Fast Approximate Hierarchical Solution of MDPs

by

Jennifer L. Barry

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

© Jennifer L. Barry, MMIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by .....
Leslie Pack Kaelbling
Professor
Thesis Supervisor

Certified by .....
Tomás Lozano-Pérez
Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Fast Approximate Hierarchical Solution of MDPs

by

Jennifer L. Barry

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In this thesis, we present an efficient algorithm for creating and solving hierarchical models of large Markov decision processes (MDPs). As the size of the MDP increases, finding an exact solution becomes intractable, so we expect only to find an approximate solution. We also assume that the hierarchies we create are not necessarily applicable to more than one problem so that we must be able to construct and solve the hierarchical model in less time than it would have taken to simply solve the original, flat model.

Our approach works in two stages. We first create the hierarchical MDP by forming clusters of states that can transition easily among themselves. We then solve the hierarchical MDP. We use a quick bottom-up pass based on a deterministic approximation of expected costs to move from one state to another to derive a policy from the top down, which avoids solving low-level MDPs for multiple objectives. The resulting policy may be suboptimal but it is guaranteed to reach a goal state in any problem in which it is reachable under the optimal policy.

We have two versions of this algorithm, one for enumerated-state MDPs and one for factored MDPs. We have tested the enumerated-state algorithm on classic problems and shown that it is better than or comparable to current work in the field. Factored MDPs are a way of specifying extremely large MDPs without listing all of the states. Because the problem has a compact representation, we suspect that the solution should, in many cases, also have a compact representation. We have an implementation for factored MDPs and have shown that it can find solutions for large, factored problems.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor

Acknowledgments

Thanks first and foremost to my wonderful advisors Leslie Kaelbling and Tomás Lozano-Pérez. Their guidance and support, from helping me to pinpoint my research interests, to listening to crazy ideas and deciphering half-thought-out explanations, to giving me new directions to explore when I was completely stuck, has been incredible.

I would also like to thank the members of the LIS group and 32-G585, both past and present, for listening to and helping me refine my ideas. Special thanks to Sarah Finney, Meg Aycinena, Nick Matsakis, Kaijen Hsiao, and Luke Zettlemoyer for answering a number of inane questions about the details of the day-to-day existence of a graduate student.

Last but certainly not least, I would like to thank my family and friends for their unflagging support and encouragement. Thanks to Maricella Foster-Molina for many long, fun email conversations and a few visits, to Lily Cohen for never forgetting to tell me to have fun and go play Ultimate, and to Michelle Tomasik for getting me outside and away from the computer, listening to my various complaints, and even occasionally checking my math! My brother Andy was my own, personal tech support, fielding panicked questions at all times of day and night. David German kept me from starving through this process and also spent hours reading drafts of ideas, more hours listening to them, and a few more reassuring me that it all was going to be OK. And, of course, my love and thanks to my parents Dan and Sue, who have done everything from originally sparking and encouraging my interest in puzzles, programming, and robotics, to proofreading, to sending me crazy robots as inspiration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

"Space is big. You just won't believe how vastly, hugely mind-bogglingly big it is. I mean, you may think it's a long way down the road to the drug store, but that's just peanuts to space." – *The Hitchhiker's Guide to the Galaxy*, Douglas Adams [Adams, 1979]

Space *is* big. More importantly in this context, it has a huge number of variables associated with it. There are, at a rough estimate, 10^{80} atoms in the universe [Contributors, 2008] and, moreover, a change to any one of these atoms has the potential to change every other one.

Of course, a robot will not care about the vast majority of those atoms. In fact, our robots will not be making decisions at the atomic level. The question is, at what level *does* the robot make decisions? Somehow, to act intelligently in the world, a robot must decide which variables currently matter while not losing track of those variables that *may* matter in the future. Specifically, we have two key issues: a large number of variables and an uncertainty about how the variables affect each other.

Markov decision processes (MDPs) have been widely used as models in this sort of problem, because they are capable of representing uncertainty. An MDP consists of a set of states, a set of actions that specify non-deterministic transitions between the states, and a reward function that gives the reward associated with taking an action in a given state. Consider, for example, a robot in a search-and-rescue job. Its task is to rescue people trapped in a burning building. We can describe this problem as an MDP where the "states" describe the current locations of the robot, the people to rescue, and the fire. The "actions" describe the robot's dynamics and abilities — it can move around the building, pick up people, and attempt to block the propagation of the fire. However, because the robot's wheels and motors are not perfect, these actions can only describe non-deterministic transitions; how much the robot moves when taking the `move-forward` action, for example, is uncertain. The "rewards" are high for rescuing people and low for leaving a person trapped.

An optimal solution to this problem exists — there is some way of going about the rescue task that minimizes the time it takes and the risk that anyone will remain

trapped. However, consider trying to solve for this solution. Even if the fire starts in only one, known room in the middle of the building, if its propagation is uncertain, the number of possible configurations for it, and hence the number of states, is quickly in the trillions. By the time the robot decides what action to take in all of these states, there will be little point.

Therefore, consider dividing the building into 20 smaller regions, solving for the optimal solution *within* each region, and then solving for a solution for moving between the regions. Once we know the value of each region and how difficult it is to transition between them, we can solve for the order in which to visit the regions. This strategy considerably reduces the computation time; the number of possible configurations in 20 small, non-interacting regions is much less than the number in one, large area.

However, an even better approach is to somehow first decide the order in which to visit regions and *then* solve for what to do within each region. Now the robot can solve the second region *while* it is rescuing the people in the first region. In addition, the robot solves fewer problems involving moving from one region to another. In our first method, it had to solve for how to move from each region to *every* adjacent region. In this method, it only has to solve for how to move from one region to the next region in the overall plan. A strategy where we first make an upper-level plan and then solve the lower-level pieces is a "top-down" strategy.

In dividing the world this way, we have, in essence, proposed a 2-level hierarchical, top-down solution for the problem. Hierarchical decomposition is widely believed to be the key to solving large planning problems. However, creating and solving these hierarchies has proved challenging. In the above example, we arbitrarily divided the building into 20 regions to create our hierarchy. We had no way of knowing in advance if 20 regions would be too many regions and degrade the quality of the solution or too few and take too long to solve. In addition, although the computational advantages of the top-down approach are clear, it is more difficult to get an accurate solution using this method because there is very little information about the value of the high-level states and actions.

In this thesis, we discuss how to create and solve hierarchical models of Markov decision processes. Our goal is an approximate, efficient, top-down algorithm.

