Automated Hierarchy Discovery for Planning in Partially Observable Domains

by

Laurent Charlin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Laurent Charlin

Abstract

Planning in partially observable domains is a notoriously difficult problem. However, in many real-world scenarios, planning can be simplified by decomposing the task into a hierarchy of smaller planning problems which, can then be solved independently of one another. Several approaches, mainly dealing with fully observable domains, have been proposed to optimize a plan that decomposes according to a hierarchy specified a priori. Some researchers have also proposed to discover hierarchies in fully observable domains.

In this thesis, we investigate the problem of automatically discovering planning hierarchies in partially observable domains. The main advantage of discovering hierarchies is that, for a plan of a fixed size, hierarchical plans can be more expressive than non-hierarchical ones.

Our solution frames the discovery and optimization of a hierarchical policy as a non-convex optimization problem. By encoding the hierarchical structure as variables of the optimization problem, we can automatically discover a hierarchy. Successfully solving the optimization problem therefore yields an optimal hierarchy and an optimal policy. We describe several techniques to solve the optimization problem. Namely, we provide results using general non-linear solvers, mixed-integer linear and non-linear solvers or a form of bounded hierarchical policy iteration. Our method is flexible enough to allow any parts of the hierarchy to be specified based on prior knowledge while letting the optimization discover the unknown parts. It can also discover hierarchical policies, including recursive policies, that are more compact (potentially infinitely fewer parameters).

Acknowledgements

My greatest thanks go out to my adviser, Pascal Poupart, for his eternal enthusiasm, his creativity without borders, his patience, and his eagerness to work, day, weekends and nights. Needless to say that without his efforts this work would not have been possible.

Many thanks also go to Romy Shioda and Shai Ben-David. To the former for enlightening me in the optimization field, for the appropriateness of her ideas and for spending the time and effort to work with me. To the latter for introducing me to what theory means, to opening my eyes to a great field through his class and for his quest of excellence. My deepest gratitude goes to both of them for reading and commenting on this thesis.

Thanks to loved ones and friends home and abroad who were there when I needed it the most and to Anne, for believing in me. Special thanks go to Arnaud Chanu for carefully reading and commenting on this thesis.

Finally, thanks to friends and colleagues at the University for enlightening discussions about science, life in general and also for making me realize that there are 10 distinct provinces in this country. Special thanks to Claude-Guy for his many advices on what grad school is all about and to my roommates Mark, Tyrel, Jeff, Kevin and Martin.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the landmark challenges that fascinates artificial intelligence researchers is the planning problem. Planning can be loosely defined as finding a way to act in an environment in order to fulfill a task. Although, as humans this is a skill long-mastered, planning is a notoriously difficult problem for computers. Challenges come from the fact that the actions executed in an environment have an inherent uncertainty associated with their outcome.

The design of effective planning techniques is an important research direction since advances could have important repercussions on many different theoretical and applied fields, by enabling machines to reason about their behaviours and to act autonomously. Planning is also at the heart of reinforcement learning [47], which is the most general class of machine learning, it is also part of a larger area referred to as reasoning under uncertainty.

Before taking a detailed look at the aspects that will characterize this thesis, let us introduce a formal framework for planning. During the 1990's, researchers in the field of planning started using probability theory to model a task and its related uncertainty. This lead to the adoption of Markov Decision Processes (MDPs) which were elaborated and first used in control theory during the 1950's [4]. By modelling the states of the environment, the stochastic transition dynamics between states and the utility of a state, MDPs provide a general methodology to model planning problems in an abstract and principled form while taking into account uncertainty. A more general class of MDPs is defined by Partially Observable MDPs (POMDPs). In POMDPs, the environment in which an agent acts is modelled as being seen through noisy sensors and thus the exact state of the agent in the environment is not known. POMDPs therefore define a more realistic model.

Although techniques other than (PO)MDPs have been used in the past to solve planning problems, (PO)MDPs have demonstrated their strengths both theoretically and empirically as the method of choice for decision-theoretic planning [46].

One of the main challenges with planning problems is that they are computationally hard to solve. In fact, the class of problems that can be modelled with POMDPs is PSPACE complete [36] and NP-Hard to approximate [31]. Although, in practice, small problems can easily be solved, real-life problems are still impossible to solve optimally. To ease planning, researchers have proposed to decompose a problem into a hierarchy of smaller problems which can be solved independently [26]. Researchers have proved that this decomposition can potentially reduce the exponential search of planning into a linear one [26], affirming the practical computational worthiness of this approach. Furthermore, constraining a plan to a hierarchy has several other advantages. In reinforcement learning it can mean that less data is needed for learning a good plan since the hierarchical constraints prune the plan search space. Decomposing the policy can also potentially make it more intuitive and easier to understand for humans. This is in addition to the fact that prior knowledge, coming from humans, about plans might often by easily expressed hierarchically.

Although trying to mimic the way humans reason is tricky and can potentially lead to dead ends, in this case there is a connection to be made between the idea of hierarchical planning and the way humans plan. As humans, we usually set out to do a task by first establishing a main goal. For example, we want to get flowers. Without, realizing it, we then subdivide this task into smaller subtasks that need to be carried in a certain sequence. For example, we need to take a taxi to the florist, enter the shop and buy flowers. In turn, each one of these subtasks can be appropriately subdivided. For example, we need to walk to the taxi, sit down in the seat, exit the taxi once the destination has been reached, walk to the door of the florist and so on. The idea is that humans don't plan everything at once, but rather solve subtasks, until the goal is achieved. Another important factor in this example, is the one of subtask sharing. In other words, once a subtask is solved, one should be able to re-use it in other contexts. For example, if one knows how to walk, one can use this knowledge to walk to the taxi as well as out of the taxi to the florist. Finally, this example also shows off that one can abstract the goal of a task while it is executing a subtask. For example, walking to a taxi is the same regardless of the overall goal.

On a more technical level, hierarchies in planning permit to subdivide the planning problem into multiple independent, and smaller, problems. When the hierarchy is given this enables the optimization of each level, one by one, of the hierarchy, therefore reducing the time required to find a good plan. When the hierarchy is discovered, as is the case in this thesis, hierarchies act as memories and thus enable to reduce the size of the final plan.

Current computerized methods for hierarchical decomposition have the major limit of requiring some form of predefined hierarchy. To our knowledge, no algorithm has been designed to take a planning problem and decompose it without having the help of a human architect. This help can take various forms, from restrictions on how the (PO)MDP should be decomposed to actual partitions of the search space, but in all cases, this prior knowledge gives additional information to an algorithm trying to solve the problem. Although an argument can be made that often human architects know specifics of a problem and thus are able to derive a good hierarchy, we think that a human should not be required in order to decompose planning problems. In fact, ideally, the decomposition problem should be framed such as to be able to incorporate prior knowledge, but not require it.

1.1 Contributions

This thesis is about hierarchical planning, specifically, the main contribution of this thesis is a novel way to discover hierarchical plans. To our knowledge our approach is the first that does hierarchy discovery in the context of POMDPs. Our approach is based on framing the hierarchy discovery and planning problem in one optimization program. Successfully solving this optimization program therefore finds the optimal hierarchy and its associated policy. Our approach is both general and principled. Although, our approach, does not deal specifically with MDPs, it can also solve problems in fully observable domains.

Previous work [11] finds policies that are optimal only in a restricted sense (i.e., with respect to a fixed hierarchy). Our approach, by discovering the hierarchy and its policy in one, leads to fully-optimal policies since policies are not restricted to any subclass.

Parts of this work has already been published in:

- L. Charlin, P. Poupart, R.Shioda. *Automated Hierarchy Discovery for Planning in Partially Observable Environments*, To appear in Advances in Neural Information Processing Systems 19, 2007. [7]

1.2 Outline

This thesis will be divided in four parts. First, a complete review of Markov Decision Processes including work in both fully and partially observable domains as well as a literature review of hierarchical planning methods will be given in Chapter 2. We will also review several optimization problems that have been proposed by researchers to solve flat, as opposed to hierarchical, planning problems. Building on this knowledge, we will introduce our contribution in Chapter 3 followed by experiments on some well-known planning problems in Chapter 4. We will end this work by reflecting on what has been done and hypothesizing on future paths to follow in Chapter 5.

Chapter 2

Background

The intention of this chapter is to define the basic building blocks leading to our contributions as well as to provide a review of the literature in hierarchy discovery for problems of decision under uncertainty.

We will first review the basic notions of planning in fully observable environments using the framework of Markov Decision Processes (MDPs). We will then explain what is hierarchy discovery and how it is applied to MDPs, by reviewing the efforts of researchers in the domain. We will then introduce a framework, extending MDPs, to plan in partially observable environments, called Partially Observable MDPs (POMDPs). We will move on to describing some approaches for hierarchy decomposition in partially observable domains. Amongst those, we will show how researchers have formalized a way to encode a plan into a finite state controller and we will talk about hierarchical finite state controllers.

In explaining the concepts that are making our contributions possible, this chapter will also introduce the reader to most of the formalism and notation that will be needed in the rest of the thesis.

Before diving into the formalism of the Markov Processes let us detail the basic challenges and ideas of planning problems. The goal of planning is coming up with a way of behaving or a plan that an agent can follow in order to attain a goal, given knowledge of the agent's environment. In an optimal scenario, following that plan should be the (expected) most efficient way of attaining the goal, where efficiency is a measure of time. The main challenge of planning is that the environment in which an agent behaves is uncertain, in

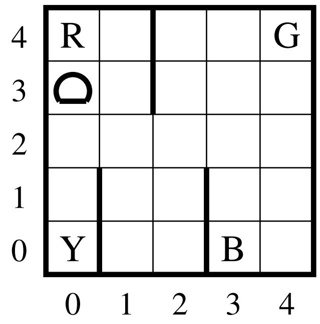


Figure 2.1: The Taxi domain as presented by Dietterich [11]

other words, behaviours of an agent can have unintended outcomes. Therefore, the agent has to be able to plan according to the stochastic model of the environment. The second challenge is a computational one, where it might be impossible to check all the possible behaviours of an agent in order to find an optimal one.

As a motivating example let us consider the Taxi domain [11], a task that will be used to clarify the main ideas throughout this chapter. The Taxi domain is depicted in Figure 2.1. In this domain, an agent, the taxi, has to pick-up and drive passengers to their desired location by moving on this grid environment one cell at a time. Passengers can only be picked-up and dropped-off at one of four locations on the figure (i.e., R, Y, B or G).

Let us imagine that there is a passenger that wants to go from R to Y. The plan of the taxi is therefore to successfully pick-up the passenger and drop him off at the prescribed locations. To do that, its behaviour, must be to go to R, pick-up the passenger, and then go to Y. The uncertainty, in this case, might be that there are several routes to get to R but some of them might be more crowded than others at certain times of the day.

2.1 Markov Decision Processes (MDPs)

Markov Decision Processes (MDPs) are probabilistic tools for optimal decision making under uncertainty. In other words, MDPs are a formalized framework to do planning, originally developed in the control theory community [4]. During the 1990's, MDPs were discovered by the AI community which used them as a principle tool in reinforcement learning and planning [46]. Since then, a wealth of research has been accomplished on the theory and algorithmic sides of MDPs [42].

This thesis will focus on the more general partially observable MDPs that will be introduced in Section 2.2 but relevant literature on hierarchy discovery in MDPs will be surveyed as it represents the stepping stone of future hierarchy discovery techniques. Specifically, the next section will formally introduce MDPs and one of their basic solving methods. It will be followed by a literature review of the main hierarchy decomposition techniques.

2.1.1 Terminology and Notation

In this section, we will review the main concepts behind MDPs.

An MDP is formally defined by a 5-tuple $\langle S, A, \Pr_s, R, \gamma \rangle$:

- S corresponds to the set of possible environment states, $s \in S$, that an agent may be in. States are assumed to be discrete and finite in size.
- A is the set of actions a available to the planning process.
- R is a reward function, mapping each (s, a)-pair to a real number corresponding to the immediate reward r of the system, $R : S \times A \to \mathbb{R}$.
- γ, the discount factor, is a real number in $[0, 1)$ that expresses the change in value of equivalent rewards received at different times. Specifically, γ enforces the idea that later rewards are worth less than more recent ones. This will be made clear in the introduction of the Bellman equation (Section 2.1.2). γ can also be thought of as a probability with which the MDP terminates at each time step. While the discount factor and termination probability views are equivalent from a probability stand point, it will be useful, later in Section 3.1, to view γ as a termination probability.
- $\Pr_s$, models the interaction of the process with the world. Specifically, it is $\Pr(s'|s, a)$, the probability of accessing a next state, $s' \in S$, given that action a was executed from state s. These transitions illustrate the Markov property. Namely, that all the information relevant to choosing the next state is contained by the current state and action. In other words, the current state and action are sufficient statistics for predicting the next state.

Let us look at the MDP framework in terms of the Taxi Domain. The taxi navigates through the states of the world, where each state is a member of the set of the cross product of the set of cells in Figure 2.1 and the set of locations of the passenger (either in the taxi or at one of the four predefined locations). The taxi can execute six actions, in addition to being able to go up, down, left or right by one cell, it can pickup and drop-off a passenger. The rewards, the goals, are to successfully pick-up and drop-off a passenger at his desired location. Finally, the uncertainty of the environment is formalized by the probability $\Pr(s'|s,a)$, which indicates that the taxi has a probability distribution over future states given an action that it executes in the current state. Practically, this uncertainty reflects the fact that the intentions of a taxi driver might not be exactly reflected in the state the taxi transitions into (e.g. the taxi might slide while turning).

