

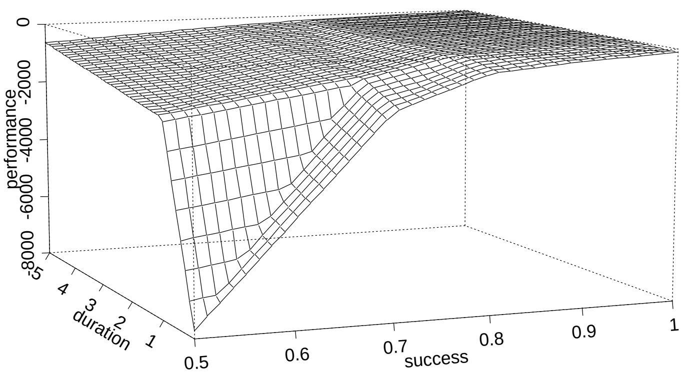


Figure 5.45: Plexus performance as a function of `success` and `duration`.

different from the other Grand Prix tracks, and the random and rally tracks have remarkably similar characteristics. Furthermore, predictions for particular tracks tells us nothing about the expected performance on other tracks, so a model based on the track will not be useful for predictions on new domains.

5.4.3 Prediction

Prediction on the basis of `success` and `duration` is not very effective; I created a model linear in `success` and the inverse of `duration` based on the performances on half of the tracks (arbitrarily chosen), and then compared predicted to actual performance on the other half of the tracks. Depending on the particular set of tracks used to create the model, the normalized mean error of the prediction varied from 1.2 to 2.5; the errors were at least as large as the values being predicted.

5.4.4 Summary

In the case where important domain parameters are categorical and not easily mapped to real numbers, little can be done in the way of modeling the performance to allow prediction of performance in other domains; in such cases, it is preferable to analyze the performance as a function of numerical attributes, as shown in the next section.

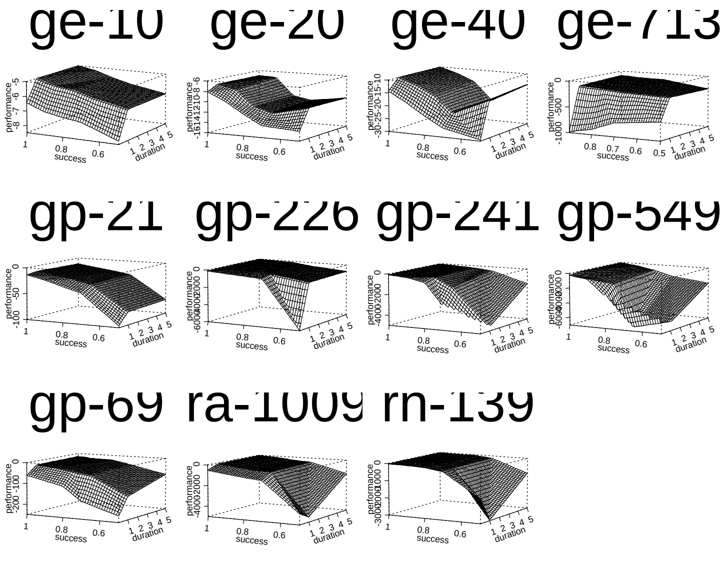


Figure 5.46: Plexus performance as a function of `success` and `duration`, for each of the different racetracks.

5.4.5 Influence of domain parameters on Plexus performance relative to optimal

General form

For the ratio of Plexus performance to optimal performance, `duration` and `track` are the most important parameters; `success` has less influence.

Figure 5.47 shows the ratio as a function of `success` and inverse `duration`. In the cases with high `success`, Plexus performs close to optimally; as `success` decreases, Plexus performs progressively worse, but for very low `success` rates, it improves again. This is because finding a good policy for low `success` rates is no more difficult than for high `success` rates (other than the deterministic case), but executing the policy is very expensive. The optimal performance is precisely the cost of execution, so the ratio is lower when the cost of execution is high.

Figure 5.48 shows the relative performance as a function of `success` and `duration`, for each of the different `tracks`. Just as for the absolute performance, `track` has a substantial influence on the form of the interactions between `success`, `duration` and relative performance. Models that ignore this parameter cannot capture these variations.

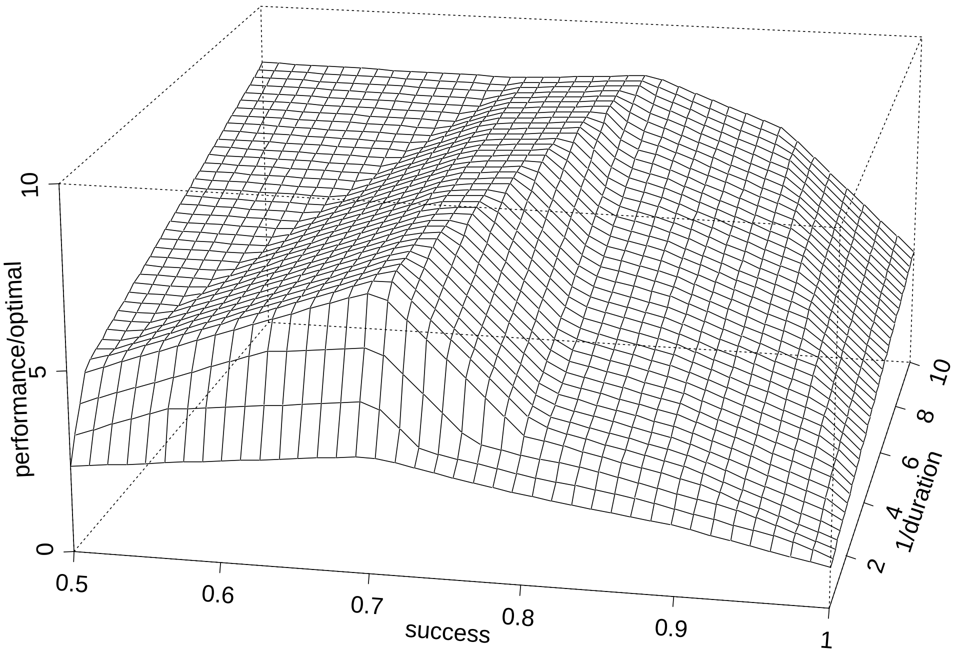


Figure 5.47: Ratio of Plexus performance to optimal, as a function of `success` and the inverse of `duration` of the actions, averaged over all `tracks`.

Prediction

Predictions based on a model linear in `success` and the inverse of `duration`, with interactions, give predictions with an error of between 0.9 and 1.4, when measured on a different set of domains from those used to construct the model.

Summary

As with the absolute performance, the categorical `track` parameter makes predictions difficult, since it cannot be used in the model. Predictions of relative performance based solely on `success` and `duration` of actions have similar error to the predictions of absolute performance.

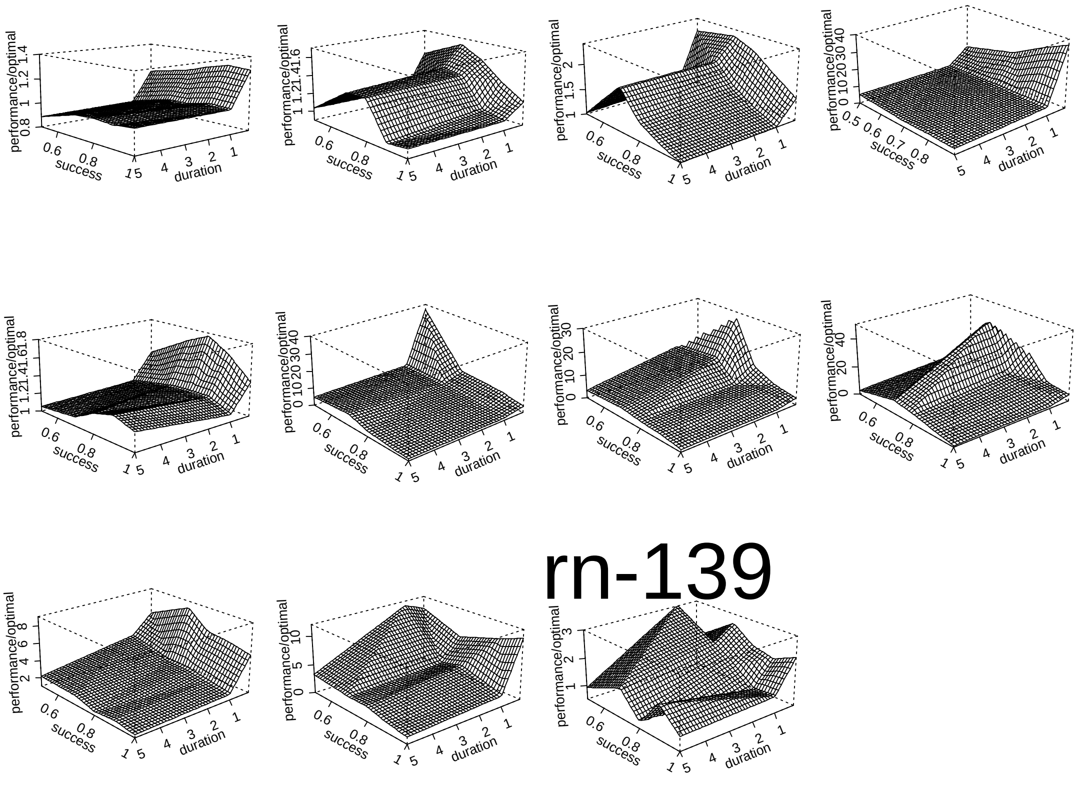


Figure 5.48: Ratio of Plexus performance to optimal, as a function of `success` and the inverse of `duration` of the actions, for each different `track`.

5.4.6 Influence of attributes on absolute Plexus performance

The attributes that most influence performance are `entropy` and `size`; `openness` and `normalized distance asymmetry` have a strong influence, while `volatility` and `distance asymmetry` do not.

The interactions between the various attributes are significant, and because there are relatively few different `tracks` in the set of domains, graphs showing performance as a function of one or two attributes are mostly unhelpful — higher dimensional combinations are necessary to capture the form of the interactions. I show only a few such graphs, and focus mainly on the predictive ability of the models.

Principal influences

The racetrack domains are unusual in that there are at most two possible outcomes of any state-action pair: either the action is successful or it is not. The action is deterministic if these two coincide: either if the action is to do nothing, or if the success probability is 1, or if the car will leave the track regardless of whether the action is successful. The no-op action only corresponds to one case in 9, so the important factor is whether the car will leave the track. In "difficult" domains, where most actions cause the car to leave the track, the entropy is consequently very low. In "easy" domains, the entropy is relatively high. This can be seen clearly in Figure 5.49, which shows `entropy` of the domains with `success` 0.9, 0.8, 0.7 and 0.5. The dotted lines show 95% confidence intervals for the means. `entropy` is 0 for all domains with `success` 1. `entropy` is therefore useful in conjunction with `success`.

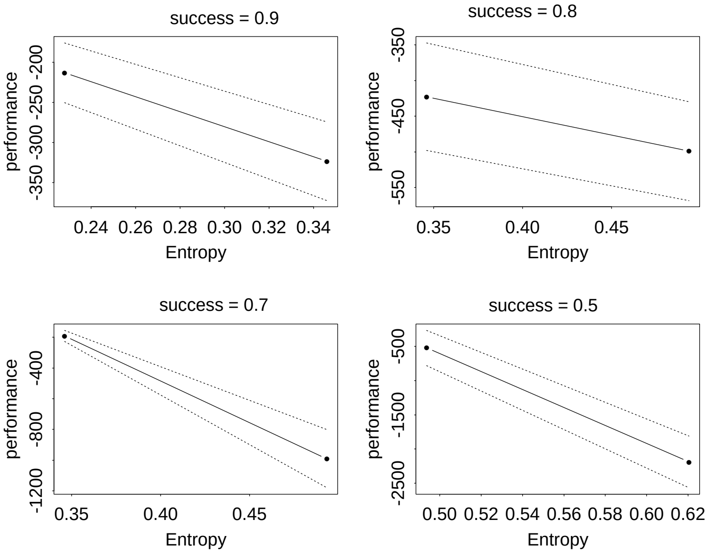


Figure 5.49: Plexus performance as a function of `entropy`, for different values of `success`.

