Stochastic Local Search for POMDP Controllers

by

Darius Braziunas

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Stochastic Local Search for POMDP Controllers

Darius Braziunas

Master of Science

Graduate Department of Computer Science

University of Toronto

2003

Gradient-based search in the space of policies representable as stochastic finite state controllers is one of the few tractable solution methods for non-trivial partially observable Markov decision processes (POMDPs). In this thesis, we illustrate a basic problem with standard gradient ascent applied to POMDPs, where the sequential nature of the decision problem is at issue, and propose a new stochastic local search method as an alternative. Our method employs certain heuristics that mimic some of the sequential reasoning inherent in dynamic programming approaches; while more computationally demanding, it can find good, even optimal, controllers where gradient-based methods commonly converge to poor local suboptima.

Acknowledgements

I would like to thank Dr. Craig Boutilier for his support and guidance, and Dr. Fahiem Bacchus for being the second reader.

Contents

Chapter 1

Introduction

Partially observable Markov decision processes (POMDPs) provide a natural model for sequential decision making under uncertainty. This model augments a well-researched framework of Markov decision processes (MDPs) [How60, Put94] to situations where an agent cannot reliably identify the underlying environment state. The POMDP formalism is very general and powerful, extending the application of MDPs to many realistic problems.

Unfortunately, the generality of POMDPs entails high computational cost. The problem of finding optimal policies for finite-horizon POMDPs has been proven to be PSPACE-complete [PT87], and their existence for infinite-horizon POMDPs — undecidable [MHC99]. Because of the intractability of current solution algorithms, especially those that use dynamic programming to construct (approximately) optimal value functions [SS73, CLZ97], the application of POMDPs remains limited to very small problems.

Policy-based solution methods search *directly* in the space of policies for the best course of action. Constraining the policy space facilitates the search and may lead tractable (although approximate) POMDP solution algorithms. *Finite-state controllers* (FSCs) are the policy representation of choice in such work, providing a compromise between the requirement that action choices depend on certain aspects of observable history and the ability to easily control the complexity of policy space being searched.

While optimal FSCs can be constructed if no restrictions are placed on their structure [Han98a], it is more usual to impose some structure that one hopes admits a good parameterization, and search through that restricted space. In this thesis, we study the problem of finding the best FSC *of a given size* for a completely specified POMDP. Even with the FSC size restriction constraint, the problem remains NP-hard [Lit94, MKKC99]; therefore, *gradient ascent (GA)* has proven to be especially attractive for solving this type of problems because of its computational properties [MKKC99, AB02].

One difficulty with gradient-based approaches, not surprisingly, is the ease with which they converge to suboptimal local optima. Our experiences with GA, specifically, have demonstrated it leads to poor local optima in problems where the *precise sequence* of actions taken is important to good performance. This is a common feature of stochastic planning problems to which POMDPs are often applied; they have very different characteristics from grid-world and other navigational problems on which GA has often been tested. While various restrictions on policy space can be used to encode prior knowledge about a problem's solution [MKKC99], such restrictions may be hard to encode naturally, and such knowledge may be hard to come by.

In this thesis, we attempt to overcome the existence of local optima of this type, while remaining within the "local search" framework. We propose a stochastic local search (SLS) technique that works in the space of FSCs, like GA, but which uses very different heuristics to evaluate moves. In particular, it incorporates intuitions—used in the dynamic programming solution to POMDPs that work in belief-state value function space—that allow moves in different directions that those permitted by simple GA. While our methods are more computationally intensive, they provide a good compromise between full dynamic programming updates and optimal search techniques like branch-and-bound, and the very restricted form of local search admitted by GA.

Our algorithm, like GA, scales well with the size of the problem; however, its complexity is directly related to the size of the FSC. Therefore, we expect that our approach will work well for large POMDPs where relatively simple policies achieve near-optimal values. Because we are searching directly in the policy space, our solution also provides a convenient form of policy for online execution.

The rest of the thesis is structured as follows. Chapter 2 provides an overview of POMDPs, with explicit emphasis on policies and solution methods that work directly in the space of policies. It introduces the POMDP value and policy iteration as well as gradient ascent algorithms, which provide the background for our SLS algorithm. Chapter 3 describes the SLS algorithm in detail, explaining main intuitions and motivations for its various parts. Chapter 5 illuminates various aspects of the SLS algorithm and compares its empirical performance to GA on several examples drawn from the research literature.

Chapter 2

POMDP solution methods

This chapter provides an overview of partially observable Markov decision processes (POMDPs), concentrating on concepts that will be used later to explain our stochastic local search procedure. The material is structured in a way that emphasizes the notion of policies and solution methods that work directly in the space of policies.

2.1 Sequential decision processes

A sequential decision process involves an *agent* that interacts synchronously with the external *environment*, or system; the agent's goal is to maximize *reward* by choosing appropriate actions. These actions and the history of the environment *states* determine the probability distribution over possible next states. Therefore, the sequence of system states can be modeled as a stochastic process.

2.1.1 MDP framework

The most commonly used formal model of fully-observable sequential decision processes is the Markov decision process (MDP) model. An MDP can be viewed as an extension of Markov chains with a set of decisions (actions) and a state-based reward or cost structure. For each possible state of the process, a decision has to be made regarding which action should be executed in that state. The chosen action affects both the transition probabilities and the costs (or rewards) incurred. The goal is to choose an optimal action in every state to increase some predefined measure of performance. The *decision* process for doing this is referred to as the Markov *decision* process.

Actions and state transitions

A state is a description of the environment at a particular point in time. Although we will deal with continuous state and action spaces when describing preference elicitation problems, we generally assume that the environment can be in a finite number of states, and the agent can choose from a finite set of actions. Let $\mathcal{S} = \{s_0, s_1, \ldots, s_N\}$ be a finite set of states. Since the process is stochastic, a particular state at some discrete *stage*, or time step $t \in \mathcal{T}$, can be viewed as a random variable S^t whose domain is the state space $\mathcal{S}$.

For a process to be *Markovian*, the state has to contain enough information to predict the next state. This means that the past history of system states is irrelevant to predicting the future:

$$Pr(S^{t+1}|S^0, S^1, \ldots, S^t) = Pr(S^{t+1}|S^t). \tag{2.1}$$

At each stage, the agent can affect the state transition probabilities by executing one of the available actions. The set of all actions will be denoted by $\mathcal{A}$. Thus, each action $a \in \mathcal{A}$ is fully described by a $|\mathcal{S}| \times |\mathcal{S}|$ *state transition* matrix, whose entry in an ith row and jth column is the probability that the system will move from state s_i to state s_j if action a gets executed:

$$p^a_{ij} = Pr(S^{t+1} = s_j|S^t = s_i, A^t = a). \tag{2.2}$$

We will assume that our processes are *stationary*, i.e., that the transition probabilities do not depend on the current time step.

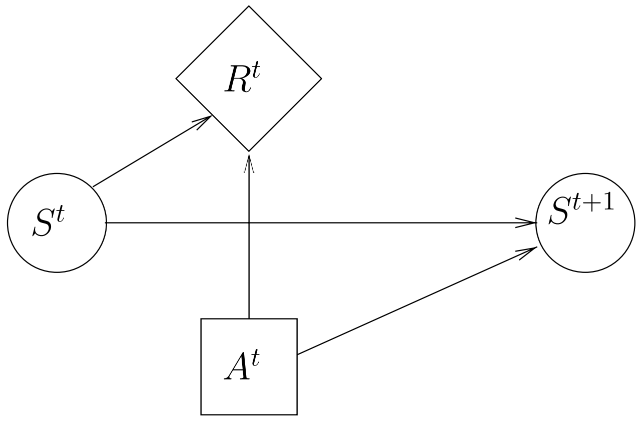


Figure 2.1: Causal relationships between MDP states, actions, and rewards. R^t is reward received at stage t, i.e., $R(S^t, A^t)$.

The transition function $T(\cdot)$ summarizes the effects of actions on systems states. $T : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{S})$ is a function that for each state and action associates a probability distribution over the possible successor states ($\Delta(\mathcal{S})$ denotes the set of all probability distributions over $\mathcal{S}$). Thus, for each $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$, the function T determines the probability of a transition from state s to state s' after executing action a, i.e.,

$$T(s, a, s') = Pr(S^{t+1} = s' | S^t = s,\ A^t = a). \tag{2.3}$$

Rewards

$R : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is a reward function that for each state and action assigns a numeric reward (or cost, if the value is negative). $R(s, a)$ is an immediate reward that an agent would receive for being in state s and executing action a.

The causal relationships between MDP states, actions, and rewards are illustrated in Figure 2.1.

2.1.2 POMDP framework

A POMDP is a generalization of MDPs to situations in which system states are not fully observable. This realistic extension of MDPs dramatically increases the complexity of POMDPs, making exact solutions virtually intractable. In order to act optimally, an agent might need to take into account all the previous history of observations and actions, rather than just the current state it is in.

A POMDP is comprised of an underlying MDP, extended with an observation space $\mathcal{O}$ and observation function $Z(\cdot)$.

Observation function

Let $\mathcal{O}$ be a set of observations an agent can receive. In MDPs, the agent has full knowledge of the system state; therefore, $\mathcal{O} \equiv \mathcal{S}$. In partially observable environments, observations are only probabilistically dependent on the underlying environment state. Determining which state the agent is in becomes problematic, because the same observation can be observed in different states.

$Z : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{O})$ is an observation function that specifies the relationship between system states and observations. $Z(s', a, o')$ is the probability that observation o' will be recorded after an agent performs action a and lands in state s':

$$Z(s', a, o') = Pr(O^{t+1} = o' \mid S^{t+1} = s',\, A^t = a). \tag{2.4}$$

Formally, a POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \mathcal{O}, Z \rangle$, consisting of the state space $\mathcal{S}$, action space $\mathcal{A}$, transition function $T(\cdot)$, reward function $R(\cdot)$, observation space $\mathcal{O}$, and observation function $Z(\cdot)$. Its influence diagram is shown in Figure 2.2.

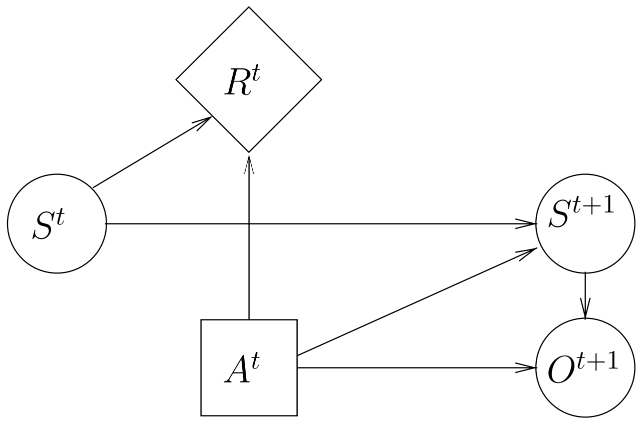


Figure 2.2: Causal relationships between POMDP states, actions, rewards, and observations.

2.1.3 Process histories

A *history* is a record of everything that happened during the execution of the process. For POMDPs, a complete system history from the beginning till time t is a sequence of state, observation, and action triples

$$\langle S^0, O^0, A^0\rangle, \langle S^1, O^1, A^1\rangle, \ldots, \langle S^t, O^t, A^t\rangle. \tag{2.5}$$

The set of all complete histories (or *trajectories*) will be denoted as $\mathcal{H}$.

Since rewards depend only on visited states and executed actions, a *system history* is enough to evaluate an agent's performance. Thus, a system history is just a sequence of state and action pairs:

$$\langle S^0, A^0\rangle, \langle S^1, A^1\rangle, \ldots, \langle S^t, A^t\rangle. \tag{2.6}$$

A system history h from the set of all system histories $\mathcal{H}_s$ provides an external, objective view about the process; therefore, value functions will be defined on the set $\mathcal{H}_s$ in the next subsection.

In a partially observable environment, an agent cannot fully observe the underlying world state; therefore, it can only base its decisions on the *observable* history. Let's assume that at the outset, the agent has prior beliefs about the world that are summarized by the probability distribution b_0 over the system states; the agent starts by executing some action a^0 based solely on b_0. The observable history until time step t is then a sequence of action and observation pairs

$$\langle A^0, O^1 \rangle, \langle A^1, O^2 \rangle, \ldots, \langle A^{t-1}, O^t \rangle. \tag{2.7}$$

The set of all possible observable histories will be denoted as $\mathcal{H}_o$. Different ways of structuring and representing $\mathcal{H}_o$ have resulted in different POMDP solution and policy execution algorithms. The concept of observable history and a closely related notion of internal memory will be central to issues addressed in this thesis.

2.1.4 Performance measures

At each step in a sequential decision process, an agent has to decide what action to perform based on its observable history. A policy $\pi : \mathcal{H}_o \mapsto \mathcal{A}$ is a rule that maps observable trajectories into actions. A given policy induces a probability distribution over all possible sequences of states and actions, for an initial distribution b_0. Therefore, an agent has control over the likelihood of particular system trajectories. Its goal is to choose a policy that maximizes some objective function that is defined on the set of system histories $\mathcal{H}_s$.

Such objective function is called a *value* function $V(\cdot)$; it essentially ranks system trajectories by assigning a real number to each $h \in \mathcal{H}_s$; a system history h is preferred to h' if and only if $V(h) > V(h')$. Formally, a value function is a mapping from the set of system histories into real numbers:

$$V : \mathcal{H}_s \mapsto \mathbb{R}. \tag{2.8}$$

In most MDP and POMDP formulations found in AI literature, the value function $V(\cdot)$ is assumed to have structure that makes it much easier to represent and evaluate. In this thesis, we will assume that $V(\cdot)$ is *additive* – the value of a particular system history is simply a *sum* of rewards accrued at each time step.

If the decision process stops after a finite number of steps H, the problem is *a finite horizon* problem. In such problems, it is common to maximize the total expected reward. The value function for a system trajectory h of length H is simply the sum of rewards attained at each stage [Bel57]:

$$V(h) = \sum_{t=0}^{t=H} R(s^t, a^t). \tag{2.9}$$

The sum of rewards over an infinite trajectory may be unbounded. A mathematically elegant way to address this problem is to introduce a *discount factor* γ; the rewards received later get discounted, and contribute less than current rewards. The value function for a total discounted reward problem is [Bel57]:

$$V(h) = \sum_{t=0}^{\infty} \gamma^t \, R(s^t, a^t), \; 0 \leq \gamma < 1. \tag{2.10}$$

This formulation is very common in current MDP and POMDP literature, including the key papers concerning policy-based search in POMDPs [Han97, Han98a, MKKC99, MPKK99]. We will use it as our performance measure in all the problems in this thesis. Another popular value function is the average reward per stage[1], used, e.g., in [AB02].

2.2 Policy representations

Generally, an agent's task is to calculate the optimal course of action in an uncertain environment and then execute its plan contingent on the history of its sensory inputs. The criterion of optimality is predetermined; in this thesis, we will use the infinite horizon discounted sum of rewards model, described above. The agent's behavior is therefore determined by its *policy* π, which in its most general form is a mapping from the set of *observable* histories to actions:

$$\pi : \mathcal{H}_o \mapsto \mathcal{A} \tag{2.11}$$

[1] $V(h) = \lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n} R(s^t, a^t)$.

Given a history

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

the action prescribed by the policy π at time t would be $a^t = \pi(h^t)$; a^0 is the agent's initial action, and o^t is the latest observation.

One of the more important concepts is that of an *expected policy value.* Taking into account a prior belief distribution over the system states b_0, a policy induces a probability distribution $Pr(h|\pi, b_0)$ over the set of system histories $\mathcal{H}_s$. The expected policy value is simply the expected value of system trajectories induced by the policy π:

$$EV(\pi) \equiv V^\pi = \sum_{h \in \mathcal{H}_s} V(h) Pr(h|\pi, b_0). \tag{2.12}$$

The value of the policy π at a given starting state s_0 will be denoted $V^\pi(s_0)$. Then,

$$EV(\pi) = \sum_{s \in \mathcal{S}} b_0(s)\, V^\pi(s). \tag{2.13}$$

The agent's goal is to find a policy $\pi^* \in \Pi$ with the maximal expected value from the set Π of all possible policies.

The general form of a policy as a mapping from arbitrary observation histories to actions is very impractical. Existing POMDP solution algorithms exploit structure in value and observation functions to calculate optimal policies that have much more tractable representations. For example, observable histories can be represented as probability distributions over system states, or grouped into a finite set of distinguishable classes using finite-suffix trees or finite-state controllers.

2.2.1 MDP policies

A POMDP where an agent can fully observe the underlying system state reduces to an MDP. Since the sequence of states forms a Markov chain, the next state depends only on the current state; the history of the previous states is therefore rendered irrelevant.

Finite horizon policies

For finite horizon MDP problems, the knowledge of the current state and stage is sufficient to represent the whole observable trajectory for the purposes of maximizing total reward (discounted or not). Therefore, a policy π can be reduced to a mapping from states and stages to actions:

$$\pi : \mathcal{S} \times \mathcal{T} \mapsto \mathcal{A}. \tag{2.14}$$

Let $\pi(s,t)$ be the action prescribed by the policy at state s with t stages *remaining* till the end of the process. The expected value of a policy at any state can then be computed by the following recurrence [Bel57]:

$$\begin{aligned} V_0^\pi(s) &= R(s, \pi(s,0)), \\ V_t^\pi(s) &= R(s, \pi(s,t)) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s,t), s')\, V_{t-1}^\pi(s'). \end{aligned} \tag{2.15}$$

The value functions in the set $\{V_t^\pi\}_{0 \le t \le H}$ are called *t-horizon*, or *t-step*, value functions; H is the horizon length — a predetermined number of stages the process goes through.

A policy π^* is *optimal* if $V_H^{\pi^*}(s) \ge V_H^{\pi'}(s)$ for all H-horizon policies π' and all states $s \in \mathcal{S}$. The optimal value function is a value function of an optimal policy: $V_H^* \equiv V_H^{\pi^*}$. A key result, called Bellman's *principle of optimality* [Bel57] allows to calculate the optimal t-step value function from the $(t-1)$-step value function:

$$V_t^*(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s')\, V_{t-1}^*(s') \right]. \tag{2.16}$$

This equation has served as a basis for value-iteration MDP solution algorithms and inspired analogous POMDP solution methods.

Infinite horizon policies

For infinite horizon MDP problems, optimal decisions can be calculated based only on the current system state, since at any stage, there is still an infinite number of time steps remaining. Without loss of optimality, infinite horizon policies can be represented as mappings from states to actions [How60]:

$$\pi : \mathcal{S} \mapsto \mathcal{A}. \tag{2.17}$$

Policies that do not depend on stages are called *stationary* policies.

The value of a stationary policy π can be determined by a recurrence analogous to the finite horizon case:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s')\ V^{\pi}(s'). \tag{2.18}$$

The agent's goal is to find a policy π^* that would maximize the value function $V(\cdot)$ for all states $s \in \mathcal{S}$. The optimal value function is

$$V^*(s) = \max_{a \in \mathcal{A}} \left[R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s')\ V^*(s') \right]. \tag{2.19}$$

Implicit policies

Equations 2.15 and 2.18 show how to find the value of a given policy π and provide the basis for policy-iteration algorithms. The calculation is straightforward and amounts to solving a system of linear equations of size $|\mathcal{S}| \times |\mathcal{S}|$.