2.1.2 Solving MDPs

In the previous section it was mentioned that planning is to come up with a sequence of behaviours, or actions, to follow in order to reach a goal. Translating this idea in the MDP framework goes as follows: At each time step, from the state it is currently in, the MDP must make a decision about which action to take. To guide its choices the MDP has the objective of maximizing the expected sum of discounted rewards:

$$V(s) = \mathbb{E}\left\{\sum_{t=0}^{\infty} \gamma^t r_t\right\}, \forall s. \tag{2.1}$$

Where $V(s)$ stands for the value of state s and r_t stands for the reward received at time t. We can see that, because of the discount factor, the same reward received at a latter time will be worth less. Formally, finding an action to execute at each state gives rise to a mapping called a policy and denoted π, where $\pi : s \to a$. The goodness of a policy is measured by the same value, the expected discounted value of all future rewards. Specifically we can extract the value of a given policy π for each state, where $V^\pi(s)$ stands for the value of following π from state s and on:

$$V^\pi(s) = \mathbb{E}\left\{\sum_{t=0}^{\infty} \gamma^t r_t | s_t = s, \pi\right\}, \forall s. \tag{2.2}$$

It is also possible to expand this equation to its recursive form by realizing that the value function is simply the sum of the current reward, $R(s,a)$, plus all future rewards:

$$V'^{\pi}(s) = R(s,\pi(s)) + \sum_{s'} \Pr(s'|s,\pi(s))V^{\pi}(s'), \forall s. \tag{2.3}$$

This recursive equation represents an easy way to evaluate the value of a policy. In general, the goal is to find the best policy, i.e., the one that maximizes the sum of future rewards. This optimal policy is denoted as π^* and it is simply the policy that has the highest value for every state:

$$V^{\pi^*}(s) \geq V^{\pi}(s), \quad \forall s, \pi \in \Pi \tag{2.4}$$

where Π is the space of all policies. Furthermore the value of a policy is a fixed point of the Bellman equation [4]:

$$V^{\pi}(s) = \max_a \left[R(s,a) + \sum_{s'} \Pr(s'|s,a)V^{\pi}(s') \right], \forall s. \tag{2.5}$$

It is also possible to extract a policy that corresponds to the optimal value function in the following way:

$$\pi^*(s) = \arg\max_a \left[R(s,a) + \sum_{s'} \Pr(s'|s,a)V^*(s') \right], \forall s. \tag{2.6}$$

Let us also mention that in general, policies can be stochastic, in which case there is a distribution over actions for each state, $\pi : s, a \rightarrow [0,1]$ (this will be useful in later sections).

In order to find good policies, several algorithms have been proposed [40, 38]. The main solving methods are either to search in the value space, the space of all value functions, and then extract a policy. This class of methods is referred to as Value Iteration (VI). Alternatively, one can iteratively search in the policy space and evaluate the policy after each iteration. In this thesis, we will focus on the latter which is called Policy Iteration (PI).

Let us first review the PI idea. PI is made up of two steps.

- First, a policy evaluation step where Equation (2.3) is evaluated with a fixed policy.

- The second step is policy improvement. This step improves the policy by picking better actions, ones that yield a greater value for a state, in order to improve the policy. This step is computed using Equation (2.6) with the updated value function calculated in the first step. The policy improvement theorem implies that if by changing the action at one state we get a better value for the state, then a new policy, π', made up of this new action and every other action from the old policy, is better than π. Mathematically, if policies π and π' differ by only one action for a specific state and for this state there is a higher value in following π' than π then

$$V^{\pi'} \geq V^{\pi}, \forall s \in S. \tag{2.7}$$

Theorem 2.1 (Policy Iteration Convergence) *Policy iteration is guaranteed to find an optimal policy in the limit, that is, given enough iterations the policy will converge to* π^*.

In fact, as soon as the value function, V^{π}, does not improve, the optimal value, V^*, has been found (recall that V^* is a fixed point of the Bellman equation). Intuitively, this optimality is guaranteed since at every policy improvement step, all the different actions from each states are considered and the best one is picked.

PI and its origins are discussed in detail by Puterman [42].

2.1.3 Hierarchy discovery using MDPs

The previous section has shown how one can iterate in the space of policies in order to converge to an optimal policy. The main limitation of this iterative method is that even if both steps are linear in terms of the number of state, as the state space increases, iteratively updating the value function becomes intractable. As an example, let us imagine that you would like to realistically model the Taxi domain. You would have to take into account not only the location of the taxi and its destination but also if the taxi has a passenger, if there are other passengers waiting and you might also include other informations such as the current time of day (for traffic purposes), the quantity of gas in the taxi and so on. Although the initial problem seemed fairly simple, the cross-product of these variables make for a large state space.

A straightforward way to deal with this limitation would be to divide larger problems into many smaller problems which in turn could share their policies. The intuition is that we can divide a large problem into smaller ones by partitioning the state and/or action space and thus, current algorithms could solve larger problems, in terms of state or action space. Such a decomposition could be done without regards to the problem, however, complicated interactions between the partitions would have to be modelled. It is therefore more appropriate to formalize methods that partition the state space with respect to the problem at hand by grouping or clustering, in the loose sense, similar states or states that usually appear one after another in a policy.

Alternatively, one could cluster actions, based on their co-occurrence in sequences. Policies over these partitions can be optimized and, in turn be assembled with other policies in order to form a structure. This structure has a natural hierarchy, where lower-level policies, known as subtasks, represent conditional sequences of actions that can be used to accomplish multiple different tasks. The main computational advantage is that, policies at different levels can be optimized independently of one another. The extent of this computational gain will be quantified later in the current section.

Let us now formalize this intuitive idea of action hierarchy. This definition of action hierarchy comes from Dietterich [11] and it is general enough to encompass definitions found in the works, detailed later in the section, that hierarchically decompose the action space.

Definition of action hierarchies A hierarchy of action is a tree where each node corresponds to a subtask, a subset of actions and states, and leaves represent concrete actions from the original planning problem. Mathematically, let us define a hierarchy of actions as a directed acyclic graph, $H_A = (N, E)$, where E is a set of edges and N a set of nodes with n_i^l corresponding to a node i on level l of the hierarchy (n_j^0 is a leaf). There is many one to many mapping from nodes to actions and states, $n_i^l \to \left(S_{li}, A_{n_i^l}\right), \forall l \neq 0$, where $S_{li} = \{s, s_t : s, s_t \subseteq S\}$, s_t represents terminal states, and $A_{n_i^l} \subseteq A$. If there's a path from n_i^l to n_j^m and, without loss of generality $l < m$, then $A_{n_i^l} \subseteq A_{n_j^m}$ (in other words, $A_{n_i^l} = \{a : a \subseteq A_{n_j^m}, l < m, l \neq 0\}$). If $l = 0$, then the mapping is one to one, $n_i^l \to a$. An example of an action hierarchy for the Taxi domain is given in Figure 2.2 (page 12), where

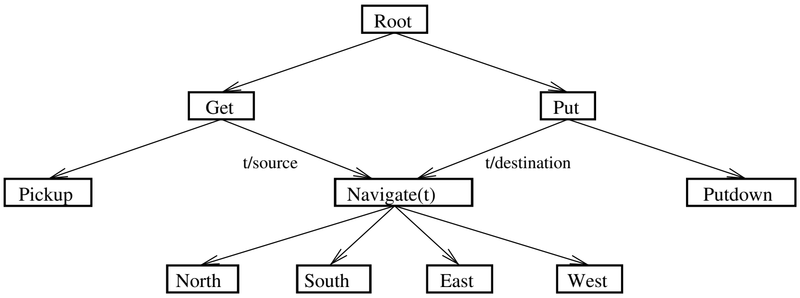


Figure 2.2: The subtasks hierarchy of the Taxi domain as presented by Dietterich [11]

each node is a subtask.

Although this definition does not necessarily account for state partitioning, as we will see, researchers have proposed different ways to decompose the state space according to the action hierarchy. Once, the state space is partitioned, subtask can be formalized by setting subgoals (states) that need to be reached to end the subtask. Each subtask can be optimized to yield the best policy over this restrained set of actions and spaces. This sub-policies can then be assembled to form a (flat or hierarchical) policy at execution time.

Several researchers [26, 27, 48] have shown that hierarchical approaches can potentially reduce the size of the search space from exponential to linear in the best case. The intuition behind the proof is the following. Let us define l as the length of the plan, the number of actions required to execute the plan, and let us imagine, without loss of generality, that at each step a binary decision has to be made. The size of the search space is therefore 2^l. In a hierarchy, the best case is that at most one decision is made per level, with l levels. The size of the search space is then constant for each level (so polynomial overall). For the proof the reader is referred to [26].

Of course, decomposing a large MDP into smaller problems also has an impact on the quality of the final policy. By constraining smaller MDPs to use only a subset of all actions or reducing their knowledge to some smaller part of the task, their policies are also restricted to a subset of all possible policies. In fact, Dietterich [11] defines two classes of hierarchical policies. First, *hierarchical optimal policies*, which stands for policies receiving the highest possible expected sum of rewards given the constraint of the hierarchy. Second, a weaker optimality called *recursive optimality* which means that the policy of each sub-MDP is optimal given its restricted action, state, and rewards space. Let us note that policies can achieve one type of optimality without the other.

There is an abundance of work related to hierarchical MDPs. Unfortunately, the majority of them focus on human-made hierarchies [11, 45, 17, 22, 25, 51, 24] or use similar prior knowledge [37] and propose ways to decompose the value function to reflect the hierarchy and optimize the subtasks. Amongst those, the work of Dietterich is particularly noteworthy [11]. A good summarization of the main ideas developed in previous work is given by Sutton et al. [45].

The rest of this section will review the main methods that automatically discover state hierarchies or parts thereof. The first approach uses the idea of action hierarchies as was defined and the other shows two different approaches which create abstractions over the state space.

Hengst [23] introduces an algorithm, HEXQ, to decompose the state and action space in a hierarchy as defined earlier. HEXQ's intuition is that the frequency at which a state variable changes value should be an indication of its level in the hierarchy. Specifically, variables that change more often while retaining their transition dynamics regardless of the value of less frequently changing, higher-level variable, should be on a lower level. For example, in the Taxi domain, the state variable that describes the position of the taxi in the world changes more often than the variable that indicates the location of a passenger (in the taxi or in one of the four pre-determined locations, see Figure 2.1). Furthermore, and this is a key point, when the taxi moves from one location to another in the world, the location of a passenger doesn't affect the transitions in-between the taxi's location variables. This concept is known as state abstraction. HEXQ abstracts these variables, by putting the taxi's location at the lowest level (the only level in which concrete actions are allowed) and the passenger's location variables at a higher level. Subgoals are identified as transitions that will change the value of a higher-level state or alternatively that are different given the value of higher-level states. For example in the Taxi domain, entering in a state where picking-up a passenger becomes possible (one of four pre-determined locations) would be an end state of the taxi location level.

HEXQ, uses a bottom up approach to optimize the policies at each level. First, it finds the policy of the lowest level. Then, this policy is seen as an abstract action that can be used by the level on top. This procedure is repeated until the top level is reached. HEXQ might have more than one MDP at each level, where each MDP represents a set of states that are strongly associated to one another.

HEXQ can therefore discover the hierarchy (the levels and the partition of the state space). The main pitfall of HEXQ is that it relies on problems specific variables changing at different time scales to create a decomposition. Such specifics might not occur in all planning problems.

Another approach is demonstrated by McGovern and Barto [33] in which they do not specifically discover hierarchies but instead they have formalized the discovery of subgoals. Subgoals are states that indicate the end of a subtask (i.e., terminal states in our definition of action hierarchies). Specifically, McGovern's method relies on identifying states that are always reached on trajectories that will lead to the goal, but not reached when sequences of actions are not successful. To do so, they frame this problem as a multiple-instance learning problem [13]. Multiple-instance learning is a supervised learning method where each instance to classify, trajectories represented as vectors of states in our case, is represented by multiple features, only one of which may explain the classification of the instance. Once identified, bottleneck states can be turned into goals that must be reached by building subtasks that end in those states. In the Taxi domain, a subgoal might be to reach a state where a passenger has been picked up, indeed, the final goal can never be reached without a passenger in the taxi.

Thrun and Schwartz [50] approach the problem of discovering hierarchy in a different manner. They focus on identifying policies that are common to a class of different tasks occurring in identical environments. The authors introduce an algorithm that discovers skills, policies over partitions of the state space. Policies that must be shared amongst similar task may not be optimal for every task. The authors address this issue by introducing a performance measure for skills that is a trade-off between a loss function of the shared policy and the description length of the policy over all tasks. The loss simply calculates how far the value function of the skill is from the optimal value function of a task. The description length considers the number of bits to represent a shared policy over all tasks. Shared actions only need to be represented once (i.e., if k tasks share the same action, then

$(k-1) * size(a)$-bits can be saved, where $size(a)$ is the number of bits needed to describe one action).

Although their method is not hierarchical in the pure sense, skills can still be seen as subtasks policies that are defined over a subset of states and as such Thrun's algorithm can be seen as discovering two level hierarchies. Thrun and Schwartz's work is also interesting because it imposes less restrictions on the skills than what is imposed on subtasks from previously mentioned work. In particular, there is no need to identify subgoals or explicit partitions of the action space. The direct consequence of these properties is that finding skills is harder that simply finding an optimal policy. The idea is that if this skill is re-used by many different tasks, it can be worthwhile to discover it. Furthermore as the authors note, it would be interesting to have skills that are able to generalize over states instead of tasks, that is, find skills using only one task. This idea would bring skills a lot closer to the other work that are being reviewed in this section.

The surveyed works show interesting approaches to discover elements of hierarchies. In addition to potentially enabling us to solve larger problem, the intuition that certain tasks can be found once and then reused by other tasks is very natural. Yet, out of all of the research that was surveyed in this section, none of the researchers have shown a general way of finding hierarchical policies. HEXQ was based on problems specifics that might not exist in all problems. McGovern and Barto's work only showed how to identify good subgoals and it is not evident how to establish a hierarchy from those. Furthermore McGovern and Barto's method will not exploit the idea of subtask sharing. Ideally, one would like to have a formal approach that would be something like a function, F, mapping an MDP problem to a (optimal) hierarchical policy, $F : \langle S, A, \Pr, R, \gamma \rangle \rightarrow \pi_h^*$ or to the closely related state and action decomposition of a problem. Thrun's method [50] is very close to achieving this mapping but the fact that it is not designed to solve the problem given only one task and that it relies on structure that might not be available in every problem is a pitfall. Furthermore, skills do not offer a true hierarchy but rather, a two-level decomposition.

This marks the end of our exploration and concerns about MDPs. In the next section, we will introduce an extended version of MDPs that can plan when the states are not directly observable. In fact, MDPs plan in what are referred to as fully observable domains.

A more realistic setting is when the environment is only partially observable because inputs informing the agent of its location are rarely perfect and as such an agent might only get approximate knowledge of where it actually is. The next section will provide the reader with an overview of a partially observable framework as well as approaches to perform hierarchical decomposition in these domains.