1.2 Related Work

In this thesis, we focus on solving MDPs where we are given the entire MDP model in advance (for a more complete discussion of MDPs see Chapter 2). However, a closely related problem is the problem of *reinforcement learning* ([Russell and Norvig, 2003], chapter 20). A reinforcement learning problem can often be stated as an MDP but, in this problem, the agent is not given the transition and reward model. Instead, it acts in the world, receiving rewards, and learns what to do based on what sequences of actions produce high rewards. Reinforcement learning problems and solving MDPs are related problems since reinforcement learning can be broken down into creating a transition and reward model and then solving it. Therefore we discuss work relating to both here.

Several approaches have shown that an appropriate hierarchical decomposition can provide considerable speed-up in solving an MDP or doing reinforcement learning. For example the options framework proposed by [Sutton *et al.*, 1999] allows a user to define macro-actions called "options" and then solves each option individually. Similarly [Dietterich, 1998] presents the MAXQ decomposition, which defines a hierarchy of sub-tasks. In another approach [Parr and Russell, 1997] divide the MDP state space into loosely coupled groups of states and solve each of these sub-problems separately. Other, similar approaches to solving MDPs or doing reinforcement learning are presented in [Parr and Russell, 1997; Hauskrecht *et al.*, 1998; Lane and Kaelbling, 2002].

Finding an appropriate hierarchy automatically, however, has proved challenging. [McGovern and Barto, 2001] present an algorithm that defines options by finding "bottleneck" regions in a space. The example they give is that of a room with a door; the door is a bottleneck and a good sub-goal is to reach the door. [Simsek *et al.*, 2005] define sub-goals similarly but use local cuts rather than diverse density to identify these goals. [Digney, 1996] takes a slightly different approach and defines sub-goals as states that are visited frequently or states that have a high reward gradient.

However, all of the above approaches to learning hierarchies operate over a relatively long time-scale, as the agent has to learn the world dynamics in the process, making this problem substantially more difficult than simply solving an MDP. The expectation is that the work associated with learning the hierarchy will pay off over the course of solving several related problems in the same or similar domains. Our goal, however, is somewhat different. We want to be able to *construct and solve* the hierarchical model in less time than it would have taken to simply solve the original flat model.

[Bakker *et al.*, 2005] presented an approach to this problem, which works well on two-dimensional navigation problems but does not seem to generalize well to other types of domains, sometimes failing to find any strategy, even a suboptimal one, for an achievable goal. The work of [Mahadevan, 2008] finds a multi-scale basis for representing value functions in MDPs and other problems; this basis can serve as an effective representation for solving the MDP efficiently, but is not hierarchical in the same sense as the other methods discussed here.

Other approaches attempt to utilize the structure of a factored MDP. Factored MDPs are specified using Boolean *state variables* that can be either true or false. For example, some of the state variables in the example in Section 1.1 might be `at-room1`, `at-room2`, `person1-rescued`, etc. The transition matrix of a factored MDP can be represented by a dynamic Bayesian network (DBN) [Boutilier *et al.*, 1995] (for a background on DBNs see [Russell and Norvig, 2003], section 15.5). Specifying MDPs in this way allows the creation of very large MDPs. By listing only 40 state variables, we can create an MDP with over one trillion states. These types of MDPs can be hard to work with because they are so large, but they also have more structure than an MDP where the transition matrix is explicitly written out.

One approach to creating hierarchical factored MDPs is to try to find a "reduced" MDP [Dean and Givan, 1997; Givan *et al.*, 2003]. The states of the "reduced" MDP are clusters of states of the original MDP that have identical dynamics. These clusters

can then themselves be treated as states of an MDP. If the reduced MDP is significantly smaller than the factored MDP, it can be solved by general MDP techniques. We discuss this idea in Section 2.5. [Dean *et al.*, 1998] and [Kim and Dean, 2002] extended this idea to MDPs with a large action space as well as a large state space.

Often, however, finding the reduced MDP is intractable or the reduced MDP does not have significantly fewer states than the original MDP. [Dean *et al.*, 1997] relax the idea of the reduced MDP, allowing clusters of states that have only ϵ-close dynamics. [Kim and Dean, 2001] take this one step farther, showing how to average the transition probabilities and rewards in a non-homogeneous cluster to create a hierarchical model. However, their algorithm is a costly one, requiring the computation of an optimal solution for the current model at every iteration.

Reduced MDPs are not the only hierarchical models of factored MDPs. Jonsson and Barto's VISA algorithm [Jonsson and Barto, 2006] extends the option framework to factored MDPs. They do a causal analysis on the transition DBN of a factored MDP to to define temporally-extended actions (options) that cause specific state variables to change in value. This algorithm requires that there be at least one instance of one-way causality in the state variable influence graph.

The Hi-MAT algorithm proposed by [Mehta *et al.*, 2008] discovers MAXQ hierarchies for factored MDPs. These hierarchies can be costly to find, however, so the real usefulness of them is in transfer learning where we hope that a hierarchy for one problem can help the algorithm learn a hierarchy for a similar problem.

In this thesis, we will present an algorithm, HDet, for both enumerated-state MDPs where the states are fully listed and for factored MDPs. This algorithm incorporates a fast clustering routine that can be repeated for every new problem, as well as a solver that returns an accurate, although possibly sub-optimal, solution.

1.3 Objectives

Consider again the problem of Section 1.1. In this problem, we faced a trade-off between computation time and accuracy. An optimal solution to the problem *did* exist, but finding it would have taken far too much time. Therefore, we settled for a less accurate, but more timely solution.

We also touched on the problem of creating a hierarchy. In the example, we arbitrarily divided the building into 20 sub-regions. Was this a good decision? Was there a more natural way to divide the building? A way that would work better with the solution method we used? There probably was. However, *finding* that better hierarchy would require computation time that we were previously using to solve the hierarchy. As discussed in Section 1.2, much work has focused on creating hierarchies that can be used for multiple problems [McGovern and Barto, 2001; Simsek *et al.*, 2005; Digney, 1996; Dean and Givan, 1997; Kim and Dean, 2001; Mehta *et al.*, 2008], but less has focused on clustering algorithms that must be efficient enough to be rerun for every problem. As well as a balance between computation time and accuracy, we also need a balance between the time it takes to create the hierarchical model and the time it takes to solve it.