On the other hand, value-iteration methods employ Equation 2.16 to calculate optimal value functions directly. Optimal policies can then be defined implicitly by value functions. First, we introduce a notion of a Q-function, or Q-value: $Q(s, a)$ is the value of executing action a at state s, and then following the optimal policy:

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s')\ V^*(s'). \tag{2.20}$$

The optimal infinite horizon policy is a greedy policy with respect to the optimal value function $V^*(\cdot)$:

$$\pi^*(s) = \arg\max_{a}\ Q(s, a). \tag{2.21}$$

Stochastic policies

A stochastic infinite horizon MDP policy is a generalization of a deterministic policy; instead of prescribing a single action to a state, it assigns a *distribution* over all actions to a state. That is, a stochastic policy

$$\psi : \mathcal{S} \mapsto \Delta(\mathcal{A}) \tag{2.22}$$

maps a state to a probability distribution over actions; $\psi(s, a)$ is the probability that action a will be executed at state s. By incorporating expectation over actions, we can rewrite the Equation 2.18 for stochastic policies in a straightforward manner:

$$V^{\psi}(s) = \sum_{a \in \mathcal{A}} \psi(s, a)\, R(s, a) + \gamma \sum_{s',\, a} \psi(s, a)\, T(s, a, s')\, V^{\psi}(s'). \tag{2.23}$$

While stochastic policies have no advantage for infinite horizon MDPs, we will use them in solving partially observable MDPs. Making policies stochastic allows to convert the discrete action space into a continuous space of distributions over actions. We can then optimize the value function using continuous optimization techniques.

2.2.2 POMDP policy trees

In partially observable environments, an agent can only base its decisions on the history of its actions and observations. Instead of a simple mapping from system states to actions, a generic POMDP policy assumes a more complicated form.

As for MDPs, we will first consider finite horizon policies. With one stage left, all an agent can do is to execute an action; with two stages left, it can execute an action, receive an observation, and then execute the final action. For a finite horizon of length H, a policy is a *tree* of height H. Since the number of actions and observations is finite, the set of all policies for horizon H can be represented by a *finite* set of policy trees.

Figure 2.3 illustrates the concept of a t-horizon policy tree. Each node prescribes an action to be taken at a particular stage; then, an observation received determines the

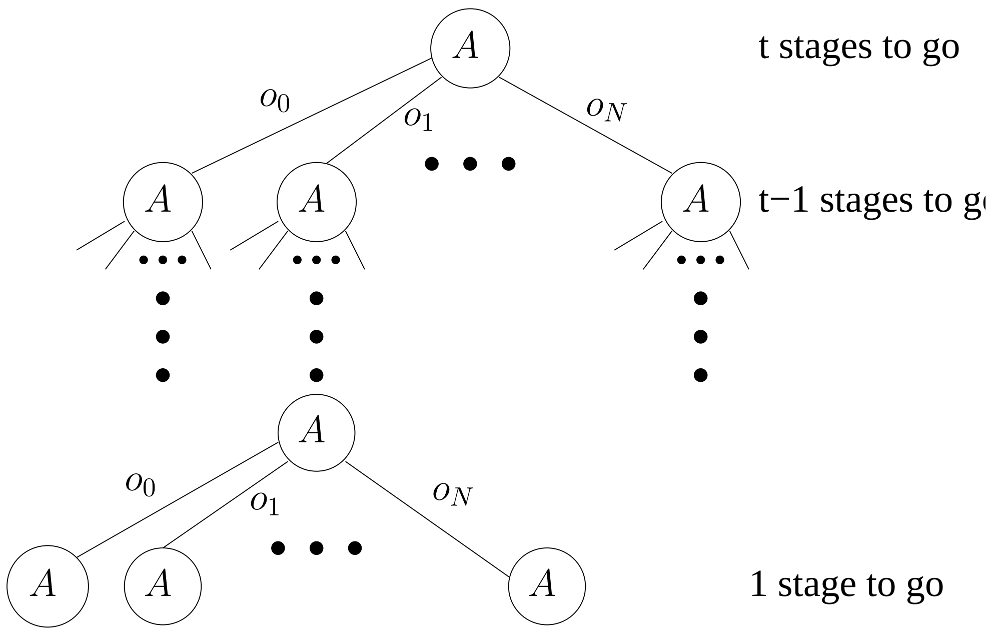


Figure 2.3: A policy tree for horizon t. For each observation, there is a branch to nodes at a lower level. Each node can be labeled with any action from the set $\mathcal{A}$.

branch to follow. A policy tree for a horizon of length H contains

$$\sum_{t=0}^{t=H-1} |\mathcal{O}|^t = \frac{|\mathcal{O}|^H - 1}{|\mathcal{O}| - 1} \tag{2.24}$$

nodes. At each node, there are $|\mathcal{A}|$ choices of actions. Therefore, the size of the set of all possible H-horizon policy trees is

$$|\mathcal{A}|^{\frac{|\mathcal{O}|^H - 1}{|\mathcal{O}|-1}}. \tag{2.25}$$

We will now present a recursive definition of policy trees using an important notion of *conditional plans*. A conditional plan $\sigma \in \Gamma$ is a pair $\langle a, \nu \rangle$ where $a \in \mathcal{A}$ is an action, and $\nu : \mathcal{O} \mapsto \Gamma$ is an *observation strategy*. The set of all observation strategies will be denoted as $\Gamma^{\mathcal{O}}$; obviously, its size is $|\Gamma|^{|\mathcal{O}|}$.

A particular conditional plan tells an agent what action to perform, and what to do next contingent on an observation received. Let Γ_t be the set of all conditional plans available to an agent with t stages left:

$$\Gamma_t = \{\langle a, \nu_t \rangle \mid a \in \mathcal{A},\ \nu_t \in \Gamma_{t-1}^{\mathcal{O}}\}. \tag{2.26}$$

In this case, $\nu_t : \mathcal{O} \mapsto \Gamma_{t-1}$ is a stage-dependent observation strategy. As a tree of height t can be defined recursively in terms of its subtrees of height $t-1$, so the conditional plans of horizon t can be defined in terms of conditional plans of horizon $t-1$. At the last time step, a conditional plan simply returns an action. A policy tree therefore directly corresponds to a conditional plan. We will use the set Γ_t to denote both the set of t-step policy trees and the equivalent set of conditional plans.

Representing policy trees as conditional plans allows us to write down a recursive expression for their value function. The value function of a non-stationary policy π_t represented by a t-horizon conditional plan $\sigma_t = \langle a, \nu_t \rangle$ is

$$\begin{aligned} &V_0^\pi(s) = R(s, \sigma_0(s)), \\ &V_t^\pi(s) = V_t^{\sigma_t}(s) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') \sum_{o \in \mathcal{O}} Z(s',a,o)\, V_{t-1}^{\nu_t(o)}(s'), \end{aligned} \tag{2.27}$$

where $\sigma_0(s)$ is the action to be executed at the last stage.

Since the actual system state is not fully known, we need to calculate the value of a particular policy tree with respect to a (initial) belief state b. Such value is just an expectation of executing the conditional plan σ_t at each state $s \in \mathcal{S}$:

$$V_t^\pi(b) = V_t^{\sigma_t}(b) = \sum_{s \in \mathcal{S}} b(s)\, V_t^{\sigma_t}(s). \tag{2.28}$$

The optimal t-step value function for the belief state b can be found simply by enumerating all the possible policy trees in the set Γ_t:

$$V_t^*(b) = \max_{\sigma \in \Gamma_t} \sum_{s \in \mathcal{S}} b(s)\, V_t^\sigma(s). \tag{2.29}$$

Thus, the t-step value function for the continuous belief simplex $\mathcal{B}$ can in principle be represented by a finite (although doubly exponential in t!) set of conditional plans and a max operator. The next section discusses some ways of making such a representation more tractable.

2.2.3 α-vectors and belief state MDPs

The previous Equation 2.29 actually illustrates the fact that the optimal t-step POMDP value function is piecewise linear and convex [Son71, Son78]. From Equation 2.28 we can see that the value of any policy tree V_t^σ is linear in b; hence, from Equation 2.29, V_t^* is simply the upper surface of the collection of value functions of policies in Γ_t.

Let α^σ be a vector of size $|\mathcal{S}|$ whose entries are the values of the conditional plan σ (or, values of a policy tree corresponding to σ) for each state s:

$$\alpha^\sigma = [V^\sigma(s_0), V^\sigma(s_1), \ldots, V^\sigma(s_N)]. \tag{2.30}$$

Equation 2.29 can then be rewritten in terms of α-vectors :

$$V_t^*(b) = \max_{\sigma \in \Gamma_t} \sum_{s \in \mathcal{S}} b(s)\, \alpha^\sigma(s) = \max_{\alpha \in \mathcal{V}_t} \sum_{s \in \mathcal{S}} b(s)\ \alpha(s). \tag{2.31}$$

Here, the set $\mathcal{V}_t$ contains all t-step α-vectors ; these vectors correspond to t-step policy trees and are sufficient to define the optimal t-horizon value function.

The optimal value function V_t is represented by the upper surface of the α-vectors in $\mathcal{V}_t$ (see Figure 2.4). Although in the worst case any policy in Γ_t might be superior for some belief region, this rarely happens in practice. Many vectors in the set $\mathcal{V}_t$ might be *dominated* by other vectors, and therefore not needed to represent the optimal value function. In Figure 2.4, vector α_3 is *pointwise* dominated by α_1, whereas vector α_1 is jointly dominated by the useful vectors α_0 and α_2 together.

Given the set of all α-vectors $\mathcal{V}_t$, it is possible to *prune* it down to a *parsimonious* subset $\mathcal{V}_t^-$ that represents the same optimal value function V_t^*:

$$V_t^*(b) = \max_{\alpha \in \mathcal{V}_t} \sum_{s \in \mathcal{S}} b(s)\ \alpha(s) = \max_{\alpha \in \mathcal{V}_t^-} \sum_{s \in \mathcal{S}} b(s)\ \alpha(s). \tag{2.32}$$

In a parsimonious set, all α-vectors (or corresponding policy trees) are *useful* [KLC98]. A vector α is useful if there is a non-empty belief region $\mathcal{R}(\alpha, \mathcal{V})$ over which it dominates all other vectors, where

$$\mathcal{R}(\alpha, \mathcal{V}) = \{b \mid b \cdot \alpha > b \cdot \alpha',\ \alpha' \in \mathcal{V} - \{\alpha\},\ b \in \mathcal{B}\}. \tag{2.33}$$

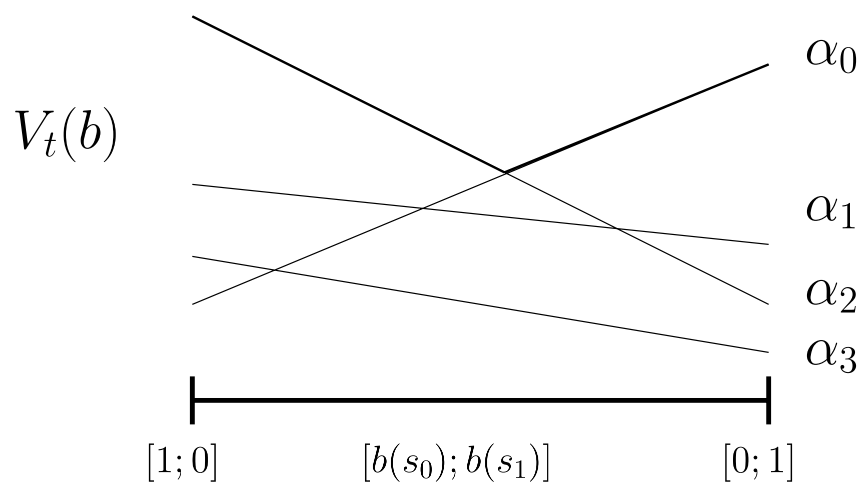


Figure 2.4: For a two-state POMDP, the belief space $\mathcal{B}$ is a one-dimensional unit interval, since $b(s_0) = Pr(s_0) = 1 - Pr(s_1)$. The horizontal axis therefore represents the whole belief space $\mathcal{B}$ on which the value function $V_t(b)$ is defined. $V_t(b)$ is the upper surface of four α-vectors . Only two of them, α_0 and α_2, are useful.

The existence of such region can be easily determined using linear programming. Various value-based POMDP solution algorithms differ in their methods of pruning the set of all α-vectors $\mathcal{V}_t$ to a parsimonious subset $\mathcal{V}_t^-$.

Implicit POMDP policies

As we already know, an explicit t-step POMDP policy can be represented by a policy tree or a recursive conditional plan. Given an initial belief state b_0, the optimal t-step policy can be found by going through the set of all useful policy trees and finding the one whose value function is maximal with respect to b_0 (see Equation 2.31). Then, executing the finite horizon policy is straightforward: an agent only needs to perform actions at the nodes, and follow the observation links to policy subtrees.

Instead of keeping all policy trees, it is enough to maintain the set of useful α-vectors $\mathcal{V}_t^-$ for each stage t. As for MDPs, an *implicit* t-step policy can be defined by doing a greedy one-step lookahead. First, we will define the Q-value function $Q_t(b, a)$ as a value of taking action a at belief state b and continuing optimally for the remaining $t-1$ stages:

$$Q_t(b,a) = \sum_{s\in\mathcal{S}} b(s)R(s,a) + \gamma \sum_{o\in\mathcal{O}} Pr(o|a,b)V^*_{t-1}(b^a_o)), \tag{2.34}$$

where b^a_o is the belief state that results from b after taking action a and receiving observation o. As we will see below, it can be calculated using the POMDP model and Bayes' theorem.

The optimal action to take at b with t stages remaining is simply

$$\pi^*(b,t) = \arg\max_{a\in\mathcal{A}} Q_t(b,a). \tag{2.35}$$

Belief state MDPs

A finite horizon POMDP policy now becomes a mapping from belief states and stages to actions:

$$\pi : \mathcal{B} \times \mathcal{T} \mapsto \mathcal{A}. \tag{2.36}$$

Astrom has shown that a properly updated probability distribution over the state space $\mathcal{S}$ is sufficient to summarize all the observable history of a POMDP agent without loss of optimality [Ast65]. Therefore, a POMDP can be cast into a framework of a fully observable MDP where belief states comprise the continuous, but fully observable, MDP state space. A belief state MDP is therefore a quadruple $\langle \mathcal{B}, \mathcal{A}, T^b, R^b \rangle$, where

- $\mathcal{B} = \Delta(\mathcal{S})$ is the continuous state space.
- $\mathcal{A}$ is the action space, which is the same as in the original POMDP.
- $T^b : \mathcal{B} \times \mathcal{A} \mapsto \mathcal{B}$ is the belief transition function:

$$\begin{aligned} T^b(b,a,b') &= Pr(b'|b,a) \\ &= \sum_{o\in\mathcal{O}} Pr(b'|a,b,o)\, Pr(o|a,b) \\ &= \sum_{o\in\mathcal{O}} Pr(b'|a,b,o) \sum_{s'\in\mathcal{S}} Z(s',a,o) \sum_{s\in\mathcal{S}} T(s,a,s')\, b(s), \end{aligned} \tag{2.37}$$

where

$$Pr(b'|a,b,o) = \begin{cases} 1 & \text{if } b_o^a = b', \\ 0 & \text{otherwise.} \end{cases} \tag{2.38}$$

After action a and observation o, the updated belief b_o^a can be calculated from the previous belief b:

$$b_o^a(s') = \frac{Z(s',a,o)\, \sum_{s\in\mathcal{S}} T(s,a,s')\, b(s)}{Pr(o|a,b)}. \tag{2.39}$$

- $R^b : \mathcal{B} \times \mathcal{A} \mapsto \mathbb{R}$ is the reward function:

$$R^b(b,a) = \sum_{s\in\mathcal{S}} b(s)\, R(s,a). \tag{2.40}$$

To follow the policy that maps from belief states to actions, the agent simply has to execute the action prescribed by the policy, and then update its probability distribution over the system states according to Equation 2.39.

The infinite horizon optimal value function remains convex, but not necessarily piecewise linear, although it can be approximated arbitrarily closely by a piecewise linear and convex function [Son78]. The optimal policy for infinite horizon problems is then just a stationary mapping from belief space to actions:

$$\pi : \mathcal{B} \mapsto \mathcal{A}. \tag{2.41}$$

It can be extracted by performing a greedy one-step lookahead with respect to the optimal value function V^*:

$$\begin{aligned} Q(b,a) &= \sum_{s\in\mathcal{S}} b(s)R(s,a) + \gamma \sum_{o\in\mathcal{O}} Pr(o|a,b)V^*(b_o^a), \\ \pi^*(b) &= \arg\max_{a\in\mathcal{A}} Q(b,a). \end{aligned} \tag{2.42}$$

2.2.4 Finite-state controllers

The optimal infinite horizon value function V^* can be approximated arbitrarily closely by successive finite horizon value functions $V_0, V_1, \ldots, V_t$, as $t \to \infty$ [Son78]. While all optimal t-horizon policies are piecewise-linear and convex, this is not always true for infinite horizon value functions. They remain convex [WH80], but may contain infinitely many facets.

Some optimal value functions do remain piecewise linear; therefore, at some horizon t, the two successive value functions V_t and V_{t+1} are equal, and therefore, optimal:

$$V^* = V_t = V_{t+1}. \tag{2.43}$$

Each vector α in a parsimonious set $\mathcal{V}^*$ that represents the optimal infinite horizon value function V^* has an associated belief space region $\mathcal{R}(\alpha, \mathcal{V}^*)$ over which it dominates all other vectors (see Equation 2.33):

$$\mathcal{R}(\alpha, \mathcal{V}^*) = \{b \mid b \cdot \alpha > b \cdot \alpha',\ \alpha' \in \mathcal{V}^* - \{\alpha\},\ b \in \mathcal{B}\}.$$

Thus, α-vectors define a partition of the belief space. In addition, it has been shown that for each partition there is an optimal action [SS73]. When an optimal value function V^* can be represented by a finite set of vectors, all belief states within one region get transformed to new belief states within the *same* single belief partition, given the optimal action and a resulting observation. The set of partitions and belief transitions constitute a *policy graph*, where nodes correspond to belief space partitions with optimal actions attached, and transitions are guided by observations [CKL94].

Another way of understanding the concept of policy graphs is illustrated in an article by Kaelbling *et al.* [KLC98]. If the finite horizon value functions V_t and V_{t+1} become equal, at every level above t the corresponding conditional plans have the same value. Then, it is possible to redraw the observation links from one level to itself as if it were the succeeding level (see Figure 2.5). Essentially, we can convert non-stationary t-step policy trees (which are non-cyclic policy graphs) into stationary cyclic policy graphs. Such policy graphs enable an agent to execute policies simply by doing actions prescribed at the nodes, and following observation links to successor nodes. The nodes partition the belief space in a way that, for a given action and observation, all belief states in a particular

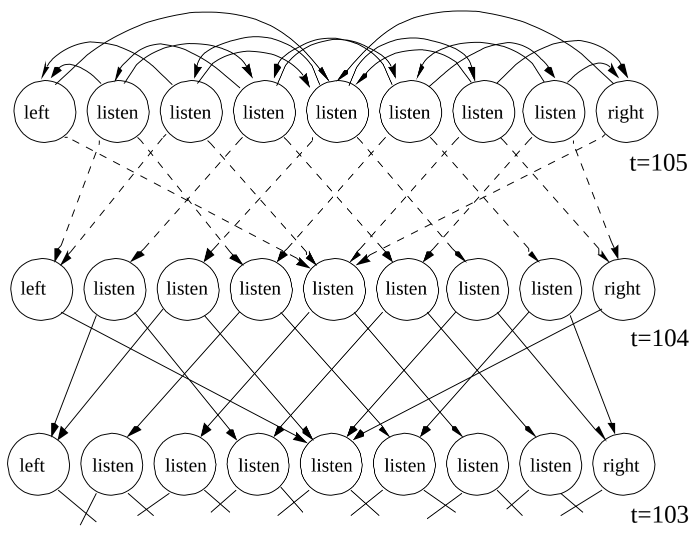


Figure 2.5: An example from [KLC98] that illustrates how policy tree branches can be rearranged to form a stationary policy.

region map to a single region (represented by another graph node).[2] Therefore, an agent does not have to explicitly maintain its belief state and perform expensive operations of updating its beliefs and finding the best α-vector for the belief state. The starting node is optimized for the initial belief state.