2.2 Partially Observable Markov Decision Processes (POMDPs)

In 1962, Drake [14] proposed an extension for MDPs to partially observable domains. The idea being that real-world environments are seen through noisy sensors and thus that the process should characterize its state by modelling the uncertainty it believes to be in. This extension of MDPs is called Partially Observable MDPs (POMDPs). The short explanation is that POMDPs can be seen as a continuous representation of MPDs and thus MDPs are a discrete state subclass of POMDPs. A more thorough explanation follows in the next section.

2.2.1 Terminology and Notation

POMDPs extend the MDP model by considering the more realistic setting in which states are not directly observable. A POMDP is provided with indications regarding its location in the form of observations, coming from the environment, after each action. Observations are typically seen as input coming from (noisy) sensors attached to the agent. Observations help the agent to situate itself may not convey as much information as states (the mapping from states to observations is surjective and/or stochastic). For this reason, in the POMDP framework, agents keep a distribution over possible states, referred to as the belief state and denoted $b(s)$. This distribution over states is the reason why POMDPs can be seen as continuous state space MDPs.

Formally, POMDPs can be described by adding two variables to MDPs to make it a 7-tuple $\langle S, A, \mathbf{O}, \Pr_{\mathbf{o}}, \Pr_s, R, \gamma \rangle$, where the new variables are:

- Observations, $o \in O$, act as indications, from the environment, about the state of the process. Observations are made available to the process after each action is executed.

- $\Pr(o|s', a)$, expresses the probability of receiving an observation o, when action a, was executed and lead to state s'.

2.2.2 Solving POMDPs

Solving MDPs meant finding a mapping from states to actions, similarly, solving a POMDP also involves finding a mapping to actions although since the process doesn't know which state it is in, there is something inherently different in the way a policy is defined. Formally, a policy is a mapping from a history of past actions and observations, (o, a), to an action, $\pi : (o, a) \rightarrow a$. In practice, since this history can be arbitrarily long and the belief state is a sufficient statistic of the history, a policy, π, is defined as a mapping from belief states, $b(s)$, to actions, a (i.e., $\pi : b(s) \rightarrow a$). After each action, $b(s)$ can be updated, simply, by using Bayes' law:

$$b_o^a(s) = k \Pr(o|s', a) \sum_{s \in S} \Pr(s'|s, a) b(s) \tag{2.8}$$

where k is the normalization constant and $b_o^a(s)$ stands for the new belief state once action a was executed and observation o received. Initially, $b(s)$ is set according to the problems specifics (or alternatively it can be set to a uniform probability over all states).

The value of a policy is defined, like in MDPs, as the expected sum of all future rewards. Again, because of the belief state the actual value of each $V(s)$ might be meaningless. Instead, taking an expected value over the belief state calculates the true value function:

$$V^{\pi}(b) = \sum_{s \in S} b(s) V^{\pi}(s), \tag{2.9}$$

where $V_{\pi}(s)$ denotes the fact that the agent is following policy π. The optimal value function, denoted $V^*(b)$, simply corresponds to the value of the best policy:

$$V^*(b) = \max_{\pi \in \Pi} \sum_{s \in S} b(s) V^{\pi}(s) \tag{2.10}$$

where Π is the set of all possible policies. This best policy that leads to the best value for every belief state is called the optimal policy and is written π^*. Extracting a policy from

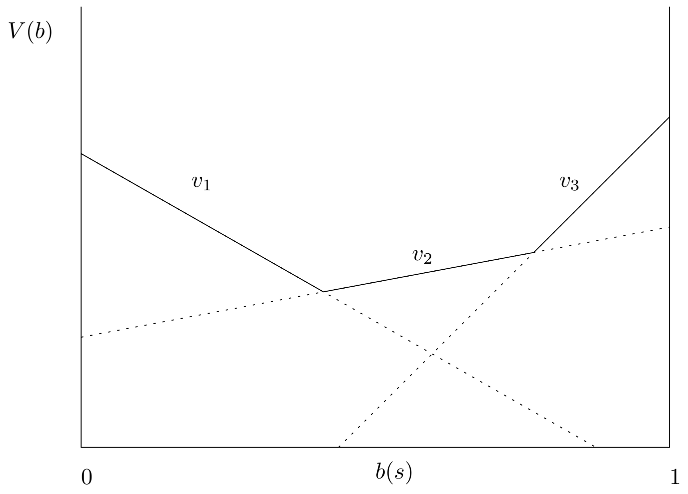


Figure 2.3: This figure shows the value function for a simple two state POMDP. The belief space can therefore be represented in 1 dimension ($|S|-1$-dimension in general). At execution, given the value of the belief state, the dominating vector would be picked (the convex upper envelope of the vectors is a solid line) and its associated action executed.

the value function is done as in MDPs. What is important from Equations (2.9,2.10) is the fact that the value function, $V(b)$, for a conditional plan is linear in terms of the belief state and it is, actually, also convex. In fact, Sondik [44] showed that the optimal value function for a finite number of steps is piecewise-linear. $V(s)$ can therefore be represented by a set of $|S|$-dimensional vectors of real numbers, $V = \{v_1, v_2, \ldots, v_k\}$ (this is depicted in Figure 2.3). Formally, the value function can then be expressed as the dot product between the belief state and the dominating vector:

$$V(b) = \max_i \sum_{s \in S} b(s) v_i(s). \tag{2.11}$$

In addition to simplifying computations, the fact that the value function is piecewise linear has several crucial ramifications that will be discussed in Section (2.3.2).

In POMDPs, the aforementioned policy iteration algorithm has been popularized by Hansen [21]. More recently, other authors have proposed techniques to perform PI in partially observable domains [40, 38]. Before we discuss these algorithms in greater details, let us review the two steps of PI in POMDPs:

- First, a policy evaluation step is performed:

$$V^{\pi}(b) = \sum_{a} \pi(b,a) \left[R(b,a) + \sum_{o \in O} \Pr(o|b,a) V^{\pi}(b_o^a) \right] \tag{2.12}$$

  where $\pi(b,a)$ stands for the probability of taking action a in belief state b when following policy π, and

$$\Pr(o|b,a) = \sum_{s'} \Pr(o|s',a) \sum_{s} \Pr(s'|s,a) b(s).$$

  Finally, the rewards parametrized by the belief state and the action is $R(b,a) = \sum_s b(s)R(s,a)$. Equation (2.12) is simply expanded from Equation (2.3) where there is an expectation over actions that the policy might lead to (i.e., the policy is stochastic). Note that in the first step $V'(s)$ can be initialized arbitrarily.

- The second step, policy improvement, can be calculated by finding the best action to do for every possible belief state:

$$\pi(b) = \arg\max_{a} \left[R(b,a) + \gamma \sum_{o} \Pr(o|b,a) V(b_a^o) \right]. \tag{2.13}$$

Performing those two steps for every possible belief state is intractable.

Sondik was the first to propose a tractable approach to search in the policy space in partially observable domains [44] in the 70's. He represented a policy as a mapping from polyhedral regions of the state space to actions. The problem is that there is no known method to evaluate a policy represented in this fashion. Instead, Sondik had to transform his policy into another representation which could then be easily evaluated using a reminiscent of Equation (2.3). Using his algorithm, the transformation of the policy in a form under which it can be evaluated is complicated and difficult to implement and as such Sondik's PI has not been widely used in practice.

Other methods have used the intuition that not every belief state is reachable to propose tractable policy iteration algorithms, Section 2.3.2 will discuss one of those.

Finally, coming back to the piecewise linearity and convexity of the value function, there is a correspondence between the update rule of Equation (2.12) and actions. In fact, let us point out that the executed action and observation resulting from the equation correspond to changing the belief value from V to V'. This will be important in Section 2.3.2 when we talk about policy iteration over finite state controllers.

2.3 Hierarchies in partially observable domains

2.3.1 Using hierarchies in POMDPs

This section will review some of the techniques that were proposed in hierarchical planning of partially observable environments. The obvious difference from approaches that deal with fully observable domains is that one cannot rely on the process reaching a specific state to enter or terminate the execution of a subtask. Moreover, for the same reason, explicit decomposition of the state space looses its meaningfulness. Therefore, the three methods that were detailed in Section 2.1.3 cannot be applied to partially observable domains as they all use the information about the location of an agent to discover their hierarchies.

The work of Wiering and Schmidhuber [51] is one of the first to consider (automatically) decomposing a POMDP's policy into multiple ones, where each sub-policy is tackled independently. To bypass the need of memorizing the history of the POMDP (through the belief state), their approach considers purely reactive policies—policies that choose their actions based solely on the latest observation.

A more general approach to hierarchical POMDPs is proposed in the work of Pineau [38] and Pineau et al. [39]. Their approach discovers state space partitions from a manually crafted action hierarchy. For example, their action hierarchy is the same as Dietterich's for the Taxi domain (see Figure 2.2). To define the state and observation partition that matches the action hierarchy (i.e., find the subset of states and observations that match all action nodes), Pineau uses a clustering algorithm, extended from an algorithm used in MDPs [9, 10] to account for observations. The idea of the clustering algorithm is to group state that have similar transition and observation probabilities given actions of a node. This idea that states keep their transitions dynamics under certain actions is reminiscent of the HEXQ algorithm [23] detailed earlier (see Section 2.1.3). Once states and observations are grouped into clusters with respect to actions, it is easy to recursively run policy or value iteration algorithms to find the correct policies for each subtask. That is, optimize the policies one action node at a time starting from the leaves of the hierarchical structure and moving up (bottom-up).

Theocharous et al. [49] propose an approach which first proceeds to a hierarchical decomposition of the states, even though their work is applied to partially observable domains, with abstract states being leaves of the decomposition and nodes being abstract state. Their approach is based on extending Hidden Markov Models to hiearchical models (HHMMs [15]) and then to Hierarchical-POMDPs. As such, abstract states are comprised of entry and exit states which indicate the beginning and the return of a sub-level. Accordingly, at execution time, the entry state of an abstract state would be called. It would then give control to a lower-level state (it can be a concrete state if it is at the lowest level). This lower level after its execution would then pass back the control to the abstract state by transitioning to its exit state. Given a state hierarchy one of the approach that authors propose is to solve for abstract actions which correspond to plans from entry to an exit state. Since this planning applies to partially observable environments, the authors then propose that at each time step the abstract action which should be followed is the one that corresponds to the most likely abstract state. Alternatively, by assuming that the next state is fully observable it is possible to pick the abstract action-state pair that yields the highest immediate reward.

Although the last two approaches differ in the way they first set the hierarchy, they both propose innovative ideas to decompose POMDPs, where the proposed algorithms, from an initial human input, proceed to a complete decomposition of the action, state and observation spaces. The main limitations of these works is their reliance on a knowledgeable human which can give them a decomposed state or action space. In the best case, similarly as for MDPs, one would like to design an algorithm that can take a POMDP and output a hierarchical decomposition (or directly a hierarchical policy), $F : \langle b_0, S, A, O, \Pr, R, \gamma \rangle \rightarrow h$ where h is a hierarchy that decomposed the initial POMDP. The space over which this decomposition should be is debatable, although it should permit solving larger POMDPs.

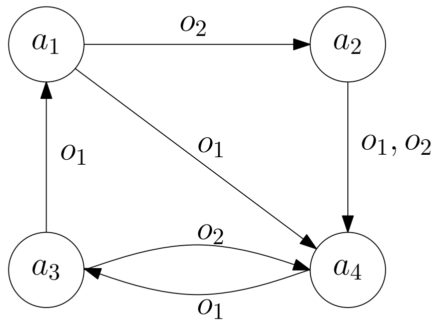


Figure 2.4: A simple deterministic FSC encoding a policy containing four actions and two observations.

2.3.2 Finite State Controllers (FSC)

The last important aspect to discuss is the representation of a policy. Naturally, policies can be represented as a mapping from states to action in the fully observable case or belief states to actions in the partially observable case but a new representation that will be detailed in this section offers interesting advantages.

In a previous section (2.2.2), we briefly touched upon the fact that a large class of POMDP policies can be expressed as Finite State Controllers (FSC). Such controllers are composed of nodes and edges and there is an injective mapping from nodes to actions and edges to observations. The FSCs can then be read as a sequence of actions and observations (ie. from a starting node do actions a_i, if observation o_j arises do a_i if observation o_k arises do action a_l). An example of such a controller is given in Figure 2.4.

Formally, let us define an FSC to be a graph $G = (N, E)$, where N is a set of nodes, and E is a set of edges. Encoding a policy, π, with an FSC consists of two mapping functions. A first many to one mapping, which assigns actions to nodes, $\alpha : n \rightarrow a$, where a is an action and n a node. A second mapping, that assigns edges to next nodes, $\beta : o, n \rightarrow n'$, where n, o are identifiers of an edge and n' the next node (the node the edge is pointing to). Given that encoding and a start node, n_0, a policy would carried out by executing the action associated with n_0 and following the edge labelled with the received observation to the next node, and so on. Here is the sequence of actions that would be executed starting from node n_0, where node n_t and observation o_t are, respectively, the node and the observation at time t:

$$n_0 \xrightarrow{\beta(n_0,o_0)} \alpha(n_1) \xrightarrow{\beta(n_1,o_1)} \cdots \alpha(n_{t-1}) \xrightarrow{\beta(n_{t-1},o_{t-1})} \alpha(n_t) \xrightarrow{\beta(n_t,o_t)} \ldots \tag{2.14}$$

In general, assuming that each action is executed at one time step, time t, the process executes the action $a_t = \alpha(n_t)$ and follows the observation to the next node, $n_{t+1} = \beta(n_t, o_t)$. The value of the encoded policy can then be calculated by solving the following set of $n \cdot s$ equations:

$$V_n(s) = \left[R(s, \alpha(n)) + \gamma \sum_{s',o} \Pr(s'|s, \alpha(n)) \Pr(o|s', \alpha(n)) V_{\beta(o,n)}(s')\right], \forall n, s. \tag{2.15}$$

These definitions can also be extended to encode stochastic policies by defining the mapping functions, α and β, to be distributions over current and next nodes. Specifically, $\Pr_\alpha(a|n)$ denotes the probability of executing action a at node n and $\Pr_\beta(n'|n, a, o)$ is the distribution over next nodes with respect to the action executed at the current node and the observation received from the environment. Accordingly, Equation (2.15) is extended to:

$$V_n(s) = \sum_a \Pr_\alpha(a|n) \left[R(s, a) + \gamma \sum_{s',o,n'} \Pr(s'|s, a) \Pr(o|s', a) \Pr_\beta(n'|n, o) V_{n'}(s')\right], \forall n, s. \tag{2.16}$$

We will refer to controllers encoding stochastic policies as stochastic controllers.