Lastly, we considered the difference between solving a problem *bottom-up* and *top-down*. A bottom-up solution, the first type of solution we proposed in the example, computes all of the possible paths at the lower levels and then uses that information to make plans at the upper levels. This type of solution may be more accurate, but is also significantly more time-consuming. A top-down solution, however, creates a top-level plan first and then solves only the lower-level pieces that are necessary to implement this plan. This type of planning is fast, but can introduce inaccuracy because the information available while making the high-level plan is limited. The algorithm proposed by [Bakker *et al.*, 2005] is a top-down algorithm, but there are many conditions under which it produces a highly inaccurate solution.

To summarize, our goals in this thesis are:

- An *approximate* algorithm for creating and solving hierarchical MDPs. Finding an optimal solution for a large MDP can be intractable. In addition, in many cases an optimal solution is not necessary; we do not always need the absolute shortest path to a goal, just a path that gets there reasonably quickly.

- An efficient algorithm for *creating* hierarchical MDPs given a flat MDP. We want to be able to *create and solve* the hierarchical MDP in significantly less time than it would take to find an optimal solution for the flat MDP. This requires an efficient algorithm for creating the hierarchical MDP, as well as one for solving it.

- A strategy for solving the hierarchical MDPs *top-down* without too much reduction in accuracy. Having a top-level plan saves significant computation at the lower levels because only the transitions necessary to the top-level plan need be considered.

1.4 Outline of Approach

In this thesis, we present the algorithm HDet for creating and solving hierarchical MDPs. We will be discussing this algorithm in detail, but here we give an overview of the basic ideas.

We begin by designing our solver algorithm. The key to the algorithm is that we assume that, at the top levels of the hierarchy, transitions are *deterministic*. For example, consider the task of shopping for food. A person may not know how many tries it will take him to unlock his car, but he can be almost certain that he can drive from his home to the grocery store. The low-level actions (unlock the car) are non-deterministic, but the high-level actions (drive to the grocery store) are not. Therefore, in our solver, we consider non-determinism at only the bottom level.

Once we have designed the solver, we can decide what properties are necessary for our clustering algorithm. There are two main approaches to clustering MDPs: clustering states that are "near" to each other or clustering states that are "look" alike. In the first, states that can transition easily to each other are put together. For instance, in the shopping example, we could cluster together the state before the

car was unlocked and the state after the car was unlocked, because we know we can transition easily from a locked car to an unlocked car.

The second method is to cluster states for which actions have similar outcomes. In this case, we could cluster together the state where the person has bought one pound of sugar and is leaving the store and the state where the person has bought one pound of flour. Assuming the task is just to get home with the groceries, whether the grocery bag contains flour or sugar is immaterial, but it would be very difficult to transition between the states.

We will use the first method for clustering. We assume that upper-level transitions are deterministic, which means that we want transitions from one group of states to another to be highly probable. Therefore, whenever we take an action, it should land us either in the current cluster or in the cluster for which we are aiming. Thus we need clusters of states that transition easily among themselves.

Our clustering method is also designed to work with the solver to create an accurate solution. Since the solver works top-down, we need to be sure that it is possible to follow a top-level plan at the lower levels. Therefore, we create a clustering algorithm that assures that, if a state could reach a goal state under an optimal plan, it can reach the goal state under the top-down plan produced by the solver.

In Chapter 2, we give an overview of MDPs. We present two types of MDPs: enumerated-state MDPs and factored MDPs. Enumerated-state MDPs require the user to list every state of the MDP, while factored MDPs can be specified with an input logarithmic in the size of the MDP.

In Chapter 3, we discuss the form of a hierarchical MDP and give an overview of the clustering and solving algorithms. In Chapter 4, we present the complete algorithm for enumerated-state MDPs and results on several well-known problems. We discuss some of the surprising results and give a run-time analysis of the algorithms.

In Chapter 5, we extend these algorithms to factored MDPs. As we discussed, these MDPs are usually more difficult to solve because they can be very large. We explain how we modify our basic clustering and solving algorithms to work with the factored representation and present some preliminary results of our implementation. We conclude in Chapter 6 with a discussion of future work.

Chapter 2

Markov Decision Processes

This thesis considers two issues in planning: uncertainty and a large number of variables. To model uncertainty, we model the world as a discrete Markov decision process (MDP). In this model, we consider the world to consist of a set of states, among which we can transition using a set of available actions. However, we assume that the actions are not necessarily deterministic: in a given state, a given action might produce a different next state every time.

Modeling the world as an MDP makes two key assumptions:

- **Markov Assumption**: We assume that the world is *Markovian*, meaning that the current state depends only on the previous state and not on all history.
- **Fully Observable**: We also assume that the world is *fully observable*, meaning that at any time we know our state with complete certainty.

In this thesis we will also make two more assumptions:

- **Discreteness**: We assume the world can be modeled by a discrete set of states and actions.
- **Known Dynamics**: We assume that the agent is given the full MDP model and therefore knows the world dynamics ahead of time.

These assumptions, especially the fully observable and discrete assumptions, may not apply to all times and places, but there are a number of problems for which they are reasonable approximations. For example, if we have a robot interacting with a human, the human may give the robot enough information for the robot to know its own state completely even though the results of its actions are uncertain.

In this chapter, we give a basic background on MDPs. We formally define MDPs, explain the value iteration algorithm, and discuss different ways of describing MDPs.

2.1 Definitions

An MDP is formally defined as the tuple $M = \{S, A, T, R, G\}$ [Russell and Norvig, 2003]. Specifically it consists of

- S: A set of states $S = \{s_1, s_2, ..., s_N\}$.
- A: A set of actions $A = \{a_1, a_2, ..., a_M\}$ that determine transition probabilities among the states.
- T: A transition function $T : S \times A \times S \rightarrow \Re$ specifying the probability that a certain action will transition one state to another.
- R: A reward function $R : S \times A \rightarrow \Re$ giving the reward of taking any action in any particular state.
- G: A subset of the states $G = g_1, g_2, ..., g_P$. Goal states are absorbing so that $T(g_i, a, s) = \delta_{g_i,s}$ where $0 < i \leq P$ and $\delta_{g_i,s}$ is the Kronecker delta.

In this thesis, we concentrate on negative MDPs. They have the property that, for any goal state $g \in G$, $R(g, a) = 0$ for all actions a and that for any other state $s \notin G$, $R(s, a) < 0$. Since we have a set of goal states, we can therefore solve the problems under the undiscounted total reward criterion, making it a 'negative' MDP [Puterman, 1994], also sometimes referred to as a 'stochastic shortest path problem' [Bertsekas, 1995]. As we will show later, all MDPs can be transformed into an equivalent negative goal MDP so this is actually a general formulation.