Similarly, the normalized distance asymmetry is a good indicator of performance in conjunction with `success`, as shown in Figure 5.50.

Figure 5.51 shows the performance as a function of `volatility` for the different `success` rates. When `success` rate is high, `volatility` has a significant effect, but with lower `success` rates this is less true, as indicated by very wide confidence levels. This is because with low `success` rates the time to execute the policy, rather than the time to find the policy. This will become more apparent in the next section, where we look at the ratio of Plexus' performance to the

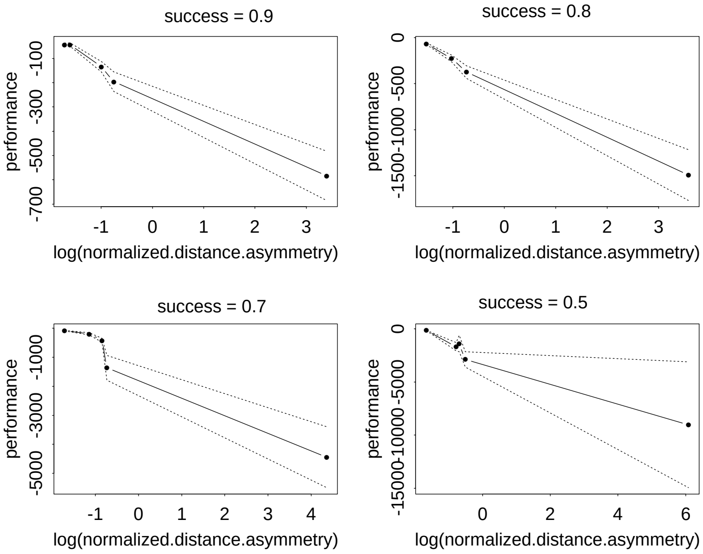


Figure 5.50: Plexus performance as a function of the log of the normalized distance asymmetry, for different values of `success`.

optimal performance.

Statistical models

Models based on the attributes `size`, `entropy` and `openness` provide better predictions than those based on the parameters `success` and `volatility`; adding other attributes does not significantly improve the performance. In particular, as might be expected from the results in the previous section, adding `volatility` does not help the predictions.

A model linear in the log of `size`, `entropy` and `openness` gives an error of approximately 1 on new domains (ranges from 0.9 to 1.2). This is somewhat better than the parameter-based model, but still considerably worse than the models for the RN1 and RN2 domains. This is probably due to the fact that the attributes (and the parameters) measure effects that are in some sense local — they correspond to the effects of short sequences of actions. In the RN1 and RN2 domains, where actions are largely reversible (the agent can relatively easily return to a state it just left), the effects of short sequences is a good indicator of the effects of longer sequences. This is not true in the racetrack domains, however, where longer sequences in a "difficult" world will almost invariably lead the agent back to the starting line. I believe that measures such as multi-step entropy would provide better predictors for domains such as the racetrack.

Interestingly, the interactions between these attributes are unimportant;

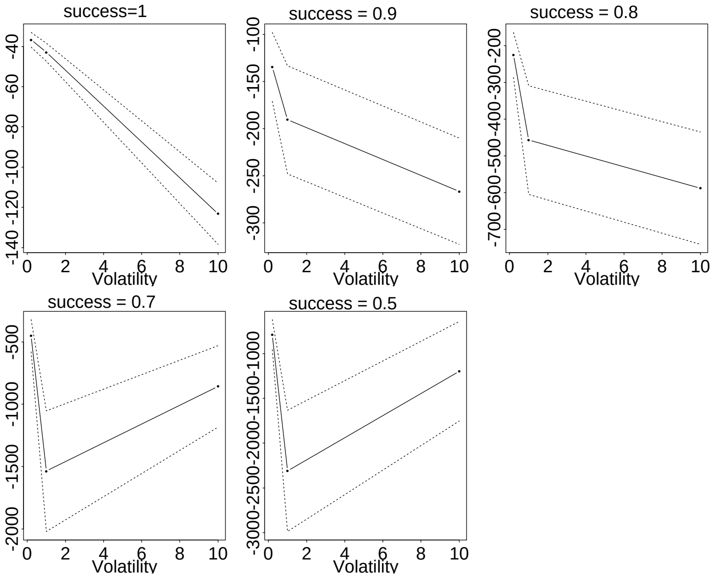


Figure 5.51: Plexus performance as a function of `volatility`, for different values of `success`.

adding interactions does not improve the models, and in fact in most cases makes the predictions on different domains worse. Other models based on attributes, using the normalized distance asymmetry in place of either the openness or the entropy, have similar predictive power. Since the normalized distance asymmetry is very expensive, a model that does not use it is preferable. Both the entropy and the openness are quite cheap to compute.

Summary

For the racetrack domains, attributes provide somewhat better predictions of the absolute performance of Plexus than parameters. The predictions are still much worse than for the RN1 and RN2 domains; this difference is probably due to the fact that the RN1 and RN2 domains are relatively reversible, whereas the racetrack domains are not. Further experimentation is required to confirm or disprove this, however.

Models linear in the log of `size`, `entropy` and `openness` provide predictions that are accurate to within approximately the magnitude of the performance; models using the normalized distance asymmetry perform equally well, but this attribute is expensive to compute, so `size-entropy-openness` model is preferable.

5.4.7 Influence of attributes on Plexus performance relative to optimal

General form

`entropy` and `volatility` are strongly correlated with the relative performance, as illustrated by Figures 5.52 and 5.53, though `volatility` is only a good predictor for high values of `success`. The other attributes are not strongly correlated with the relative performance.

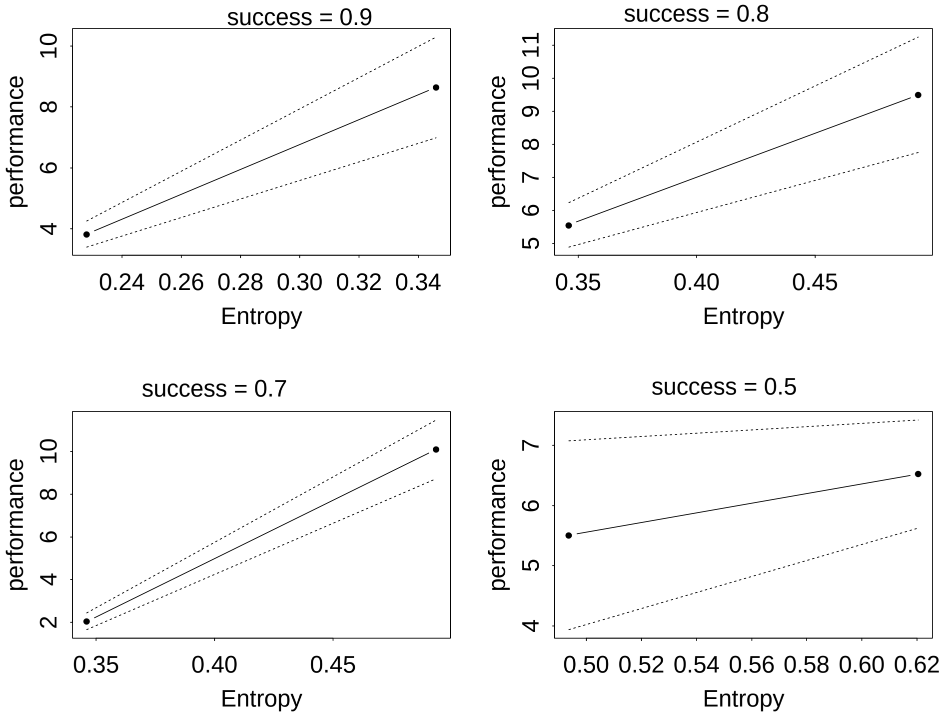


Figure 5.52: Ratio of Plexus performance to optimal, as a function of `entropy`, for each of four levels of `success` parameter

Summary

The ratio of Plexus' performance to optimal ranges from 0.7, for the random track, to nearly 50 for large domains with Grand-Prix type tracks and low success rates. Of course, Plexus cannot perform better than the optimal on average, so values of the ratio less than 1 are due to noise in the data.

Models based on attributes are less good predictors of the ratio of Plexus performance to optimal performance, giving predictions with an error of about

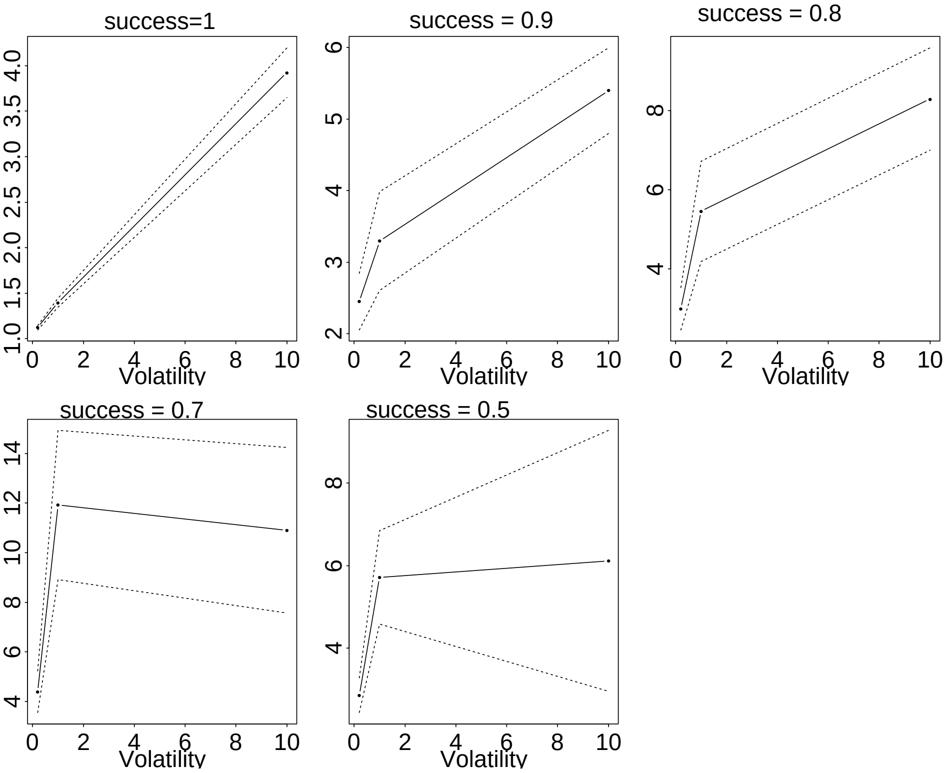


Figure 5.53: Ratio of Plexus performance to optimal, as a function of `volatility`, for each of four levels of `success` parameter

1.5: this is probably not useful for practical purposes, and we would need a larger base of data to form better models of the relative performance.

5.4.8 Influence of attributes on Plexus performance relative to other planners

General form

Figure 5.54 shows the performance of different planners as a function of `size`. The figure is shown in two parts since the cautious planner performs so much worse than the others. The changes in direction are due to domains with similar sizes but different tracks. As before, cautious is by far the worst in these domains, taking an order of magnitude longer than Plexus. For small sizes, the performances of Plexus, recover and replan are comparable; replan and recover do better on the medium-sized worlds, while Plexus does better on the larger worlds.