Policy Optimization with Finite State Controllers

Hansen [21] has introduced a policy iteration technique using FSCs. Let us recall that policy iteration is made up of two steps. First, a policy evaluation step that calculates the value of a policy. Second, a policy improvement step that returns an incrementally better policy. Equation (2.16) shows how to evaluate a policy encoded in a controller. Hansen's contribution was to show a simple and efficient algorithm to perform policy improvement. Specifically, he showed a simple correspondence between policy improvement and operations on FSCs. By doing the dynamic programming update, V, a set of k vectors corresponding to the value function, is transformed into an improved set V' which is equivalent to transforming an FSC, F, into an improved FSC F'. Let us look at it more closely. Each node of an FSC has a value function, the value of a complete FSC can therefore be represented, similarly to V, as a vector V^F. Adjusting V^F to reflect the changes from V to V' improves F to F'. In fact, from V' several possibilities arise to transform F to F'. Here are the three transformation possibilities with respect to the vectors of V', as proposed by Hansen (let us keep in mind that each vector $v_i \in V'$ represents an action):

- If the action and outgoing edges corresponding to the new vector also correspond to a node of F keep this node in F'.
- If a new vector in V' pointwise dominates a vector in V then replace the corresponding node in F and its outgoing edges in F'.
- Else, simply add a new node to F that corresponds to this new vector of V'.

By evaluating the new policy encoded in F' and re-applying the policy improvement step the FSC will converge to the optimal FSC [21].

While guaranteeing an optimal policy, the main disadvantage of Hansen's work is that policies can require arbitrarily large (or even infinite) controllers to represent the optimal policy. A remedy to this situation comes from the work of Poupart and Boutilier [41] where they propose a *bounded* policy iteration algorithm that searches for the best policy in a space of restricted controllers, controllers with a fixed number of nodes. What is important to realize is that any acceptable POMDP policy will form a basis of solution to Equations (2.16), hence, the system has more than one solution. Furthermore, fixing the policy, by setting the probabilities $\Pr_\alpha$ and $\Pr_\beta$, completely determines the value function. As such one can search in policy space and evaluate the goodness of a policy by solving this system of equations. A more robust approach for searching in the space would be to frame this problem as an optimization problem where the value function and the policy are seen as variables [41]. In their paper, Poupart and Boutilier suggest to frame the policy improvement step as an optimization problem:

The variables of this program are $\Pr_\alpha(a|n)$, $\Pr(n', a|n, o)$ ($V_n(s)$ is fixed during the policy optimization step). This program is run for every node, n. The intuition behind the optimization is that, by maximizing ϵ, it ensures that the policy improvement step will

$$
\begin{aligned}
&\max_{\Pr_\alpha,\Pr_\beta} \quad \epsilon \\
&\quad \text{s.t.} \\
&\qquad V_n(s) + \epsilon \leq \sum_a \left[\Pr_\alpha(a|n) R(s,a) + \gamma \sum_{s',o,n'} \Pr(o|s',a) \Pr(s'|s,a) \Pr(n',a|n,o) V_{n'}(s') \right] \forall s \\
&\qquad \sum_a \Pr_\alpha(a|n) = 1 \qquad\qquad\qquad \text{and } \Pr_\alpha(a|n) \geq 0 \ \forall A \\
&\qquad \sum_{n'} \Pr(n',a|n,o) = \Pr_\alpha(a|n) \ \forall A \qquad \text{and } \Pr(n',a|n,o) \geq 0 \ \forall A
\end{aligned}
$$

Table 2.1: Poupart and Boutilier's linear program for bounded policy iteration [41]

make every node better than in the previous iteration. Of course, this method may fall into local optimum and as such the authors propose a heuristic to bypass obvious maxima.

In order to reduce the representation complexity, Poupart and Boutilier [41] also propose to merge the x variables by making the following transformation:

$$\Pr(n',a|n,o) = \Pr_\alpha(a|n) \Pr_\beta(n'|n,a,o). \tag{2.17}$$

Accordingly, relevant variables are replaced by $\Pr(n',a|n,o)$ in the optimization problem.

Following Poupart's work, Amato et al. [2] proposed an approach to frame both policy evaluation and improvement in one optimization program which is shown in Table 2.2. The variables of the optimization program, $V_n(s)$, $\Pr_\alpha$, $\Pr_\beta$ are under-braced to highlight the quadratic constraints and linear objective of the program (this is called a quadratically constrained optimization program). The objective is to maximize the expectation of the value function at the starting node where, b_0 is the initial belief state. The first constraint is simply the system of linear equations (2.16) which represent the Bellman equation. The next four constraints enforce the fact that $\Pr_\alpha$ and $\Pr_\beta$ are distributions. The starting node, n_0, is determined arbitrarily before the optimization starts.

To summarize, the idea is to find the best stochastic policy, represented by $\Pr_\alpha$ and $\Pr_\beta$, by running an optimization with the Bellman equation as constraints. Since this problem

$$
\begin{aligned}
&\max_{y} \quad \sum_{s} b_o(s)\, \underbrace{V_{n_o}(s)}_{y} \\
&\text{such that} \\
&\quad \underbrace{V_n(s)}_{y} = \sum_{a} \left[\underbrace{\Pr_{\alpha}(a|n)}_{x_1} R(s,a) + \sum_{s',o,n'} \Pr_{\gamma}(s'|s,a) \Pr(o|s',a) \underbrace{\Pr(n',a|n,o)}_{x_2} \underbrace{V_{n'}(s')}_{y} \right] \quad \forall n, s \\
&\text{and} \quad \underbrace{\Pr_{\alpha}(a|n)}_{x_1} \geq 0 \quad \forall a, n \qquad\qquad \text{and} \quad \sum_{a} \underbrace{\Pr_{\alpha}(a|n)}_{x_1} = 1 \quad \forall n \\
&\text{and} \quad \underbrace{\Pr(n',a|n,o)}_{x_2} \geq 0 \quad \forall n', a, n, o \qquad \text{and} \quad \sum_{n'} \underbrace{\Pr(n',a|n,o)}_{x_2} = 1 \quad \forall n, a, o
\end{aligned}
$$

Table 2.2: Quadratically constrained optimization program for policy optimization by Amato et al. [2]

is non-convex, the optimization methods used can only guarantee a local maximum. To solve this problem, Amato et al. use a non linear solver called *SNOPT* [18].

As in Poupart and Boutilier's work, the number of nodes must be fixed at the beginning of the optimization but they show no way of withdrawing this limit. Rather, Amato et al. argue that their approach commits less of an approximation since it considers all the node at all times and that in practice this lets them find good controllers which are smaller than the ones found using other approaches.

Hierarchical Finite State Controllers

In previous sections (Section 2.1.3 and 2.3.1), we have introduced planning methods that divide the search space into a hierarchy in order to gain tractability. In the previous section we showed how an FSC could encode a flat policy. The next logical step is to merge these two ideas in introducing hierarchical finite state controllers (HFSC) that, thus, encode hierarchical policies. Combining the two approaches was first proposed by Hansen and Zhou [20]. The idea is that multiple flat controllers will interact together in a structure that forms a hierarchy (see Figure 2.5). Higher-level controllers, through abstract

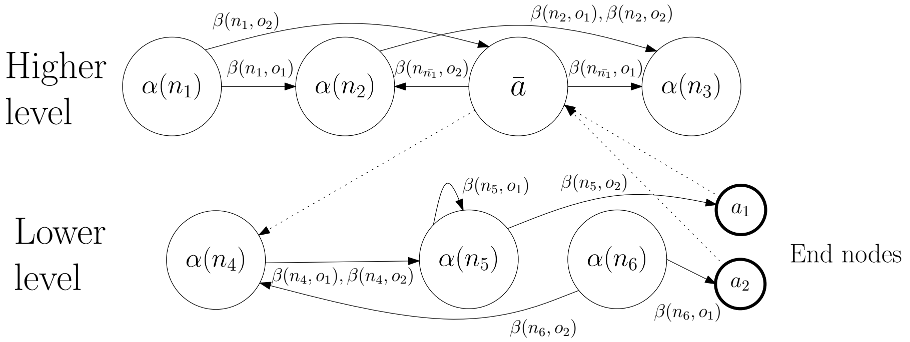


Figure 2.5: A simple deterministic HFSC as described by Hansen and Zhou [20]. Dotted edges stand for vertical transitions and end nodes n_{end} are in bold. This HFSC encodes a policy containing two actions (a_1, a_2) and two observations (o_1,o_2).

actions, can pass the control to lower level ones, specifically, they pass the control to a predetermined initial node of the lower-level controller. Lower-level controllers will give control back to their parent once they reach an end node. Mathematically the main difference between FSCs and HFSCs is that an HFSC has a function, $\alpha(n)$, to map to abstract actions, $\alpha : n \rightarrow \bar{a}$, where the set of abstract actions is a superset of the regular actions, $\bar{a} \in \bar{A} \supset A$. Abstract actions correspond to subcontrollers. Furthermore, every controller must also have a set of end nodes, $n_{end} \in N$, which indicate the termination of a controller and, hence, the return the control to the calling controller higher in the hierarchy. Terminal nodes don't have knowledge of any information from the calling controller. Consequently, the value of a terminal node is simply the immediate reward associated with the action of the node (i.e., $V_{n_{end}}(s) = R(s, a_{n_{end}}), \forall s \in S$). Hansen and Zhou also add the constraints that the number of levels of an HFSC must be finite and leaves of the HFSC must only contain mapping to concrete (primitive) actions.

The main role of terminal nodes, as their name implies, is to provide for termination of a subcontroller. In fully observable domains, subtasks are usually set to terminate whenever a certain state is reached. In partially observable domains, the same cannot be achieved and setting an appropriate belief state in which the controller should finish does not make much sense. So, a controller terminates when it reaches a terminal node. In Hansen's framework, there is one terminal node for each possible action. As such, terminal nodes are not optimized over.

Let us look at some of the details of Hansen and Zhou's formulation as their approach constitutes the basis of the next chapter. One of the interesting question that they tackle is how to calculate the transition probabilities of an abstract action. In other words, on a higher-level controller how can one calculate the probability of the state which occurs after a lower-level controller has returned the control. The authors define the probability of returning from node n_i of a subcontroller in state $\tilde{s}$ if it was started in state s as $P_{n_i}(\tilde{s}|s)$. It is defined, simply, for every node as:

$$P_n(\tilde{s}|s) = \sum_{s',o} \Pr(s', o|s, \alpha(n)) \Pr_{\beta(n,o)}(\tilde{s}|s') \tag{2.18}$$

and for a terminal node as:

$$\Pr(\tilde{s}|s)_{n_{end}} = \begin{cases} 0 & \text{if } s = \tilde{s} \\ 1 & \text{if } s \neq \tilde{s} \end{cases} \tag{2.19}$$

Once the policy, the *alpha* and the *beta* functions are set, this recursive equation examines the path that can be followed and returns the probability of finishing the controller in state $\tilde{s}$ of the POMDP.

Hansen and Zhou also deal with the problem of observation abstraction. Specifically, what observation to associate with each abstract action. One idea is to send the whole, or part of the history of the subcontroller as a special observation to the higher-level controller. In that way subcontrollers don't act as black boxes and interesting information might be given to the higher-level controllers. In their paper, Hansen and Zhou opt to return a null observation, losing the history of the process but also lowering computational requirements to solve hierarchical POMDPs.

Hansen and Zhou prove that policies which can be encoded in their controllers have a form of optimality close to the recursive optimality as defined by Dietterich (see Section 2.1.3). The idea is that since lower-level controllers do not return any part of their execution history (through an observation for example) to higher-level controllers, their HFSCs can only attain policies that are optimal per level.

The other issue that Hansen and Zhou's paper deal with is how to properly discount the rewards of higher-level controllers, given that the duration (i.e., the number of time steps) of subcontrollers is unknown. Their approach calculates the mean duration of a subcontroller, using this duration they can properly discount the future rewards of a higher-level controller. The problem with this approach is the assumption that general subcontrollers must always terminate. Although subcontrollers can be crafted in such a way, for example by appropriately augmenting the rewards of terminal nodes, this is not true in the general case. Furthermore, the mean duration time for the level is not sufficient to properly discount rewards at higher levels, instead, the mean duration time should be calculated for every possible end state. In the next section, we will show a way to deal with this issue.

Hansen and Zhou's framework is very meaningful because it constitutes the first approach at representing hierarchical policies in hierarchal controllers. Nevertheless, the fact that the hierarchy is set by the programmer and that the number of levels must be finite are two of the shortage that lead us to the work described in this thesis.

To finish off this discussion on HFSCs, let us mention that there is also an important computational gain to using HFSCs. If HFSCs allow for subtask sharing, in other words if multiple abstract actions are allowed on each level and each abstract action ends at the same lower-level node, there can be an exponential reduction in the number of nodes (and edges) that need to represent a policy. This comes from the fact that subtasks do not need to remember which abstract action they were called from. In other words, because multiple abstract action can share exactly the same subcontroller there is a linear (in terms of number of abstract nodes) number of nodes that can be saved at each level, and exponential number of nodes over the whole hierarchy.

2.4 Conclusion

This chapter formalized the basic concepts of MDPs, POMDPs and it introduced the idea of hierarchical planning. It showed how policies could be encoded by flat and hierarchical FSCs. Furthermore, it showed the limitations of current hierarchical techniques in both fully and partially observable environments which sets the table for the next chapter that will explore our proposed solutions to some of those shortcomings.