The solution to an MDP is a *policy* π which specifies an action $\pi(s)$ for every state s. The *optimal policy* is the policy that specifies the action for each state that will maximize the future expected reward of that state. Formally, the optimal policy π^* satisfies [Russell and Norvig, 2003]

$$\pi^* = \arg\max_{\pi} E\left[\sum_{t=0}^{\infty} R(s_t, \pi(s_t))|\pi\right]. \tag{2.1}$$

Finding an optimal policy is a non-trivial problem. Since the model is non-deterministic, for each state the policy specifies not just a single state sequence, but many possible sequences. Specifically we can define a *value function* $V^\pi(s)$ that gives the future expected reward for a state s under policy π:

$$\begin{aligned} V^\pi(s) &= E\left[\sum_{t=0}^{\infty} R(s_t, \pi(s_t))|\pi, s_0 = s\right] \\ &= R(s, \pi(s)) + \sum_{s' \in S} T(s, \pi(s), s')V^\pi(s'). \end{aligned} \tag{2.2}$$

The optimal value function is then the one defined by

$$V^*(s) = \max_{a \in A}\left[R(s, a) + \sum_{s' \in S} T(s, a, s')V^*(s')\right]. \tag{2.3}$$

Equation 2.3 is known as the *Bellman equation*. An optimal policy is any policy π^* such that $V^{\pi^*} = V^*$. Note that there are $|S|$ equations of the form of equation 2.3,

Algorithm 1 ValueIteration(mdp, ϵ)

```
1: ∀s ∈ S, V(s) ← 0
2: δ ← ε + 1
3: while δ > ε do
4:    δ ← 0
5:    for all s in S_/G do
6:       v ← max_{a∈A} [R(s,a) + Σ_{s'∈S} T(s,a,s')V(s')]
7:       if |V(s) − v| > δ then
8:          δ ← |V(s) − v|
9:       end if
10:      V(s) ← v
11:   end for
12: end while
13: return V
```

giving us $|S|$ equations and $|S|$ unknowns (considering $V(s)$ to be an unknown for each $s \in S$). We would like to be able to solve these equations for $V(s)$, but they contain the non-linear "max" operator. Therefore, there is no straightforward analytical way of solving the system of equations. Instead, an optimal solution can be found using the *value iteration* algorithm.

2.2 Value Iteration

Value iteration ([Russell and Norvig, 2003], section 17.2) is an iterative method for finding an optimal value function for an MDP. With some slight book-keeping, this algorithm also leads to an optimal policy for the MDP.

The algorithm is initialized by setting the value of each state to an arbitrary value (in practice, usually 0). We then iterate through every non-goal state of the MDP updating the value function using a *Bellman update*:

$$V(s) \leftarrow \max_{a \in A} \left[R(s,a) + \sum_{s' \in S} T(s,a,s')V(s') \right]. \tag{2.4}$$

We repeat this process until we reach an equilibrium. Pseudo-code for the algorithm is shown in Algorithm 1.

It can be shown that value iteration converges to the optimal solution for a negative MDP provided that the action space of the MDP is finite and we begin with $V = 0$ ([Bertsekas, 1995], section 5.4). We give an outline of that proof here, beginning with some notational definitions.

Let V be a value function for a negative goal MDP $M = \{S, A, T, R, G\}$. Then,

for $s \in S$, we define:

$$U(V)(s) = \max_{a \in A} \left[R(s,a) + \sum_{s' \in S} T(s,a,s')V(s') \right] \tag{2.5}$$

$$V_0(s) = 0 \tag{2.6}$$

$$V_\infty(s) = \lim_{k \to \infty} U^k(V_0)(s). \tag{2.7}$$

Note that, on ending value iteration, we have something close to V_∞. Therefore, we must show that for negative goal MDPs

$$V_\infty = V^*. \tag{2.8}$$

Lemma A: The function U is monotonic so that if J and J' are functions such that $J(s) \geq J'(s)$ for all s, $U(J)(s) \geq U(J')(s)$ for all s.

Proof: Since $J(s) \geq J'(s)$ for all s, for all actions a we must have that

$$R(s,a) + \sum_{s' \in S} T(s,a,s')J(s') \geq R(s,a) + \sum_{s' \in S} T(s,a,s')J'(s') \tag{2.9}$$

and therefore that

$$\max_{a \in A} \left[R(s,a) + \sum_{s' \in S} T(s,a,s')J(s') \right] \geq \max_{a \in A} \left[R(s,a) + \sum_{s' \in S} T(s,a,s')J'(s') \right]. \tag{2.10}$$

Corollary A: For a negative MDP $M = \{S, A, T, R, G\}$, for all $s \in S$ and all k

$$U^k(V_0)(s) \geq V^*(s). \tag{2.11}$$

Proof: We proceed by induction.

Base Case: Since $R(s,a)$ is strictly negative for non-goal states (recall that $V^*(g) = V_0(g) = 0$ for $g \in G$) then for all $s \in S$

$$V_0(s) \geq V^*(s). \tag{2.12}$$

Induction Step: Assume that for some $k > 0$, $U^{k-1}(V_0)(s) \geq V^*(s)$. Then, by Lemma A, $U(U^{k-1}(V_0)(s)) \geq U(V^*(s))$. Now note that by the definitions of U and V^* given in equations 2.5 and 2.3 respectively, we have that $U(V^*) = V^*$ ([Bertsekas, 1995] proves this another way in chapter 5, proposition 8). Therefore, $U^k(V_0)(s) \geq V^*(s)$.

Lemma B: If value iteration converges, it converges to the optimal solution.

Proof: If value iteration converges, then we must have that

$$V_\infty(s) = U(V_\infty)(s) \tag{2.13}$$

since this is the criteria for convergence. Firstly note that by Lemma A and Corollary A,

$$V_0 \geq U(V_0)(s) \geq U^2(V_0)(s) \geq ... \geq U^k(V_0)(s) \geq ...V^*(s) \tag{2.14}$$

which, in the limit, gives that

$$\lim_{k\to\infty} U^k(V_0)(s) = V_\infty(s) \geq V^*(s). \tag{2.15}$$

However, since we assume value iteration converges, there is a policy π — the one we are using to evaluate V_∞ when we run value iteration — that yields V_∞. Therefore, by the definition of the optimal value function we must have

$$V^*(s) \geq V_\infty(s). \tag{2.16}$$

Thus

$$V^*(s) = V_\infty(s) \tag{2.17}$$

and if value iteration converges, it converges to the optimal solution.