Of course, not all POMDP problems allow for optimal infinite horizon policies to be represented by a finite policy graph. Since such a graph cannot be extracted from a suboptimal value function, a policy in such cases is usually defined implicitly by a value function and calculated using Equation 2.35.

However, limiting the size of a policy provides a tractable way of solving POMDPs *approximately*. Although generally the optimal policy depends on the whole history of observations and actions, one way of facilitating the solution of POMDPs is to assume that an agent has a finite memory. We can represent this finite memory by a set of internal

[2]Note that this is true only if the optimal infinite horizon value function can be represented by a finite number of α-vectors.

states $\mathcal{N}$. The internal states are fully observable; therefore an agent can execute a policy that maps from *internal* states to actions.

The *action selection* function determines what action to execute at each internal memory state $n \in \mathcal{N}$. In addition to the mapping from internal states to actions, we also need to specify the dynamics of the internal process, i.e., describe the transitions from one internal state to another. The internal memory states can be viewed as nodes, and the transitions between nodes will depend on observations received. Together, the set of nodes and the transition function constitute a policy graph, or a *finite-state controller* (FSC).

FSC model

A deterministic policy graph π is a triple $\langle \mathcal{N}, \psi, \eta \rangle$, where

- $\mathcal{N}$ is a set of controller nodes n, also known as internal memory states.
- $\psi : \mathcal{N} \mapsto \mathcal{A}$ is the action selection function that for each node n prescribes an action $\psi(n)$.
- $\eta : \mathcal{N} \times \mathcal{O} \mapsto \mathcal{N}$ is the node transition function that for each node and observation assigns a successor node n'. $\eta(n, \cdot)$ is essentially an observation strategy for the node n, described above when discussing policy trees and conditional plans.

In a *stochastic* FSC, the action selection function ψ and the internal transition function η are stochastic. Here,

- $\psi : \mathcal{N} \mapsto \Delta(A)$ is the stochastic action selection function that for each node n prescribes a distribution over actions:
$$\psi(n, a) = Pr(A^t = a | N^t = n). \tag{2.44}$$
- $\eta : \mathcal{N} \times \mathcal{O} \mapsto \Delta(\mathcal{N})$ is the stochastic node transition function that for each node and observation assigns a probability distribution over successor nodes n'; $\eta(n, o, n')$ is

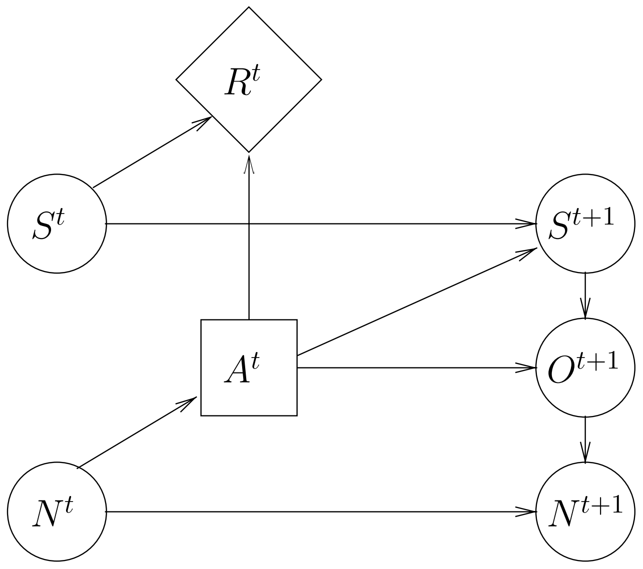


Figure 2.6: The joint influence diagram for a policy graph and a POMDP. The sequence of FSC nodes coupled with POMDP states is Markovian.

the probability of transition from node n to node n' after observing $o' \in \mathcal{O}$:

$$\eta(n, o', n') = Pr(N^{t+1} = n' | N^t = n,\, O^{t+1} = o'). \tag{2.45}$$

2.2.5 Cross-product MDP

In the way that an MDP policy $\pi : \mathcal{S} \mapsto \Delta(\mathcal{A})$ gives rise to a Markov chain defined by the transition matrix T, a POMDP policy, represented by a finite graph, is also sufficient to render the dynamics of a POMDP Markovian. The cross-product between the POMDP and the finite policy graph is itself a finite MDP, which will be referred to as the *cross-product MDP*. The structure of both the POMDP and the policy graph can be represented in the cross-product MDP. The influence diagram for such a *coupled* process is shown in Figure 2.6.

Given a POMDP $\langle \mathcal{S}, \mathcal{A}, T, R, \mathcal{O}, Z \rangle$ and a policy graph with the node set $\mathcal{N}$, the new cross-product MDP $\langle \bar{\mathcal{S}}, \bar{\mathcal{A}}, \bar{T}, \bar{R} \rangle$ can be described as follows [MKKC99]:

- The state space $\bar{\mathcal{S}} = \mathcal{N} \times \mathcal{S}$ is the Cartesian product of external system states and internal memory nodes; it consists of pairs $\langle n, s \rangle,\ n \in \mathcal{N},\ s \in \mathcal{S}$.
- At each state $\langle n, s \rangle$, there is a choice of action $a \in \mathcal{A}$, and a conditional observation strategy $\nu : \mathcal{O} \mapsto \mathcal{N}$, which determines the next internal node for each possible observation. The new action space $\bar{\mathcal{A}} = \mathcal{A} \times \mathcal{N}^{\mathcal{O}}$ is therefore a cross product between A and the space of observation mappings $\mathcal{N}^{\mathcal{O}}$. A pair $\langle a, \nu \rangle$ is a *conditional plan*, where $a \in \mathcal{A}$ is an action and $\nu \in \mathcal{N}^{\mathcal{O}}$ is a deterministic observation strategy.
- $\bar{T} : \bar{\mathcal{S}} \times \bar{\mathcal{A}} \mapsto \bar{\mathcal{S}}$ is the transition function:

$$\bar{T}(\langle n, s \rangle, \langle a, \nu \rangle, \langle n', s' \rangle) = T(s, a, s') \sum_{o | \nu(o) = n'} Z(s', a, o). \tag{2.46}$$

- The reward function $\bar{R} : \bar{\mathcal{S}} \times \bar{\mathcal{A}} \mapsto \mathbb{R}$ becomes:

$$\bar{R}(\langle n, s \rangle, \langle a, \nu \rangle) = R(s, a). \tag{2.47}$$

Policy graph value

Given a (stochastic) policy graph $\pi = \langle \mathcal{N}, \psi, \eta \rangle$ and a POMDP $\langle \mathcal{S}, \mathcal{A}, T, R, \mathcal{O}, Z \rangle$, the generated sequence of node-state pairs $\langle N^t, S^t \rangle$ constitutes a Markov chain [Han97, Han98a, MKKC99]. In a way analogous to Equation 2.23, the value of a given policy graph can be calculated using Bellman's equations:

$$\bar{V}^{\pi}(\bar{s}) = \bar{R}^{\pi}(\bar{s}) + \gamma \sum_{\bar{s}'} \bar{T}^{\pi}(\bar{s}, \bar{s}')\, \bar{V}^{\pi}(\bar{s}'), \tag{2.48}$$

where $\bar{s}, \bar{s}'$ are node-state pairs in $\bar{\mathcal{S}}$, and

- $\bar{T}^{\pi}$ is the transition matrix. Given stochastic functions $\psi(\cdot)$ and $\eta(\cdot)$, the transition matrix is analogous to Equation 2.23 for MDPs, although now we need to take expectation not only over actions a, but also over observations o:

$$\bar{T}^{\pi}(\langle n, s \rangle, \langle n', s' \rangle) = \sum_{a,o} \psi(n, a)\, \eta(n, o, n')\, T(s, a, s')\, Z(s', a, o). \tag{2.49}$$

- $\bar{R}^\pi$ is the reward vector:

$$\bar{R}^\pi(\langle n, s\rangle) = \sum_a \psi(n,a)R(s,a). \tag{2.50}$$

2.3 Exact solution algorithms

2.3.1 Value iteration

MDP value iteration

Value iteration for MDPs is a standard method of finding the optimal infinite horizon policy π^* using a sequence of optimal finite horizon value functions $V_0^*, V_1^*, \ldots, V_t^*$ [How60]. The difference between the optimal value function and the optimal t-horizon value function goes to zero as t goes to infinity:

$$\lim_{t\to\infty} \max_{s\in\mathcal{S}} |V^*(s) - V_t^*(s)| = 0. \tag{2.51}$$

It turns out that the optimal value function can be calculated in a finite number of steps given the *Bellman error* ϵ, which is the maximum difference (for all states) between two successive finite horizon value functions. Using Equation 2.16, the value iteration algorithm for MDPs can be summarized as follows:

- Initialize $t = 0$ and $V_0(s) = 0$ for all $s \in \mathcal{S}$.

- While $\max_{s\in\mathcal{S}} |V_{t+1}(s) - V_t(s)| > \epsilon$, calculate $V_{t+1}(s)$ for all states $s \in \mathcal{S}$ according to the following equation, and then increment t:

$$V_{t+1}(s) = \max_{a\in\mathcal{A}} \left[R(s,a) + \gamma \sum_{s'\in\mathcal{S}} T(s,a,s')\ V_t(s') \right].$$

This algorithm results in an implicit policy (which can be extracted using Equation 2.21) that is within $2\epsilon\gamma/(1-\gamma)$ of the optimal [Bel57].

POMDP value iteration

As described above, any POMDP can be reduced to a continuous belief-state MDP. Therefore, value iteration can also be used to calculate optimal infinite horizon POMDP policies:

- Initialize $t = 0$ and $V_0(b) = 0$ for all $b \in \mathcal{B}$.
- While $\sup_{b \in \mathcal{B}} |V_{t+1}(b) - V_t(b)| > \epsilon$, calculate $V_{t+1}(b)$ for all states $b \in \mathcal{B}$ according to the following equation, and then increment t:

$$V_{t+1}(b) = \max_{a \in \mathcal{A}} \left[R^b(b,a) + \gamma \sum_{b' \in \mathcal{B}} T^b(b,a,b')\, V_t(b') \right]. \tag{2.52}$$

The previous equation can be rewritten in terms of the original POMDP formulation as

$$V_{t+1}(b) = \max_{a \in \mathcal{A}} \left[\sum_{s \in \mathcal{S}} b(s) R(s,a) + \gamma \sum_{o \in \mathcal{O}} Pr(o|a,b) V_t(b_o^a) \right], \tag{2.53}$$

where $Pr(o|a,b)$ is

$$Pr(o|a,b) = \sum_{s' \in \mathcal{S}} Z(s',a,o) \sum_{s \in \mathcal{S}} T(s,a,s')\, b(s). \tag{2.54}$$

Although the belief space is continuous, any optimal finite horizon value function is piecewise linear and convex and can be represented as a finite set of α-vectors (see Section 2.2.3). Therefore, the essential task of all value-iteration POMDP algorithms is to find the set $\mathcal{V}_{t+1}$ representing value function V_{t+1}, given the previous set of α-vectors $\mathcal{V}_t$.

Various POMDP algorithms differ in how they compute value function representations. The most naive way is to construct the set of conditional plans $\mathcal{V}_{t+1}$ by enumerating all the possible actions and observation mappings to the set $\mathcal{V}_t$. The size of $\mathcal{V}_{t+1}$ is then $|\mathcal{A}||\mathcal{V}_t|^{|\mathcal{O}|}$. Since many vectors in $\mathcal{V}_t$ might be dominated by others, the optimal t-horizon value function can be represented by a parsimonious set $\mathcal{V}_t^-$. The set $\mathcal{V}_t^-$ is the smallest subset of $\mathcal{V}_t$ that still represents the same value function V_t^*; all α-vectors in $\mathcal{V}_t^-$ are useful at some belief state (see Section 2.2.3). To compute $\mathcal{V}_{t+1}$ (and $\mathcal{V}_{t+1}^-$), we only need to consider the parsimonious set $\mathcal{V}_t^-$.

Some algorithms calculate $\mathcal{V}_{t+1}^{-}$ by generating $\mathcal{V}_{t+1}$ of size $|\mathcal{A}||\mathcal{V}_t^{-}|^{|\mathcal{O}|}$, and then *pruning* dominated α-vectors, usually by linear programming. Such algorithms include Monahan's algorithm [Mon82, Whi91], and Incremental pruning [ZL96, CLZ97]. Other methods, such as Sondik's One-pass [Son71, SS73], Cheng's Linear Support [Che88], and Witness [KLC98], build the set $\mathcal{V}_{t+1}^{-}$ directly from the previous set $\mathcal{V}_t^{-}$, without considering non-useful conditional plans. Even the fastest of exact value-iteration algorithms can currently solve only toy problems.

As for MDPs, for a given ϵ, the implicit policy extracted from the value function is within $2\epsilon\gamma/(1-\gamma)$ of the optimal policy value.

2.3.2 Policy iteration

Policy iteration algorithms proceed by iteratively improving the policies themselves. The sequence $\pi_0, \pi_1, \ldots, \pi_t$ then converges to the optimal infinite horizon policy π^*, as $t \to \infty$. Policy iteration algorithms usually consist of two stages: *policy evaluation* and *policy improvement*.

MDP policy iteration

First, we summarize the policy iteration method for MDPs [How60]:

- Initialize $\pi_0(s) = a$, for all $s \in \mathcal{S}; a \in \mathcal{A}$ is an arbitrary action. Then, repeat the following policy iteration and improvement steps until the policy does not change anymore, i.e., $\pi_{t+1}(s) = \pi_t(s)$ for all states $s \in S$.

- Policy evaluation. Calculate the value of policy π_t (using Equation 2.18):

$$V^{\pi_t}(s) = R(s, \pi_t(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi_t(s), s')\, V^{\pi_t}(s').$$

- Policy improvement. For each $s \in \mathcal{S}$ and $a \in \mathcal{A}$, compute the Q-function $Q_t(s,a)$:

$$Q_{t+1}(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s')\, V^{\pi_t}(s'). \tag{2.55}$$

Then, improve the policy π_{t+1}:

$$\pi_{t+1}(s) = \arg\max_{a \in \mathcal{A}} Q_{t+1}(s, a) \text{ for all } s \in \mathcal{S}. \tag{2.56}$$

Policy iteration tends to converge much faster than value iteration in practice. However, it performs more computation at each step; policy evaluation step requires a solution of a $|\mathcal{S}| \times |\mathcal{S}|$ linear system.

POMDP policy iteration

For value iteration, it is important to be able to extract a policy from a value function (see Section 2.2.3). For policy iteration, it is important to be able to represent a policy so that its value function can be calculated easily. Here, we will describe a POMDP policy iteration method that uses an FSC to represent the policy explicitly and independently of the value function.

The first POMDP policy iteration algorithm was described by Sondik [Son71, Son78]. It used a cumbersome representation of a policy as a mapping from a finite number of polyhedral belief space regions to actions, and then converted it to an FSC in order to calculate the policy's value. Because the conversion between the two representations is extremely complicated and difficult to implement, Sondik's policy iteration is not used in practice.

Hansen proposed a similar approach, where a policy is directly represented by a finite state controller [Han97, Han98a]. His policy iteration algorithm is analogous to the policy iteration in MDPs. The policy is initially represented by a deterministic finite-state controller π^0. The algorithm then performs the usual policy iteration steps: evaluation and improvement. The evaluation of the controller π is straightforward; during the improvement step, a dynamic programming update transforms the current controller into an improved one. The sequence of finite-state controllers $\pi_0, \pi_1, \ldots, \pi_t$ converges to the optimal policy π^* as $t \to \infty$.

Policy evaluation

In exact policy iteration, each controller node corresponds to an α-vector in a piecewise-linear and convex value function representation. Since our policy graph is deterministic, $\psi(n)$ outputs the action associated with the node n, and $\eta(n, o)$ is the successor node of n after receiving observation o. The α-vector representation of a value function can be calculated using the cross-product MDP evaluation formula from before (Equation 2.48):

$$\bar{V}^{\pi}(\langle n, s\rangle) = R(s, \psi(n)) + \gamma \sum_{s', o} T(s, \psi(n), s')\, Z(s', \psi(n), o)\, \bar{V}^{\pi}(\eta(n, o), s'). \tag{2.57}$$

$\bar{V}^{\pi}(\langle n, s\rangle)$ is the value of state s of an α-vector corresponding to the node n:

$$\bar{V}^{\pi}(\langle n_i, s\rangle) \equiv \alpha_i(s). \tag{2.58}$$

Thus, evaluating the cross-product MDP for all states $\bar{s} \in \bar{\mathcal{S}}$ is equivalent to computing a set of α-vectors $\mathcal{V}^{\pi}$. Therefore, policy evaluation step is fairly straightforward and its running time is proportional to $|\mathcal{N} \times \mathcal{S}|^2$.

Policy improvement

Policy improvement step simply performs a standard dynamic programming backup during which the value function V^{π}, represented by a finite set of α-vectors $\mathcal{V}^{\pi}$, gets transformed into an improved value function V', represented by another finite set of α-vectors $\mathcal{V}'$. Although in the worst case the size of $\mathcal{V}'$ can be proportional to $|\mathcal{A}||\mathcal{V}^{\pi}|^{|\mathcal{O}|} = |\mathcal{A}||\mathcal{N}|^{|\mathcal{O}|}$ (where $|\mathcal{N}|$ is the number of controller nodes at the current iteration), many exact algorithms, such as Witness [CKL94] or Incremental pruning [CLZ97], fare better in practice.

In the policy evaluation step, a set of α-vectors $\mathcal{V}^{\pi}$ is calculated from the finite-state controller π using Equation 2.57. Then, the set $\mathcal{V}'$ is computed using dynamic programming backup on the set $\mathcal{V}^{\pi}$. The key insight in Hansen's policy iteration algorithm is observation that the new improved controller π' can be constructed from the new set $\mathcal{V}'$ and the current controller π by following three simple rules:

- For each vector $\alpha' \in \mathcal{V}'$:
  - If the action and successor links of α' are identical to the action and conditional plan of some node that is already in π, then the same node will remain unchanged in π'.
  - If α' pointwise dominates some nodes in π, replace those nodes by a node corresponding to α', i.e., change the action and successor links to those of the vector α'.
  - Else, add a node to π' that has the action and observation strategy associated with α'.
- Prune any node in π that has no corresponding α-vector in $\mathcal{V}'$ as long as that node is not reachable from a node with an associated vector in $\mathcal{V}'$.

If the policy improvement step does not change the FSC, the controller must be optimal. Of course, this can happen only if the optimal infinite horizon value function does have a finite representation. Otherwise, a succession of FSCs will approximate the optimal value function arbitrarily closely; an ϵ-optimal FSC can be found in a finite number of iterations [Han98b].

Like MDP policy iteration, POMDP policy iteration in practice requires fewer steps to converge. Since policy evaluation complexity is negligible compared to the worst-case exponential complexity of the dynamic-programming improvement step, policy iteration appears to have a clearer advantage over value iteration for POMDPs [Han98a].