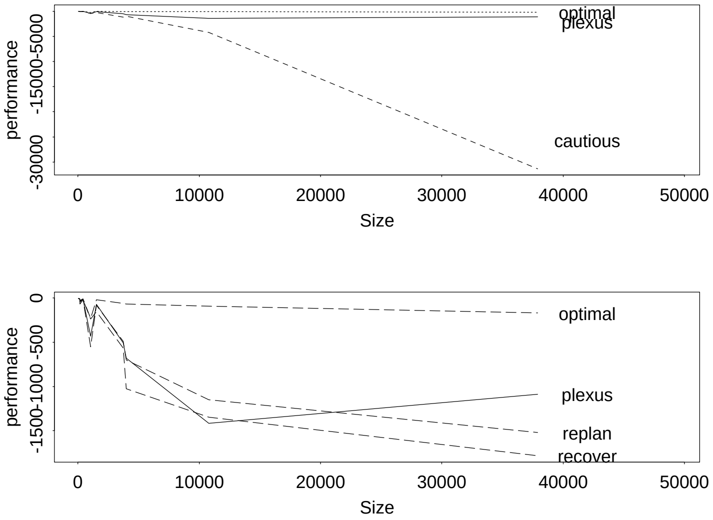


Figure 5.54: Performance of planners as a function of `size`.

Figure 5.55 shows the performances as a function of `volatility`. Cautious again is much worse than the others; Plexus, replan and recover are again comparable.

Similar figures for the other attributes are very difficult to interpret, since the interactions between the attributes complicate the relationships.

The figures are somewhat misleading in that they appear to show that replan and recover perform almost as well as Plexus. In fact, this is due to the averaging over several domains; if one compares individual domains, of the 161 domains, Plexus performs significantly better than the others in 16 cases, while the second best, recover, performs significantly better in only 5, and all of the 5 domains have the longest action duration (5 seconds). Generally it is the cases with more time-pressure (shortest action durations) that are of interest.

To see this more clearly, I show the ratio of Plexus' performance to recover's performance, as a function of success (Figure 5.56) and volatility (Figure 5.57). From these graphs we see that recover performs better with low volatilities (rewards are negative, so a ratio greater than 1 means Plexus is performing better than recover). Recover also performs better with high success values.

Summary

Plexus out-performs the other planners in more cases than any other planner, and the cases in which it performs worse are the least interesting ones (with low volatility). The difference between Plexus and recover is not very large; the

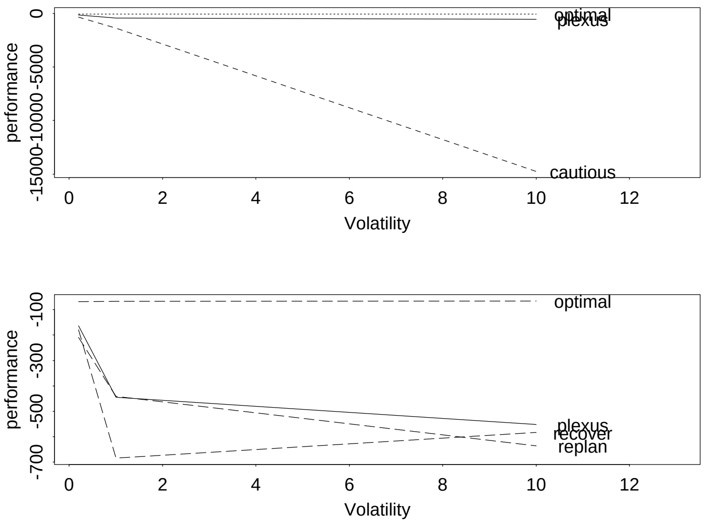


Figure 5.55: Performance of planners as a function of `volatility`.

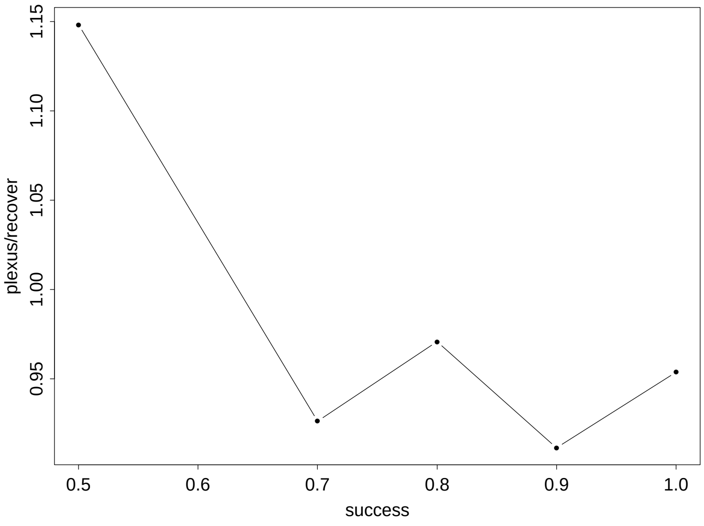


Figure 5.56: Relative performance of Plexus to recover, as a function of `success`.

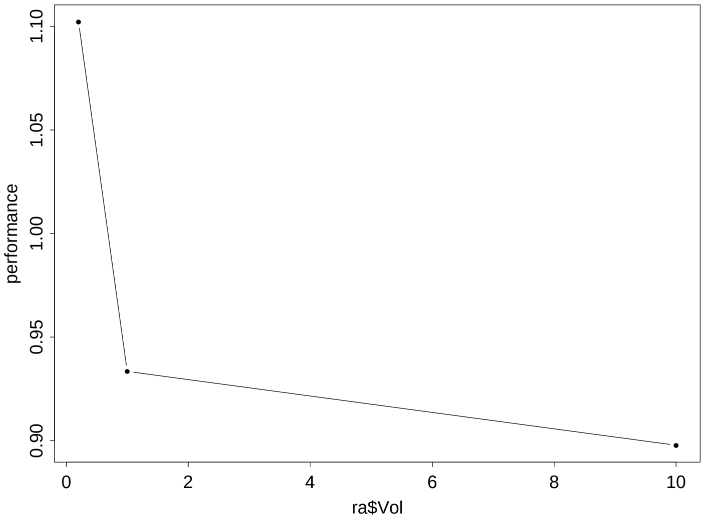


Figure 5.57: Relative performance of Plexus to recover, as a function of `volatility`.

average performance of Plexus over all the domains is −390, whereas that of recover is −490. Replan works well in these domains, because of the "inertia" in this type of domain — the agent cannot stay in the same state while looking for a new path, since its velocity will continue to carry it through the state space (until it returns to the starting line, at which point the velocity is zeroed). This is a disadvantage for Plexus and for recover, but not so much so for replan.

5.5 Summary

In this chapter I have presented an analysis of the performance of various different planners on three sets of problems. Each set is specified by a number of parameters, some continuous and some discrete.

I took a relatively small sample of problems from these 3- or 4-dimensional spaces and used it to form a model of the performance of Plexus. I showed that this small sample is sufficient to predict the performance of Plexus to within a factor of 2 on new domains taken from the set. The models also provide intuitions about the general influence of the parameters on the planner's performance.

By looking at the ratio of the planner's performance to the performance under an optimal policy, one can determine the areas in which the planner can be improved. Comparing the performance of different planners shows the regions of the space of problems for which different planners are most appropriate.

Chapter 6

Conclusions and future work

In this dissertation, I discuss solutions to sequential decision problems (with completely observable state) with the following characteristics:

- Uncertainty in outcomes — the outcomes of actions are not deterministic; either there are exogenous influences, or the actions themselves may fail.
- Real-time response required — the planner must make decisions in a timely manner.
- Large state space — the state space is too large for off-line optimal policy computation.

Most classical AI planning approaches assume that either the world operates deterministically, or that it does so most of the time, making it possible to plan under the assumption of determinism; they deal poorly with problems in which uncertainty is a fundamental component. The standard operations research and engineering approach is to construct an optimal policy off-line; this is not practical for problems with large state-spaces, or repeated problems with different goals.

In the solution I present here, the problem of uncertainty is handled naturally by the Markov decision problem framework I use; expectations of future reward are computed based on all possible outcomes of actions. When the exact computation of expected reward is prohibitively expensive, an approximation is made by considering only the most likely outcomes.

To address the problem of timely responses, the agent is composed of two parts: an executive and a planner. The executive uses a look-up table to choose actions based on the current state; its responses are thus effectively immediate. Initially, the executive starts with a random policy (or uses some domain-specific heuristic). The planner constructs a policy (a mapping from states to actions), using a model of the world's operation to predict the effect of different actions in different circumstances. Periodically, when the planner has a policy it thinks is better than the one the executive is currently using, it passes the policy to the executive, which will use that one until further notice. The policies are not guaranteed to improve, but they improve in expectation, and in practice improve quite rapidly.

To deal with large state spaces, the planner does not construct a policy over the entire state space, but instead focuses on the states that are likely to be visited in the near future. It constructs a *partial policy*, a mapping from a subset of the states to actions. The executive uses the partial policy when it is defined for the current state, and otherwise falls back on the default policy. As the agent moves through the state space, the planner can prune away states that have been passed — that are no longer likely to be visited — and add more states ahead of the agent. If the agent enters a state that is not in the partial policy, the planner finds a path back to the envelope of the policy. The agent thus uses an *anytime algorithm*: it finds approximate solutions early, and continually refines them as long as time is available.

I have shown that in most cases Plexus out-performs the other planners with which I compared it: RTDP [BBS93], recover and replan (two search-based planners). In certain types of problems, the search-based planners perform better; these problems are typically the least difficult (*i.e.,* the ones with the best performance under an optimal policy).

In the second part of the dissertation, I considered the following problem: given performance data on a collection of planning problems that vary by some number of parameters, can one predict the performance of Plexus (and other planners) for new (but similar) domains?

Predicting performance has several applications. When the problem can be specified in several different ways (*e.g.,* as one large problem or as several smaller problems to be solved sequentially), one can choose the specification with the best performance characteristics. A second use for such predictions is to choose the best planner for a particular task. I show examples where Plexus out-performs several other planners, and examples where simpler, search-based, planners perform better. A model of the relative performance of two planners gives a disciplined way of choosing the better planner. Finally, comparison with the optimal performance (the expected performance of an agent using the optimal policy) allows one to discern the cases where a planner performs particularly poorly, in order to focus on improving those cases, or cases where a planner performs close to optimally, in which case further improvement would be a waste of time.

My thesis is that predictions of this type can be made by measuring performance on a fairly small set of domains and extrapolating or interpolating to other domains, basing the prediction on the values of a few attributes. I have shown that this is true in all of the domains I have conducted experiments on.

Each class of domains was specified by varying a number of parameters; these parameters were strong predictors of the performance in most cases, which was expected, since the parameters were explicitly chosen to influence performance. Domain parameters are not always useful predictors, however; they are sometimes categorical, *i.e.,* they have no convenient numerical interpretation. In this case, clearly, they are of no use for predicting performance for new values of the parameter.

To solve this problem, I define a set of domain *attributes* that measure certain aspects of the underlying Markov decision problem, and are therefore domain independent. These attributes were also strong predictors of performance, in most cases better than the domain parameters; they are by definition continuous, and so avoid the problem of domain parameters. I have shown that attributes are useful predictors of performance in cases where the domain parameters are inadequate, and that they provide a convenient uniform set of metrics for characterizing planning problems.

This work can be extended in a number of directions. Experiments on classes of problems that differ significantly from the ones I used would determine to what extent these results can be generalized; it would be particularly useful to look at problems with a richer reward structure and ones with larger state spaces. Many of the restrictions listed in Section 2.2 can be relaxed; the most useful of these would be to allow actions of varying durations and to allow for unobservable state. The statistical models that I use are quite limited, due mainly to the paucity of data; more sophisticated models would provide better results if used on more extensive data.

Appendix A

Proofs and comments

A.1 Convergence of policy iteration

I will show that the multiple-update version of policy iteration converges to an optimal policy in a finite number of steps. The procedure is defined in Section 3.5.2. Similar proofs of this theorem can be found in many standard textbooks on Markov processes, *e.g.,* Derman [Der70], Bertsekas [Ber76]. I present a proof here for completeness.