Lemma C: If the action set A is finite for all $s \in S$ then value iteration converges.

Proof: We proceed by contradiction. Assume that for some state s^*

$$V_\infty(s^*) > U(V_\infty)(s^*). \tag{2.18}$$

Let a_k be the action that maximizes

$$U^{k+1}(V_0)(s^*) = \max_{a\in A}\left[R(s^*, a) + \sum_{s'\in S'} T(s^*, a, s')U^k(V_0)(s')\right] \tag{2.19}$$

for some k. Then, since $|A(s^*)|$ is finite, there must exist some $a^* \in A$ such that $a^* = a_k$ for all k in an infinite subset K of the positive integers. Therefore, for all $k \in K$

$$U^{k+1}(V_0)(s) = R(s^*, a^*) + \sum_{s'\in S} T(s^*, a^*, s')U^k(V_0)(s'). \tag{2.20}$$

Since K is infinite, we can take the limit as $k \to \infty$ giving us

$$\begin{aligned} V_\infty(s^*) &= R(s^*, a^*) + \sum_{s'\in S'} T(s^*, a^*, s')V_\infty(s'). \\ &= \max_{a'\in A}\left[R(s^*, a') + \sum_{s'\in S} T(s^*, a', s')V_\infty(s')\right] \end{aligned} \tag{2.21}$$

where equation 2.21 follows by the definition of a^*. Therefore, we have that $V_\infty(s^*) = U(V_\infty)(s^*)$ contradicting Equation 2.18 and, if the action set is finite, value iteration does converge.

Theorem: If the action set A of a negative goal MDP is finite for all states, then value iteration converges to the optimal solution.

Proof: This follows directly from Lemmas B and C.

[Bertsekas, 1995], Proposition 6, Chapter 7 also shows that, if every state can reach a goal state with probability ϵ in m steps, the rate of convergence of value iteration is linear in $(1-\epsilon)^{\frac{1}{m}}$. We can bound m above by $|S|$ giving that, if we require that value iteration algorithm reach an error of no more than δ, it should converge in

$$Time(iteration) = O(|S| \log_{1-\epsilon} \delta). \tag{2.22}$$

which is linear in the number of states. For every iteration, we may need to look at $|S|^2$ states so that value iteration scales cubically with the size of the state space

$$Time(VI) = O(|S|^3). \tag{2.23}$$

2.3 Discounted MDPs vs Goal MDPs

In this thesis we work with negative goal MDPs, meaning that we specify certain goal states and require that the reward for non-goal states be strictly negative. This describes a set of problems for which the agent has a specific goal it needs to obtain. However, there is another type of problem we may want to solve: namely the problem of "live long and prosper" where there is no set goal state, and we need to find a good policy for a world that goes on forever. In this world we do not necessarily restrict the reward function to be negative. For this type of MDP where there are no longer guaranteed to be absorbing states, however, the value function as defined in equation 2.2 can become positive or negative infinity for all states. Therefore we introduce the idea of "discounting", making states further in the future have less impact on the value of the current state. Specifically we define a discount factor $\gamma < 1$ and define the value function to be

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V(s'). \tag{2.24}$$

The Bellman equations are then

$$V^*(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V(s') \right] \tag{2.25}$$

and we can do value iteration on these equations as before. Note that equations 2.25 and 2.3 are identical if we set $\gamma = 1$.

However, although discounted MDPs appear to be very different from goal MDPs, we can show that we can turn any discounted MDP into a negative goal MDP. We follow the proof outlined in [Bertsekas and Tsitsiklis, 1996], pgs 39-40 although we take the notion of equivalence from [Bonet and Geffner, 2009].

Let V_M^π be the value function for an MDP M under policy π. We consider two MDPs R and M to be *equivalent* if they have the same set of non-goal states and if there are two constants, α and β, such that for every policy π, $V_R^\pi(s) = \alpha V_M^\pi(s) + \beta$. That is, R and M are equivalent if their value functions are related by a linear transformation. A transformation U is *equivalence-preserving* if $U[M]$ and M are equivalent for MDP M.

Notice that, since linear transformations preserve order, if R and M are equivalent they will have the same optimal policy. Therefore, we just need to show that every discounted MDP has an equivalent negative goal MDP.

Firstly, note that adding a constant to the reward function is an equivalence-preserving transformation. Let D be a discounted MDP with reward function $R(s,a)$ and let $D + C$ be the same MDP except with reward function $R(s,a) + C$. Then

$$\begin{aligned} & V_D^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s') V_D^\pi(s') \\ \Rightarrow V_D^\pi(s) + \frac{C}{1-\gamma} \quad & = R(s,\pi(s)) + \frac{C}{1-\gamma} + \gamma \sum_{s' \in S} T(s,\pi(s),s') V_D^\pi(s') \\ & = R(s,\pi(s)) + C + \gamma \sum_{s' \in S} T(s,\pi(s),s') \left(V_D^\pi(s') + \frac{C}{1-\gamma} \right) \end{aligned} \tag{2.26}$$

where the last step is valid because $\sum_{s' \in S} T(s,\pi(s),s') = 1$. Now note that if we define $W^\pi(s) = V_D^\pi(s) + \frac{C}{1-\gamma}$ then

$$W^\pi(s) = R(s,\pi(s)) + C + \gamma \sum_{s' \in S} T(s,\pi(s),s') W^\pi(s). \tag{2.27}$$

But equation 2.27 is the Bellman equation for $V_{D+C}^\pi(s)$ in $D + C$. Therefore we must have that $V_{D+C}^\pi(s) = V_D^\pi(s) + \frac{C}{1-\gamma}$ so D and $D + C$ are equivalent.

Now we show that eliminating the discount factor is in fact not only possible but an equivalence-preserving transformation. Consider a negative MDP M with discount factor γ. We transform it to an MDP M' in the following manner:

- We add to the set of states a single, absorbing goal state g with reward 0. Let S_M be the set of states of M and $S_{M'}$ be the set of states of M'.

- For each state s in S_M, we set $T_{M'}(s,a,s') = \gamma T_M(s,a,s')$ if $s' \in S_M$ and $T_{M'}(s,a,g) = 1 - \gamma$. Since g is absorbing $T_{M'}(g,a,s) = \delta_{g,s}$. Note that $T_{M'}$ is normalized.