Controllers found by Hansen's policy iteration are optimized for all possible initial belief states. The convexity of the value function is preserved because the starting node maximizes the value for the initial belief state. From the next section onward, we will usually assume that an initial belief state is known beforehand, and our solutions will take computational advantage of this fact. Optimal controllers can be much smaller if they do not need to be optimized for all possible belief states [KLC98, Han98a].

2.4 Gradient-based optimization

Exact methods for solving POMDPs remain highly intractable, in part because optimal policies can be either very large, or, worse, infinite. For example, in exact policy iteration, the number of controller nodes might grow doubly exponentially in the horizon length; in value iteration, it is the number of α-vectors required to represent the value function that multiplies at the same doubly exponential rate.

An obvious approximation technique is therefore to restrict the set of policies; the goal is then to find the best policy within that restricted set. Since all policies can be represented as (possibly infinite) policy graphs, a widely used restriction is to limit the set of policies to those representable by *finite* policy graphs, or finite-state controllers, of some *bounded* size. This allows to achieve a compromise between the requirement that courses of action should depend on certain aspects of observable history, and the ability to control the complexity of the policy space.

Many previous approaches rely on the same general idea. While Hansen's exact policy iteration does not place any constraints on the policy graph structure, other techniques take computational advantage of searching in the space of structurally restricted FSCs. Littman [Lit94], Jaakola *et al.* [JSJ95], Baird and Moore [BM99] search for optimal reactive, or memoryless, policies; McCallum [McC95] considers variable-length finite horizon memory; Wiering and Schmidhuber [WS97] attempt to find sequences of reactive policies; and, Peshkin *et al.* [PMK99] constrain the search to external memory policies. All of these techniques[3] are special cases of searching in the space of finite policy graphs.

The restricted policy space that we will consider in this thesis is representable by a limited size stochastic finite-state controller (see Section 2.2.4). Here, we describe the details of a gradient-based policy search method, introduced by Meuleau *et al.* [MKKC99, MPKK99]. The main idea of gradient-based POMDP policy search methods

[3]Some of these approaches are applied in a reinforcement learning setting. In this thesis, we assume that the model of the world is fully known.

is to reformulate the task of finding optimal POMDP policies as a classical non-linear numerical optimization problem. If the stochastic FSC is appropriately parameterized so that its value is continuous and differentiable, the gradient of the value function can be computed analytically in polynomial time with respect to the size of the cross-product MDP ($|\mathcal{N} \times \mathcal{S}|$), and used to find locally optimal solutions.

Policy graph value

We can rewrite Equation 2.48, which calculates the value of a stochastic policy graph π, in a more concise matrix and vector form:

$$\bar{V}^{\pi} = \bar{R}^{\pi} + \gamma \bar{T}^{\pi} \, \bar{V}^{\pi}. \tag{2.59}$$

$\bar{V}$ and $\bar{R}$ are vectors of length $|\mathcal{N}| \, |\mathcal{S}|$, and $\bar{T}$ is an $|\mathcal{N}| \, |\mathcal{S}|$ by $|\mathcal{N}| \, |\mathcal{S}|$ matrix. Since $\bar{T}$ is a stochastic matrix and the discount factor $\gamma < 1$, the matrix $I - \gamma \bar{T}$ is invertible [Put94]; we can thus solve Equation 2.59 for $\bar{V}$:

$$\bar{V}^{\pi} = (I - \gamma \bar{T}^{\pi})^{-1} \, \bar{R}^{\pi}. \tag{2.60}$$

Notice that $\bar{V}^{\pi}$, $\bar{T}^{\pi}$, and $\bar{R}^{\pi}$ depend on the policy graph $\pi = \langle \mathcal{N}, \psi, \eta \rangle$. Therefore, for a given number of nodes $|\mathcal{N}|$, the vector $\bar{V}^{\pi}$ could be optimized by choosing the right functions ψ and η. To convert this problem to a classical non-linear optimization problem, we need to make sure that the objective function is a scalar as well as appropriately parameterize the functions ψ and η.

Prior beliefs

The value vector $\bar{V}^{\pi}$ contains the total discounted cumulative reward for each system state s and graph node n. The total expected reward depends on the state and node in which an agent starts; this could be quantified by an agent's prior beliefs about the world. Let $\bar{b}_0$ be an $|\mathcal{N}| \, |\mathcal{S}|$ vector of probabilities representing the agent's prior beliefs about the states $\mathcal{S}$ and policy graph nodes $\mathcal{N}$. That is,

$$\begin{aligned}\sum_{n,s} \bar{b}(\langle n,s\rangle) = 1, \\ \bar{b}(\langle n,s\rangle) \geq 0 \text{ for all } n \in \mathcal{N}, s \in \mathcal{S}.\end{aligned} \tag{2.61}$$

Then, the total expected cumulative discounted reward E^{π} is just

$$E^{\pi} = \bar{b}_0 \cdot \bar{V}^{\pi}. \tag{2.62}$$

To simplify the problem, we will assume that the agent always starts in node n_0; it is a valid simplification if the initial policy graph structure is symmetric for all nodes. The agent's prior knowledge about the world is summarized by the belief vector b_0. Therefore,

$$\bar{b}(\langle n,s\rangle) = \begin{cases} b(s), & \text{if } n = n_0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.63}$$

Soft-max parameterization

To parameterize the functions ψ and η, we will employ a commonly used *soft-max* distribution function [MPKK99, AB02]. Let $\mathbf{x}^{\psi}$ and $\mathbf{x}^{\eta}$ be parameter vectors for the respective functions ψ and η. $\mathbf{x}^{\psi}$ will be indexed by a node n and an action a; $\mathbf{x}^{\eta}$ will be indexed by a node n, an observation o, and the successor node n'. We will use the notation $\mathbf{x}^{\psi}[n,a]$ to denote the ψ parameter indexed by n,a, and $\mathbf{x}^{\eta}[n,o,n']$ will be the η parameter indexed by n,o,n'. Then,

$$\psi(n,a) = \psi(a|n;\mathbf{x}^{\psi}) = \frac{e^{\mathbf{x}^{\psi}[n,a]}}{\sum_{\bar{a}\in\mathcal{A}} e^{\mathbf{x}^{\psi}[n,\bar{a}]}}, \tag{2.64}$$

$$\eta(n,o,n') = \eta(n'|n,o;\mathbf{x}^{\eta}) = \frac{e^{\mathbf{x}^{\eta}[n,o,n']}}{\sum_{\bar{n}'\in\mathcal{N}} e^{\mathbf{x}^{\eta}[n,o,\bar{n}']}}. \tag{2.65}$$

Because we use soft-max, the parameterized functions ψ and η still represent probability distributions; that is,

$$\begin{aligned} \sum_{a \in A} \psi(a|n; \mathbf{x}^\psi) &= 1, \\ \sum_{n' \in \mathcal{N}} \eta(n'|o, n; \mathbf{x}^\eta) &= 1, \\ \psi(a|n; \mathbf{x}^\psi) &\geq 0 \quad \text{for all } a \in \mathcal{A}, n \in \mathcal{N}, \\ \eta(n'|n, o; \mathbf{x}^\eta) &\geq 0 \quad \text{for all } n, n' \in \mathcal{N}, o \in \mathcal{O}. \end{aligned} \tag{2.66}$$

Objective function

Let $\mathbf{x}$ denote the combined vector of parameters $\mathbf{x}^\psi$ and $\mathbf{x}^\eta$. By substituting Equation 2.60 into 2.62, we finally get an unconstrained continuous objective function $f(\cdot)$ of parameters $\mathbf{x}$:

$$f(\mathbf{x}) = \bar{b}_0 \; (I - \gamma \bar{T}^\pi)^{-1} \; \bar{R}^\pi, \tag{2.67}$$

where (see Equations 2.49 and 2.50)

$$\bar{T}^\pi(\langle n, s \rangle, \langle n', s' \rangle) = \sum_{a,o} \psi(a|n; \mathbf{x}^\psi) \, \eta(n'|n, o; \mathbf{x}^\eta) \, T(s, a, s') \, Z(s', a, o), \tag{2.68}$$

$$\bar{R}^\pi(\langle n, s \rangle) = \sum_{a} \psi(a|n; \mathbf{x}^\psi) \, R(s, a), \tag{2.69}$$

and $\bar{b}_0, T(\cdot), R(\cdot), Z(\cdot)$ are supplied by the POMDP model. The number of parameters $|\mathbf{x}|$ depends on the POMDP model and the size of the policy graph (i.e., the size of the cross-product MDP):

$$|\mathbf{x}| = |\mathbf{x}^\psi| + |\mathbf{x}^\eta| = |\mathcal{N}||\mathcal{A}| + |\mathcal{N}||\mathcal{O}||\mathcal{N}|. \tag{2.70}$$

This presents two advantages to gradient-based methods of solving POMDPs: the number of parameters does not depend on the size of the state space $\mathcal{S}$, and the size of internal memory $\mathcal{N}$ can be controlled by a user.

Gradient calculation

Since the objective function $f(\mathbf{x})$ is a complicated series matrix expansion with respect to its parameters, function value based optimization techniques will be ineffective. To perform numerical optimization, we will need to employ first-order information about our objective function.

Because of the soft-max parameterization, the gradient of $f(\mathbf{x})$ can be calculated analytically. From Equation 2.62,

$$\frac{\partial f}{\partial x} = \bar{b}_0 \frac{\partial \bar{V}}{\partial x}. \tag{2.71}$$

From Equation 2.60,

$$\frac{\partial \bar{V}}{\partial x} = (I - \gamma \bar{T})^{-1} \left[\frac{\partial \bar{R}}{\partial x} + \gamma \frac{\partial \bar{T}}{\partial x} (I - \gamma \bar{T})^{-1} \bar{R} \right]. \tag{2.72}$$

Partial derivatives with respect to $\bar{T}$ and $\bar{R}$ can be calculated from Equations 2.68 and 2.69:

$$\frac{\partial \bar{T}}{\partial x^{\psi}} = \sum_{a,o} \frac{\partial \psi(a|n; \mathbf{x}^{\psi})}{\partial x^{\psi}} \eta(n, o, n') \, T(s, a, s') \, Z(s', a, o), \tag{2.73}$$

$$\frac{\partial \bar{T}}{\partial x^{\eta}} = \sum_{a,o} \psi(n, a) \frac{\partial \eta(n'|n, o; \mathbf{x}^{\eta})}{\partial x^{\eta}} \, T(s, a, s') \, Z(s', a, o), \tag{2.74}$$

$$\frac{\partial \bar{R}}{\partial x^{\psi}} = \sum_{a} \frac{\partial \psi(a|n; \mathbf{x}^{\psi})}{\partial x^{\psi}} R(s, a), \tag{2.75}$$

$$\frac{\partial \bar{R}}{\partial x^{\eta}} = 0. \tag{2.76}$$

Finally, we can find the derivatives of ψ and η from the analytical expression of the soft-max function (see Equations 2.64 and 2.65):

$$\frac{\partial \psi(a|n; \mathbf{x}^{\psi})}{\partial x^{\psi}[\bar{n}, \bar{a}]} = \begin{cases} (1 - \psi(n, a)) \, \psi(n, a), & \text{if } n = \bar{n},\ a = \bar{a}, \\ -\psi(n, a) \, \psi(\bar{n}, a), & \text{if } n = \bar{n},\ a \neq \bar{a}, \\ 0, & \text{if } n \neq \bar{n}. \end{cases} \tag{2.77}$$

$$\frac{\partial \eta(n'|n, o; \mathbf{x}^{\eta})}{\partial x^{\eta}[\bar{n}, \bar{o}, \bar{n}']} = \begin{cases} (1 - \eta(n, o, n')) \, \eta(n, o, n'), & \text{if } n = \bar{n},\ o = \bar{o},\ n' = \bar{n}', \\ -\eta(n, o, n') \, \eta(\bar{n}, o, n'), & \text{if } n = \bar{n},\ o = \bar{o},\ n' \neq \bar{n}', \\ 0, & \text{if } n \neq \bar{n} \text{ or } o \neq \bar{o}. \end{cases} \tag{2.78}$$

Local optimization

Many numerical optimization techniques, such as steepest-descent, quasi-Newton or conjugate gradient, can be used to search for local minima employing the analytically calculated gradient information. All the experiments reported in this thesis used the quasi-Newton method with Broyden, Fletcher, Goldfrab and Shanno (BFGS) update and quad-cubic line search, implemented in Matlab's Optimization Toolbox [Mat02].

Chapter 3

Stochastic local search procedure

In this chapter, we present a new approach to the approximate solution of POMDPs using finite-state controllers. Using certain heuristics that follow some of the sequential reasoning inherent in dynamic programming approaches, we supplement gradient ascent with stochastic local search techniques. This allows us to solve some problems where gradient ascent methods get stuck in very poor local suboptima.

3.1 Motivating example

We now introduce a simple problem containing typical local optima which gradient ascent methods commonly fail to overcome. This example should also help understand intuitions behind the actual local search algorithm. Such intuitions will be formalized and described in the following sections of this chapter.

Consider a very simple planning problem in which the optimal solution consists of performing a certain action c until the precondition p for a subsequent action d is observed; action d then leads to a high-reward goal state g. The actions can be stochastic in their effect and the observations can be noisy. Furthermore, actions c and d are very costly, but the reward associated with the goal g more than compensates for their expected costs; d only achieves g (with reasonable probability) if p is true; p is only made

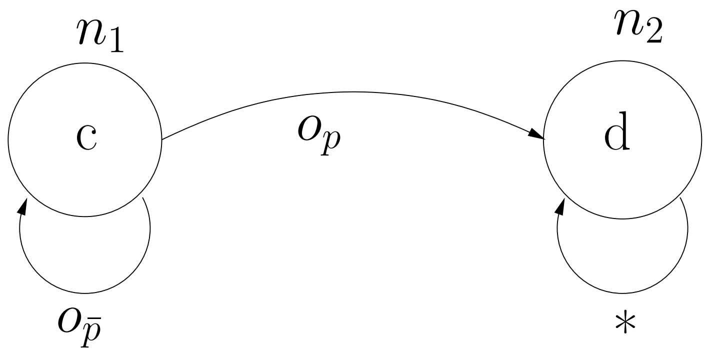


Figure 3.1: An optimal FSC for the Planning problem.

true by c; all other actions (at any state) have small costs and lead to rewards that are small relative to the costs of actions c and d, and the reward of the goal state g.

Let's assume that observation o_p gets emitted with certainty when the precondition p is true; $o_{\bar{p}}$ is any other observation. The optimal policy for this POMDP can be represented using a simple 2-node policy graph shown in Figure 3.1. Starting at node n_1, the best course of action is to execute c until observation o_p is received. On observation o_p, the process moves to the internal state n_2, at which action d eventually leads to a high-reward goal.

Planning POMDP

We can formalize this simple planning problem as a POMDP in the following way. The state space $\mathcal{S}$ consists of four states s_0, s_p, g, g':

- s_0 is the starting state at which the precondition p is false. We will assume that agents have full knowledge of the initial state.
- s_p is the state where the precondition p for action d is true.
- g is the high-reward goal state. Reaching g terminates the process and gives an agent a reward of +500.

- g' is an alternate goal state that gives an agent a reward of +10. One should think of g' as summarizing the state space reachable by actions that do not follow the optimal sequence c, d. The goal g' is a suboptimal goal; we shall see that gradient ascent will commonly tend to select actions that lead to such suboptimal states.

The action space $\mathcal{A}$ consists of actions $c, d, a_0, a_1, \ldots, a_k$. k actions a_i lead to suboptimal regions summarized by the goal state g'. In our simple model, executing any action a_i will cause the transition to g'. Actions c and d are costly (each costs -100), but executing the sequence c, d enables to achieve a reward of +500 at goal state g. Action c has a high probability of causing a transition to state s_p. Action d has high cost (-1000) unless executed at state s_p; it can then achieve goal g with high probability. For the following concrete POMDP, we will set $k = 18$; that is, the action space size is 20.

To keep things simple, let's assume that transitions and observations are deterministic, and the discount factor is very close to 1. Then, the optimal policy of executing the action sequence c, d has a value of $+300 = 500 - 100 - 100$; a suboptimal policy of executing an action a_i at the starting node has a value of +10 (see Figure 3.1).

Why GA fails

Quite surprisingly, gradient ascent will rarely find a policy represented by the optimal 2-node controller shown in Figure 3.1. In 100,000 GA trials of a 2-node controller, 96.29% of them resulted in suboptimal policies of value +10; allowing for more capacity did not help: with 10-node controllers, the failure rate was 97.47%. If the action space is large enough, a random instantiation of this FSC is very unlikely to be optimal or serve as a good start point for gradient ascent.[1] Suppose we attempt to solve this problem using gradient ascent, starting from some random initial policy graph, and suppose no node selects action c or d with significant probability (if the number of "other" actions a_i is

[1]To keep things simple, we focus on a small two-action sequence; for longer sequences, typical of planning problems, the odds of a random initial FSC including any significant subsequence is negligible.

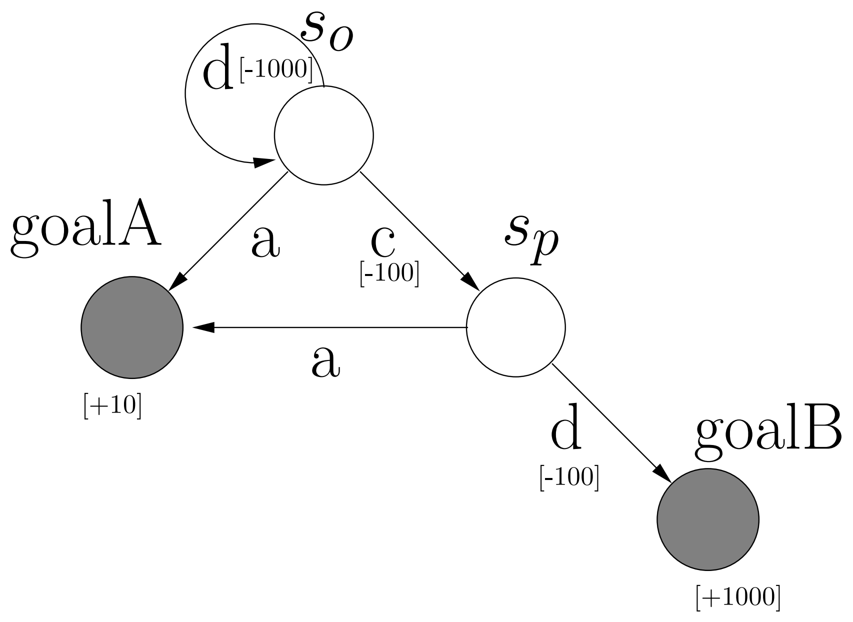


Figure 3.2: Simple planning problem

large, this is a very reasonable assumption). Since the probability of c being executed is small, the probability of the precondition p being true at any belief state reachable using the current FSC is small; hence, increasing the probability of d at any node will decrease controller value, preventing GA from moving in that direction. Similarly, since d is unlikely to be executed, the value of increasing the probability of c at any node is negative, preventing GA from moving in that direction. The safe alternative of selecting an action a_i seems more attractive. Indeed, the nature of this problem is such that GA will be forced to move *away* from the optimal FSC. The sequential nature of the problem, and the fact that the actions that make up the optimal policy are *undesirable* unless their counterparts are in place, make the landscape very hard to navigate using GA.