A.1.1 Policy iteration over entire state space

Theorem 1 *If π is a (complete) policy, the policy iteration procedure terminates in a finite number of steps, with an optimal policy under the infinite horizon discounted value definition of optimality.*

First, I show that if a policy is not optimal, then for some state, there is an action other than that of the policy that produces a higher Q value. That is, the policy iteration algorithm will not stop until it has found an optimal policy. Next, I show that changing any number of actions while improving all of their Q values cannot decrease the value of any other state. Thus, the policy iteration algorithm will improve the value of all states at each iteration. Finally, I show that the policy iteration algorithm must converge to an optimal policy, since the set of policies is finite.

Lemma 1 *For Markov decision problems with a finite number of states and a finite number of actions, using the discounted infinite horizon optimality criterion, there exists an optimal policy that is deterministic and stationary (independent of the past history of the system).*

For a proof of this, see, *e.g.,* Derman [Der70]. Since the domain-task formulation of the problem is equivalent to the MDP formulation (see Section 2.1), there exists an optimal policy for any domain-task problem.

Lemma 2 *If π is a (complete) policy on domain $\{\mathcal{S}, \mathcal{A}, \tau\}$ and task $\{\rho, s_0, \mathcal{X}, \gamma\}$, and π is not optimal then there exists a state s and an action a such that $Q_pi(s,a) > V_\pi(s)$.*

Proof: if π is not optimal, there exists a policy π' that dominates π:

$$\begin{array}{l} \forall s \in \mathcal{S}, V_{\pi'(s)} \geq V_\pi(s) \\ \exists s \in \mathcal{S}, V_{\pi'(s)} > V_\pi(s) \end{array}$$

Consider, for each state, the difference between the value under the better policy and the value under the current policy. More precisely, let $d : \mathcal{S} \longrightarrow \mathbb{R}$ defined by

$$d(s) = V_{\pi'}(s) - V_\pi(s).$$

Clearly for some state, at least, d is strictly positive, since π' dominates π. Let w be a state that maximizes d. Since the number of states is finite, such a state exists. I will show that for the state w that maximizes $d, Q_\pi(w, \pi'(w)) > V_\pi(w)$. This says that given any policy, if we find the state for which π is most pessimistic, compared to π' we can improve the value of that state by choosing the action specified by the better policy. Of course, this is not a constructive proof, but the next lemma will provide a mechanism for applying policy iteration.
By definition of w,

$$\forall s \in \mathcal{S}, d(s) \leq d(w) \ ;$$

rewriting this using the definition of d, we see that

$$\forall s \in \mathcal{S}, V_\pi(s) \geq V_{\pi'}(s) - d(w) \ .$$

Now we consider the value of the state w under the optimal policy's action for that state. By definition of Q,

$$Q_\pi(w, \pi'(w)) = \rho(w, \pi'(w)) + \gamma \sum_{s \in \mathcal{S}} \tau(w, \pi'(w), s) V_\pi(s).$$

Replacing $V_\pi(s)$ by $V_{\pi'}(s) - d(w)$,

$$Q_\pi(w, \pi'(w)) \geq \rho(w, \pi'(w)) + \gamma \sum_{s \in \mathcal{S}} \tau(w, \pi'(w), s) \left(V_{\pi'}(s) - d(w)\right),$$

that is,

$$Q_\pi(w, \pi'(w)) \geq \rho(w, \pi'(w)) + \gamma \sum_{s \in \mathcal{S}} \tau(w, \pi'(w), s) V_{\pi'}(s) - \gamma \sum_{s \in \mathcal{S}} \tau(w, \pi'(w), s) d(w) \ .$$

The first two terms on the right of the inequality are, by definition, $V_{\pi'}(w)$, so

$$Q_\pi(w, \pi'(w)) \geq V_{\pi'}(w) - \gamma \sum_{s \in \mathcal{S}} \tau(w, \pi'(w), s) d(w) \ .$$

Since $d(w)$ is not dependent on s, we can take it out of the sum; since the sum of the transition probabilities from a particular state is 1, we obtain

$$Q_\pi(w, \pi'(w)) \geq V_{\pi'}(w) - \gamma d(w) \ .$$

But since $V_{\pi'}(w) - V_\pi(w) = d(w)$,

$$Q_\pi(w, \pi'(w)) \geq V_\pi(w) + d(w) - \gamma d(w) \ ,$$

or

$$Q_\pi(w, \pi'(w)) \geq V_\pi(w) + d(w)(1 - \gamma) \ .$$

Since we assume that $\gamma < 1$, and since $d(w) > 0$,

$$Q_\pi(w, \pi'(w)) > V_\pi(w) \ .$$

That is, there exists some state w and some action a for which the Q value is better than the value of w. Thus, as long as the policy is not optimal, the policy iteration algorithm will not terminate. □

Lemma 3 *If a policy π is not optimal then the policy π' produced by one round of policy iteration will dominate π.*

Consider a policy π and its associated value function V_π. Since π is not optimal, there are states with Q values higher than their V value. Let W be the set of such states:

$$W = \{w \in \mathcal{S}, \exists a \in \mathcal{A}, Q_\pi(w, a) > V_\pi(w)\} \ . \tag{A.1}$$

Consider the new policy π' defined by:

$$\pi'(s) = \begin{cases} \pi(s) & \text{if } s \notin W \\ \arg_a \max Q_\pi(w, a) & \text{if } s \in w \end{cases}$$

Let k be a state that minimizes the function d (difference of value under π' and π); since the number of states is finite, such a state exists. I will show that $V_{\pi'}(k) \geq V_\pi(k)$, and then that for some state, $V_{\pi'}$ is strictly greater than V_π.

There are two possible cases: $k \in w$ or $k \notin w$:

1. $k \notin W$

   Then $\pi(k) = \pi'(k)$, so

$$\begin{array}{rcl} V_\pi(k) & = & \rho(k, \pi(k)) + \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi(k), s') V_\pi(s') \\ V_{\pi'}(k) & = & \rho(k, \pi(k)) + \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi(k), s') V_{\pi'}(s') \end{array} \qquad \text{(A.2)}$$

   from which we obtain:

$$d(k) = \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi(k), s')(V_\pi(s') - V_{\pi'}(s'))$$

   i.e.,

$$d(k) = \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi(k), s') d(s')$$

   By definition, $d(k) \leq d(s')$ for any state s', so

$$d(k) \geq \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi(k), s') d(k)$$

   since the $\tau(k, \pi(k), s')$ sum to 1,

$$d(k) \geq \gamma d(k).$$

   Therefore $d(k) >= 0$.

2. $k \in W$

   We know from A.1 that $Q_\pi(k, \pi'(k)) > V_\pi(k)$. From this and the definition of $V_{\pi'}(k)$ and $Q_\pi(k, \pi'(k))$,

$$\begin{array}{rcl} V_{\pi'}(k) & = & \rho(k, a) + \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi'(k), s') V_{\pi'}(s') \\ V_\pi(k) & < & \rho(k, a) + \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi'(k), s') V_\pi(s') \end{array}$$

   Hence, subtracting the second equation from the first,

$$d(k) > \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi'(k), s') d(s'),$$

   and since k minimizes d,

$$d(k) > \gamma \sum_{s' \in \mathcal{S}} \tau(k, \pi'(k), s') d(k),$$

   i.e.,

$$d(k) > \gamma d(k),$$

   but $0 \leq \gamma < 1$, so

$$d(k) > 0$$

Thus $\forall s \in \mathcal{S}, d(s) >= 0$, so $V_{\pi'}(s) \geq V_\pi(s)$.

Now I show that all states in W have strictly higher values under π' than under π.

$$\forall w \in W, Q_\pi(w, \pi'(w)) = \rho(w, \pi'(w)) + \sum_{s' \in \mathcal{S}} \tau(w, \pi'(w), s') V_\pi(s') \ .$$

But by the above, $\forall s' \in \mathcal{S}, V_\pi(s') \leq V_{\pi'}(s')$. Thus

$$\forall w \in W, Q_\pi(w, \pi'(w)) \leq \rho(w, \pi'(w)) + \sum_{s' \in \mathcal{S}} \tau(w, \pi'(w), s') V_{\pi'}(s') \ .$$

The right-hand side of this inequality is simply $V_{\pi'}(w)$, so

$$\forall w \in W, Q_\pi(w, \pi'(w)) \leq V_{\pi'}(w) \ .$$

By definition, $V_\pi(w) < Q_\pi(w, \pi'(w))$, so

$$\forall w \in W, V_\pi(w) < V_{\pi'}(w) \ .$$

Therefore all the states in W have strictly higher values under the new policy π' than the old policy π. From this, and the fact that all states had higher or equal values under π', we conclude that $V_{\pi'}$ dominates V_π $\Box$.

I have now shown that if a policy π is not optimal, the policy iteration algorithm will find at least one state whose Q value can be improved, and that the policy resulting from changing all actions to maximize Q values will produce a policy that dominates π. Since there are a finite number of policies, policy iteration must converge on an optimal policy $\Box$.

A.1.2 Policy iteration over an envelope

Now consider the policy iteration algorithm applied to a partial policy. The algorithm is the same as in the case of the entire state space, except that policy improvement is applied only to states in the envelope, and value computation is done slightly differently; the equations defining the values of states are:

$$\begin{aligned}
V_o(s) &= \rho(s, \pi_d(s)) \left(\sum_{i=1}^{Eo} \gamma^i \right) \\
P_{in}(s) &= \sum_{s' \in \mathcal{E}} \tau(s, \pi(s), s') \\
V_e(s) &= \begin{cases} \frac{\rho(s,\pi(s)}{1-\gamma} & \text{if } P_{in}(s) = 0 \\ \frac{1}{P_{in}(s)} \sum_{s' \in \mathcal{E}} \tau(s, \pi(s), s') V(s') & \text{otherwise} \end{cases} \\
V(s) &= \begin{cases} \rho(s, \pi(s)) + \gamma \left(P_{in}(s) V_e(s) + (1 - P_{in}(s))(V_o(s) + \gamma^{Eo} V_e(s) \right) & \text{if } s \in \mathcal{E} - \mathcal{X} \\ 0 & \text{if } s \in \mathcal{X} \end{cases}
\end{aligned}$$

We are given a domain $D = \{\mathcal{S}, \mathcal{A}, \tau\}$, a task $T = \{\rho, s_0, \mathcal{X}, \gamma\}$, and a partial policy $\pi : \mathcal{E} \longrightarrow \mathcal{A}$. We construct a new domain $D' = \{\mathcal{S}', \mathcal{A}', \tau'\}$, and a new task $T' = \{\rho', s_0{}', \mathcal{X}', \gamma'\}$. A policy on D will trivially induce a policy on D', and the policy will be complete. I will show that the value computation step of policy iteration on D (defined in Section 3.5.3) produces the same values as standard value computation on D'. I next show that the policy improvement step will make the same changes as the standard policy improvement step on D'. Finally, I show that an optimal policy on D' induces a policy on D that is optimal under the partial policy definition of state value.

Recall that Eo is an estimate of the number of steps the agent will be out of the envelope. In general, Eo will change as the system evolves, but it is fixed during the execution of the phases in a cycle; for our purposes here, it can therefore be considered as a constant. For each state s in the envelope and for each action a in $\mathcal{A}$, we define a set of new states $\{O_{s,a,1}, O_{s,a,2}, \ldots, O_{s,a,Eo}\}$, as shown in Figure A.1 (repeated from Figure 3.13). The new set of states, $\mathcal{S}'$, consists of the envelope and, for each state s in the envelope, $\mid \mathcal{A} \mid \times Eo$ new states, denoted $O_{a,i}$, for $a \in \mathcal{A}$ and $i \in [1, Eo]$.