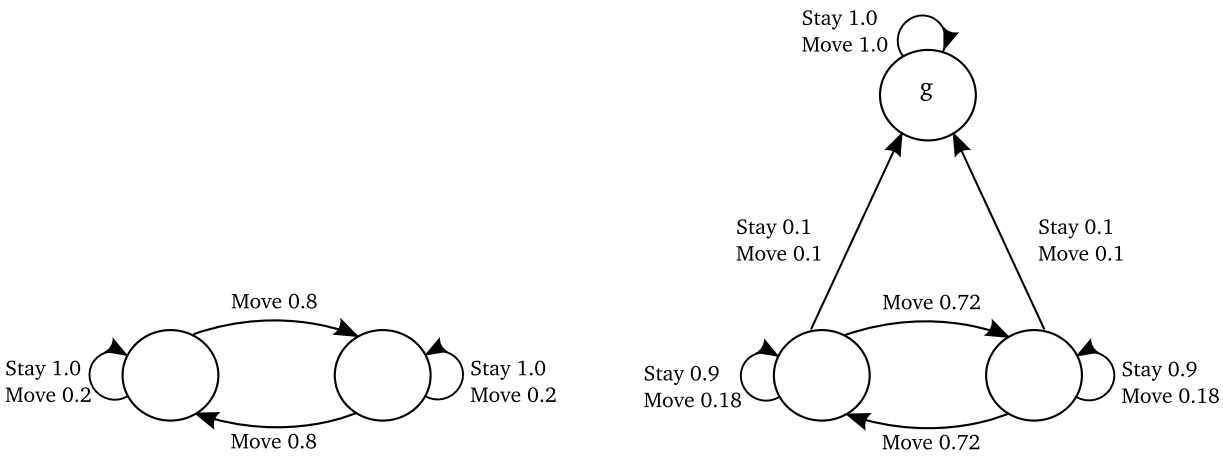

Figure 2-1: Converting from an MDP with a discount factor of 0.9 (left) to a goal MDP (right).

This transformation is shown in Figure 2-1.

Now consider the Bellman equation for $V^{\pi}_{M'}(s)$:

$$\begin{aligned} V^{\pi}_{M'}(s) &= R(s,\pi(s)) + \sum_{s' \in S_{M'}} T_{M'}(s,\pi(s),s') V^{\pi}_{M'}(s') \\ &= R(s,\pi(s)) + \gamma \sum_{s' \in S_M} T_M(s,\pi(s),s') V^{\pi}_{M'}(s') + T_{M'}(s,\pi(s),g) V^{\pi}_{M'}(g) \\ &= R(s,\pi(s)) + \gamma \sum_{s' \in S_M} T_M(s,\pi(s),s') V^{\pi}_{M'}(s') \qquad (2.28) \end{aligned}$$

where equation 2.28 follows because, since g is a goal state with reward 0, $V^{\pi}_{M'}(g) = 0$. Therefore, for states s in S_M, $V^{\pi}_{M'}(s) = V^{\pi}_{M}(s)$ and M and M' are equivalent.

Thus, we can transform a discounted MDP D into an equivalent negative goal MDP M by

1. Subtracting a constant C from the reward function where C is a constant large enough that $R_D(s,a) - C < 0$ for all $s \in S_D$ and $a \in A_D$. Let this MDP be $D - C$.

2. Eliminating the discount factor from $D - C$ by adding an absorbing goal state as discussed above.

Thus the class of MDPs we consider is actually the most general class of discrete MDPs. Note that there are also a set of MDP problems with continuous state and/or action spaces. In order for our algorithms to work for those problems, the state and action spaces would need to be discretized. However, our methods apply to any discrete MDP.

2.4 Enumerated-State and Factored MDPs

There are many ways of defining MDPs. In this thesis, we focus on two: enumerated-state MDPs and factored MDPs.

Enumerated-State MDPs are defined by listing the full action and state spaces. For example, a navigation problem might be described this way by listing all of the possible locations in the problem. The transition function for such an MDP is usually specified as a sparse matrix. For this type of definition, algorithms can be polynomial in the number of states and actions in the MDP.

Factored MDPs are another way of specifying MDPs. A factored MDP consists of n state variables, each of which can be either true or false. Although there are solvers that work with both factored state and action spaces [Dean *et al.*, 1998; Kim and Dean, 2002], in this thesis we still specify the actions by listing them. The transition function can be specified as a dynamic Bayesian network [Boutilier *et al.*, 1995], although we never use an explicit representation of it.

Factored MDPs are useful because MDPs with huge state spaces can be represented very compactly. The hope is that we can take advantage of this compact representation to solve the MDP efficiently. Since, in this case, the number of states in the MDP is exponential in the input size, we require algorithms that use time and space polynomial in the number of state variables (*logarithmic* in the number of states).

2.5 Minimized-State MDPs

It is clear from Section 2.2 that it is possible to solve MDPs optimally using value iteration. However, value iteration is prohibitively slow on large MDPs. Therefore, we present a method for "shrinking" the state space of MDPs.

Minimized-state MDPs were proposed by [Dean and Givan, 1997]. A minimized-state MDP has a state space that is a partition of the state space of the original MDP and basically acts like an MDP itself, but with fewer states. Here we give an overview of converting an MDP to a minimized-state MDP, beginning with a few definitions.

- *Stable*: Consider a partition P of the state space S. We call a block C of this partition *P stable with respect to a block B of P and action a* if and only if every state in C has the same probability of being carried into B by action a. The block C is *stable* if it is stable with respect to every block in P and every action.

- *Homogeneous*: A partition P is *homogeneous* if every block is stable.

[Dean and Givan, 1997] show that for a given starting partition of the state space, P, there exists a unique coarsest homogeneous partition, which we will refer to as $MM(P)$. We refer to the blocks of $MM(P)$ as the "minimized states of P". The

starting partition of interest is the partition P_R, which partitions states on their reward values. Specifically, in any block of P_R, for any action a, all states in that block have the same reward value. Since $MM(P_R)$ is a refinement of P_R, we know that this property will hold in $MM(P_R)$. Therefore, $MM(P_R)$ is a partition of the state space such that we can define reward and transition functions for the partition, considering each block of the partition to be a "state". Thus $MM(P_R)$ acts itself as an MDP and, moreover, the optimal solution to $MM(P_R)$ is the optimal solution to the original MDP. We refer to $MM(P_R)$ as a "reduced MDP".