Intuitively, action d would be considered useful at a belief state in which precondition p held (i.e., where probability of being in state s_p is high enough). Unfortunately, since c is never executed, such a belief state is unreachable given the current FSC. However, it is easy to verify that action d is good at some belief state in the context of the current controller. More precisely, a conditional plan $\langle d, \nu \rangle$ installed at node n_2—where, e.g., ν loops back to the same node n_2 for all observations—would have high value in any belief state where p is sufficiently probable. Then, if d is already installed at node n_2, choosing c at n_1 and transitioning to n_2 on receiving observation o_p would attain high value.

How to choose moves

So, how should we identify and select conditional plans, such as $\sigma = \langle d, \nu \rangle$, that are necessary for the optimal controller? We already saw that GA would not recommend this plan because it is not useful at any belief region that is reachable from the starting belief state $b_0 = [1; 0]$ *given the current controller* (let $b = [Pr(s_0); Pr(s_p)]$ be a representation of agent's beliefs; we allow uncertainty only over non-goal states). However, this plan is very good at a belief region around the belief state $[0; 1]$, which is not reachable if action c is not executed at node n_1. Therefore, we can ask the following question:

Is there a belief state b^σ at which the plan σ is better than any other plan σ'?

A straightforward linear program (LP) can not only answer this question, but also return the largest difference δ^σ by which a conditional plan σ is better than any other plan σ':

$$\delta^\sigma = \max_b \left[Q(b, \sigma) - \max_{\sigma' \in \Sigma} Q(b, \sigma') \right]. \tag{3.1}$$

Let $\mathcal{B}^\sigma$ be the belief region such that any belief state in $\mathcal{B}^\sigma$ maximizes this expresion for δ^σ. We define b^σ as the belief state that maximizes the Q-value of the plan σ with the constraint that b^σ has to be in the $\mathcal{B}^\sigma$ region:

$$b^\sigma = \arg\max_{b \in \mathcal{B}^\sigma} Q(b, \sigma). \tag{3.2}$$

The notions of δ-values and Q-value maximizing belief states b^σ will be described in detail in Section 3.2.3.

Assuming we start with a FSC initialized with uniform action and transition function distributions, we get the δ-values for all conditional plans, which are listed Table 3.1.

For our simple POMDP with two observations, a shorthand notation for the conditional

Table 3.1: δ and Q-value heuristics for conditional plans

Conditional plan σ	δ^σ	Q-value at b^σ	b^σ
$\langle c,\ [n_1\, n_1]\rangle$	-84.7368	-74.7368	[1;0]
$\langle d,\ [n_1\, n_1]\rangle$	0	400	[0;1]
$\langle a_i,\ [n_1\, n_1]\rangle$	0	10	[1;0]
$\langle c,\ [n_1\, n_2]\rangle$	-84.7368	-74.7368	[1;0]
$\langle d,\ [n_1\, n_2]\rangle$	0	400	[0;1]
$\langle a_i,\ [n_1\, n_2]\rangle$	0	10	[1;0]
$\langle c,\ [n_2\, n_1]\rangle$	-84.7368	-74.7368	[1;0]
$\langle d,\ [n_2\, n_1]\rangle$	0	400	[0;1]
$\langle a_i,\ [n_2\, n_1]\rangle$	0	10	[1;0]
$\langle c,\ [n_2\, n_2]\rangle$	-84.7368	-74.7368	[1;0]
$\langle d,\ [n_2\, n_2]\rangle$	0	400	[0;1]
$\langle a_i,\ [n_2\, n_2]\rangle$	0	10	[1;0]

plan $\langle a,\ [n\ m]\rangle$ means "execute action a, and on observation $o_{\bar{p}}$ go to node n, while on observation o_p go to node m".

The δ^σ column in this table answers our question. If $\delta^\sigma \geq 0$ for the plan σ, then there is a belief state b^σ at which this plan is as good or better than any other plan. Otherwise, the plan is not very useful given the current controller. For example, there is no belief state at which executing action c will be a good decision *given the current controller*.[2] On the other hand, just by considering δ-values, any other action seems to be attractive.

Thus, δ-values themselves are not enough to serve as good heuristics for evaluating

[2]The fact that observation mappings do not seem to matter (only actions do) is an artifact of this simple problem; in general, full conditional plans will have different values depending on the internal node transitions.

conditional plans (note that because all four plans with action d achieve a value of +400 at belief state [0;1], their δ-value is 0). Both actions a_i and d have δ-values of zeros at their respective Q-value difference maximizing belief states b^σ. In order to evaluate conditional plans, we therefore propose to look at their Q-values at "witness" belief states. In this case, it is clear that any conditional plan with the action d is much better than the rest, since they achieve a value of +400 at the witness belief state [0;1]. To pick a conditional plan, we can therefore stochastically sample from a distribution that is biased by the heuristic values. It is very likely that one of the conditional plans with action d will get chosen, since all of them have a heuristic value $h(\sigma) = Q(b^\sigma, \sigma)$ of +400, which is much bigger than the next best value of +10. Let $\sigma^* = \langle d,\, [n_2\, n_2] \rangle$ be the chosen conditional plan.

Our local search procedure will consider adjustments to the FSC of this type: if a plan has high value at some belief state b, even if it can't be realized by the current controller, we will consider making that move, i.e., adjusting the FSC parameters in that direction. Before we formalize and explain various possibilities for moves in the FSC space, let us define a move $m = \langle n_2, \sigma^* \rangle$ which consists of installing a conditional plan σ^* at node n_2. Of course, if we make this move by adjusting the parameters at node n_2 toward the plan $\langle d,\, [n_2\, n_2] \rangle$, we decrease the value of the FSC. For example, the initial value of the controller when initialized with uniform distributions was -47.0914; after the move m, its value became -1808.90. Should we subsequently resort to moving in a direction that improves FSC value, we would naturally want to "undo" this move. For this reason, moves of this type will be held on a *tabu list* for some period of time. By doing this, we will give the algorithm a chance to "catch up" to the move. Specifically, since the plan σ^* at node n_2 has high value at belief states near b^{σ^*}, by holding this node fixed, we give the FSC a chance to find a policy for the rest of the FSC that will induce this region of belief space at node n_2. In this example, by holding n_2 fixed, the plan $\langle c,\, [n_1\, n_2] \rangle$ at node n_1 will now look attractive (indeed, with n_2 fixed, GA would move in this direction). In a sense, this process simulates the reasoning inherent in value iteration over belief space.

After executing the move m (i.e., installing the deterministic conditional plan σ^* at node n_2), we get the heuristic values for conditional plans in the modified controller, which are displayed in Table 3.2. These new heuristic values provide new insights about

Table 3.2: δ and Q-value heuristics for conditional plans after a move

Conditional plan σ	δ^σ	Q-value at b^σ	b^σ
$\langle c,\, [n_1\, n_1]\rangle$	-361.66	-65.22	[1;0]
$\langle d,\, [n_1\, n_1]\rangle$	0	400	[0;1]
$\langle a_i,\, [n_1\, n_1]\rangle$	-286.44	10	[1;0]
$\langle c,\, [n_1\, n_2]\rangle$	0	296.44	[1;0]
$\langle d,\, [n_1\, n_2]\rangle$	0	400	[0;1]
$\langle a_i,\, [n_1\, n_2]\rangle$	0	10	[1;0]
$\langle c,\, [n_2\, n_1]\rangle$	-361.66	-65.22	[1;0]
$\langle d,\, [n_2\, n_1]\rangle$	0	400	[0;1]
$\langle a_i,\, [n_2\, n_1]\rangle$	0	10	[1;0]
$\langle c,\, [n_2\, n_2]\rangle$	0	296.44	[1;0]
$\langle d,\, [n_2\, n_2]\rangle$	0	400	[0;1]
$\langle a_i,\, [n_2\, n_2]\rangle$	0	10	[1;0]

our local search procedure. Any plan with action d still seems the most attractive even though we already have the action d at node n_2. Installing such plans at other nodes would, intuitively, waste the controller "capacity". Since node n_2 already represents a belief region near the witness belief state b^{σ^*}, we can disregard any conditional plans whose witness belief states are already represented in the current controller.

Plans $\langle c,\, [n_1\, n_2]\rangle$ and $\langle c,\, [n_2\, n_2]\rangle$ now look the most attractive; installing them at node n_1 would actually result in the optimal controller. This toy example, however, is misleading because our starting belief state [1;0] happens to be the same as the witness belief state for these two plans. In most cases, however, witness belief states might not be reachable given the current controller. Thus, after instantiating some controller nodes with potentially useful plans, we will have to either search for the plans that are good with respect to the initial belief state b_0 or hope that gradient ascent will manage to find better controllers after some nodes were chosen to represent potentially valuable conditional plans. In our algorithm, we actually do both.

In the following sections, we make such high-level intuitions more precise, identify further issues that must be addressed, and incorporate them into a stochastic local search procedure.

3.2 Stochastic local search framework

In this section we describe how our algorithm fits in the general framework of stochastic local search methods, and explain its key details and the relevant terminology.

Stochastic local search (SLS) has become a popular and successful approach to solving hard combinatorial optimization problems. Many techniques, from simple iterative improvement to evolutionary algorithms, fall under an umbrella of SLS methods. Here, we introduce the general definition of SLS, and show how the POMDP policy search can be cast into the SLS framework.

Given a combinatorial problem, an SLS algorithm for solving its arbitrary instance is defined by the search space, a set of feasible solutions, a neighborhood relation, a step function, an initialization function, a termination predicate, an objective function, and an evaluation function [Hoo98].

Below, we outline our POMDP algorithm in SLS terminology.

- The *search space* is the set of finite state controllers of a given size $|\mathcal{N}|$, parameterized by the stochastic action selection function ψ and the node transition function η.

- All FSCs in the search space are valid POMDP policy representations, and therefore, are *feasible* solutions.

- The *neighborhood* of a specific FSC can be defined as the set of all the controllers that can be reached by making a single *move*. Therefore, the neighborhood relation depends on what constitutes a move in the search space. Different alternatives for moves will be discussed later; generally, the move will be a change of either action or transition distributions (or both) of a single controller node. Gradient ascent, however, modifies distributions at all policy graph nodes at once in the direction of the gradient.

- The *step function* assigns each member of the search space a probability over its neighbors. In our case, the step function will be defined implicitly by the internal parameters of a procedure that chooses moves.

- Our *initialization* function returns a randomly initialized finite state controller.

- The *objective function* is the expected controller value with respect to the initial belief state b_0 (see Equation 2.67). It is used by gradient ascent to find locally optimal solutions. However, in order to escape from such local optima, we will need a different evaluation function.

- An *evaluation* function also maps search space positions into real numbers; however, it can be different from the objective function. The evaluation function is used for assessing or ranking candidate neighborhood solutions, and provides guidance toward high-quality, or optimal solutions. In our algorithm, we use a Q-value heuristic function $h(\cdot)$ to evaluate moves in the search space. Moves that have high heuristic value lead to potentially valuable FSCs; therefore, the evaluation function is defined implicitly by the heuristic function $h(\cdot)$, which will be described in detail later.

3.2.1 Moves, conditional plans, actions

Such terms as moves, actions, and conditional plans will be used extensively, so we need to make their meaning as precise as possible. Some of the notation was already explained in the cross-product MDP section (2.2.5) above.

action Action $a \in \mathcal{A}$ is one of the direct actions available to the agent.

observation mapping (strategy) Conditional observation mapping $\nu : \mathcal{O} \mapsto \mathcal{N}$ determines the internal transitions between nodes given observations. The set of observation mappings is $\mathcal{N}^{\mathcal{O}}$.

conditional plan Conditional plan $\bar{a} = \sigma = \langle a, \nu \rangle$, $a \in \mathcal{A}$, $\nu \in \mathcal{N}^{\mathcal{O}}$, is an element of the cross-product MDP action space $\bar{\mathcal{A}}$ and consists of an action a and a deterministic observation mapping ν. We can apply a conditional plan to any node, but by itself a conditional plan is not "attached" to any particular node $n \in \mathcal{N}$.

move Deterministically installing a specific action, observation mapping, or a conditional plan *at a specific policy graph node* n (or changing respective probabilities toward a particular action, observation, or a conditional plan in a stochastic graph) could constitute a local move in the FSC space. While all these alternatives are worth considering, in our actual implementation a move $m = \langle n, \sigma \rangle$ is a conditional plan σ ascribed to particular node n. If $\mathcal{M}$ is the set of such deterministic local moves in the space of finite graphs, then its size is $|\mathcal{N}||\mathcal{A}||\mathcal{N}|^{|\mathcal{O}|}$.

Executing moves

Making a move $m = \langle n, \sigma \rangle$ involves changing the parameters of the stochastic functions ψ and η for a node n. While we could simply set the probability of the conditional plan σ to 1, during a stochastic search, it might be desirable to retain stochasticity in an POMDP policy. If we allow non-zero probabilities to other plans σ' at the node n, the next local search step will still consider other actions and observation strategies, if their value is high (even though their probability is low).

Therefore, when we make a move, the parameters at the node n are adjusted in the direction of the plan σ. In practice, we increase the probability of σ by a fraction *moveDist* of the difference between the current probability of σ and 1, and then normalize the probabilities of the remaining plans. If $\sigma = \langle a, \nu \rangle$, the FSC functions ψ and η get modified as follows:

Algorithm 1 Function **makeMove**

```
Input:
  $\psi$   // action selection function
  $\eta$   // observation strategy function
  $n$   // node at which the move is executed
  $\langle a, \nu \rangle$   // conditional plan to be installed
Output:
  $\psi'$   // new action selection function
  $\eta'$   // new observation strategy function

  $\psi(n, \cdot) \leftarrow$ changeDistribution$(\psi(n,\ \cdot), a)$
  for each $o \in \mathcal{O}$ do
    $\eta(n, o, \cdot)) \leftarrow$ changeDistribution$(\eta(n, o, \cdot),\ \nu(o))$
  end for
```

Algorithm 2 Function **changeDistribution**

```
Input:
  global $changeDist$   // the move fraction parameter
  $dist$   // original probability distribution
  $index$   // index of outcome whose probability will change
Output:
  $newDist$   // new probability distribution

  $oldSum \leftarrow 1 - dist(index)$ // probability of all other outcomes
  $newDist(index) \leftarrow dist(index) + (1 - dist(index)) \cdot moveDist$
  $newSum \leftarrow 1 - newDist(index)$ // probability of all other outcomes

  // normalize probabilities of other outcomes
  for each $outcomeIndex \neq index$ do
    $newDist(outcomeIndex) \leftarrow dist(outcomeIndex) \cdot \frac{newSum}{oldSum}$
  end for
```

The fraction parameter *moveDist* would usually range from 0.5 to 1 (deterministic move). In our experiments, we found out that $moveDist = 0.95$ works well; such high value does not necessarily mean that nearly deterministic moves are better; it could simply be due to the structure of the limited number of POMDP problems we tested.

3.2.2 Q-values

The Q-value of a move (or action, or conditional plan) at a state $s \in \mathcal{S}$ is determined by its immediate reward at s, and the value of executing the current policy afterward.

Here, we define several Q-values for several possible interpretations of moves in the finite policy graph space. In our actual algorithm, a move will correspond to installing a conditional plan at a specific node; however, it is worthwhile considering other possibilities.

Conditional plans

Let's assume that the current policy graph is π, and its value is V^π. First, we consider the set of full conditional plans Σ. A conditional plan σ is a pair $\langle a, \nu \rangle$, where $a \in A$ and $\nu \in \mathcal{N}^{\mathcal{O}}$. The Q-value $Q^\pi(\bar{s}, \sigma)$ is simply

$$Q^\pi(\bar{s}, \sigma) = \bar{R}^\pi(\bar{s}) + \gamma \sum_{\bar{s}'} \bar{T}^\pi(\bar{s}, \bar{s}') V^\pi(\bar{s}'). \tag{3.3}$$

In terms of the underlying POMDP, we get:

$$\begin{aligned} Q^\pi(\langle n, s\rangle, \langle a, \nu\rangle) &= R(s,a) + \gamma \sum_{n',\, s'} T(s,a,s') \sum_{o|\nu(o)=n'} Z(s',a,o)\; V^\pi(\langle n', s'\rangle) \\ &= R(s,a) + \gamma \sum_{s'} T(s,a,s') \sum_{n'} V^\pi(\langle n', s'\rangle) \sum_{o|\nu(o)=n'} Z(s',a,o). \end{aligned} \tag{3.4}$$

Notice that the expression for $Q^\pi(\langle n, s\rangle, \langle a, \nu\rangle)$ does not depend on the node n, since ν is a conditional observation mapping. Thus, for conditional plans σ we can use simpler notation:

$$Q^\pi(\langle \cdot, s\rangle, \sigma) = Q^\pi(s, \sigma), \tag{3.5}$$

and

$$Q^\pi(s, m) = Q^\pi(s, \sigma), \tag{3.6}$$

where $m = \langle n, \sigma \rangle \in \mathcal{M}$ is a move (node + conditional plan), as defined above.

Using full conditional plans as moves has both its advantages and disadvantages. The biggest problem is that the number of such plans is exponentially dependent on the number of observations:

$$|\mathcal{M}| = |\mathcal{A}||\mathcal{N}|^{|\mathcal{O}|}. \tag{3.7}$$

Therefore, for bigger observation spaces, we need to resort to sampling or limiting the graph connectivity in order to enumerate such moves. On the other hand, the combinatorial nature of full conditional plans helps overcome the local optima problems of gradient ascent.

Partial observation strategies

It is also useful to define a Q-value $Q^\pi(\cdot, \langle a, \rho \rangle)$ for an action $a \in \mathcal{A}$ and a *partial* observation strategy $\rho \in \mathcal{O} \times \mathcal{N}$. If $\rho = \langle o, \tilde{n} \rangle$, then the probability distributions remain the same for all observations except o; if an agent receives observation o, it transitions deterministically to the node $\tilde{n}$.

$$Q^\pi(\langle n, s \rangle, \langle a, \rho \rangle) = R(s, a) + \gamma \sum_{n',\, s'} T^{\langle a,\rho \rangle}(\langle n, s \rangle, \langle n', s' \rangle)\, V^\pi(\langle n', s' \rangle), \tag{3.8}$$

where

$$T^{\langle a,\rho \rangle}(\langle n, s \rangle, \langle n', s' \rangle) = T(s, a, s') \left[I(\rho, n')\, Z(s', a, o) + \sum_{o' \neq o} \eta(n, o', n')\, Z(s', a, o') \right]. \tag{3.9}$$

$I(\rho, n')$ is an indicator variable:

$$I(\rho, n') = I(\langle o, \tilde{n} \rangle, n') = \begin{cases} 1 & \text{iff } \tilde{n} = n', \\ 0 & \text{otherwise.} \end{cases} \tag{3.10}$$

The move space now becomes of manageable size:

$$|\mathcal{M}| = |\mathcal{N}||\mathcal{A}||\mathcal{O}||\mathcal{N}|. \tag{3.11}$$

The Q-value does depend on the node at which action a and partial observation mapping ρ get installed since n appears on the right-hand side of Equation 3.8.

Such moves are less likely to lead to local optima than moves that consider changing action and partial observation strategies independently (as GA does). However, our experience has shown that in most cases they still exhibit similar suboptimal behavior as gradient ascent.

3.2.3 Heuristic function

We would like to find a heuristic for evaluating a set of possible moves in the policy graph space. From now on, we will assume that our moves are full conditional plans ascribed to a controller node. However, since the Q-value does not depend on the node, given any conditional plan, the value of the move that includes that plan is the same for all FSC nodes $n \in \mathcal{N}$.