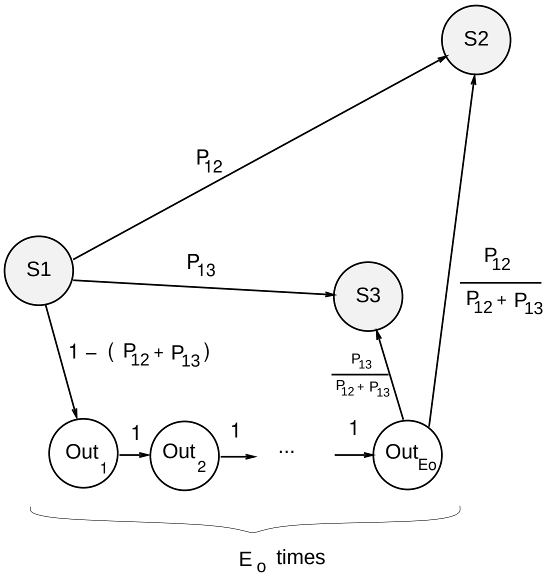


Figure A.1: Value of a state adjoining the fringe

The transitions are defined below; between states in the envelope, the transition probabilities remain the same. Transitions out of the envelope become transitions to the first state in a chain; there is a different chain for each state-action pair. Once in the chain, all actions have the same effect; the system moves through the chain one step at a time, getting the same reward as for the original state-action pair. All of the chains are of the same length: Eo. At the end of the chain, the transition probabilities are similar to those in the original state — the only difference is that in all cases, the system returns to the envelope. In the specification below, the function τ' is assumed to be 0 anywhere it is not explicitly defined. In the transitions definitions, s and s' are states in the envelope $\mathcal{E}$, a and a' are actions in $\mathcal{A}$, and i is an integer between 1 and $Eo - 1$ inclusive. Since the actions chosen have no influence on the outcome for the new states, I will not distinguish policies on D from those on D'.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{S}' = \mathcal{S} \cup \mathcal{S} \times \mathcal{A} \times \{1, 2, \ldots, Eo\}} \\
\multicolumn{3}{l}{\mathcal{A}' = \mathcal{A}} \\
\quad \tau'(s, a, s') & = & \tau(s, a, s') \\
\quad \tau'(s, a, O_{s,a,1}) & = & 1 - P_{in}(s, a) \\
\quad \tau'(O_{s,a,i}, a', O_{s,a,i+1}) & = & 1 \\
\quad \tau'(O_{s,a,Eo}, a', s') & = & (1/P_{in}(s, a))\tau(s, a, s') \quad \text{if } P_{in}(s, a) \neq 0 \\
\quad \tau'(O_{s,a,Eo}, a', s) & = & 1 \qquad\qquad\qquad\qquad\quad\; \text{if } P_{in}(s, a) = 0 \\
\multicolumn{3}{l}{\gamma' = \gamma} \\
\multicolumn{3}{l}{s_0 = s_0} \\
\multicolumn{3}{l}{\mathcal{X}' = \mathcal{X}} \\
\multicolumn{3}{l}{\rho'(s, a) = \rho(s, a)} \\
\multicolumn{3}{l}{\rho'(O_{s,a,i}, a') = \rho(s, a)} \\
\multicolumn{3}{l}{\rho'(O_{s,a,Eo}, a') = \rho(s, a)}
\end{array}
$$

Lemma 4 *The value computation step of policy iteration, performed on a policy π over D', will result in the same values for the states of the envelope as the modified value computation on D, described in Section 3.5.3.*

This can be verified trivially from the definition of D'.

Lemma 5 *The policy improvement step of policy iteration, performed on a policy π over D', will result in the same choice of actions as the modified policy improvement step on D.*

In the same way that the value of a state in D is the same as that in D', the Q-value of a state under an alternative action is the same in both domains. Since the choice of actions for the states that are in D' but not in D does not influence the values of the states in D', the actions that maximize the Q values are the same in both domains.

Since the two steps of policy iteration have the same effect in both domains, policy iteration on D will produce an optimal policy for D', which is optimal for D. $\square$

A.2 Representing undiscounted rewards with discounting

In Section 2.1, I imposed the restriction that the discount factor γ be strictly less than 1. Here I show that under certain conditions, a value for $\gamma < 1$ can be chosen such that the optimal policy is also optimal when $\gamma = 1$.

In the case where $\gamma = 1$, the value function induced by a policy π is defined by:

$$V_\pi(s) = \sum_{t=0}^{\infty} E(R_t)$$

where R_t is the expected reward received on the tth step of executing policy π after starting in state s. I allow the value function to range over the extended reals: $\mathbb{R} \cup \{-\infty, \infty\}$. It is clear that the value function may not be well-defined for all states and policies. In order to guarantee that it is well-defined, I impose the requirement that the reward function is non-negative, *i.e.,*

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \rho(s, a) \leq 0. \tag{A.3}$$

It is then clear that the value of a state under a policy must be either a negative real or $-\infty$.

In addition, I make the assumption that there exists a policy π that induces a value function that is finite for all states:

$$\exists \pi, \forall s \in \mathcal{S}, V_\pi(s) \in \mathbb{R}. \tag{A.4}$$

In effect, this assumption states that the performance of the agent under this policy can always be measured by a finite quantity; equivalently, that no sequence of states is "infinitely bad"; this seems a reasonable assumption in most cases. In my thesis I will discuss cases where this assumption does not hold. This assumption implies that all ergodic states have reward 0, since if the system is started in an ergodic set, it will visit every state in the set infinitely often, and if any of the states have non-zero reward, the sum of rewards will be infinite.

Clearly, the existence of such a policy implies that the optimal policy also induces a value function that is finite for all states; let π be the optimal policy without discounting.

Finally, I assume that the Markov chain induced by the optimal policy π is absorbing (Section 2.1), *i.e.,* all ergodic sets are singletons. This assumption is reasonable, because once the system has entered an ergodic set, the rewards accumulated thereafter are 0, so the set can be collapsed to a single state without affecting the value of other states.

I will show that under these assumptions, there exists a $\gamma \in [0,1)$ such that π is optimal under the performance model with discount factor γ.

Let V be the value function induced by π under the undiscounted model, and let V_γ be the value function induced by the same policy π under the model with discount factor γ.

Let $r_n(s)$ be the expected instantaneous reward obtained after n steps starting from state s. Clearly $r_n(s)$ does not depend on the discounting model.

Lemma 6

$$\exists \alpha \in (-\infty, 0), \beta \in [0,1), \forall m \in \mathbb{N}, \forall s \in \mathcal{S}, r_m(s) > \alpha\beta^m.$$

i.e., the series $r_m(s)$ converges to 0 at least as quickly as a power series.

Since the Markov chain induced by the policy is absorbing, for any state s, there is some absorbing state s' that is reachable from s. That is, there exists a sequence of states $\{s_0 = s, s_1 \ldots s_k = s'\}$ such that for all $i < k-1$, $\tau(s_i, \pi(s_i), s_{i+1}) \neq 0$. Note that this sequence may be chosen such that no state appears in it twice, so the length of the sequence is less than the number of states in the domain. Let

$$p(s) = \prod_{i=0}^{i=k-1} \tau(s_i, \pi(s_i), s_{i+1})$$

$$p = \min_{s \in \mathcal{S}} p(s)$$

Let $N = |\mathcal{S}|$, and define:

$$\hat{p} = \left(\min_{s \in \mathcal{S}, a \in \mathcal{A}, s' \in \mathcal{S}, \tau(s,a,s') \neq 0}, \tau(s,a,s') \right)^N$$

Clearly, $\hat{p} < p$.

If the system is in any state s at time t_1, it will be in an absorbing state at time $t_1 + N$ with probability at least $\hat{p}$, since there is some path from s to an absorbing state with probability at least p.

Therefore, the probability that the system will *not* be in an absorbing state after N steps is no larger than $1-\hat{p}$. This inequality holds regardless of the initial state s, the probability that the system will not be in an absorbing state after kN steps is at least $(1-\hat{p}^N)^k$. Let $b = 1 - \hat{p}^N$. Since $\hat{p} > 0$, $0 \leq b < 1$

The expected reward after n steps starting in state s, $r_n(s)$, is the sum of the rewards for each state weighted by the probability that the system will be in that state after n steps. It is thus at least equal to the smallest instantaneous reward multiplied by the probability that the system is not in an absorbing state, since absorbing states have 0 instantaneous reward.

Let

$$R = \min_{s \in \mathcal{S}, a \in \mathcal{A}} \rho(s, a).$$

Using the bound on the probability that the system is not in an absorbing state, for any state s, $r_{kN}(s) \geq Rb^k$.

Clearly, since r is an increasing sequence, for any integer m and any state s,

$$r_m(s) = r_{N\frac{m}{N}}(s) \geq r_{N\lfloor\frac{m}{N}\rfloor}(s) \geq r_{\frac{m}{N}+1}(s) \geq Rb^{\frac{m}{N}+1}(s) = Rb(b^{\frac{1}{N}})^m.$$

Letting $\alpha = 2Rb$ and $\beta = b^{\frac{1}{N}}$, I have shown that

$$\forall m \in \mathbb{N}, s \in \mathcal{S}, r_m(s) > \alpha\beta^m.$$

Since $0 \leq b < 1$ and $R < 0$, $0 \leq \beta < 1$ and $\alpha < 0$, as desired$\Box$.

Theorem 2

$$\forall \epsilon > 0 \ \exists \gamma \in [0, 1), \forall s \in \mathcal{S}, \mid V(s) - V_\gamma(s) \mid < \epsilon$$

By definition

$$\begin{array}{rcl} V(s) & = & \sum_{i=0}^{\infty} r_i(s) \\ V_\gamma(s) & = & \sum_{i=0}^{\infty} \gamma^i r_i(s) \end{array}.$$

Since $0 \leq \gamma < 1$, $V(s) < V_\gamma(s)$. From the definitions,

$$V_\gamma(s) - V(s) = \sum i = 0^\infty (1 - \gamma^i)(-r_i(s)).$$

By Lemma 6,

$$V_\gamma(s) - V(s) < \sum i = 0^\infty (1 - \gamma^i)(-\alpha\beta^i).$$

Hence,

$$V_\gamma(s) - V(s) < -\alpha \sum i = 0^\infty (1 - \gamma^i)(\beta^i),$$

i.e.,

$$V_\gamma(s) - V(s) < -\alpha \left(\sum i = 0^\infty \beta^i - \sum i = 0^\infty (\gamma\beta)^i \right).$$

Summing the power series,

$$V_\gamma(s) - V(s) < -\alpha \left(\frac{1}{1 - b} + \frac{1}{1 - \gamma^\beta} \right).$$

By choosing γ sufficiently close to 1, this quantity can be made arbitrarily small.$\Box$.

A.3 Convergence of performance measure EP_n

Below is a sketch of the proof that the performance measure EP_n is well-defined; the details are straightforward.

Recall that for an agent α and a history H_n,

$$V(H_n) = \sum_{i=0}^{i=n-1} \gamma^i \rho(s_i, a_i),$$

and

$$EP_n(\alpha) = \sum_{H_n \in \mathcal{H}_n} \mathrm{PR}_\alpha(H_n) V(H_n).$$

Lemma 7 *$EP_n(\alpha)$ is a Cauchy sequence,* i.e.,

$$\forall \epsilon > 0, \exists N \in \mathbb{N}, \forall n, m > N, | EP_n(\alpha) - EP_m(\alpha) | < \epsilon$$

We decompose the value accumulated over n steps into that accumulated during the first m steps plus that accumulated during the last $n-m$ steps. This latter is bounded above and below by constants multiplied by γ^m; therefore if N is sufficiently large, the difference between EP_n and EP_m must be less than γ^N times a constant. This can be made smaller than an arbitrary ϵ by suitable choice of N.

This sequence is defined on the reals, which form a complete metric space, and Cauchy sequences in complete metric spaces converge (see, *e.g.,* [Wei73]). Thus each element of $\alpha^n Q^n$ converges, so

$$EP = \lim_{n \to \infty} EP_n$$

is unique.