Starting with any partition P, we can find $MM(P)$ by iteratively identifying any block $B \in P$ that is not stable with respect to some block $C \in P$ and action a and splitting it into sub-blocks that *are* stable with respect to C and action a. We refer to the action of splitting block B in partition P to make it stable with respect to C and a by the operator $SPLIT(B, C, a, P)$. When no unstable blocks are left, the algorithm terminates. Moreover $MM(P)$ is unique with respect to P; the order in which the splits are done has no effect. The number of splits is upper bounded by twice the number of blocks in $MM(P)$.

The hope is that the number of blocks in $MM(P_R)$ is significantly less than the number of states in the original MDP, allowing us to run value iteration in a reasonable amount of time. However, there is no guarantee that the number of blocks in $MM(P_R)$ is less than the number of blocks in the original MDP (consider, for example, an MDP where every state has a different reward value). In addition, the $SPLIT$ operation is non-trivial and may result in representations of the partition that expand exponentially in size. Therefore, although reduced MDPs can be used to solve large MDPs optimally in some cases, they are not a general solution.

Having defined MDPs and discussed basic methods for solving them, we will define hierarchical MDPs and present an overview of our algorithm HDet in the next chapter.

Chapter 3

Creating and Solving Hierarchical Markov Decision Processes

In this chapter we discuss various methods for creating and solving hierarchical Markov decision processes (HMDPs). We begin by defining our hierarchical model, then describe our process for solving it and, lastly, discuss how to create it.

3.1 Hierarchical Markov Decision Processes

Although MDPs can be solved optimally, for large problems value iteration takes an impractical amount of time. However, for many problems, the optimal solution is not necessary. Thus we attempt to find an approximate solution quickly by utilizing a hierarchy.

We construct a hierarchical MDP (HMDP) from an MDP $M = \{S, A, T, R, G\}$ represented in one of the ways described in Section 2.4. We will go into more detail about this construction later in the thesis (see sections 3.3, 4.3, 5.2), but here we explain the form our HMDP takes.

An HMDP of depth L is a depth-L tree. The leaves of the tree, at level 0, are the states of the original MDP. Internal nodes of the tree represent (possibly overlapping) sets of nodes at the lower levels. A diagram of a two level HMDP is shown in Figure 3-1.

In talking about an MDP, we will use the following vocabulary:

- **Primitive state**: A state of the original, flat MDP, the leaves of the depth-L tree.
- **Primitive action**: An action of the flat MDP.
- **Macro-state**: A level 1 macro-state is a cluster of primitive states. A level $l > 1$ macro-state is a cluster of level $l - 1$ macro-states. Each non-leaf node of our tree is a macro-state.

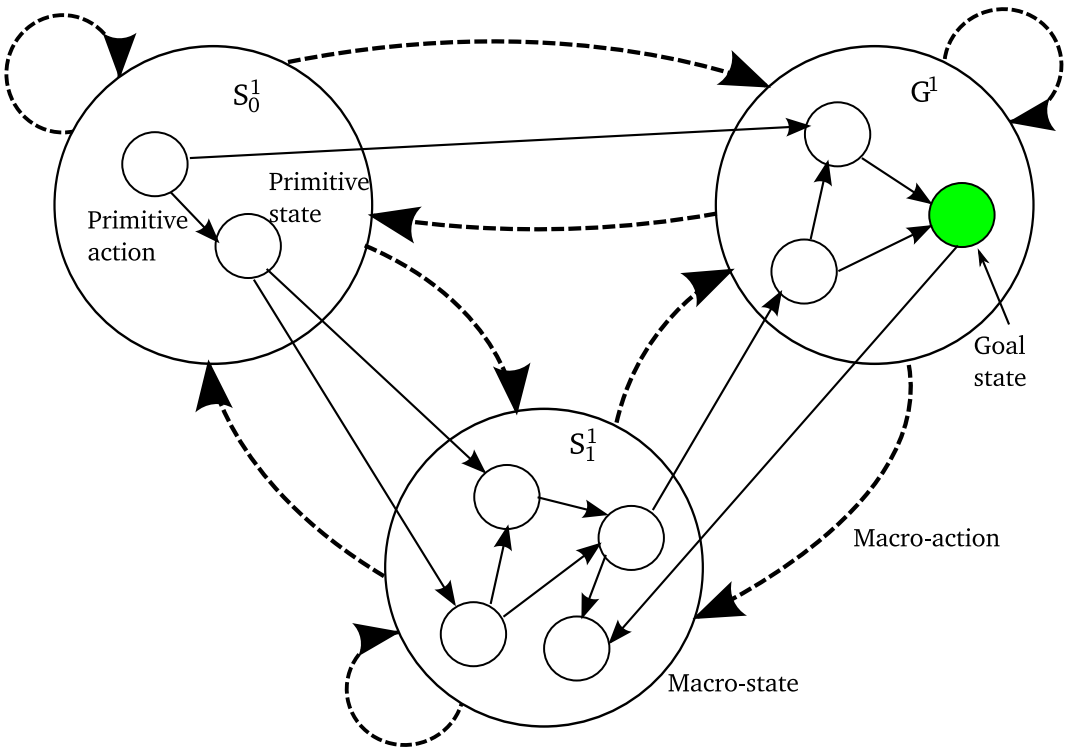

Figure 3-1: A two-level HMDP.

- **Sub-state**: A sub-state of a macro-state is any direct descendent of the macro-state. Sub-states may be either primitive states or macro-states. We use the notation $j' \in j$ to indicate that j' is a sub-state of j.

- **Goal macro-state**: A goal macro-state is a macro-state that has a primitive goal state $g \in G$ as a descendent.

- **Macro-action**: Macro-actions represent an attempt to transition between macro-states. For any non-leaf level of the tree with n nodes, we have n^2 macro-actions, one for each pair of macro-states.

One way we may try to represent an HMDP is to represent each level of the HMDP as an MDP in its own right. At level 0, the leaf level, this is easy: the leaf level is the original, flat MDP. Now consider level $l > 0$ and try to construct an MDP M^l. At this level, we can define the "states" of M^l to be the macro-states at level l, the "actions" to be the macro-actions, and the "goal states" to be the goal macro-states. The transition and reward functions, however, are harder to define. In addition, as we have discussed, as we move away from dealing with primitive states, the world tends to become less uncertain. Therefore, in this thesis, we will not attempt to model non-determinism at the upper levels. Rather we will assume that all macro-actions are deterministic and assign each a transition cost based on the underlying transition probabilities.