Chapter 3

Discovering Hierarchical Policies

In the previous chapter, we reviewed the foundations of POMDPs and showed how a POMDP's policy could be represented as a FSC. This representation, in turn, was extended to hierarchical controllers, as defined by Hansen and Zhou [20], which could encode hierarchical policies. Hansen and Zhou's idea is excellent since it represents a general way to encode hierarchical policies. However, their work does not address the problem of discovering the hierarchy, the structure of hierarchical controllers. The previous chapter has also exposed other methods to decompose planning in a hierarchical structure. The key factor that no research has addressed, thus far, is how to completely come up with a hierarchical structure given a general planning problem. That is, how to find a mapping from a planning environment to a hierarchical policy ($F : \langle b_0, S, A, O, \Pr, R, \gamma\rangle \rightarrow \pi_h$, where π_h stands for a hierarchical policy).

Our main contribution, that will be detailed in this chapter, is a method to discover such a mapping from a POMDP to an optimal hierarchical policy. This constitutes, to our knowledge, the first approach which requires no prior knowledge and discovers the hierarchical structure of a policy, in a principled manner, applicable to any POMDP. The only parameter that will need to be set by a practitioner is the number of nodes of the discovered hierarchy.

We will start by defining our notion of HFSC which, although being similar to the one proposed by Hansen, proposes several extensions which make it more powerful. Then, we will move on to present how we frame the hierarchy discovery problem as a non-linear

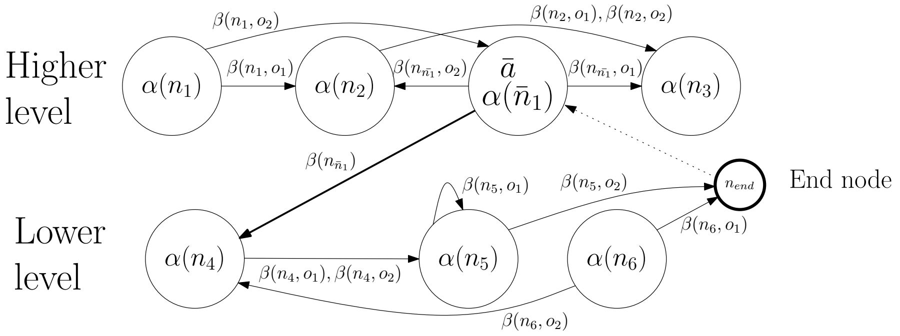


Figure 3.1: An extended version of HFSCs. The most notable differences, with respect to the work of Hansen and Zhou [20], are the unique terminal node n_{end}, the action executed by the abstract node once the control is returned ($\alpha(\bar{n})$) and the parametrization of the vertical transition outgoing from the abstract action, $\bar{a}$. On the figure, bold edges stand for vertical transitions starting a subcontroller and dotted edges represent the termination of a subcontroller.

non-convex optimization program. We will also introduce several methods to solve this program. The effectiveness of these methods will be discussed, from a theoretical point of view, before empirical results are presented in the next chapter. We will end this chapter with a brief discussion of how to fix the number of nodes of a policy.

3.1 Extended definition of Hierarchical Finite State Controllers

Hansen and Zhou [20] were the first to propose hierarchical controllers to encode hierarchical policies. In this section, we will extend their idea of HFSC in preparation of our main contribution that will be presented in the next section.

First, in our definition, abstract nodes of HFSCs are like regular nodes, associated with an action (in addition to a sequence of actions by the subcontroller)—see Figure 3.1. This action is executed once the subcontrollers finishes and passes the control back to the abstract node. This idea alleviates the need for terminal nodes to execute an action.

Furthermore, in this way and following Hansen's idea, subcontrollers don't have to return any information to higher-level controllers. The next observation is simply available after the concrete action has been executed by the abstract node. Finally, the vertical transition of the abstract node is also parametrized, that is, the first node of a subcontroller is not known a priori.

Second, Hansen's method requires that each subcontroller have $|A|$ termination nodes (where A is the set of all actions). In our definition, since terminal nodes don't execute an action, terminal nodes need only to indicate the end of a subcontroller. For this reason, we only need one terminal node for the entire HFSC. This node is special because it is the only one that does not execute an action but edges from any node can point to it. The terminal node doesn't explicitly contain an outward edge but it gives the control back to the abstract node from which the subcontroller started. These first two differences also have the advantage that there is a uniform parametrization of actions, in other words, every node, concrete or abstract, executes an action.

Third, Hansen talks about calculating mean duration times for the execution of a controller in order to properly discount the rewards of higher-level controllers. That is, one should know the number of steps executed by abstract actions in order to properly adjust the decay of rewards at higher-level controllers through the discount factor γ. As we pointed out earlier (see Section 2.3.2), this idea will not work if the subcontrollers have a non-zero probability of never finishing. As a remedy, we absorb the discount factor into the transition probability, denoting $\Pr_\gamma(s'|s,a) = \gamma \Pr(s'|s,a)$. As such, $\Pr_\gamma$ is seen as having a $1-\gamma$ probability of terminating at each step. Probabilistically, this is equivalent to discounting the future rewards.

Fourth and final, in our definition, nodes are not associated with a level a priori. That is, the only thing that is fixed is the number of nodes of the entire controller. The number of levels as well as which nodes can go on which levels will be completely determined when the hierarchy is discovered.

Let us finally mention that these changes from Hansen's controllers do not affect the representation capabilities of our controllers. In fact, our approach reduces the size of controllers by eliminating all but one terminal node.

Another interesting property with our definition of controller is that it allows recursive

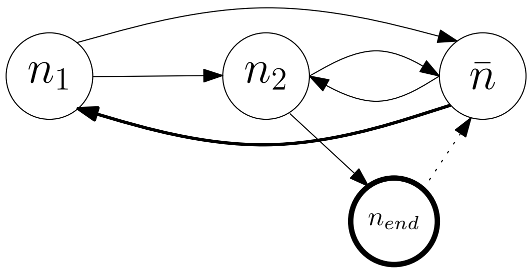


Figure 3.2: Example of a recursive controller where the parameters have been omitted for clarity purposes. The bold edges represent a vertical transition starting a subcontroller and the dotted edges represent the termination of a subcontroller.

controllers to be defined.[1] Recursive controllers are simply controllers that can recursively call themselves (see Figure 3.2). The main advantage to recursive controllers is that they can in theory represent by a finite number of nodes policies that would otherwise require an infinite number of nodes. In practice, the memory needed to execute recursive controllers is indefinite. Although we offer no formal proof, there is a parallel to be made from controllers to formal languages. Controllers, as defined by Hansen and Zhou, are equivalent to regular languages, as HFSC presented in the section could encode context-free grammars.[2]

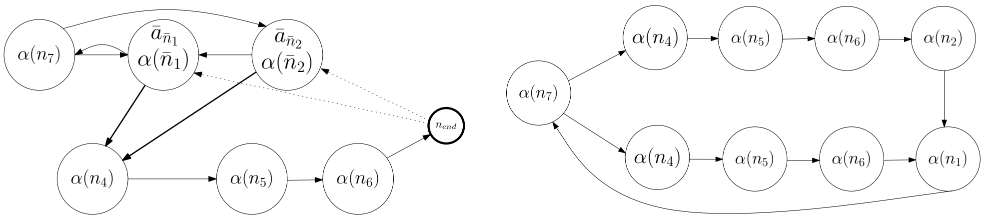
3.2 Hierarchy Discovery and Policy Optimization

Now that we have re-defined part of the HFSCs, we introduce a formulation for hierarchy discovery and policy optimization in the form of a non-convex non-linear quarticly-constrained optimization program. The formulation constitutes the bulk of our contribution.

It was mentioned, in Section 2.3.2, that the principle advantage of hierarchical controllers is that they can encode more expressive policies given a fixed number of nodes.

[1]Let us note that although they share the same name this concept has no relation with Dietterich's, previously mentioned, definitions of recursive optimality [13] Recursive optimality was a metric to evaluate a restricted class of policies as here, recursivity refers to fact that controllers can call themselves.

[2]The reader who wishes to explore this parallel, is referred to [32]



(a) Typical controller in which a lower-level subcontroller is re-used by two, higher-level, nodes.

(b) The equivalent flat controller

Figure 3.3: The main advantage of hierarchical controllers is that they can encode more expressive policies compared to flat controllers of the same size.

That is, by re-using subcontrollers, calling them from different higher-level nodes, exponentially many nodes and edges can be saved. A simple, two-level, illustrative example is given in Figure 3.3. Figure 3.3(a), is a hierarchical controller where two abstract nodes can call the same subcontroller. Figure 3.3(b), is the larger flat controller needed to encode the same policy as the hierarchical controller. Another way of explaining why hierarchical controllers may require less nodes is by viewing subcontrollers as subroutines which can be called from different contexts. Higher-level abstract nodes may, in turn, be viewed as a memory which retains specific information which is not needed by the subcontroller. Subcontrollers can execute without any regards for the nodes at higher levels, specifically, they do not have to remember from which abstract node they were called. A method which can discover hierarchies therefore has access to a richer class of policies given a fixed number of nodes.

Our proposed formulation extends previously proposed formulations by Poupart and Boutilier [41] and Amato et al. [2] for non-hierarchical controllers. The intuition behind Poupart and Amato's approach, as was discussed in Section 2.3.2, is to encode the policy as a set of variables in an optimization problem. Solving the optimization problem, which optimizes over the value of a policy, yields to a policy discovery method. In this case, the policy could be seen as horizontal transitions of a flat FSC. The main structural difference between a flat FSC and a HFSC is that the latter also contains vertical transitions, transitions between higher-level and lower-level subcontrollers. By encoding these vertical transitions in the optimization problem, it becomes possible to discover the hierarchy in addition to the (hierarchical) policy. In fact, successfully and optimally solving such an optimization problem would yield a hierarchical decomposition of the original problem and an optimal hierarchical policy of a fixed size. The optimization problem which represents such an idea (detailed explanations follow) is presented in Table 3.1. Maximizing this optimization problem yields an optimal hierarchical controller with a fixed number of nodes ($|N|$) that encodes an optimal policy. The optimization is a quarticly-contrained non-convex optimization problem. The quartic constraints come from the four variables multiplied together in Equation (3.4b). It is possible to demonstrate that the problem is non-convex by realizing that the Hessian matrix ($\nabla^2 f(w, x, y, z)$) of the function is not positive semidefinite for all its variables [43]. This is caused by the Bellman constraints, Equations (3.2,3.3). To give a proper overview of the optimization problem, let us first look at the variables (i.e., the underbraced expressions) and then at the constraints of the problem.

The variables of the program are underbraced in the above formulation. They are, first, the value function at each node-state pair, $V_n(s)$ and $V_{\bar{n}}(s)$, for concrete nodes, n, and abstract nodes $\bar{n}$, respectively, denoted as y's. Values for abstract nodes have a different formulation, that take the form of a constraint, that will be detailed later on. Second, the variables parameterizing the policy, $\Pr(n', a|n, o)$, are denoted by x's. In addition to associating actions to nodes (in the same way that the α function did in Hansen's work [21]), these variables can be seen as setting the horizontal transitions of the HFSC. As mentioned earlier, these variables are the aggregation of $\Pr(a|n)$ and $\Pr(n'|n, a, o)$. The new variables, with respect to the work of Amato et al. [2], are the z's representing vertical transitions, passing the control to a lower-level controller from a higher-level one. They are presented as $\Pr(n_{beg}|\bar{n})$, that is a distribution of the nodes of a subcontroller conditional on the abstract nodes (of a higher-level controller). The uncertainty modelled by variables that are probability distributions make the HFSC and the hierarchical policy stochastic. Finally, the occupancy distribution, denoted as $oc(n', s'|n, s)$ was first formalized in the POMDP setting in [40]. They are close to what Hansen [20] refers to as *state transition probabilities for an abstract action*, denoted $\Pr_n(\tilde{s}|s)$ (see Section 2.3.2). In our case it is not a probability but rather it denotes the frequency by which, starting from state s in

$$\max_{w,x,y,z} \quad \sum_{s \in S} b_0(s) \underbrace{V_{n_0}(s)}_{y} \tag{3.1}$$

$$\text{s.t.} \quad \underbrace{V_n(s)}_{y} = \sum_{a \in A, n' \in N} \Big[\underbrace{\Pr(n', a | n, o_k)}_{x} R(s, a)$$
$$+ \sum_{s' \in S, o \in O} \Pr_\gamma(s'|s, a) \Pr(o|s', a) \underbrace{\Pr(n', a|n, o)}_{x} \underbrace{V_{n'}(s')}_{y} \Big] \quad \forall s, n \tag{3.2}$$

and

$$\underbrace{V_{\bar{n}}(s)}_{y} = \sum_{n_{beg} \in N} \underbrace{\Pr(n_{beg}|\bar{n})}_{z} \Big[\underbrace{V_{n_{beg}}(s)}_{y} + \sum_{s_{end} \in S, a \in A, n' \in N} \underbrace{\text{oc}(s_{end}, n_{end}|s, n_{beg})}_{w} \Big[$$
$$\underbrace{\Pr(n', a|\bar{n}, o_k)}_{x} R(s_{end}, a) + \sum_{s' \in S, o \in O} \Pr_\gamma(s'|s_{end}, a) \Pr(o|s', a) \underbrace{\Pr(n', a|\bar{n}, o)}_{x} \underbrace{V_{n'}(s')}_{y} \Big] \Big] \quad \forall s, \bar{n} \tag{3.3}$$

and

$$\underbrace{oc(s', n'|s_0, n_0)}_{w} = \delta(s', n', s_0, n_0) + \sum_{s \in S, o \in O, a \in A} \Big[\tag{3.4}$$

$$\left. \sum_{n \in N} \underbrace{oc(s, n|s_0, n_0)}_{w} \Pr_\gamma(s'|s, a) \Pr(o|s', a) \underbrace{\Pr(n', a|n, o)}_{x} \right\} n \text{ concrete} \tag{3.4a}$$

$$\left. \begin{aligned} &+ \sum_{s_{end} \in S, n_{beg} \in N, \bar{n} \in \bar{N}} \underbrace{oc(s, \bar{n}|s_0, n_0)}_{w} \Pr_\gamma(s'|s_{end}, a) \Pr(o|s', a) \underbrace{\Pr(n', a|\bar{n}, o)}_{x} \\ &\underbrace{oc(s_{end}, n_{end}|s, n_{beg})}_{w} \underbrace{\Pr(n_{beg}|\bar{n})}_{z} \Big] \quad \forall s_0, s', n_0, n' \end{aligned} \right\} \bar{n} \text{ abstract} \tag{3.4b}$$

Table 3.1: Hierarchy discovery and policy optimization framed as a quarticly-constrained optimization problem

node n, the agent could end up in state s' in node n'. $\delta(s', n'|s_0, n_0)$ is simply an indicator function where,

$$\delta(s', n'|s_0, n_0) = \begin{cases} 1 & \text{iff } s' = s_0 \text{ and } n' = n_0 \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

In general, oc variables may therefore have a value greater than 1. However, in Equation (3.3) their values are contained in the interval $[0, 1]$ and as such are almost the same as the *state transition probabilities for an abstract action* of Hansen and Zhou. The main difference is that our approach uses the indicator function δ instead of having a special case for terminal nodes. Furthermore, we explicitly express the transition to yet another subcontroller in Equation (3.4b).