A.4 Performance criteria

In order to determine whether the algorithms described above are effective at producing optimal or good policies, we need to be clear about the criteria for performance. The "right" criterion obviously depends on the application; here are some common formulations of performance:

1. Maximize expected cumulative reward over a finite horizon
2. Maximize expected discounted cumulative reward over an infinite horizon
3. Maximize expected undiscounted cumulative reward over an infinite horizon

4. Maximize expected average reward over an infinite horizon

Finite horizon problems are generally solved using dynamic programming techniques that are computationally much too expensive for the type of domain we use, hence I will not consider them.

Maximized discounted cumulative reward is the performance measure that corresponds to our current implementation. There are two cases where using this measure seems reasonable to me. The first is for tasks that are subject to termination at any time, with equal probability of termination at each time-point. The Traffic World is one such task; arguably if a high-level controller monitors the environment and terminates the current planner/executive when its model of the world becomes too inaccurate, the lifetime of the planner can be modeled by this type of termination. The second case is one where future reward is discounted in the economic sense. Problems like the Non-combative Evacuation Operations probably fall into this category; evacuating people early is worth more than evacuating them late.

Maximizing undiscounted cumulative reward is the performance measure that seems most natural for tasks for which there exists a proper policy, in particular tasks of achievement, such as those in the Robot Navigation domain.

Maximizing expected average reward is appropriate for problems in which there is no natural termination, but discounting is not desirable. This seems the most reasonable measure of performance for variations on the Traffic World in which it is assumed that the model of the environment will be valid indefinitely. The usual argument against this definition of performance is that different executions of the system may have the same average reward in the limit while having very different "average-adjusted" values (the limit of the difference between the average reward times the number of steps and the cumulative reward), and this performance measure does not allow for any distinction between them. Schwartz' suggestion of measuring performance by both the average and average-adjusted values (the pair is called the *total value*, and ordering is lexicographic) takes care of this problem.

For example, a company will usually accept a small probability that an action will lead to bankruptcy (and hence an average future reward of 0), to get a large immediate reward, rather than refusing the immediate reward to get an average future reward that is small but strictly greater than 0.

Furthermore, notice that in some sense the use of a discounting factor that is very close to 1 approximates the total value measure Schwartz proposes; basically it maps a total value of $\langle \rho, \sigma \rangle$ to $\rho/(1-\gamma)+\sigma$. If gamma is close enough to 1, the ordering on the reals provides a close approximation to the ordering on total values. The cases where it differs are precisely those in which humans tend to reject the total value type of ordering: where the difference in average rewards is very small, while the difference in average-adjusted rewards is large and of opposite sign.

This can be taken as an argument for discounting, but unfortunately it does not get around one big problem with discounting: how to choose an appropriate discounting factor that treads the line between inefficiency and a criterion of optimality with undesirable consequences (*e.g.,* not guaranteed to terminate on a halting problem).

Appendix B

Worlds, agents, and their interactions

In this appendix I describe the domains and agents that are used as examples throughout the thesis, and outline the implementation details concerning their interactions.

PEST (Planning and Execution System Testbed) is a system that co-ordinates agents and worlds, either in simulation or in actual operation. It provides a set of programs that implement the worlds and agents described in previous chapters. It also provides a program that co-ordinates the execution and communication of an agent and a world.

Given a world and a planner, Pest simulates execution of the agent-environment system defined by the two. It ensures that the CPU time allocated to the agent corresponds to the duration of the actions as specified by the world, regardless of the presence of other processes on the host. This is essential for accurate performance measurement. It uses a multi-threaded approach to deal with concurrent operation of the executive and the planner. Currently the system operates only on Sun Solaris platforms, using Solaris threads.

B.1 Agent-world interaction

The agent and the world are represented by UNIX processes, using the standard stream interface. An additional process may be specified to display the current state of the world. Most commonly, the agent is a planning program and the world is a simulation of an actual environment. This is not necessarily the case; for example, the agent may be a program that drives a user-interface to a human, and the world may be a program sending commands to, and receiving perceptual information from, a mobile robot.

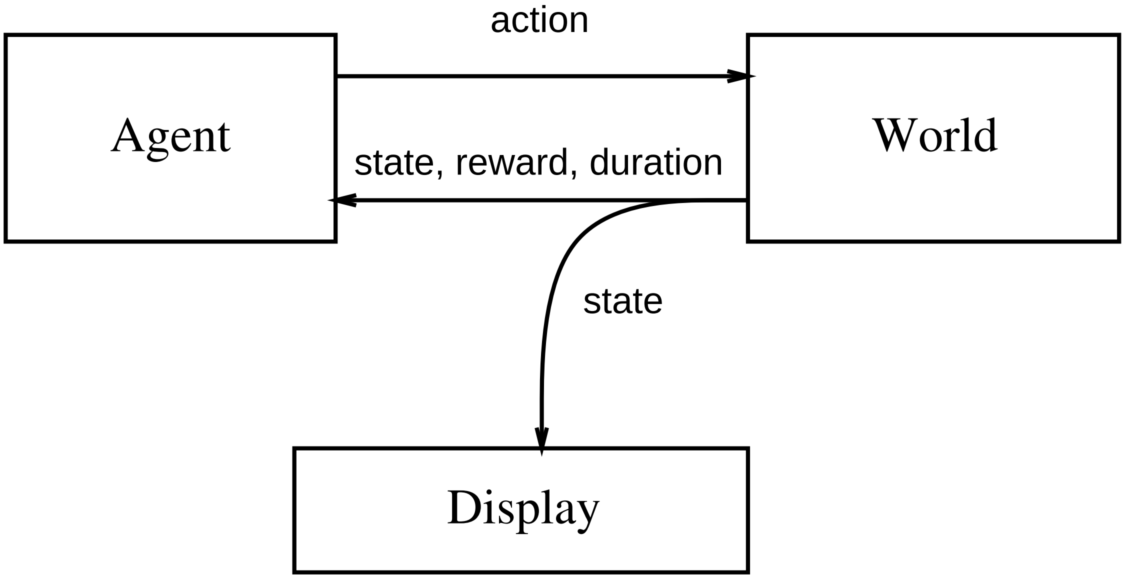


Figure B.1: Interactions between agent, world, and display

Figure B.1 depicts the interactions of the three processes. The sequences of operations are as follows:

Agent

```
Read state
While not end of file on input
    Write action
    While input not available
        Internal processing
    Read state, reward, duration
```

The agent initially reads the current state of the world from standard input. It must immediately[1] write an action to standard output. It may then do any internal processing desired, until it receives a new state, along with the reward and duration of the previous transition, on standard input. This cycle repeats indefinitely. The reward and duration are supplied to the agent in the event that it does not have access to a model of the world.

World

---

[1] A fixed period of time is allowed, negligible compared to the duration of actions.

```
Write initial state s
While s is not terminating
    Read action
    Write new state s, reward, duration
```

The world writes its initial state to standard output and then repeatedly reads an action from standard input, makes the state transition internally (this might involve, *e.g.,* forwarding the command to a robot and then sensing the new state of the environment and the robot) and writes the new state, the immediate reward, and the duration of the transition to standard output. The world need not take as long as the duration specifies before responding; Pest will ensure that the agent is allowed the appropriate amount of time.

Display

```
While not end of file on input
    Read state
    Display state
```

The display simply reads states and displays them in some manner, typically using a graphical interface.

B.2 Agents and worlds

I have implemented a number of different agents and worlds in order to conduct the experiments described in this dissertation; they are described below.

B.2.1 RTDP

RTDP implements the real-time dynamic programming algorithm presented by Barto, Bradtke and Singh in [BBS93]. This agent requires a world model, as defined below. I present some comparisons of the performance of RTDP and the Plexus planner in Section 5. My implementation of RTDP is very simple, and its performance could be greatly improved by techniques such as prioritized sweeping [MA93].

RTDP starts by assigning an optimistic value to each state. In all the cases I discuss, rewards are negative, so we ascribe a value of 0 to every state. Then RTDP goes into a cycle of trials. Each trial consists of a number of simulated steps in the environment, starting from the current state. It computes the expected value of each action using the current state values, and chooses the action with the best value. Then the action is simulated — a state is chosen according to the probability distribution of the action outcome — and the process is repeated at that state. After a fixed number of such steps, the trial is halted, and the next trial is started from the current state again. The length of trials is gradually lengthened, in a kind of iterative deepening search. This cycle is repeated indefinitely, producing ever more accurate values for the states.

I expected RTDP to be very sensitive to the initial trial length and the rate of increase of the trial length, but in a set of experiments on the RN2 domains, I found very little difference in performance when these parameters were varied. For all the RTDP experiments reported here, I used a trial length of 1000 and increased this by 1% each time a trial failed to reach a goal state.

B.2.2 Recover

Recover implements a search-based agent. It uses the best-first search algorithm from the explore phase, described in Section 3.8, to find a path from the initial state to a goal state. It uses the partial policy induced by this path (the envelope consists of the states in the path, and the actions are those used by the search procedure) as long as the agent remains in the envelope. If the agent falls out of the envelope, the recover planner does a best-first search to any state of the envelope or goal. If this search reaches a new goal, the entire old envelope is discarded. If the search reaches a state on the old path, the portion of the path after this state is kept, along with the new path to the state. The portion of the old path before this state is discarded.

B.2.3 Replan

Replan implements a search-based agent similar to recover, except that in the case of a plan "failure" (the agent falls out of the envelope), it discards the old envelope and searches from the current state to a goal.

B.2.4 Plexus phase-cycle planner

The Plexus phase-cycle planner, described in Section 3.3.5, is implemented in its general form: an initial sequence of phases followed by a recurrent sequence of phases. At the moment, no flow-control is provided; this would be a useful addition, since conditioning phase execution on results obtained during previous phases would be a simple step towards the more general deliberation-scheduling

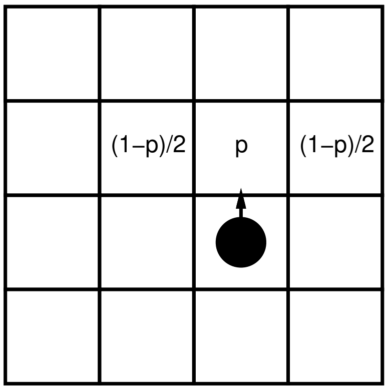


Figure B.2: Possible outcomes of an action in the RN1 domain

planners described in earlier Plexus papers (*e.g.,* [DKKN93b]). The phases currently implemented are: strengthen, prune, explore, policy iteration and value iteration, described in Chapter 3.

B.3 Domains

B.3.1 Simple Robot Navigation (RN1)

The RN1 class of domains is designed for testing agents; they model a robot moving in an uncluttered area.

Domain

The domain consists of a rectangular grid of cells, denoted L-¡x¿-¡y¿. L-0-0 is the top left cell and L-1-0 is one cell to its right. The robot occupies one cell at any given time-point; the state of the world is just the location of the robot. The actions available to the robot are `north`, `south`, `east`, `west` and `stay`.

A parameter `success` (a real between 0 and 1) specifies the probability that the action will be successful, moving the robot one cell in the specified direction. The remainder of the probability mass is equally divided between the two diagonals (*e.g.,* North-East and North-West if the action was `north`. If the result of the action would move the robot off the grid, it remains in place. The result of the action `stay` is deterministic, leaving the robot in the same location.

Figure B.2 shows a 4x4 domain. The robot is in state L-2-2, and takes the action `north`. The figure shows the three possible outcomes and their probabilities, with p being the probability of success, `success`.

Task

The task given to the agent is to minimize the number of steps taken before reaching a designated state `goal`, starting in a state `start`. See Section B.3.2 for a description of the reward function used to represent this task.