The following description of the linear program also works if moves are elementary actions $\mathcal{A}$ or a pair of an action and a partial observation mapping for a particular observation $o \in \mathcal{O}$; in the latter two cases, we have to evaluate moves at specific nodes, since the Q-value of such moves depends not only on the underlying system state s, but also on the policy graph node n.

We can either consider all possible moves $\mathcal{M}$ or a subset of moves $\mathcal{M}' \subseteq \mathcal{M}$, in case we either have some additional heuristic for limiting the set of moves, or we just resign to sampling from the set $\mathcal{M}$ for computational reasons. For each move $m \in \mathcal{M}'$ we need to have its Q-value vector $Q^\pi(\cdot, m)$ for all states s. We will assume that our current policy graph is π and drop the superscript, i.e., $Q^\pi(s,m) \equiv Q(s,m)$.

We will also use the notation $Q(b,m)$ to denote the Q-value of the move m at belief

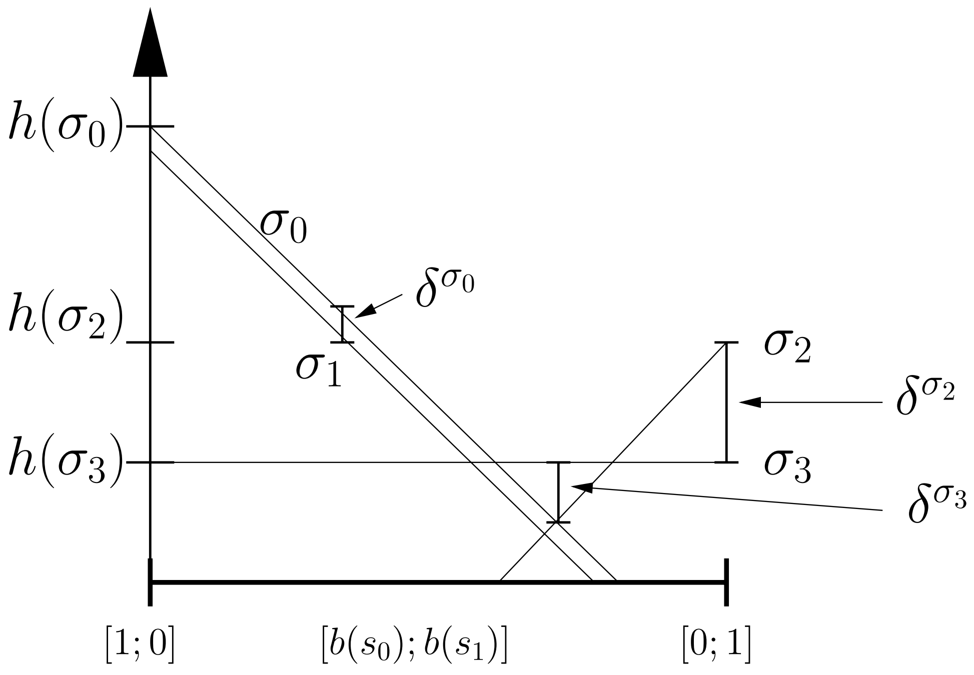


Figure 3.3: Heuristic and δ-values for several conditional plans.

state b. The value of $Q(b, m)$ is simply the expectation with respect to b:

$$Q(b, m) = \sum_{s \in \mathcal{S}} b(s)\, Q(s, m). \tag{3.12}$$

Gradient-based approaches fall into the category of methods that consider the value of moves *with respect to belief states b that are reachable from the initial belief state* b_0 in the current policy graph. However, other moves could be potentially useful at other belief states, if the controller structure changed so that those belief states became reachable from b_0. Our goal is to rank conditional plans according to a heuristic function that takes into account the potential value of a plan at belief states that might not be reachable in the current FSC. Therefore, we propose the following heuristic for evaluating moves: For each move $m \in \mathcal{M}'$,

1 Find the belief state b_1^m such that the *difference* between the Q-value of the move m and any other move in $\mathcal{M}'$ is maximized:

$$b_1^m = \arg\max_b \left[Q(b, m) - \max_{m'} Q(b, m') \right], \ m' \in \mathcal{M}' - \{m\}. \tag{3.13}$$

We will denote by δ^m the maximum difference between the Q-value of m and any other move at belief state b_1^m:

$$\delta^m = Q(b_1^m, m) - \max_{m'} Q(b_1^m, m'), \text{ for all } m' \in \mathcal{M}' - \{m\}. \tag{3.14}$$

2 If $\delta^m \geq 0$, then there exists a belief state b_1^m at which the move m is as good as any other, so we should consider it. However, there might be the whole region of belief states that yield the same δ^m; we denote it as $\mathcal{B}^m$. Since the magnitude of δ^m is less significant, we would like to find the best possible Q-value for the move m subject to the constraint that $b \in \mathcal{B}^m$, i.e. we optimize within belief states that achieve δ^m. Our heuristic value for the move m is thus $h(m) = Q(b_2^m, m)$, where

$$b_2^m = \arg\max_b \; Q(b, m), \; b \in \mathcal{B}^m, \tag{3.15}$$

or, equivalently, given δ^m from Step 1,

$$b_2^m = \arg\max_b \; Q(b, m), \tag{3.16}$$

$$\text{subject to } Q(b, m) \geq \max_{m'} Q(b, m') + \delta^m, \; m' \in \mathcal{M}' - \{m\}.$$

$b^m \equiv b_2^m$ can be thought of as a "witness" belief state for the move m. The second equation allows to formulate the problem as a linear program. Thus, we can calculate the heuristic $h(m)$ for each move $m \in \mathcal{M}'$ by sequentially solving two linear programs with $|\mathcal{S}| + 1$ variables and $|\mathcal{M}'| - 1$ constraints.

Figure 3.3 plots the Q-value vectors for a hypothetical POMDP. Intuitively, we would like to select the plan σ_0, since it is valuable over a large belief region $\mathcal{B}^{\sigma_0}$. Its δ-value δ^{σ_0}, however, is quite small compared to δ^{σ_2} and δ^{σ_3}, because of the parallel vector σ_1. Therefore, we prefer to look at Q-values. After the first LP, the witness belief state returned by the LP algorithm may lie anywhere within the region $\mathcal{B}^{\sigma_0}$; Q-values in this region range from $h(\sigma_3)$ to $h(\sigma_0)$, which makes it hard to differentiate among the three plans σ_0, σ_2, and σ_3. The second LP maximizes the Q-value of σ_0 within the region $\mathcal{B}^{\sigma_0}$; the maximum is the heuristic value $h(\sigma_0)$.

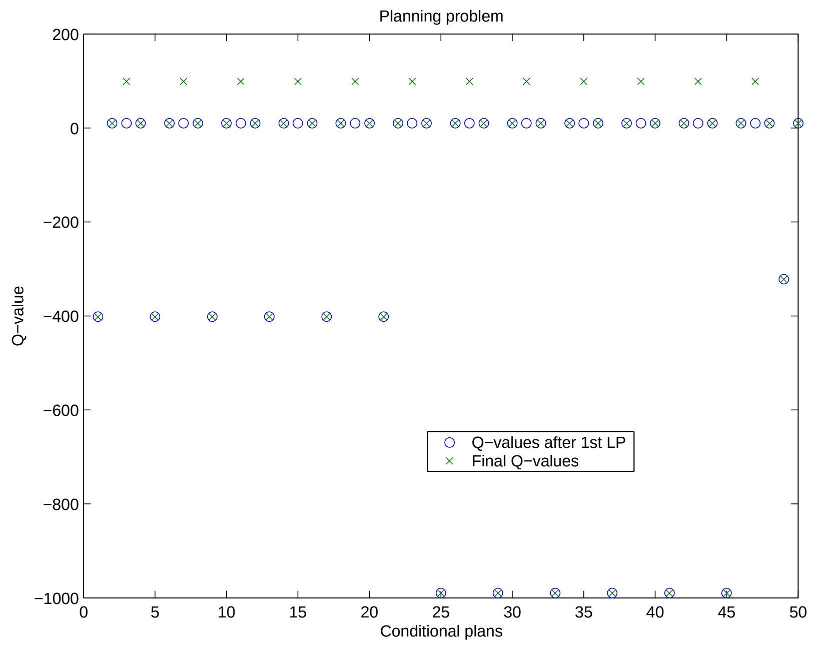


Figure 3.4: Q-values of 50 conditional plans after the 1st and 2nd LP (from the Planning POMDP of Section 4.2).

Figure 3.4 shows the Q-values of 50 conditional plans from the Planning POMDP described in Section 4.2. After the first LP, we get the Q-values at witness belief states b_1 (depicted by circles). The second LP increases the Q-values of some plans (depicted by crosses) by one hundred and differentiates the moves that lead to an optimal controller.

3.2.4 Tabu search

Tabu search is a general local search method that utilizes search history to guide the search process and escape from local suboptima [Glo89, Glo90]. A common meta-strategy is to maintain a list, called *tabu list*, of recent moves that represents the short-term search memory. The moves on the list are not considered when selecting the next move. The rationale is that in such a case, the tabu moves are given a chance to prove their "usefulness", since the effect of local optima attractors is minimized and some cycles are avoided.

In our algorithm, the moves recommended by the heuristic function $h(\cdot)$ are good only at certain belief regions, and often reduce the objective function value. If we later attempt to improve the controller value, such moves can get undone without fulfilling their long-term potential. Therefore, we add the moves to a tabu list with a hope that meanwhile the controller changes so that belief regions in which the tabu moves have high-value become reachable from the initial belief state. In the current implementation, the tabu list is simply a queue of a fixed size, pre-determined by the size parameter tl.[3] The elements of the list are simply the nodes of the finite policy graph; adding a new node to the list removes the oldest one in the list.

Witness belief list

In addition to the tabu list, we keep another list that represents a different aspect of short-term search history. Each move m that is chosen according to the heuristic value $h(m)$ also has an associated witness belief state b^m at which its Q-value is highest, and a belief region $\mathcal{B}^m$ in which it dominates other moves. While the witness belief state b^m is accessible with no extra computational cost as a by-product of the Q-value heuristic LP, there is no easy way to represent the whole region B^m. It often happens that there are many good moves in the same belief region; installing more than one of them at different nodes might simply waste the controller capacity.

We therefore view the witness belief state b^m as a representative of the whole region B^m and add it to a separate witness belief tabu list. If another move is considered whose witness belief state is "near" a tabu belief state, we assume that they share the same belief region; therefore, such a move is not useful. Maintaining a witness belief tabu list allows one to rule out subsequent moves that duplicate the effect of previous conditional plans.

[3]More generally, the tabu list size could be dynamically determined by the history of past configurations (one example is *Reactive Search* [Bat96]).

The "nearness", or distance between two belief states can be defined in a variety of ways. For our experiments, we used the belief discretization technique of Geffner and Bonnet [GB98], but more suitable measures warrant further research. Given an integer resolution parameter $r > 0$, the probabilities $b(s)$ are discretized into r discrete levels. Two belief states b and b' are close if their discretized representation is the same, that is $round(b(s) \cdot r)/r = round(b'(s) \cdot r)/r$, for all $s \in \mathcal{S}$.

Witness belief states get associated with the nodes at which their respective conditional plans are installed. Once a new move is executed at a node, its witness belief state is removed.

3.3 Algorithm

We can now describe our stochastic local search in the space of finite state controllers. At each iteration, or step, of our SLS algorithm, we perform two kinds of moves: one or more "local" moves and a "global" move. In the local stage, we instantiate nodes with conditional plans that are good at some belief state by choosing moves according to the Q-value heuristic $h(\cdot)$. In the global stage, we select the move that increases the overall value of the controller with respect to the initial belief state. At the end of each iteration, we perform a gradient ascent on the resulting FSC, and record the maximum value attained. Here is the outline of the SLS procedure (for the pseudocode version, see Algorithms 3–5 below):

While search termination criteria are not met, do the following:

- Perform local moves:
  - *sample* a conditional plan σ according to the Q-value heuristic $h(\sigma)$ (plans with higher h-values are given greater weight in the sampling distribution) while ensuring that no node in the FSC already has a witness belief state b^σ;

  - choose a non-tabu node n which is either not reachable from the starting node or which leads to the highest increase in the FSC value when instantiated with the plan σ;
  - perform the local move $\langle n, \sigma \rangle$, add the node n to the move tabu list and the witness belief state b^σ to the witness belief tabu list.

- Perform a global move:
  - Sample a given number of conditional plans, consider installing them at non-tabu nodes, and select the move according to the increase in the FSC value with respect to the prior belief b_0;
  - make the selected move, add the node to the tabu-list and remove the witness belief state ascribed to that node.

- Run gradient ascent starting from the current FSC and record the value achieved (GA does not change the current FSC).

This procedure can be viewed as a particular case of *iterated local search* and *tabu search* [Hoo98]. Iterated local search uses two types of SLS steps to avoid getting stuck in local optima of the objective function: one for reaching local optima as efficiently as possible, and the other for effectively escaping from local optima.

In our algorithm, local (and, to an extent, global) moves play the role of *perturbation* procedure. They attempt to modify the current solution in a way which cannot be immediately undone by the subsequent greedy optimization phase. This allows the process to escape into a region of a different local attractor. Gradient ascent then provides the most efficient way to reach the new local optimum.

We now describe the three stages of our algorithm in more detail.

3.3.1 Local moves

In the local stage (see Algorithm 4), we attempt to find moves that bring high rewards for some belief states (even though such belief states might not be reachable in the current controller). Given a set of conditional plans (either all possible, or a random sample), we evaluate them using the Q-value heuristic described above. We then remove plans whose witness belief states are near some state on the witness belief tabu list. Finally, we sample one conditional plan from a probability distribution that gives greater weight to plans with higher h-values. We use the soft-max, or Boltzmann function, to weigh the plans in Σ:

$$Pr(\sigma) = \frac{e^{h(\sigma)\,\theta}}{\sum_{\sigma' \in \Sigma} e^{h(\sigma')\,\theta}}, \tag{3.17}$$

where θ is the *temperature* parameter. Since it is not of central importance, in our experiments we choose this parameter for each problem by inspecting the heuristic function values; in the future, it would certainly be desirable to automate this aspect of the algorithm.

Together with the heuristic $h(\sigma)$, the linear program also returns the witness belief state b^{σ} at which this heuristic value is achievable. In general, there may be a lot of conditional plans that are good for the same belief state; if they get installed at different nodes, controller capacity gets wasted. To prevent that, we attach a witness belief state for a conditional plan to a node at which it gets installed. The witness belief tabu list $beliefList$ thus holds paired nodes and belief states, or, in our case, their discretized forms. The resolution parameter r, described in Section 3.2.4, was chosen by hand; for our experiments, r ranged from 10 to 50.

There are two main heuristics for selecting a non-tabu node to which we apply the conditional plan:

- We could just pick a move (i.e., the selected conditional plan + a node) that results in the highest controller value among all non-tabu nodes; however,

- We also need to consider unreachable nodes, i.e. the nodes that have no incoming transitions. Applying a conditional plan to such a node will not change the controller value; therefore, unreachable nodes will not be considered in the global or GA stage. However, such nodes represent unused capacity of the controller, and instantiating them with good conditional plans is one way of eventually making them useful.

To balance the two heuristics, we probabilistically choose one or another. In our experiments, if there are any unreachable nodes, we choose such a node with probability 0.9, and the node that leads to the highest increase in value — with probability 0.1.

3.3.2 Global moves

In this stage (see Algorithm 5), we would like to choose a move that increases the overall controller value. One of the simplest ways is to sample a number of conditional plans, apply them to nodes, and select the move according to the increase in controller value. The selected node is added to the tabu list.

The global stage is essentially a form of stochastic hill-climbing. While the local stage instantiates nodes with moves that are useful for *some* belief states, what we ultimately care about is the value of the policy graph with respect to the prior belief. Therefore, in the global stage we select moves that increase the controller value; in most cases, such moves would link to the nodes instantiated with useful moves in the local stage. Thus, the global stage verifies the usefulness of moves proposed in the local stage.

3.3.3 Gradient ascent

The previous stages propose good starting points for GA; they are designed to prevent the GA from getting stuck in the same local suboptimum. At the end of each step, we still run GA in order to reach a (new, ideally) local optimum in a computationally efficient manner; with good starting points proposed by our algorithm, we can hope that one of these local optima will happen to be global.

Algorithm 3 Policy search algorithm

```
Input:
    global $POMDP$   // encoding of POMDP
    $nNodes$   // number of free nodes in policy graph
    $nSamplesLocal$   // number of moves to consider in Local stage
    $nSamplesGlobal$   // number of moves to consider in Global stage
    $tl$   // the size of tabu list
    $nLocalMoves$   // number of moves to make in Local stage
Output: $\pi^*$   // the best policy graph found

1:  global $tabuList \leftarrow$ initializeTabuList($tl$)
2:  global $beliefList \leftarrow$ initializeBeliefList($nNodes$)
3:  $\pi \leftarrow$ randomGraph($nNodes$)
4:  $v^* \leftarrow -Inf$

5:  repeat
6:      // Local stage
7:      for $i = 1..nLocalMoves$ do
8:          $chosenMove \leftarrow$ getLocalMove($nSamplesLocal, \pi$)
9:          $beliefList \leftarrow$ addBelief($chosenMove.node, chosenMove.belief$)
10:         $tabuList \leftarrow$ addTabu($chosenMove.node$)
11:         $\pi \leftarrow$ makeMove($chosenMove, \pi$)
12:     end for

13:     // Global stage
14:     $chosenMove \leftarrow$ getGlobalMove($nSamplesGlobal, \pi$)
15:     $beliefList \leftarrow$ removeBelief($chosenMove.node$)
16:     $tabuList \leftarrow$ addTabu($chosenMove.node$)
17:     $\pi \leftarrow$ makeMove($chosenMove, \pi$)

18:     // Gradient ascent stage
19:     $v', \pi' \leftarrow$ gradientAscent($\pi$)
20:     if $v' \geq v^*$ then
21:         $v^* \leftarrow v'$
22:         $\pi^* \leftarrow \pi'$
23:     end if

24: until search termination condition is met
```

Algorithm 4 Function **getLocalMove**

```
Input:
    global $POMDP$   // encoding of POMDP
    global $tabuList$   // list of tabu nodes
    global $beliefList$   // witness belief states associated with nodes
    $\pi$   // current policy graph
    $nSamplesLocal$   // number of moves to consider in Local stage
Output: $chosenMove = \langle node, condPlan, belief \rangle$

 1: $sampledCondPlans[\,] \leftarrow$ sampleCondPlans($nSamplesLocal$)
 2: $QvalueHeuristics[\,]$, $witnessBeliefs[\,] \leftarrow$
    getQvalueHeuristics($sampledCondPlans[\,]$)
    // get Q-value heuristic and associated witness belief state
    // for each sampled move by solving LP
 3: sample conditional plan $chosenCondPlan$:
    – according to Q-value heuristic; and,
    – so that no node in graph has the same witness belief state
 4: $chosenBelief \leftarrow$ witness belief associated with $chosenCondPlan$

 5: $maxV \leftarrow -Inf$
 6: for $iNode = 1..nNodes$ do // Choose node
 7:    if $iNode$ is not in $tabuList$ then
 8:       $\pi' \leftarrow$ makeMove($\langle iNode, chosenCondPlan \rangle, \pi$)
 9:       $v' \leftarrow value(\pi')$
10:       if $v' \geq maxV$ then
11:          $maxV \leftarrow v'$
12:          $chosenNode \leftarrow iNode$
13:       end if
14:    end if
15: end for
16: if there are unconnected nodes then
17:    if some random choice criterion is met then
18:       $chosenNode$ is one of unconnected nodes
19:    end if
20: end if
21: $chosenMove.node \leftarrow chosenNode$
    $chosenMove.condPlan \leftarrow chosenCondPlan$
    $chosenMove.belief \leftarrow chosenBelief$
22: return $chosenMove$
```

Algorithm 5 Function **getGlobalMove**

Input:

global $POMDP$ // encoding of POMDP
global $tabuList$ // list of tabu nodes
π // current policy graph
$nSamplesGlobal$ // number of moves to consider in Global stage

Output: $chosenMove = \langle node, condPlan \rangle$

```
 1: $sampledCondPlans[\,] \leftarrow$ **sampleCondPlans**($nSamplesGlobal$)
 2: $maxV \leftarrow -Inf$
 3: for $i = 1..n$ do
 4:   for $iNode = 1..nNodes$ do
 5:     if $iNode$ is not in $tabuList$ then
 6:       $iCondPlan \leftarrow sampledCondPlans[i]$
 7:       $\pi' \leftarrow$ **makeMove**($\langle iNode, iCondPlan \rangle, \pi$)
 8:       $v' \leftarrow value(\pi')$
 9:       if $v' \geq maxV$ then
10:         $maxV \leftarrow v'$
11:         $chosenNode \leftarrow iNode$
12:         $chosenCondPlan \leftarrow iCondPlan$
13:       end if
14:     end if
15:   end for
16: end for
17: $chosenMove.node \leftarrow chosenNode$
18: $chosenMove.condPlan \leftarrow chosenCondPlan$
19: return $chosenMove$
```

Chapter 4

Experiments

The following set of experiments illuminates various aspects of the SLS algorithm and compares its performance to GA on several examples drawn from the research literature. The algorithm parameters, such as the number of local moves per iteration, the number of conditional plans sampled, and the size of the tabu list, have been tuned to specific problems. Our SLS procedure was implemented in Matlab and run on Xeon 2.4GHz computers; the linear programs were solved using ILOG CPLEX 7.1 LP optimizer.