Now that we have defined the variables let us look at the different parts of the optimization problem.

- Equation (3.1) constitutes the objective which is simply the expected value at the beginning node, n_0. n_0 is arbitrarily fixed and it is a node at the highest level of the HFSC. b_0 is the initial belief state, which is a parameter of the planning problem.

- Equation (3.2) is a constraint encoding the value of a concrete node as given by Bellman's equation. This is the same as the formulation of Amato et al. [2].

- Equation (3.3) is a constraint of the optimization problem representing the value function for abstract-node/state pairs $(\bar{n}, s)$. The intuition is that it is made up of the value function of the subcontroller and of the value of the next node on the same level. For comprehension purposes, this constraint is depicted in Figure 3.4. The first part is a summation over all the nodes of the subcontroller that can be attained and $V_{n_{beg}}$ stands for the value of that first node in the subcontroller. The rest of the constraint, preceded by $oc(s_{end}, n_{end}|s_0, n_0)$ is:

  $$\Pr(n', a|\bar{n}, o_k)R(s_{end}, a) + \sum_{s' \in S, o \in O} \Pr_\gamma(s'|s_{end}, a) \Pr(o|s', a) \Pr(n', a|\bar{n}, o)V_{n'}(s')$$

  It is very similar to the version of Bellman's equation from the previous constraint as it stands for the transition to the node after $\bar{n}$ on the same level. s_{end} is the state that

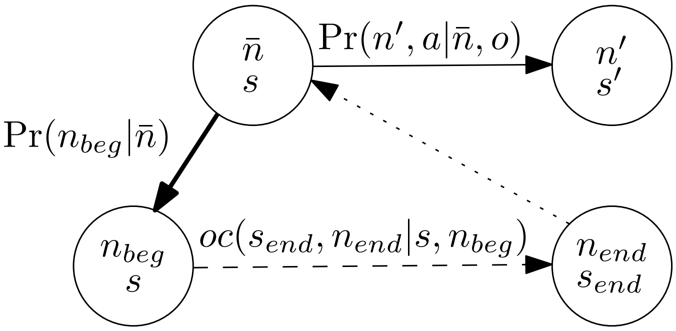


Figure 3.4: A controller representing Equation (3.3). Solid edges represent a direct connection between two nodes. Long-dashed edges shows that there's a directed path between two nodes and short-dashed edges represent a vertical transition.

the subcontroller will end in and n' is the next node. The *oc* variables are therefore used to give an expected value of the possibility of terminating the subcontroller in state s_{end}.

- Equation (3.4) represents the constraint of the general *oc*. Note that contrary to Bellman's equation for V, the *oc* variables are updated from the end (from n') to the first node n_0 in the reverse order of the sequence of actions generated by the policy. For clarity this constraint is divided into two parts:

  Equation (3.4a) represents the update of the *oc* when the policy went to node n' from a concrete node n.

  Equation (3.4b) is similar to the first part except that it deals from a transition to an abstract node $\bar{n}$ to n'. This implies that a second subcontroller (and possibly many more subcontrollers) needs to be examined. This is shown by the last two terms of the equation:

$$oc(s_{end}, n_{end}|s, n_{beg}) \Pr(n_{beg}|\bar{n}) \tag{3.6}$$

  This constraint is depicted in Figure 3.5.

- The last constraints that were not part of Equations (3.1,3.2,3.3,3.4), are the constraints that force variables x, z to be probability distributions. These constraints

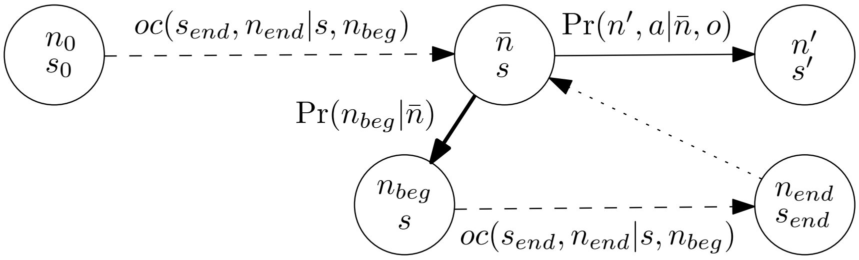


Figure 3.5: Controller representing Equation (3.3). Solid edges represent a direct connection between two nodes. Long-dashed edges show that there's a directed path between two nodes and short-dashed edges represent a vertical transition.

are much like the ones in Poupart's formulation [41]:

$$\underbrace{\Pr(n_{beg}|\bar{n})}_{z} \geq 0 \quad \forall n_{beg}, \bar{n} \qquad \text{and} \quad \sum_{n'} \underbrace{\Pr(n_{beg}|\bar{n})}_{z} = 1 \quad \forall \bar{n} \tag{3.7}$$

$$\text{and} \quad \underbrace{\Pr(n', a|n, o)}_{x} \geq 0 \quad \forall n', a, n, o \qquad \text{and} \quad \sum_{n',a} \underbrace{\Pr(n', a|n, o)}_{x} = 1 \quad \forall n \tag{3.8}$$

$$\text{and} \quad \sum_{n'} \underbrace{\Pr(n', a|n, o)}_{x} = \sum_{n'} \underbrace{\Pr(n', a|n, o_k)}_{x} \quad \forall a, n, o, \text{ and a fixed } o_k \in O. \tag{3.9}$$

.

Using Dietterich's definitions (see Section 2.1.3), it can be said that the policy corresponding to the optimum objective of the optmization problem, is at least hierarchical optimal. In reality, since the hierarchy discovery is part of the optimization, finding a global maximum to the optimization problem yields the optimal policy given a fixed number of nodes.

In theory, our method allows for recursive controllers to be found using the exact optimization program shown in Table 3.1. In practice, allowing an abstract nodes to recursively call itself from a vertical transition will lead to a controller only doing a concrete action after an indefinite number of abstract actions have been executed. This has a direct consequence in the optimization problem that the value of abstract nodes can grow unboundly. A simply solution is to restrict abstract nodes to execute vertical transitions to concrete nodes. However this is more restrictive than needed. A more general restriction only requires that vertical transitions form an acyclic graph—in other words abstract nodes cannot recursively transition vertically to one another but one abstract node can execute a vertical transition to another abstract node. This assures that in practice a concrete action is eventually executed. This constraint can be formalized as follows:

$$\Pr(\bar{n}'|\bar{n}) = 0 \text{ if } label(\bar{n}') \leq label(\bar{n}), \forall \bar{n}', \bar{n}$$

where the *label* function refers to the labelling of all abstract nodes. This labelling induces an ordering on the nodes where a node which makes a vertical transition to another has a smaller number than that other. This constraints, therefore, guarantees that cycles cannot exist.

3.3 Algorithms and Solution Techniques

The way we have formulated hierarchy discovery as an optimization problem is a principled approach that solves both hierarchy discovery and policy optimization. The main issue is how to successfully solve the highly non-linear and non-convex program. Since non-convex functions contain multiple local maxima (extrema in the general case) it is not possible to guarantee convergence to global optima unless all possible solutions are examined (which is at best intractable). This section will examine, explain and justify the different methods that we have put in place to successfully solve the problem.

Let us first take a closer look at our problem from an optimization point of view. In this section, and from now on, we will refer to the variables of the optimization program as their underbraced symbols from Equations (3.1,3.2,3.3,3.4), that is x, y, w, z. In its current form all the variables of the program are continuous. Here is a list of the lower and upper bounds of all the variables:

$$\begin{aligned}
\frac{R_{min}}{1-\gamma} \leq y(n,s) & \leq \frac{R_{max}}{1-\gamma}, & \forall n \in \{N \cup \bar{N}\} \\
0 \leq w(s',n',s_0,n_0) & \leq \frac{1}{1-\gamma}, & \forall s', s_0 \in S, n', n_0 \in \{N \cup \bar{N}\} \\
0 \leq w(s_{end}, n_{end}, s, n_{beg}) & \leq 1, & \forall s_{end}, s \in S, n_{end}, n_{beg} \in \{N \cup \bar{N}\} \\
0 \leq z(n_{beg}, \bar{n}) & \leq 1, & \forall n_{beg} \in N, \bar{n} \in \bar{N} \\
0 \leq x(n',a,n,o) & \leq 1, & \forall n', n \in N, a \in A, o \in O
\end{aligned}$$

Where $R_{max} = \max\{R\}$ and similarly $R_{min} = \min\{R\}$ and $R_{max,min}/(1-\gamma)$ indicate the extrema rewards that a process would acquire if it could receive the maximum (or minimum) reward at each time step for infinitely many time steps (infinite horizon). These constitute tight upper and lower bounds on the value function in the sense that there exist POMDPs for which the constraints are equalities.

3.3.1 Non-Linear Solvers

Given the nature of the problem, the first idea is to use a non-linear solver. In fact, Amato et al. [2] successfully used such a solver called *SNOPT* [18] to optimize non-hierarchical (flat) FSCs. *SNOPT* is a general purpose Sequential Quadratic Programming (SQP) solver. SQP is an iterative method that is comprised of major and minor iterations. In the major iterations, the problem is reformulated as a Quadratic Programming (QP) second order Taylor approximation problem. A QP is a program with a convex quadratic objective and linear constraints. The nonlinear constraints are, therefore, linearized using approximations. The idea of solving multiple QPs to solve the original problem is to generate a sequence of solutions that satisfy the linear constraints and will converge to satisfy the non-linear constraint and find local optimality. The QP is itself solved using an iterative method, hence the minor iterations.

The intuition is that there is an important approximation made by *SNOPT* because it reduces the quartic problem to a series of quadratic problems. Nonetheless, non-linear solvers, such as *SNOPT*, can be used to tackle our optimization program in its most general form, that is without any modifications.

3.3.2 Mixed-Integer Non-Linear Programing Solvers

The hierarchy discovery optimization problem with its original constraints contains only continuous variables. Specifically, having continuous x's and z's define stochastic hierarchical controllers which in turn can be seen as a stochastic policy. An alternative is to have deterministic controllers where a node executes a certain action with probability one and transfers to the next node, given an observation, with the same certainty. The x and z variables therefore take either 0 or 1 as values and they still have to sum up to one for any given node. Since, in practice, controllers are often almost deterministic, restricting those variables has a modest impact on the final solution. From an optimization point of view this makes these variables binary variables (or integer with values 0 and 1). Setting all of the x and z variables to integer variables gives rise to a new class of programs called mixed integer non-linear programs (MINLP). It is called a mixed-integer program because it contains a mixture of integer and continuous variables. This doesn't change the fact that the problem is non-linear, but it gives access to a new class of solvers for MINLPs. Furthermore, although MINLPs are usually computationally harder than NLPs (given the same number of variables), transforming our problem into a MINLP adds other constraints since only one x and one z per node or abstract node respectively can take a non-zero value. Nevertheless, the problem is still non-convex.

There exist several methods to solve MINLP problems [19]. We will give a brief overview of the branch and bound (BB) method. The BB method is the one that is exploited[3] by *MINLP_BB* [29] and *Bonmin* [5], the two MINLP solvers that will be used for empirical evaluations in the next chapter.

The idea of the BB method is to frame the optimization as a tree search where a problem is to be solved at each node [19]. Each node therefore corresponds to a problem and each branch corresponds to a new constraint added to the child of a parent. Specifically,

[3]Note that *Bonmin* offers a choice of two methods. A BB-method and an Outer Approximation (OA) method. Because of the higher level of maturity of the BB method in the *Bonmin* software, only it will be used.

each node contains an NLP problem that is the relaxed version of the MINLP, with the integer variables relaxed to continuous variables. This problem is solved using an NLP solver. At each node, starting from the root node, if the solution from the relaxed NLP problem satisfies the integrity constraints then the search is over. If not, two new relaxed NLP problems are branched from the initial one. For each of them, one of the original integer variables, let us call it v, which hasn't seen its integrity constraint met is added a constraint. In one branch, it is given an upper bound in the form of $v \leq \lfloor v_{nlp} \rfloor$ and in the other branch it is lower bounded by $v \geq \lceil v_{nlp} \rceil$ where v_{nlp} stands for the non-integer value for variable v returned by the NLP solver. A branch is then chosen and the NLP solver is called with this new relaxed NLP problem.

There are three cases that will make the search in the tree backtrack. In the first case, the relaxed NLP problem is infeasible and thus it is a leaf node. In the second case the node returns an integer solution and the solution is therefore a lower bound to the problem (given that the objective is to be maximized). The third possibility is that a solution is returned but its objective value is lower than the current lower bound. Again in this case there is no need to explore the subtree of the node since this solution is an upper bound of all of its descendants.

The BB method is very simple and it is particularly good when the NLP problem is fairly easy to solve. It is, however, not the most efficient since it has to solve an NLP problem, which is meaningless unless it returns an integer solution, at each node in the tree [30].

3.3.3 Mixed-Integer Linear Programming Solvers

In the previous approach some continuous variables were restricted to be integer variables. Although restricting the policy variables in such a way meant going from stochastic policies to deterministic policies, the optimization problem remained non-linear and non-convex.

The next logical approach would be to reformulate the problem to obtain a problem on which solvers can be guaranteed to find a global maximum, by using an approximated formulation. Before we present two ideas to do so, let us have a more theoretical look at how sources of non-linearity can be removed.