B.3.2 Robot Navigation (RN2)

Motivation

This type of domain, which I refer to as RN2, model a mobile robot moving around a static office-building environment. The design of the domains was based on experience with a Real World Interface mobile robot base equipped with sonars as its only sensor [KBD91, DCK90, Bas92]. One of the main problems with navigation using mobile robots is the speed with which small errors in dead reckoning accumulate, due to irregularities in the floor surface, wheel slippage, or misalignment of the wheels. Analysis of the sonar readings in an office building[LDW89] is more reliable when the sonars are known to be roughly perpendicular to a flat surface, since deviations from the normal will lead to specular reflections, giving wildly incorrect results. Consequently our approach includes an attempt to realign the robot to be facing one of the four canonical directions after each major movement — hence the discretization of robot direction.

Another consequence of these problems is that the robot may have become sufficiently poorly oriented after a movement that an attempt to realign causes a 90-degree error in the final heading. Finally, since the map used by the robot contains only high-level information (adjacency of rooms and corridors), small obstacles in the path of the robot are avoided using local operations which may cause errors in the final location of the robot. The Robot Navigation model was designed to represent this type of error.

Domain

The physical layout of the world is described as a connected square grid; some squares in the grid are off-limits, but the agent can move between any two adjacent grid squares. Each grid square is called a location. The state of the robot is the location of the robot along with its direction, one of `north`, `east`, `south` and `west`. The actions are `go`, `stay`, `left`, `right`, `about`. Nominally, `go` moves the robot to the next location in front of it, `left` and `right` rotate the robot 90°, `about` rotates the robot 180°, and `stay` does nothing. In general the outcomes of the actions are non-deterministic; for example, there may be some probability that the robot overshoots into the location beyond the one it was supposed to move to, or that a rotation will not produce the desired effect.

For the experiments described in earlier chapters, I used a map of the fourth floor of the CIT building at Brown University. I discretized the map coarsely into about 160 locations, as shown in Figure B.3. The "sinks" marked in this figure are explained below. To increase the size of the problem, I created new domains by connecting up copies of the grid; there are four locations that protrude from the rectangle containing all the others, one in each direction. These form passageways from one copy to the next. I always arranged the copies in a square; this produces many possible paths from one location to another. The four sizes of domain I used were 1x1, 2x2, 3x3 and 4x4.

Movement probabilities

The five movement probabilities define the normal operation of the robot. Three affect the outcome of a `go` action:

- `goSuccess` is the probability that when the action `go` is taken, the robot will succeed: the resulting state is one square ahead of the robot, and the direction remains unchanged.

- `goStay` is the probability that when the action `go` is taken, the robot will in fact remain in the same state.

- `goOver` is the probability that when the action `go` is taken, the robot will overshoot and end in the state two squares ahead, with the same direction.

If the square ahead of the robot is not part of the grid, the `goOver` and `goSuccess` probabilities are added to the probability of staying in the same state. If the square two ahead of the robot is not part of the grid, the `goOver` probability is added to the probability of success. These adjustments are realistic in this domain.

The remainder of the probability mass is evenly distributed amongst those of the eight other possible outcomes that are feasible (part of the grid); the robot may stay in the same location, move one location to the left or to the right or backwards, and it may stay facing in the same direction or rotate in the direction it slipped.

The other two movement probabilities affect the outcome of the three turn actions `left`, `right` and `about`. The turn actions always leave the robot in the same location.

- `turnOver` is the probability that the robot will turn 90 degrees more than the turn specified.

- `turnUnder` is the probability that the robot will turn 90 degrees less than the turn specified.

The remainder of the probability mass is associated with a successful turn. Thus for the action `left`, there is probability `turnUnder` that the robot will remain in the same state, probability $1 - \texttt{turnUnder} - \texttt{turnOver}$ that the robot will turn 90 degrees to the left, and probability `turnOver` that it will turn 180 degrees. The stay action is always successful.

If the success parameter described in Section B.3.2 is p, then $\texttt{goSuccess} = p, \texttt{goOver} = (1-p)/4, \texttt{goStay} = (1-p)/4$ — the probability of a successful go (move forward one grid square) is p; there is a probability $(1-p)/4$ of overshooting or not moving, and the remainder of the probability distribution is distributed evenly amongst the other 8 possible outcomes: the robot may turn left or right 90°, may move diagonally, *etc.*. The uncertainty in the turn actions was defined by $\texttt{turnUnder} = \texttt{turnOver} = (1-p)/2$.

Partial sinks

A set of locations may be specified as "partially absorbing"; in this case, the transitions from the four states at that location have a modified probability that the `go` action will leave the robot in the same state. The remainder of the probability mass is distributed with the same relative values as specified by the rules above. That is, if τ is the transition function when no sinks are specified, and state s is given "sinkness" x, the new transition function τ' is described by:

$$\tau'(s, \mathsf{go}, s) = x$$

$$\forall s' \in \mathcal{S}, \tau'(s, \mathsf{go}, s') = \tau'(s, \mathsf{go}, s') \times \left(\frac{1-x}{1-\tau(s, \mathsf{go}, s)} \right) .$$

In the special case where $\tau(s, \mathsf{go}, s) = 1$, the transitions for s are left unchanged. Specifying the sinkness of a location as 1 produces a recurrent set consisting of the four states at that location.

The sinks parameter varied from 0 to 3. At 0, there were no sinks. At 1, 5 states were given a sinkness of 0.995, and 10 states were given a sinkness of 0.95. These numbers of states are for the 1x1 worlds; in larger worlds, commensurately more states are turned into sinks. The sinks parameter is linear in the number of sinks, so at sink level 2, 10 states have a sinkness of 0.995, *etc.*. The set of sinks at any sinks level is a subset of the sinks at higher sinks levels, to ensure that the "difficulty" of the task was increasing monotonically with the parameter.

Task

The task given the robot is to minimize the number of steps taken to reach a particular state g of the world from an initial state s_0. This is encoded by defining the termination states as $\mathcal{X} = \{g\}$, the start state as specified, and the reward function ρ by

$$\rho(s,a) = \begin{cases} -1 & \text{if } s \neq g \\ 0 & \text{if } s = g \end{cases} .$$

That is, the agent receives -1 reward for every action taken until it reaches the goal; thereafter it receives no reward. Thus the agent is trying to maximize the expected discounted sum of the instantaneous rewards obtained until the environment occupies the goal state. Clearly, if the discounting factor were 1, the performance measure would be exactly as desired, since there is no "economic" discounting – a time step now is the same as a time step later, and the system will not be halted unless the agent reaches the goal. However, convergence of the policy iteration algorithm is not guaranteed in the case of $\gamma = 1$, so we choose γ very close to 1; in practice, $1 - 10^{-6}$. This is therefore an approximation of the true performance measure, but in practice it is an accurate approximation.

B.3.3 Racetrack (RA)

Motivation

The racetrack domains are based on a game called Race Track described by Martin Gardner [Gar73], and adapted to the Markov decision problem formalism by Barto, Bradtke and Singh [BBS93]. The problem models a race car moving on a track discretized into a grid; a state of the world encodes the location of the car and its velocity vector. The agent controls the race car's acceleration, not its speed directly. If the race car ever crosses a track boundary other than the goal line, it is immediately returned to the starting position with zero velocity.

This class of problems is mainly interesting because it incorporates a notion of inertia — the agent does not always have the option of taking a safe action that will result in no great loss of performance.

Domain

The domain is defined by a bitmap specifying the layout of the track, the starting cells and the goal cells. Figure B.4 shows an example of such a track, with the starting line in black on the left, and the goal line in black on the right. The track is 3 cells wide in most places.

The tracks ranged in size from 10 to 1000 track cells, and I loosely divided them into three categories: Grand-prix tracks have basically a single (though possibly wide) path from the start to the goal, with a number of curves requiring slow speeds. Geometric tracks are straight lines or simple shapes. Rally tracks have several possible paths, with tradeoffs between width and curvature, for example. Finally, random tracks are half-density randomly placed track cells in a background of off-track cells, with a single start and goal cell placed roughly opposite each other. I manually scanned the random tracks for connectivity, and eliminated any in which the goal was not reachable from the start state.

Figure B.5 shows a portion of a track with a car at position $(2, 9)$. The states are represented by the location of the race car and its integer velocity; the state shown in the upper part of the figure is L=(2,9),V=(1,-3). The actions available to the agent are to accelerate, decelerate or coast in both X and Y directions, written (-1,-1), (-1, 0), *etc.*. In the figure, if the agent takes action (1,0), the new velocity is calculated: $(1, -3) + (1, 0) = (2, -3)$; this is used to determine the amount by which the car moves: $(2, 9) + (2, -3) = (4, 6)$. The new state is therefore L=(4,6),V=(2,-3). A measure of uncertainty governs the actions of the agent; with probability `success`, the new state will be chosen as described above, but with probability $1 -$ `success`, the acceleration will not take effect, and the new state will be computed using the old velocity. In the example above, this would lead to state L=(3,6),V=(1,-3). To get an idea of the problem facing the agent, imagine driving a race car around a track where you are allowed to accelerate or brake or turn only at discrete intervals, and where some of the time when you step on the brake, the car doesn't slow down.

The path between the old location of the car and the new location is checked for validity — if it passes through any grid cell that is not part of the track, the car is warped back to a randomly chosen cell on the starting line, and its velocity is set to 0.

Task

The task is to cross the goal line after as few actions as possible. If the agent makes a transition from a state at location (x_1, y_1) to a state at location (x_2, y_2), and if the line $(x_1, y_1), (x_2, y_2)$ passes through one of the goal locations (x_g, y_g), and if none of the locations on the line between (x_1, y_1) and (x_g, y_g) is off the track, the new state is terminating.

This is implemented with a single terminating state; each action is checked for the above conditions, and if they hold, the transition is made to the terminating state. The reward at each step is -1, unless the agent reaches the terminating state, in which case it is 0.