We look at the following domains:

- **Load/Unload** is a commonly used toy POMDP from literature. Our results verify that the SLS algorithm performs no worse than gradient ascent.
- **Planning** problem is used to demonstrate problems with gradient ascent and justify the specifics of our algorithm.
- **Heaven/Hell** is a harder problem from POMDP literature. It is a challenging domain for our algorithm, and (with minor modifications) an insurmountable problem for gradient ascent.
- **Preference Elicitation** is a promising domain of application for our algorithm. We experiment both with discrete and continuous state spaces.

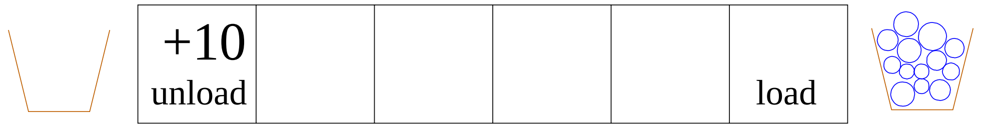


Figure 4.1: Load/Unload problem

4.1 Load/Unload

Load/Unload is a simple problem that has been used to demonstrate the benefits of gradient-based policy search [PMK99, MKKC99, Abe01].

Problem description

In the Load/Unload POMDP (see Figure 4.1), an agent moves between "Load" (L) and "Unload" (U) locations and receives a reward of 10 every time it enters the "Unload" location after having first visited the "Load" location. The environment is only partially observable, because the agent cannot discriminate among the intermediate locations; the observation space is thus $\mathcal{O} = \{load, unload, null\}$. The agent can perform two actions: $\mathcal{A} = \{left, right\}$. Following the problem settings in [MKKC99, Abe01], we assume that transition and observation probabilities are deterministic. The discount factor is 0.99, and the agent starts in the "unload" location with no load.

Results

The Load/Unload problem is fairly straightforward and readily amenable to GA. It has been used to test various policy-based POMDP algorithms because of two main reasons: first, since the problem is quite easy, the results obtained are expected to reflect an upper bound on the performance of algorithms on a variety of problem instances; second, no matter how many intermediate locations are specified (i.e., regardless of the size of the system state space $\mathcal{S}$), the optimal policy can be expressed by a two-node policy graph, depicted in Figure 4.2.

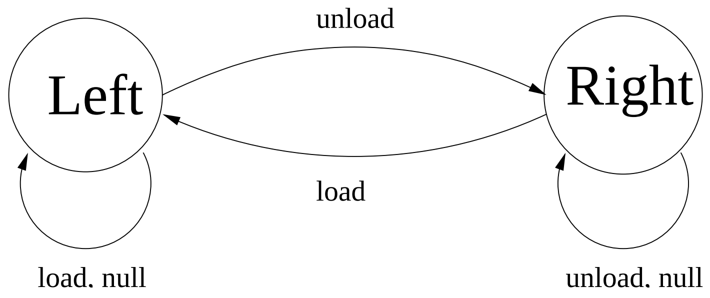


Figure 4.2: Optimal policy graph for the Load/Unload problem

We tested our algorithm on this problem to ensure that it performs as well as GA on relatively “easy” problems. With 2 nodes and 6 locations (10 states), GA with random starting FSCs finds the optimal policy 46.4% of the time (in 1000 trials), a near-optimal policy 44.3% of the time, and fails (converging on a poor policy) 9.3% of the time. The optimal policy value is 9.5538; a near-optimal policy results if the first action is to move left instead of going straight to the “load” location; its value is 9.4583. Poor policies have values ranging from 0 to 6.0205.

Our SLS algorithm always finds the optimal controller, taking on average 4.11 iterations (averaged over 1000 runs). Because the FSC size is so small, the computational cost for SLS is not much greater than GA. GA takes on average 0.20 seconds; one SLS iteration (with GA time included) — 0.40 seconds. Figure 4.3 shows how the complexity of gradient ascent increases as a function of state space size (for each state space size, the GA time is averaged over 20 runs with random initial FSCs); a similar diagram was displayed in [MKKC99], although their absolute execution times seem to be much slower.[1]

[1]For example, GA is reported to take about 1000 seconds for a 200-state POMDP.

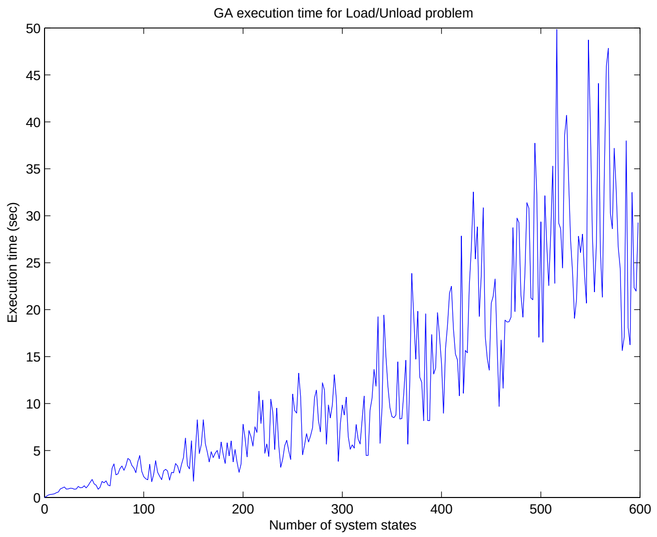


Figure 4.3: Convergence time of gradient ascent for a 2-node FSC as a function of state space size $\mathcal{S}$.

4.2 Planning

The next POMDP is designed to test the intuitions described in Section 3.1. The *Planning* problem requires a sequence of actions to be performed, causing severe difficulties for GA.

Problem description

The state space is composed of four variables with the following domains: $U = \{u_1, u_2, u_3\}$, $V = \{v, \bar{v}\}$, $G = \{g, \bar{g}\}$, $D = \{d, \bar{d}\}$. The actions k, l, m, n have conditional effects on these variables. Action k causes U to take value u_2 if executed when u_1 holds (denoted $k : u_1 \rightarrow u_2$); similarly, $k : u_2 \rightarrow d$ and $k : u_3 \rightarrow d$. The other actions have these effects—$l : u_1 \rightarrow g$; $l : u_2 \rightarrow u_3$; $l : u_3 \rightarrow d$; $m : u_1 \rightarrow g$; $m : u_2 \rightarrow d$; $m : u_3 \rightarrow v$; $n : u_1 \rightarrow g$; $n : u_2 \rightarrow d$; and $n : u_3 \rightarrow d$. The initial state is $\langle u_1, \bar{v}, \bar{g}, \bar{d} \rangle$, and states where either V, G, or D is true are terminal. The reward function is additive: d has a cost of -1000 (disaster), g is worth +10 (good), and v is worth +100 (very good). Action effects

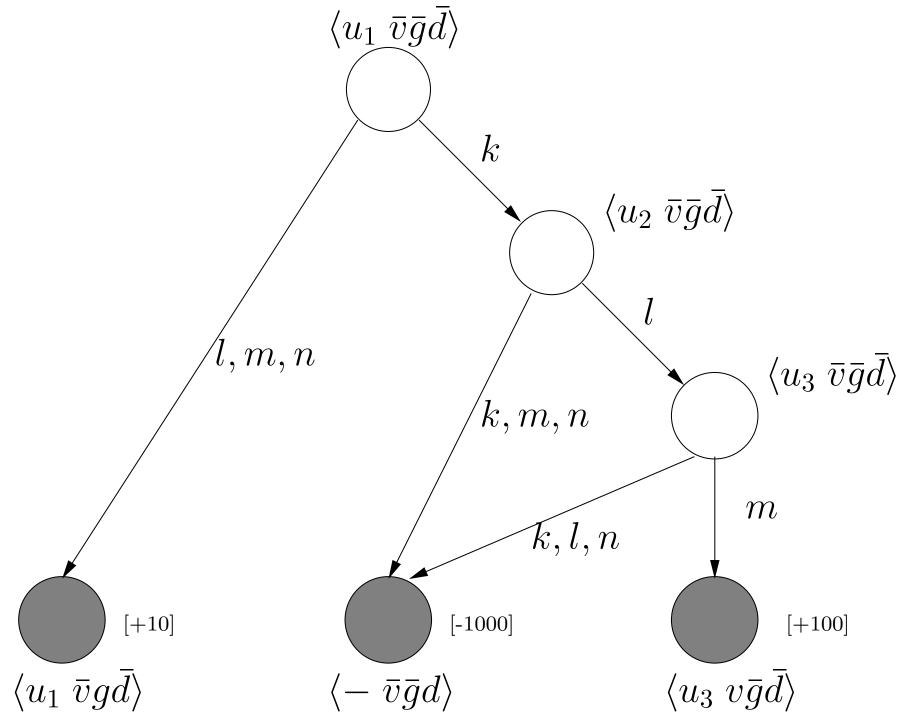

$\langle u_1\ \bar{v}\bar{g}\bar{d}\rangle$
k
$\langle u_2\ \bar{v}\bar{g}\bar{d}\rangle$
l
$\langle u_3\ \bar{v}\bar{g}\bar{d}\rangle$
l, m, n
k, m, n
m
k, l, n
[+10]
[-1000]
[+100]
$\langle u_1\ \bar{v}g\bar{d}\rangle$
$\langle -\ \bar{v}\bar{g}d\rangle$
$\langle u_3\ v\bar{g}\bar{d}\rangle$

Figure 4.4: State transition diagram for Planning POMDP.

are deterministic. The diagram of the problem setting is shown in Figure 4.4.

Results

The optimal policy is to execute the sequence k, l, m, attaining a reward of 100. However, in virtually all randomly initialized FSCs, increasing the probability of action k while keeping everything else constant leads to a decrease in FSC value; therefore, GA methods almost always converge to a suboptimal policy, choosing actions l, m, n at the outset and receiving 10. Using a controller with 6 nodes, we ran GA 6000 times: the policy reached achieves the suboptimal value of 10 in every trial. In 6000 trials, our SLS algorithm found the optimal FSC each time, taking on average 3.14 iterations. On average, GA takes 0.29 seconds; one SLS iteration, 3.08 seconds (with no conditional plan sampling).

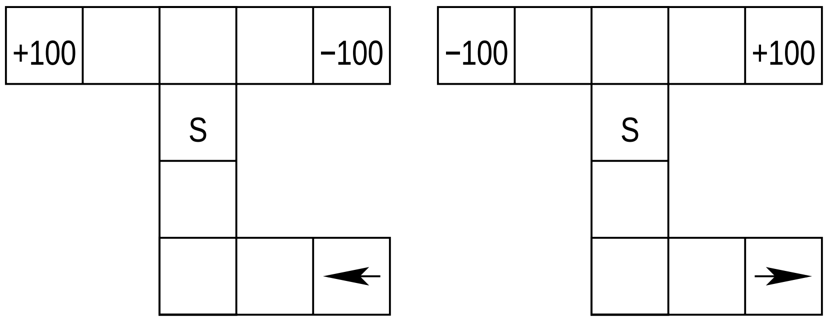


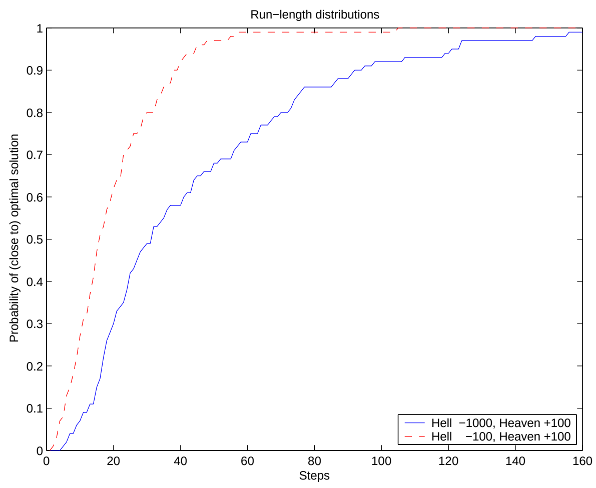
Figure 4.5: Heaven/Hell problem with left and right worlds.

4.3 Heaven/Hell

The Heaven/Hell problem appeared (in various forms) in [AB02, Abe01, Thr99, GB98].

Problem description

The agent starts with equal chance in either "left" or "right" worlds (in the location marked by "S" in Figure 4.5), and, because of partial observability, does not know which. The (left or right) arrow (Figure 4.5) conveys information about the location of "Heaven" (positive reward); if the agent does not observe the arrow, it risks falling into "Hell" (negative reward). The arrow locations are fully observable, emitting observations $left$ or $right$. Heaven and Hell locations are absorbing – the process terminates once the agent reaches one of them. The top and bottom cells in the center column are fully observable; any other state is aliased by the $default$ observation. The observation space thus consists of five observations: $left, right, top, bottom, default$. The agent has four actions at its disposal, and can move $up, down, left, right$; it stays in the same location if an action is not appropriate (such as moving left in one of the starting states). In our experiments, the state transitions were deterministic, and there was no observation noise. An optimal FSC needs eight nodes.



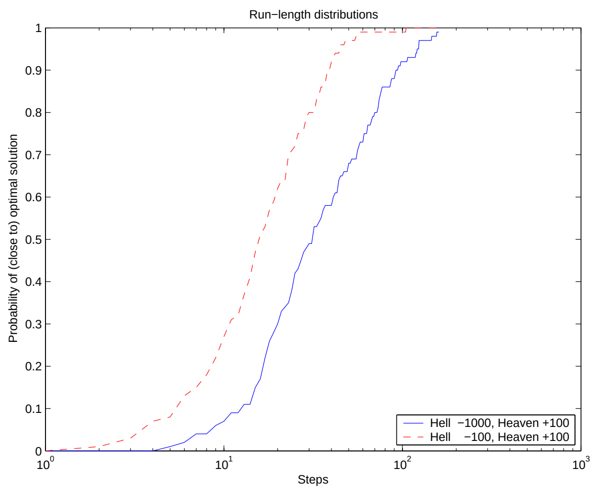


Figure 4.6: Run-length distributions for the two variants of Heaven/Hell (100 trials). On the right, RLDs are plotted on a logarithmic scale.

Results

In [AB02, GB98], both Heaven and Hell have symmetric rewards (e.g., +100 and -100). Since the average reward of going up and blindly choosing either Heaven or Hell is zero, and information cost is also zero, GA methods have a chance of finding a reasonably good policy (since with regard to the initial belief state, going down is no worse than going up). Our experiments with 12-node controllers show that when Heaven and Hell locations have symmetric rewards, GA achieves approximately half of the optimal value (but never attains the optimal). However, it is not hard to make this problem practically unsolvable by GA. If we increase the Hell penalty to -1000, GA will almost always choose the safe alternative of bumping into walls and receiving zero reward, even though the optimal FSC has not changed. Our SLS procedure, on the other hand, does find the optimal solution in both versions of the problem.

To get a sense of how FSC quality improves over time, we plot *run-length distributions* (RLDs) for our method, showing the empirical probability of finding a near (within 10%) optimal FSC as a function of SLS iterations (see Figure 4.6). As we see, after a reasonable

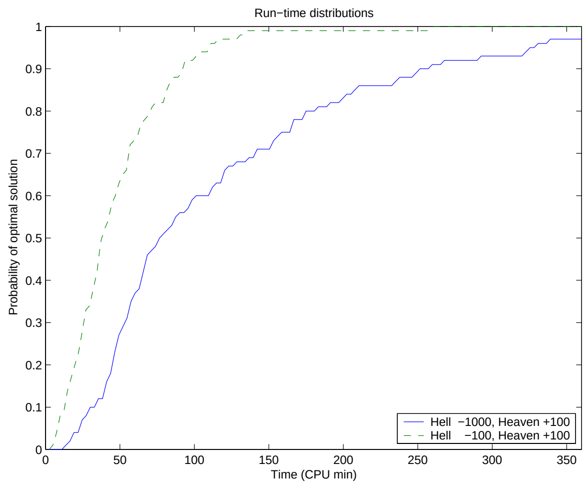


Figure 4.7: Run-time distributions for the two variants of Heaven/Hell (100 trials). Asymmetric rewards make the problem considerably harder to solve.

number of steps, the probability of finding the correct FSC is quite high. The asymmetric problem is, not surprisingly, somewhat harder to solve. Rough computation time (for -1000 penalty case) per iteration is 164 seconds (GA time included); GA time is on average 53 seconds. The run-time distribution (RTD) is shown in Figure 4.7.

For the Heaven/Hell problem, we adopted the technique of [AB02] in order to speed up our algorithm: instead of allowing transitions to any node for each observation, we restrict the number of outgoing links to 3, including a mandatory self-transition. The structure of the FSC (i.e., the allowed connections between nodes) is chosen randomly at the beginning of each trial. This trick reduces the space of conditional plans considered, which currently is the main complexity bottleneck for our algorithm.

4.4 Preference elicitation

A final problem we consider is the preference elicitation (PE) problem described in [Bou02]. The objective is to optimally balance the cost of queries and the gain provided by the elicited information with respect to the quality of the final decision.

Problem description

Preference elicitation is a process of determining user utility functions to the extent necessary to make a decision on their behalf. The main idea is that an optimal decision can usually be made without the full knowledge of preferences. The interaction process can be viewed as a sequence of questions and answers; at any time, there is a trade-off between the quality of the decision an agent makes and the amount of information it must obtain from a user about the relevant preferences. A question is only worth asking if its expected value with regard to the decision quality outweighs the cost.