First, the multiplication of a continuous variable with a binary variable can be replaced by a new continuous variables by disjunctive programming [3]. Imagine a binary variable denoted B and a continuous variable, C with upper bound U_C and lower bound L_C. The product BC can be replaced by a new variables D by adding the following constraints:

$$\begin{aligned} C + (B-1)U_C \leq & D \leq C + (B-1)L_C \\ BL_C \leq & D \leq BU_C \end{aligned}$$

If B is 0, D is forced to be 0 by the second constraint and $C - L_C \geq 0$ and $C - U_C \leq 0$ which is trivially true. When, B is 1, the first constraint becomes $C \leq D \leq C$ which is the wanted effect. Applying disjunctive programming therefore adds a variable, and four constraints but reduces the degree of the polynomial without any approximation.

Once, the policy variables, x and z, have been constrained to be binary, the disjunctive programming approach can be directly applied to the hierarchy discovery program. In fact, it immediately linearizes Equations (3.2,3.4a) and reduces the degree of Equations (3.3,3.4b) to being quadratic. Let us now present two other methods to deal with these remaining non-linearities.

Mixed-Integer Linear Programming (MILP)

Using disjunctive programming, it was shown how products of continuous and integer variables can be linearized by adding a new variable. The problem in the hierarchy discovery program is that applying this trick still yields quadratic constraints where the continuous variables y's and w's are multiplied. A straightforward approach is to transform another set of variables from continuous to binary. It wouldn't make much sense to do it for the y values since they represent the goodness of a controller. Instead, making some of the w variables binary, the ones which give the values for the complete path of a subcontroller (i.e., $oc(s_{end}, n_{end}|s, n_{beg})$), implies that a subcontroller will only lead to a single end state. Proceeding in this way makes the problem a Mixed-Integer Linear Program (MILP) and a global solution, of this approximated formulation, is therefore guaranteed (given enough time).

The method that is used to solve a MILP by solvers such as *CPLEX* [1] is a BB method much like the one that was presented in the previous section (Section 3.3.2). The main difference is that at each node the relaxed problem is a Linear Program (LP) instead of a NLP. In addition, *CPLEX* implements a variety of tricks to reduce the number of nodes to explore.

Bounded Hierarchical Policy Iteration (BHPI)

A second idea to linearize the program once disjunctive programming has been applied, is to fix certain variables to linearize the constraints. In fact this can be accomplished by having an iterative approach where in the first step, some variables are fixed and in the second the fixed variables are updated. Formally:

- Fix part of the continuous variables y and w. Specifically, fix the $V'_n(s')$ in Equation (3.3) as well as the $oc(s, \bar{n}|s_0, n_0)$ in Equation (3.4b). Doing so permits the use of disjunctive programming, without any approximation, on the remaining variables. Then, optimize for the remaining variables until an optimum is found.

- Update the fixed variables. For the y's this is a simple matter of updating the fixed values of the first part based on the unfixed ones. For the oc variables, it is a matter of solving the linear system of Equation (3.4).

Unlike Poupart et Boutilier's Bounded Policy Iteration (BPI) [41], BHPI is not guaranteed to converge. However, in practice, it has been our experience that it is usually monotonically increasing.

By first improving the policy and, in the second step updating the value function, this procedure is reminiscent of Policy Iteration (see Section 2.2.2), hence its name, Hierarchical Policy Iteration. This approach can be solved by invoking *CPLEX* to solve the first step of the iterative approach. Although the second step doesn't required any optimization, but rather a solution to a system of equations, it can also be solved by *CPLEX* by optimizing for a dummy objective.

3.3.4 Setting the number of nodes

In this chapter, we have assumed that in addition to the planning problem, the optimization program was also given a set of nodes, $\{N\}$, that define the size of the hierarchical controller. Although the work of Poupart [41] has proposed a method to adapt the size of this set as needed during optimization, this is not addressed in this thesis. Instead, since optimal controllers may have infinitely many nodes, our approach guides for selecting the maximum number of nodes given the computational resources available to perform the optimization.

3.4 Conclusion

This chapter introduced a general method to do hierarchy discovery and policy optimization in partially observable environments. This method searches for optimal policies in a space of hierarchical policies. This space is not a subspace of the policy space given a fixed number of nodes. In fact, since hierarchical policies can express equivalent flat policies with less node, this new search space is a superset of the original space. In addition, several techniques to solve the resulting non-convex formulation were proposed. The first two methods will solve the non-convex problem directly as the last two methods approximate the original formulation and add a number of variables and constraints, but ensure that a global maximum to the approximated problem is found. The next chapter will offer an empirical evaluation of these techniques.

Chapter 4

Empirical Evaluation

4.1 Introduction

In Chapter 3, we introduced an optimization problem which can find both the optimal hierarchy and optimal policy (given a fixed number of nodes) of a POMDP. This idea surpasses previous definitions of hierarchical optimality by Dietterich [11]. The main advantage of this approach is to formalize the hierarchy discovery problem by formulating the search as an optimization problem which can be tackled with general purpose optimization solvers. Unfortunately, the proposed optimization problem is non-convex and as such a global maximum is not guaranteed to be found (in the MILP and BHPI cases we can only guarantee that global maxima on an approximation of the original problem will be found). Furthermore, as other approaches partition the search space in order to gain tractability, our approach does not, leaving the problem to be at least as hard as solving for flat policies. One advantage of searching in the space of hierarchical policies is the intuition that planning problems, especially as they get larger, often have a natural hierarchy that can potentially be exploited by our formulation. Furthermore, given a fixed number of nodes, a hierarchical solution is more expressive than a flat policy due to the fact that it can re-use substructures. In other words, it can represent a richer policy than an equivalent size flat policy because it is hierarchical.

To test the validity of our approach, it would be interesting to show a theoretical upper bound on the difference between the optimum and the local optimum that will be found, however it seems unlikely to give meaningful results in practical settings. Specifically, for non-convex problems, there may be a large gap between the objective values of the primal and dual problem. Hence, the definitive way of testing the successfulness of our method is through an experimental evaluation. The next sections will introduce three well-known problems previously used in the literature[1]. The results of using the different optimization techniques mentioned in Section 3.2 will be compared and discussed.

4.2 Problems

Before diving in to the experiments, let us explain the chosen experimental methodology.

All the different formulations of the problem were encoded using the AMPL language. In addition to being a modelling language designed for mathematical programming, AMPL is also an environment with which many optimization solvers can interface.

The linear optimization methods (MILP and BHPI) were run using *CPLEX* version 10.0 [1], the NLP solvers used were *SNOPT 5.3* [18] and *filterSQP* [16], and the MINLP solvers used were *MINLP_BB* [29] (using *filterSQP* as its continuous solver) and Bonmin (using Ipopt as its continuous solver) [5].

In the next sections, experimental results are further explained using two main parameters: the objective value (the expected value function) attained by the optimization problem and the required time to achieve such a result. Note that time measurements depend on the parameter settings of each solver. Since the optimal setting is unknown, it may be possible for each solver to achieve better results with different settings.

4.2.1 Paint

The Paint problem was introduced by Kushmerick et al. [28] and it is used by Pineau [38] to test the validity of her proposed hierarchical planning algorithm (PolCA+). The setting of the Paint problem is a shop where parts should be painted and shipped if they pass the inspection (indicating that the part is unflawed) and reject if they do not. The state space

[1] All of these problems are made available, in an easily usable format, thanks to Tony Cassandra, from his website, `http://www.pomdp.org`

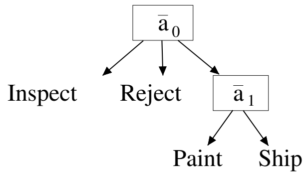


Figure 4.1: The paint task hierarchy as presented by Pineau [38].

refers to the state of the part to be painted. It contains three boolean features, whether the part is *flawed*, whether it is *blemished* and whether it is *painted*, but only allows for four combinations of these variables: the states are *unflawed-unblemished-unpainted*, *unflawed-unblemished-painted*, *flawed-unblemished-painted*, *flawed-blemished-unpainted*. The states are inferred through observations which are either *blemished* or *unblemished*. There are four possible actions: *Paint*, *Inspect*, *Reject*, *Ship*. The action hierarchy, as given by Pineau [38], is given in Figure 4.1.

The rewards occur when a piece is rejected or when it is shipped. If it is correctly shipped (the part was *unflawed-unblemished-painted*) the system receives a +1 reward, if it is incorrectly shipped the system receives a −1 reward. Similarly, with the reject action, when a *flawed-blemished-unpainted* part is rejected, the reward is +1 and −1 if any other part is rejected. Initially, the part has equal probability of being *unflawed-unblemished-unpainted* or *flawed-blemished-unpainted*.

There are two sources of uncertainty. First, the observations which have a 25% chance of reporting the incorrect blemish value of the part. Second, the paint action has a 10% chance of leaving a part unpainted and the same probability of leaving a blemish on a painted part. The other actions, ship and reject, are deterministic (both actions lead to one of the initial states).

This problem, although small, constitutes a good first test for our optimization problem. Furthermore, the fact that Pineau has shown a manual decomposition of the action space (Figure 4.1) and has also shown that the optimal policy follows this decomposition [38] makes for a good comparison.

Table 4.1 shows the results for this problem. Non-linear techniques, apart from *SNOPT*, find optimal values in a very short time. BHPI is also doing well although its iterative

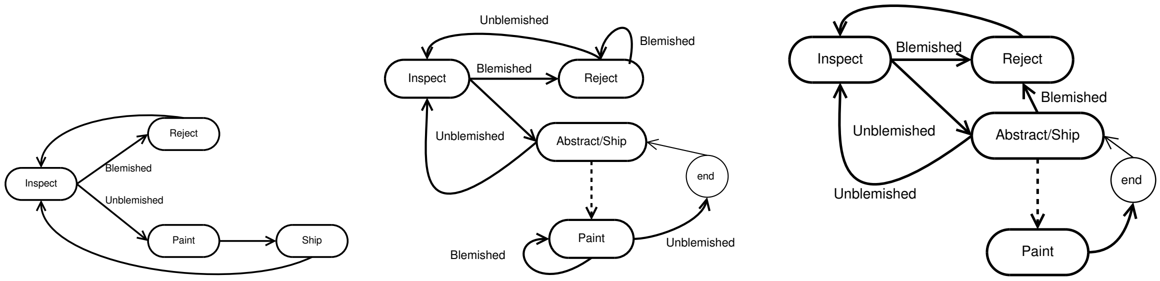
	4(3/1)	
	Time	V
Optimal	–	3.29
NLP-SNOPT	2s	0.48146
NLP-filter	< 1s	3.29
BHPI	13s	3.29
MILP	3028s	2.077
MINLP_BB	< 1s	3.29
Bonmin	42s	3.12

Table 4.1: Experiment results with the Paint problem ($|S| = 4, |A| = 4, |O| = 2$). All experiments were done with 2 levels and 4 nodes, three at the top level and one on the lower level (i.e., 4(3/1))

approach is not the fastest. The hope was that by having fewer variables to solve it could scale better – this is certainly the case compared to the MILP method. In this problem MILP deals with over 9000 variables and BHPI about 5000 variables (these numbers are after a pre-solve operation which simplifies the problem by consolidating variables and eliminating redundant constraints). Figure 4.2 compares the policy found with *MINLP_BB* and BHPI to the optimal policy (as shown by Pineau [38]). The three policies are equivalent and their structure is very similar. The only difference between the optimal flat policy and the optimal policy found using BHPI is that after the *ship* action, the solution found by BHPI rejects the next piece if it looks blemished. The policy found by *MINLP_BB* offers a similar concept, where, after a reject, the next part is rejected if it is blemished. This policy also paints part that seem blemished which makes sense since painting can make the part ready for shipping (by hiding the blemish).

4.2.2 Shuttle

The Shuttle program was first proposed by Chrisman [8]. It is a simple delivery task by a spaceship between two space stations. Each station, separated from the other by an outer space state, has a dock on which the agent must backup with merchandise taken from the



(a) Optimal flat policy [38] (b) Policy found using *MINLP_BB* (c) Policy found using BHPI

Figure 4.2: Comparison of three policies for the paint problem. The first is the optimal flat policy shown in [38] and the two others are the policies discovered by two of our solving methods.

last station (stations are identified as least recently visited, lrv, and most recently visited, mrv). The states represent the position of the spaceship:

$$S_p = \{\mathit{space, outside\_lrv\_station, outside\_mrv\_station}\}$$

and its orientation with respect to the two stations:

$$S_o = \{\mathit{facing\_lrv, facing\_mrv}\}$$

Furthermore, the spaceship can dock in both of the stations for a total of eight different states ($|S| = |S_p| \times |S_o| + |\{\mathit{dock\_mrv, dock\_lrv}\}|$). The spaceship has three actions that it can execute: *turn_around*, *backup* and *go_forward*. To properly dock to a station it must have its back facing the station and execute the *backup* action. The spaceship is given a +10 reward if it correctly docks to the least recently visited station (the least recently visited station is recognizable by the agent), or -3 if it enters a station while facing it (collision), other events are associated a 0 reward. Both transitions and observations are noisy and limited as the agent cannot see a station unless it is directly outside of it.

The results presented in Table 4.2 are similar to the ones of the Paint problem. The new aspect in the evaluation of this problem is that we tried four different number of nodes on two levels. What is most surprising is that increasing the number of nodes doesn't

	4 Nodes (3/1)		6 Nodes (4/2)		7 Nodes (4/3)		9 Nodes (5/4)	
	Time	V	Time	V	Time	V	Time	V
Optimal	–	32.7						
NLP-SNOPT	2s	31.87	6s	31.87	26s	31.87	1449s	30.27
NLP-filter	4s	18.92	82s	31.87	994s	18.97	2375s	12.07
BHPI	85s	18.92	7459s	27.93	10076	31.87	10518	3.73
MINLP_BB	4s	18.92	221s	27.68	N/A		N/A	
Bonmin	141s	24.49	9758s	29.12	N/A		N/A	
MILP	$> 20000s$	N/A	$> 20000s$	N/A	$> 20000s$	N/A	$> 20000s$	N/A

Table 4.2: Experimental results with the Shuttle problem ($|S| = 8$,$|A| = 3$,$|O| = 5$), where (i/j) stands for number, i, of nodes on the higher level and j, the number of nodes on the lower level. N/A can mean two things. Either that a result is not available given the time limit in which case this limit will be indicated in the adjacent column, either that the solver was not able to find a feasible solution.

always amount to finding better policies, although by increasing the size of the search space it always amounts to longer running times. Neither MINLP methods (*MINLP_BB* and *Bonmin*) can solve this larger problem (both were unable to find a feasible solution). Let us also mention that contrary to its performance on the Paint problem, *SNOPT* finds near-optimal results in very short running times, surpassing the results of *filterSQP*.