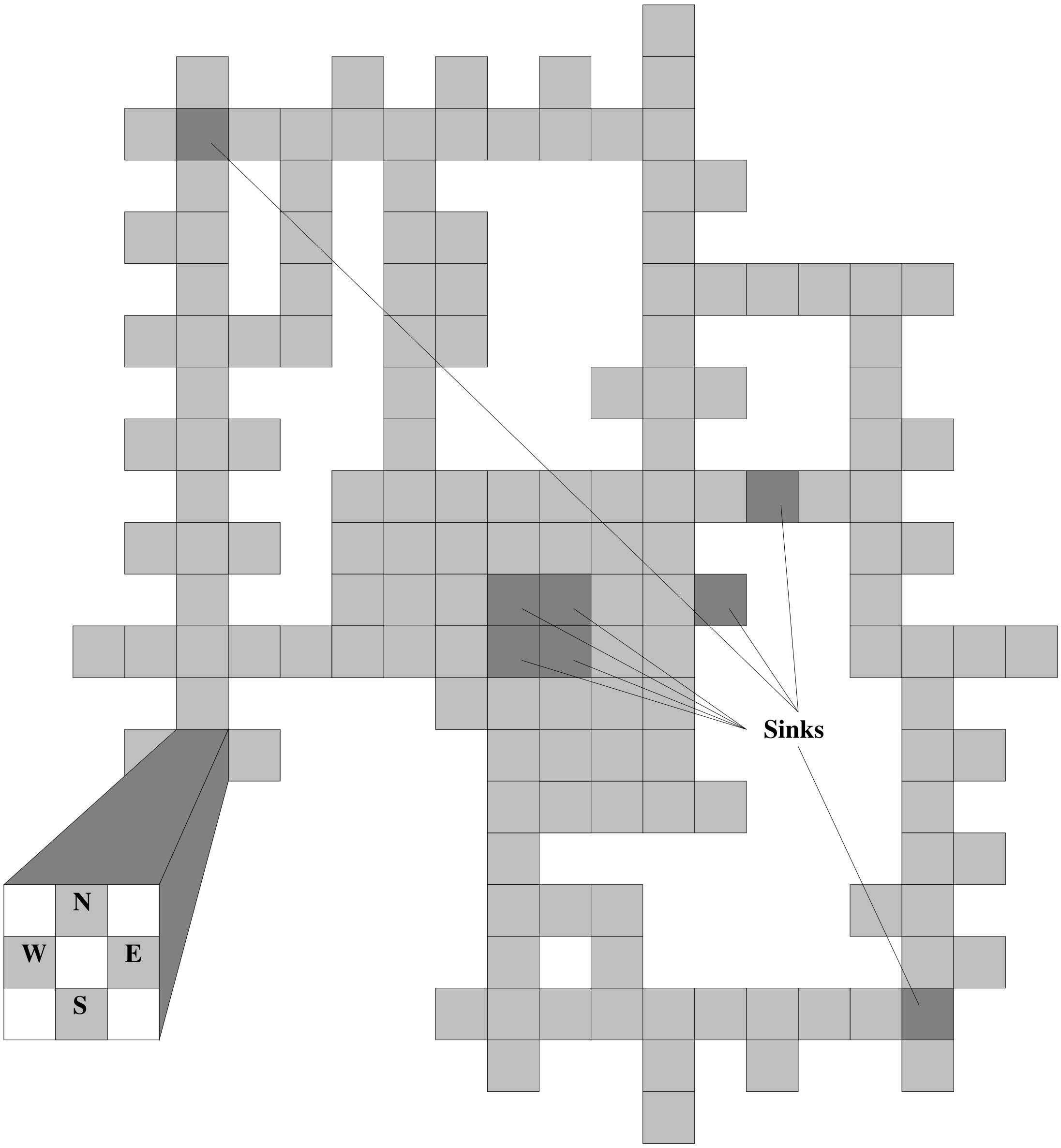


Figure B.3: Spatial layout of the 1x1 RN2 problem

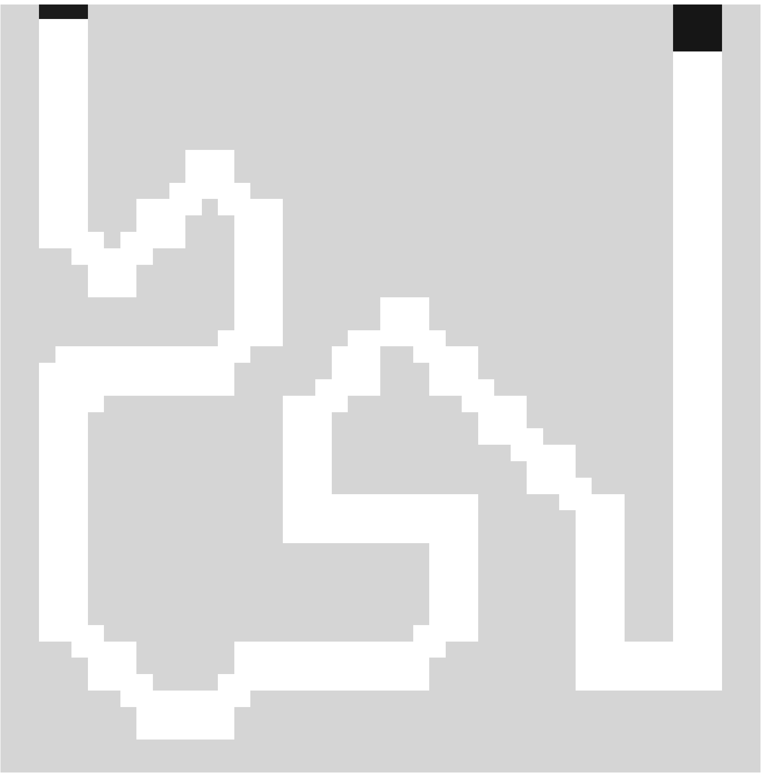


Figure B.4: Racetrack GP-549

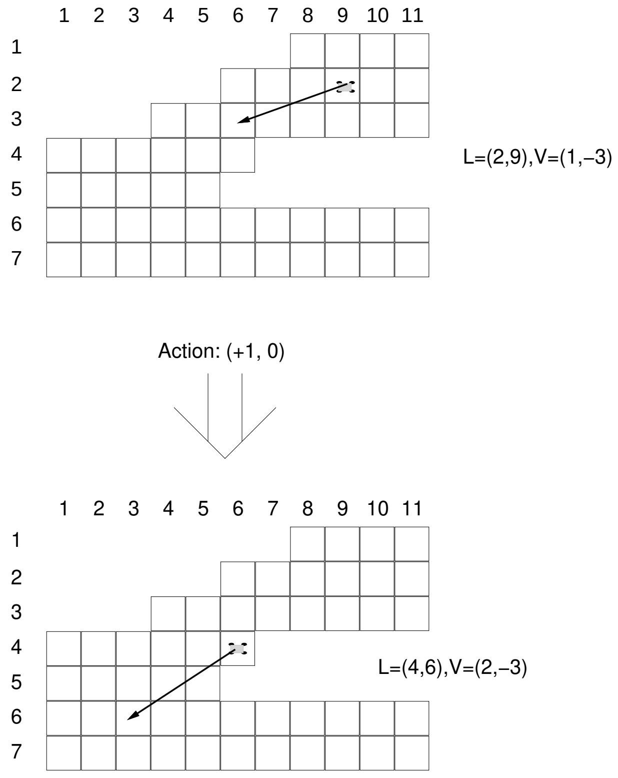


Figure B.5: Actions in the racetrack domains

Bibliography

[AC87] P. E. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, 1987.

[Asp49] A. A. Aspin. *Biometrika*, 36(290), 1949.

[Bas92] Kenneth Basye. A framework for map construction. Technical report, Brown University Department of Computer Science, Providence, RI, 1992.

[BBS93] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Submitted to AIJ Special Issue on Computational Theories of Interaction and Agency*, 1993.

[BC89] D. P. Bertsekas and D. A. Castanon. Adaptive aggregation for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34(6):589–598, 1989.

[BD94] Craig. Boutilier and Richard Dearden. Using abstractions for decision theoretic planning with time constraints. In *Proceedings AAAI-94*. AAAI, 1994.

[Bel57] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[Ber76] Dimitri P. Bertsekas. *Dynamic Programming and Stochastic Control.* Academic Press, 1976.

[Bro85] Rodney Brooks. *A Robust Layered Control System for a Mobile Robot.* Massachusetts Institute of Technology Artificial Intelligence Laboratory, MITAIL, September 1985.

[CKL94a] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, 1994.

[CKL94b] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Algorithms for partially observable markov decision processes. Technical Report Forthcoming, Brown University, Providence, Rhode Island, 1994.

[DB88] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings AAAI-88*, pages 49–54. AAAI, 1988.

[DB90] Mark Drummond and John Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings AAAI-90*, pages 138–144. AAAI, 1990.

[DCK90] Thomas Dean, Theodore Camus, and Jak Kirman. Sequential decision making for active perception. In *Proceedings of the DARPA Image Understanding Workshop*, pages 889–894. DARPA, 1990.

[Der70] Cyrus Derman. *Finite State Markovian Decisian Processes.* Academic Press, 1970.

[DHW94] Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the AAAI Spring Symposium on Decision Theoretic Planning*, 1994.

[DKKN93a] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Deliberation scheduling for time-critical sequential decision making. In *Eighth Conference on Uncertainty in Artificial Intelligence*, 1993.

[DKKN93b] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning with deadlines in stochastic domains. In *AAAI-93.* AAAI, 1993.

[DKKN94] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence, in press*, 1994.

[Gar73] Martin Gardner. Mathematical games. *Scientific American*, (228:108), January 1973.

[Gre93] William H. Greene. *Econometric analysis.* Macmillan, New York, New York, 1993.

[Heg94] Matthias Heger. Consideration of risk in reinforcement learning. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 105–111, San Francisco, CA, 1994. Morgan Kaufmann.

[How66] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, 1966.

[HS82] Daniel P. Heyman and Matthew J. Sobel. *Stochastic Models in Operations Research (Volume II).* McGraw-Hill, 1982.

[JF92] A. Jalali and M. J. Ferguson. Computationally efficient algorithms for on-line optimization of markov decision processes. *Automatica*, 28(1):107–118, 1992.

[JS88] Mark Jerrum and Alistair Sinclair. Conductance and the rapid mixing property for markov chains: the approximation of the permanent resolved. *?*, 1988.

[KBD91] Jak Kirman, Kenneth Basye, and Thomas Dean. Sensor abstractions for control of navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2812–2817, 1991.

[KHW94] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. In *Proceedings AAAI-94*. AAAI, 1994.

[KK71] H. J. Kushner and A. J. Kleinman. Mathematical programming and the control of markov chains. *International Journal of Control*, 13(5):801–820, 1971.

[Kno91] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings AAAI-91*, pages 686–691. AAAI, 1991.

[Kor87] Richard E. Korf. Real-time heuristic search: First results. pages 133–138, 1987.

[KS60] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand, New York, 1960.

[KS76] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Springer-Verlag, 1976.

[KT74] Samuel Karlin and Howard M. Taylor. *A First Course in Stochastic Processes*. Academic Press, Sand Diego, California, second edition edition, 1974.

[LD94] Shieu-Hong Lin and Thomas Dean. Exploiting locality in temporal reasoning. In E. Sandewall and C. Backstrom, editors, *Current Trends in AI Planning*, Amsterdam, 1994. IOS Press.

[LDW89] John J. Leonard and Hugh F. Durrant-Whyte. Active sensor control for mobile robotics. Technical Report OUEL-1756/89, Oxford University Robotics Research Group, 1989.

[MA93] Andrew W. Moore and Christopher G. Atkeson. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In *Advances in Neural Information Processing 5*, San Mateo, California, 1993. Morgan Kaufmann.

[MDS93a] David Musliner, Eduund Durfee, and Kang Shin. Circa: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.

[MDS93b] David Musliner, Eduund Durfee, and Kang Shin. World modeling for the dynamic construction of real-time control plans. *submitted to AIJ*, 1993.

[Mih89] Milena Mihail. Conductance and convergence of markov chains — a combinatorial treatment of expanders. *Proceedings of the 31st ACM Symposium on Foundations of Computer Science*, pages 526–531, 1989.

[NK94] Ann Nicholson and Leslie Pack Kaelbling. Toward approximate planning in very large stochastic domains. In *Proceedings of the AAAI Spring Symposium on Decision Theoretic Planning*, Stanford, California, 1994.

[Put94] M. L. Puterman. *Markov Decision Processes.* John Wiley & Sons, New York, 1994.

[PW93] Jing Peng and Ronald J. Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4):437–454, 1993.

[R8́4] Alfréd Rényi. *A Diary on Information Theory.* John Wiley and Sons, New York, New York, 1984.

[Ros70] Sheldon M. Ross. *Applied Probability Models with Optimization Applications.* Holden-Day, 1970.

[San88] James C. Sanborn. A model of reaction for planning in dynamic environments. 1988.

[Sch87] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[Sch93] Anton Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference in Machine Learning*, pages 298–305, June 1993.

[Sci93] Statistical Sciences. *S-Plus Programmer's manual, Version 3.2.* StatSci, a division of MathSoft Inc., 1993.

[SPS94] Michael I. Jordan Satinder P. Singh, Tommi Jaakkola. Model-free reinforcement learning for non-markovian decision problems. 1994.

[Sut90] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of Machine Learning*, pages 216–224, 1990.

[The61] H. Theil. *Economic Forecasts and Policy.* North-Holland, 1961.

[Wat89] C. J. C. H Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

[WB93] Ronald J. Williams and Leemon C. III Baird. Tight performance bounds on greedy policies based on imperfect value functions. Technical Report Technical Report NU-CCS-93-13, Northeastern University, College of Computer Science, Boston, MA, November 1993.

[Wei73] Alan J. Weir. *Lebesgue Integration and Measure.* Cambridge University Press, 1973.

[Wil85] David E. Wilkins. Recovering from execution errors in sipe. *Computational Intelligence*, 1(1), February 1985.