Boutilier [Bou02] introduces the concept of preference elicitation as a POMDP that takes into account the value of future questions when determining the value of the current question. We can assume a system that makes decisions on behalf of a user; such a system has a fixed set of choices (actions, recommendations) whose effects are generally known precisely or can be modeled stochastically. The main idea of this formulation is to probabilistically quantify the system's uncertainty about a user's true utility function by maintaining a probability distribution over possible functions; the distribution gets updated after each interactive step. The system interacts with a user in a sequential way; at each step it either asks a question, or determines that it has enough information about a user's utility function to make a decision. As each query has associated costs, the model allows the system to construct an optimal interaction policy which takes into account the trade-off between interaction costs and the value of provided information.

In particular, the state space of the preference elicitation POMDP is the set of possible utility functions; actions can be either queries about a user's utility function or terminal decisions; observation space is the set of possible responses to queries. The dynamics of the system is simplified by the fact that the state transition function is trivial: the underlying utility functions never change throughout the interaction process; the observation function maintains a probability distribution of a particular response to a given query for a specific utility function; and, the reward function simply assigns costs

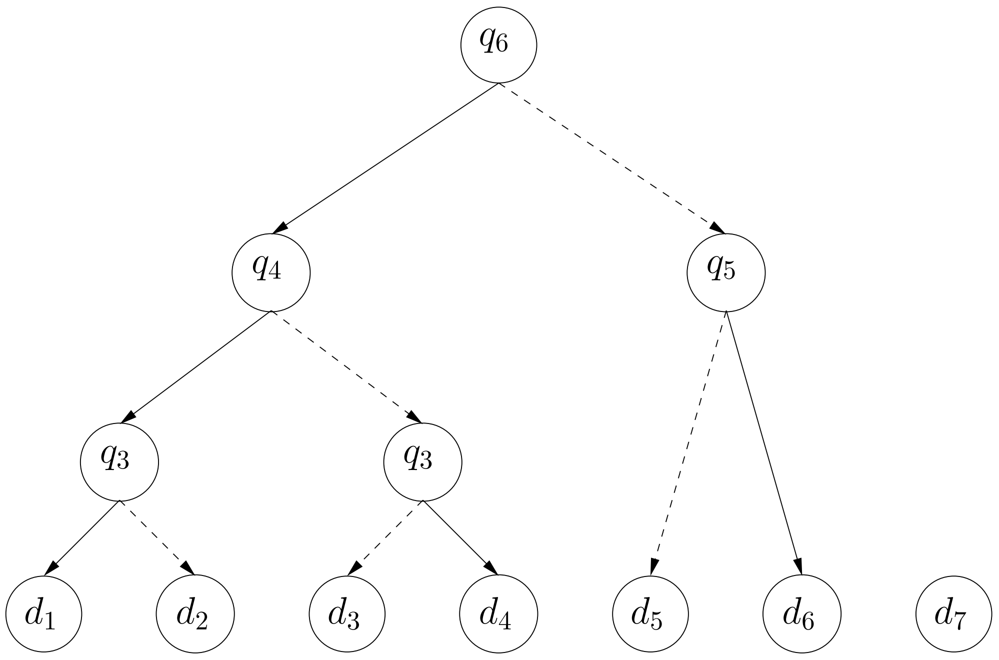


Figure 4.8: Optimal controller for the discrete preference elicitation problem. Solid lines represent *yes* observation links; dashed lines — *no* observation links.

to queries and expected utilities to decisions.

Solving the preference elicitation POMDP is a difficult task. In realistic situations, the state space is continuous and multi-dimensional, so standard methods for solving finite-state POMDPs are no longer applicable. Boutilier [Bou02] presents a value-iteration based method that exploits the special structure inherent in the preference elicitation process to deal with parameterized belief states over the continuous state space; belief states are represented by Gaussian mixture models. We attempted to solve this problem using our SLS algorithm.

Results

We tackle two variants of the problem described in [Bou02]. In the first, we discretize the utility space to six states and the number of actions to 14. In the second, we sample from a continuous state space, but discretize the action space to 70 actions. The decision scenario is the same for both variants. There are seven outcomes $\{s_1, \ldots, s_7\}$, and seven decisions $\{d_1, \ldots, d_7\}$. The decisions d_i, $i \leq 5$, each have a 50% chance of causing outcomes s_i and s_{i+1}, while d_6 causes either s_6 or s_1. Decision d_7 is guaranteed to realize outcome s_7.

In the discrete setting, the 6 utility functions are quantified as follows:

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
u_1	0.9	0.9	0.1	0.1	0.1	0.1	0.3
u_2	0.1	0.9	0.9	0.1	0.1	0.1	0.3
u_3	0.1	0.1	0.9	0.9	0.1	0.1	0.3
u_4	0.1	0.1	0.1	0.9	0.9	0.1	0.3
u_5	0.1	0.1	0.1	0.1	0.9	0.9	0.3
u_6	0.9	0.1	0.1	0.1	0.1	0.9	0.3

This table shows the utility values $u_i(s_j)$ assigned to outcome s_j by utility function u_i.

In our POMDP, the discount factor is 0.99, query cost is 0.02, and observation space is $\{yes, no\}$. When observation probabilities are noiseless, the optimal FSC has 11 nodes (see Figure 4.8). Since this makes it easy to calculate the optimal value, the experiments reported in Figures 4.9 and 4.10 assume noiseless observations. With noise probabilities of 0.03, the optimal controller has at least 15 nodes.

In order to achieve good results, an optimal policy has to execute a precise sequence of queries, and then make an appropriate decision. Since decisions are always terminal, for preference elicitation problems, we assign each decision to a single FSC node. In Figure 4.8, the seven bottom nodes are decision nodes. For queries, we use the following notation: q_i denotes the query “Is the utility of outcome i less than 0.9?”. By following the optimal policy, we can achieve a value of 0.8233. Gradient ascent converges to local suboptima with values no higher than 0.6552. This is illustrated in Figure 4.9, where we plot the best value attained (averaged over 100 trials) at any time by 17 and 22-

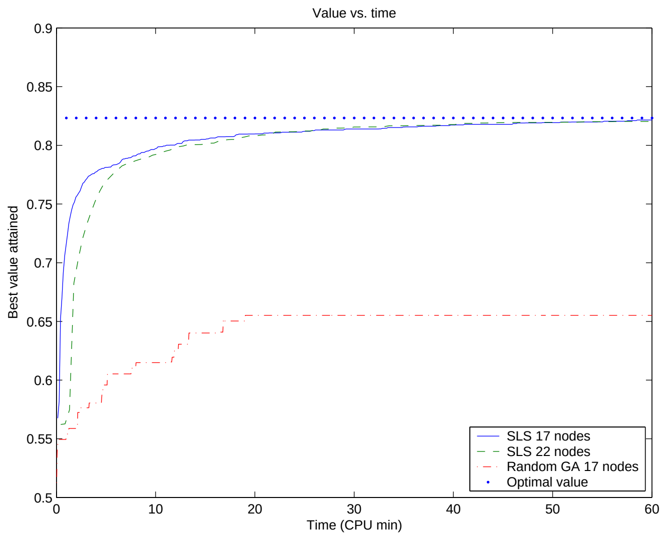


Figure 4.9: Anytime performance of discrete preference elicitation SLS (100 trials).

node controllers using our SLS algorithm vs. pure gradient ascent on random starting configurations. In terms of CPU time the performance of the 17 and 22-node controllers is similar. GA never finds an optimal FSC.

Figure 4.10 shows a plot of run-length distributions and run-time distributions for 17 and 22-node controllers. We can see that 22-node controllers achieve the optimal value in fewer iterations than 17-node controllers.[2] On the other hand, in terms of running time, 17-node controllers perform better because they take less time per step.

We also briefly experimented with sampling from continuous utility spaces. In this case, the utility priors are given by the mixture of uniforms with the following six components (each weighted equally) [Bou02]:

[2]Parameter values for 10 node controller: $nSamplesLocal = 100$, $nSamplesGlobal = 200$, $nLocalMoves = 3$, $tl = 5$; 15-node controller: $nSamplesLocal = 300$, $nSamplesGlobal = 200$, $nLocalMoves = 3$, $tl = 10$.

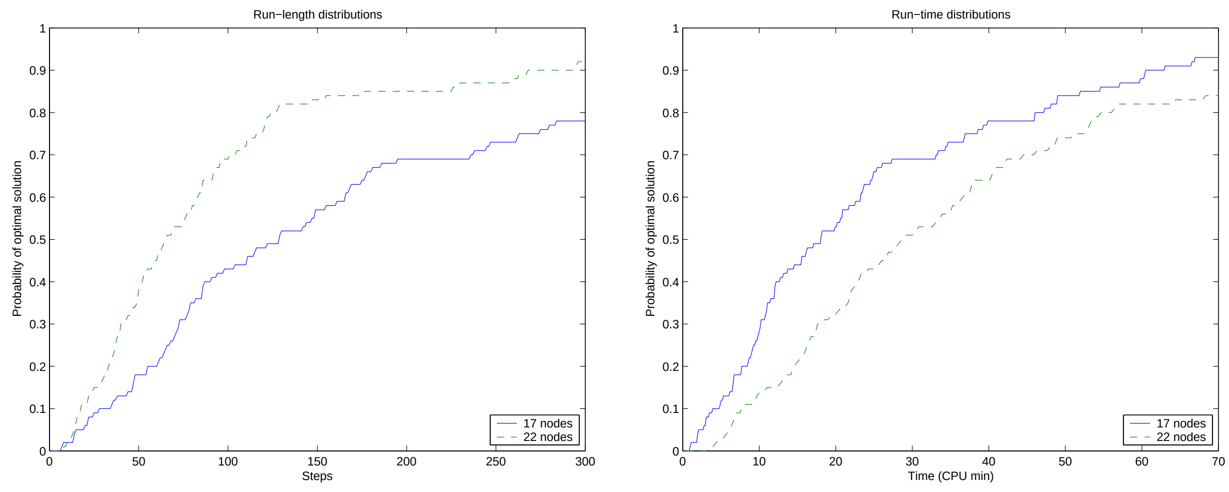


Figure 4.10: Run-length (left) and run-time (right) distributions for discrete preference elicitation POMDP (100 trials).

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
b_1	[.9 1]	[.9 1]	[0 .1]	[0 .1]	[0 .1]	[0 .1]	[.7 .8]
b_2	[0 .1]	[.9 1]	[.9 1]	[0 .1]	[0 .1]	[0 .1]	[.7 .8]
b_3	[0 .1]	[0 .1]	[.9 1]	[.9 1]	[0 .1]	[0 .1]	[.7 .8]
b_4	[0 .1]	[0 .1]	[0 .1]	[.9 1]	[.9 1]	[0 .1]	[.7 .8]
b_5	[0 .1]	[0 .1]	[0 .1]	[0 .1]	[.9 1]	[.9 1]	[.7 .8]
b_6	[.9 1]	[0 .1]	[0 .1]	[0 .1]	[0 .1]	[.9 1]	[.7 .8]

For each belief component b_i and outcome s_j, the table shows the range for which b_i assigns positive uniform density to $u(s_j)$. This prior reflects the fact that the user prefers some pair of adjacent outcomes; however, the pair is unknown to the agent. Outcome s_7 is considered to be a safe alternative.

To solve this continuous space POMDP, we use our SLS algorithm, but *sample* 20 states (utility functions) at each iteration. Then, we calculate the observation function and the reward function for the sampled states. The action space remains discrete, but increased to 70 actions — we now allow nine different queries about the utility of each

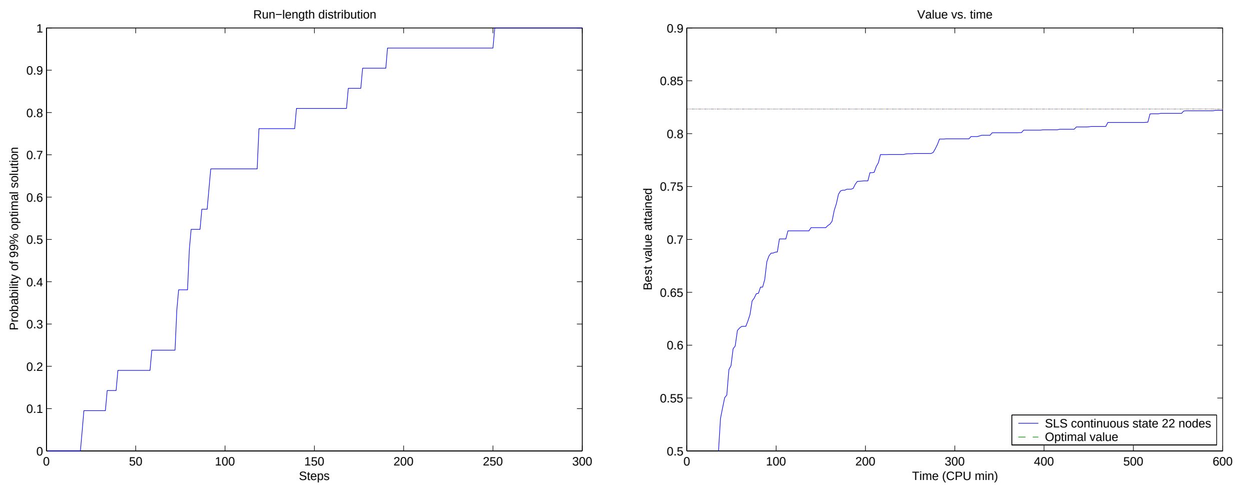


Figure 4.11: Run-length distribution (left) and the best average value attained vs. execution time (right) for continuous preference elicitation problem (22-node controllers, 21 trials). One SLS step takes roughly 140 seconds.

outcome[3] ($7 \times 9 = 63$), and still have 7 terminal decisions. Although our work with continuous preference elicitation problems is at very early stages, the results are quite promising — our SLS procedure does find optimal or near-optimal controllers (see Figure 4.11).

[3]Queries are now of the type "Is utility of outcome s_j less than k", where k is any fraction in the set $\{0.1, 0.2, \ldots, 0.9\}$.

Chapter 5

Conclusions

This thesis provides two main contributions to POMDP research. First, we clearly identified and illustrated the importance of a basic problem with gradient-based FSC search methods — their convergence to local suboptima. Most examples in the previous literature happened to have structure favorable to GA approaches; we showed that whenever the precise sequence of actions is required, GA can lead to arbitrarily poor policies. The problem is inherent in settings where the trade-off between the potential value of information and its cost has to be carefully considered (e.g., in sequential preference elicitation).

Our second contribution is a procedure for stochastic local search in the space of POMDP controllers that combines a computationally attractive GA technique with heuristics that help guide the search toward good (even optimal) FSCs. While more intensive than GA, experiments demonstrate its effectiveness in interesting classes of POMDPs. Our SLS procedure should be useful where exact POMDP solutions are intractable, and GA methods, while more effective, lead to very poor policies. For such POMDPs our method provides an any-time algorithm that seems to perform much better than GA.

The main drawback of the current algorithm is the evaluation of conditional plans whose number is exponential in the number of observations. Our future research will concentrate on finding appropriate heuristics for sampling from the space of conditional plans, allowing much larger FSCs to be dealt with. Ideally, we can find a way to generate useful plans incrementally, like in the Witness POMDP algorithm [KLC98].

We are also planning to pursue the application of our search framework to preference elicitation problems, where continuous state, action, and observation spaces present challenges to value-based POMDP solution methods.

Bibliography

[AB02] Douglas Aberdeen and Jonathan Baxter. Scalable internal-state policy-gradient methods for POMDPs. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 3–10, 2002.

[Abe01] Douglas Aberdeen. Internal-state policy-gradient algorithms for infinite-horizon POMDPs. Technical report, Research School of Information Science and Engineering, Australian National University, Canberra, Australia, July 2001.

[Ast65] K. J. Aström. Optimal control of Markov decision processes with incomplete state estimation. *J. Math. Anal. Appl.*, 10:174–205, 1965.

[Bat96] Roberto Battiti. Reactive search: Toward self-tuning heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, chapter 4, pages 61–83. John Wiley and Sons Ltd, 1996.

[Bel57] Richard E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, 1957.

[BM99] Leemon Baird and Andrew Moore. Gradient descent for general reinforcement learning. *Advances in Neural Information Processing Systems 11*, 1999.

[Bou02] Craig Boutilier. A POMDP formulation of preference elicitation problems. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 239–246, Edmonton, 2002.

[Che88] Hsien-Te Cheng. *Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, University of British Columbia, Vancouver, 1988.

[CKL94] 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*, pages 1023–1028, Seattle, 1994.

[CLZ97] Anthony R. Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact method for POMDPs. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 54–61, Providence, RI, 1997.

[GB98] Hector Geffner and Blai Bonet. Solving large POMDPs by real time dynamic programming. In *Working Notes, Fall AAAI Symposium on POMDPs*, 1998.

[Glo89] F. Glover. Tabu Search – Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[Glo90] F. Glover. Tabu Search – Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[Han97] Eric A. Hansen. An improved policy iteration algorithm for partially observable MDPs. In *Proceedings of Conference on Neural Information Processing Systems*, pages 1015–1021, Denver, CO, 1997.

[Han98a] Eric A. Hansen. Solving POMDPs by searching in policy space. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 211–219, Madison, WI, 1998.

[Han98b] Eric J. Hansen. *Finite-memory control of partially observable systems.* PhD thesis, University of Massachusetts Amherst, Amherst, 1998.

[Hoo98] Holger H. Hoos. *Stochastic Local Search—Methods, Models, Applications.* PhD thesis, TU Darmstadt, Darmstadt, Germany, 1998.

[How60] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, 1960.

[JSJ95] Tommi Jaakkola, Satinder P. Singh, and Michael I. Jordan. Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 345–352. The MIT Press, 1995.

[KLC98] Leslie Pack Kaelbling, Michael Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[Lit94] Michael L. Littman. Memoryless policies: Theoretical limitations and practical results. In Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, 1994. The MIT Press.

[Mat02] The MathWorks. MATLAB Optimization Toolbox 2.2. `http://www.mathworks.com/products/optimization`, 2002.

[McC95] R. Andrew McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 387–395, Lake Tahoe, Nevada, 1995.

[MHC99] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable decision problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 541–548, Orlando, 1999.

[MKKC99] Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony R. Cassandra. Solving POMDPs by searching the space of finite policies. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 417–426, Stockholm, 1999.

[Mon82] George E. Monahan. A survey of partially observable Markov decision processes: Theory, models and algorithms. *Management Science*, 28:1–16, 1982.

[MPKK99] Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling. Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 427–436, Stockholm, 1999.

[PMK99] Leonid Peshkin, Nicolas Meuleau, and Leslie P. Kaelbling. Learning policies with external memory. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 307–314, San Francisco, CA, 1999.

[PT87] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

[Son71] Edward J. Sondik. *The optimal control of partially observable Markov Decision Processes*. PhD thesis, Stanford university, Palo Alto, 1971.

[Son78] Edward J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26:282–304, 1978.

[SS73] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

[Thr99] Sebastian Thrun. Monte Carlo POMDPs. In *Proceedings of Conference on Neural Information Processing Systems*, pages 1064–1070, Denver, 1999.

[WH80] C. C. White and D. Harrington. Application of Jensen's inequality for adaptive suboptimal design. *Journal of Optimization Theory and Applications*, 32(1):89–99, 1980.

[Whi91] Chelsea C. White. A survey of solution techniques for the partially observed Markov decision process. *Annals of Operations Research*, 32:215–230, 1991.

[WS97] Marco Wiering and Juergen Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1997.

[ZL96] Nevin L. Zhang and Wenju Liu. Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Hong Kong University of Science and Technology, 1996.