4.2.3 4x4 Maze

The 4x4 Maze is a very simple (four by four) grid-environment, see Figure 4.3, proposed by Cassandra et al. [6] where the agent has to reach the goal state denoted by a star, having started at a randomly chosen state amongst the fifteen others. The agent can execute four actions (*North, South, East, West*) to try to reach the four adjacent cells, but all the states, except the goal, are un-differentiable given observations. The agent must then act blindly until it reaches the goal state and receives a reward. All the state transitions are deterministic, except when the goal state is reached in which case the agent, as we mentioned earlier, has a uniform probability of being re-positioned on any of the 15

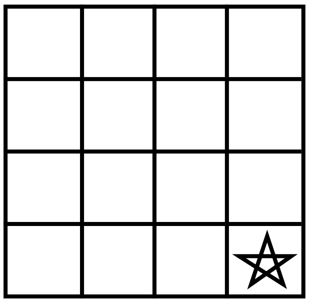


Figure 4.3: The 4x4 maze environment as shown by Cassandra et al. [6]

	4(3/1)	
	Time	V
Optimal	–	≈3.71
NLP-SNOPT	3s	3.15
NLP-filter	26s	3.73
BHPI	397s	3.21
MILP	too large	
MINLP	30s	3.73
Bonmin	332s	3.66

Table 4.3: Experiment results with the Maze problem ($|S| = 16$,$|A| = 4$,$|O| = 2$)

remaining squares.

Results are presented, using a two level and four node controller, in Table 4.3. Results are similar to the previous problems, where each solver gets relatively close to the optimal value in varying time.

4.3 Discussion and Conclusion

The experiments show that our approach performs well on some small-world problems. Although, these problems are small POMDPs, they end up being medium to large size optimization problems (from a few thousand variables for the NLP approach to around one hundred thousand variables for the BHPI approach) and as such, scalability is the main problem of our approach. Non-linear solvers that work with less variables for a given problem compared to linear approaches seem to be able to get to good solutions faster. The approximate approach consisting of reducing the non-convex problem to a full MILP doesn't seem to have much success. On the other hand the BHPI approach is a lot more promising. It shows that an iterative method can be very successful, taking advantage of the fact that the number of variables at each iteration is reduced since some variables are fixed. Keeping this in mind, it would be interesting to develop such a method while using a non-linear solver to use the best of both worlds. Although, this method can be directly applied to the non-linear case, preliminary experiments have shown that this method does not always converge to a local maximum.

All the experiments that we have run are quite small. One has to remember that our approach does not have access to a pre-made hierarchy and thus it is at least as hard to solve the current problem than to solve the original POMDP. Furthermore, the intuition that searching in the space of hierarchical policies might help the optimization did not turn out to ease the search in any flagrant way. This is explainable considering the difficulty to find sharable substructures given the size of the controllers used to solve the test problems. Nonetheless, if, part of, or a complete hierarchy is provided, it would be a simple matter to encode it by adding restrictions in the optimization problem. As such, our method could scale up by using prior information in a format that would be similar to other methods. The advantage of our method is that any type of partial prior information can be used by the system. Therefore, a human practitioner could leverage his knowledge by encoding it into the optimization program, while optimizing the remaining transitions that are unknown to him.

Chapter 5

Conclusion

For the past decade Markov decision processes have been at the heart of decision-theoretic planning. In practice, since real-life problems often involve large state spaces, Markov processes in fully and especially in partially observable environments become intractable to solve. Several researchers have proposed to manually decompose a process' action space into a hierarchy, creating smaller subtasks that can be solved independently.

Several authors have mentioned that the next logical step is to discover or learn hierarchies [49, 12], although few researchers have actually proposed ways to do so and only in fully observable domains. This thesis remedies this shortcoming by introducing an optimization program that can discover the optimal hierarchy and the optimal policy in partially observable domains. It is the first method that searches in a space of hierarchical policies which is not a subset of the space of flat policies. As such, hierarchical policies found by it are potentially more expressive than their flat counterparts given a fixed number of nodes. Our approach also allows for recursive controllers which can represent, with a finite number of nodes, policies that would otherwise require infinitely many nodes.

The proposed optimization program is non-convex and as such finding a good solution is not guaranteed. We have presented experiments on some small problems which verify the success of solving the optimization problem using several techniques. Proposed techniques focus on linear reformulations or approximations of the original program.

5.1 Future Work

Since our approach is the first to propose to discover hierarchies for POMDPs there are many new directions that would be interesting to pursue.

The scalability of the method will have to be improved so that it can solve larger problems. In addition to the sheer merit of solving larger domain POMDPs, being able to solve larger problems will also allow for comparison between discovered hierarchies and human-made hierarchies. Furthermore, it will potentially allow even larger problems to be solved by combining human prior knowledge and our automatic discovery method.

Practically, since it would be surprising to stumble on a new solver that can successfully solve much larger problem in reasonable times, approximate methods, will have to be investigated in greater details. Experimental results indicate that non-linear approaches work relatively well. Therefore, a non-linear iterative method that can be proved to be monotonically increasing would seem to be a natural next step. Having said this, there might also be merit in trying out a class of optimization solvers which focus on finding global optimum, global optimizers [34, 35].

Other approximation methods should also be considered. In the spirit of previous work, future research could focus on finding a principled way to partition the state space as we discover part of the hierarchy or maybe to constrain the search by forcing it to partition the state space.

In one of our approximation methods we set the occupancy frequency to be deterministic which did not turn out to be a success. Nonetheless, since the occupancy frequency is completely determined by the policy and the hierarchy there must be another way to approximate it. For example, the occupancy frequency variables (oc variables from Table 3.1) could be pre-calculated off line by only considering whether or not two states can be reached given the number of nodes at a sub-level, without regards for the actual policy.

Recursive controllers are very attractive although no experiment has shown that they could be successfully discovered. Intuitively problems in the natural language field might be able to benefit from such controllers although we have not been able to find such an application thus far.

On a theoretical level, there might be a need to find a convex relaxation of the problem. Although a relaxation from any quadratically constrained linear program to semi-definite programming (SDP) is possible [52], current SDP solvers do not seem to be able to solve problems involving thousands of variables. Alternatively, there might exist other ways to frame the problem, for example using control theory, so that larger instances are successfully solvable.

This work could also be extended in two different fields. First, although applying this technique to fully observable domains is trivial, by removing the observation probabilities in the optimization program, this approach might not be the best since each node would require $|S|$-outgoing arcs. A more pleasing approach would see states represented as nodes in which case there would be a maximum of $|S|$ nodes. In any case, there must be a way to re-formulate the hierarchy discovery problem specifically for MDPs. The idea would be to discover the action hierarchies as they are presented in Section 2.1.3.

Another extension would be to see if this work can be adapted to a reinforcement learning setting. Typically, in reinforcement learning transition probabilities are completely unknown and an agent must explore the world in search of rewards. As such, can our idea can be extended to build a hierarchy online ? If so, can this hierarchy help in the exploration versus exploitation trade off, by rapidly identifying structure that can be shared ? It would seem that many more exciting questions will arise from such a quest.

Bibliography

[1] User's manual, ILOG CPLEX 10.0, January 2006.

[2] C. Amato, D. Bernstein, and S. Zilberstein. Solving POMDPs using quadratically constrained linear programs. In *To appear In International Joint Conferences on Artificial Intelligence (IJCAI)*, 2007.

[3] E. Balas. Disjunctive programming. *Annals of Discrete Mathematics*, 5:3–51, 1979.

[4] Richard Bellman. A problem in the sequential design of experiments. *Sankhya A*, 16:221–229, 1956.

[5] P. Bonami, A. Waechter, L. Biegler, A. Conn, G. Cornuejols, I. Grossmann, C. Laird, J. Lee, A. Lodi, F. Margot, and N. Sawaya. An algorithmic framework for convex mixed integer nonlinear programs. Technical Report RC23771, IBM Research Report, October 2005.

[6] 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 (AAAI-94)*, volume 2, pages 1023–1028, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[7] Laurent Charlin, Pascal Poupart, and Romy Shioda. Automated hierarchy discovery for planning in partially observable environments. In B. Schölkopf, J.C. Platt, and T. Hofmann, editors, *To appear in Advances in Neural Information Processing Systems 19*, Cambridge, MA, 2007. MIT Press.

[8] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence, (AAAI-92)*, pages 183–188, 1992.

[9] Thomas Dean and Robert Givan. Model minimization in markov decision processes. In *AAAI/IAAI*, pages 106–111, 1997.

[10] Thomas Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *Proceeding of the Thirteenth Conference on Unvertainty in Artificial Intelligence*, pages 124–131, 1997.

[11] T. Dietterich. The MAXQ method for hierarchical reinforcement learning. In Jude W. Shavlik, editor, *ICML*, pages 118–126. Morgan Kaufmann, 1998.

[12] T. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Jounral of Artificial Intelligence Research JAIR*, 13:227–303, 2000.

[13] Thomas G. Dietterich, Richard H. Lathrop, and Tomas Lozano-Perez. Solving the multiple instance problem with axis-parallel rectangles. *Artif. Intell.*, 89(1-2):31–71, 1997.

[14] Alvin W. Drake. *Observation of a Markov Process through a noisy channel*. PhD thesis, Massachusetts Institute of Technology, 1962.

[15] Shai Fine, Yoram Singer, and Naftali Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 32(1):41–62, 1998.

[16] R. Fletcher and S. Leyffer. User manual for filterSQP, 1998.

[17] M. Ghavamzadeh and S. Mahadevan. Hierarchical policy gradient algorithms. In T. Fawcett and N. Mishra, editors, *Proceedings of the International Conference on Machine Learning ICML*, pages 226–233. AAAI Press, 2003.

[18] P. Gill, W. Murray, and M. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005.

[19] Ignacio E. Grossmann. Review of nonlinear mixed-integer and disjunctive programming techniques. *Optimization and Engineering*, 3(3):227–252, September 2002.

[20] E. Hansen and R. Zhou. Synthesis of hierarchical finite-state controllers for POMDPs. In E. Giunchiglia, N. Muscettola, and D. Nau, editors, *ICAPS*, pages 113–122. AAAI, 2003.

[21] Eric A. Hansen. An improved policy iteration algorithm for partially observable mdps. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *NIPS*. The MIT Press, 1997.

[22] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *Fourteenth International Conference on Uncertainty In Artificial Intelligence*, pages 220–229, 1998.

[23] Bernhard Hengst. Discovering hierarchy in reinforcement learning with hexq. In Claude Sammut and Achim G. Hoffmann, editors, *ICML*, pages 243–250. Morgan Kaufmann, 2002.

[24] Natalia Hernandez-Gardiol and Sridhar Mahadevan. Hierarchical memory-based reinforcement learning. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 1047–1053. MIT Press, 2000.

[25] Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *ICML*, pages 167–173, 1993.

[26] Craig Knoblock. A theory of abstraction for hierarchical planning. *Change of Representation and Inductive Bias*, pages 81–104, 1990.

[27] R.E. Korf. Planning as search: a quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[28] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An algorithm for probabilistic planning. *Artifical Intelligence*, 76(1-2):239–286, 1995.

[29] Sven Leyffer. User manual for MINL_BB. http://www-unix.mcs.anl.gov/~leyffer/solvers.html, 1999.

[30] Sven Leyffer. Integrating SQP and branch-and-bound for mixed integer nonlinear programming. *Computational Optimization and Applications*, 18:295–309, 2001.

[31] Christopher Lusena, Judy Goldsmith, and Martin Mundhenk. Nonapproximability results for partially observable markov decision processes. *Journal of Artificial Intelligence Research (JAIR)*, 14:83–103, 2001.

[32] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.

[33] Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In Carla E. Brodley and Andrea Pohoreckyj Danyluk, editors, *ICML*, pages 361–368. Morgan Kaufmann, 2001.

[34] A. Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, 13:271–370, 2004.

[35] I. Nowak. *Relaxation and Decomposition Methods for Mixed Integer Nonlinear Programming, Numerical methods in approximation theory*, volume 152. Springer, 2005.

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

[37] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1997.

[38] J. Pineau. *Tractable Planning Under Uncertainty: Exploiting Structure*. PhD thesis, Robotics Institute, Carnegie Mellon University, 2004.

[39] J. Pineau, N. Roy, and S. Thrun. A hierarchical approach to POMDP planning and execution. In *Workshop on Hierarchy and Memory in Reinforcement Learning (ICML)*, 2001.

[40] Pascal Poupart. *Exploiting Structure to efficiently solve large scale partially observable Markov decision processes*. PhD thesis, University of Toronto, 2005.

[41] Pascal Poupart and Craig Boutilier. Bounded finite state controllers. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *NIPS*. MIT Press, 2003.

[42] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[43] Andrej Ruszczynski. *Nonlinear optimization.* Princeton University Press, Princeton, NJ, 2006.

[44] E. Sondik. The optimal control of partially observable decision processes over the infinite horizon: Discounted cost. *Operations Research*, 26(2):282–304, 1978.

[45] R. Sutton, D. Precup, and S. Singh. Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[46] Richard S. Sutton. On the significance of markov decision processes. In Wulfram Gerstner, Alain Germond, Martin Hasler, and Jean-Daniel Nicoud, editors, *ICANN*, volume 1327 of *Lecture Notes in Computer Science*, pages 273–282. Springer, 1997.

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

[48] Josh D. Tenenberg. Abstraction in planning, reasoning about plans. pages 213–280, 1991.

[49] G. Theocharous, S. Mahadevan, and L. Kaelbling. Spatial and temporal abstractions in POMDPs applied to robot navigation. Technical Report MIT-CSAIL-TR-2005-058, Computer Science and Artificial Intelligence Laboratory, MIT, 2005.

[50] S. Thrun and A. Schwartz. Finding structure in reinforcement learning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, Cambridge, MA, 1995. MIT Press.

[51] M. Wiering and J. Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1997.

[52] Henry Wolkowicz, Romesh Saigal, and Lieven Vandenberghe, editors. *Handbook of Semidefinite Programming: Theory, Algorithms, and Applications*, volume 27 of *International series in operations research & management science*. Kluwer, 2000.