Therefore, each level $l > 0$ of the HMDP can be represented as $M^l = \{S^l, A^l, C^l, G^l\}$ where S^l is a set of macro-states, A^l is a set of macro-actions representing transitions between the macro-states, $C^l : S^l \times S^l \rightarrow \Re$ is a cost function, and G^l is the set of goal macro-states. At the primitive level $M^0 = M = \{S, A, T, R, G\}$ is the original expression for the flat MDP.

3.2 Solving HMDPs

The solver takes as input a hierarchical MDP and a minimum transition probability ϵ and computes a *hierarchical policy* π. The policy π specifies, for each macro-state at level $l > 0$, a policy, π^l, which prescribes behavior for each level $l-1$ sub-state. At levels $l > 1$, the policy maps each level $l-1$ macro-state i to some other level $l-1$ macro-state j, signifying that when the system is in a primitive state contained in the macro-state i it should attempt to move to some primitive state in the macro-state j. At level 1 macro-states, the policy is a standard MDP policy, mapping the primitive states that are children of that macro-state to primitive actions. The solution to the original MDP can be found in the union of the policies of the level 1 macro-states by methods that will be explained more fully in Sections 4.2 and 5.4.

Before beginning our discussion of the solver, we give a few definitions.

3.2.1 Definitions

- **Adjacent**: A primitive state j is *adjacent* to primitive state i if there exists an action a such that $T(i, a, j) > \epsilon$.
- **Reachable**: A primitive state j is *reachable* from primitive state i if j is adjacent to i, or if j is adjacent to some state k that is reachable from state i.
- **Stranded**: A sub-state $j' \in j$ is *stranded under policy* π if it could get to a goal state under an optimal policy but cannot get to the goal state under the hierarchical policy π. An example of a stranded sub-state is shown in Figure 3-2.

The solver works in two passes: an upward pass and a downward pass.

3.2.2 Upward Pass

As we mentioned in Section 3.1, we constrain the upper levels of the hierarchy to have deterministic transitions. In the upward pass, the solver computes the *cost matrix* for these transitions. This matrix is composed of real values and, for each level, $l < L$, specifies an estimated cost of moving between macro-states i and j at level l. The aim is to get a rough estimate of the expected cost incurred to transition between every pair of macro-states at each level.

First, we consider costs at level 0. Because our actions are stochastic, we could compute the optimal cost exactly by constructing n MDPs, one in which each of the

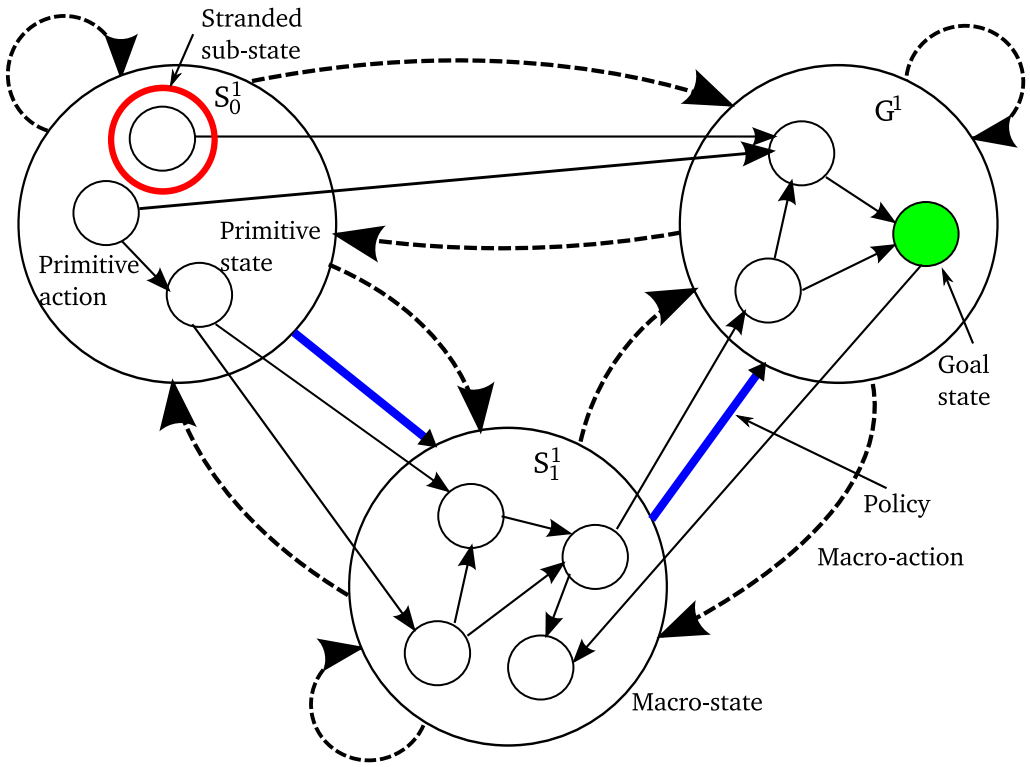

Figure 3-2: A stranded sub-state in a two-level hierarchy. The hierarchical policy is shown by the thick blue lines. The sub-state circled in red is "stranded" because, although it can actually reach the goal state, it cannot do so following the hierarchical policy.

primitive states is a goal, and solving for the optimal value function. The resulting value functions would encode the expected costs to move between each pair of primitive states. Solving each of these MDPs would take as long as solving the original problem, so this is clearly not a reasonable approach.

Instead, we make a quick approximation that makes the assumption that any action a taken in primitive state i with the goal of landing in primitive state j does in fact make a transition to j with probability $T(i,a,j)$; but that with the remaining probability mass, it stays in state i. If we imagine executing such an action a repeatedly, then in expectation the number of steps it will take to move to j is

$$E[steps] = \sum_{k=0}^{\infty}(1 - T(i,a,j))^k = \frac{1}{T(i,a,j)}. \tag{3.1}$$

Each of these steps costs $-R(i,a)$. Note that we are changing from (negative) rewards to (positive) costs (because later we will be solving for minimum-cost paths), which accounts for the negation. Finally, we can select whichever action would minimize

this cost, in expectation. So, our quick cost estimate is

$$C^0(i,j) = \min_{a \in A} \frac{-R(i,a)}{T(i,a,j)}. \tag{3.2}$$

If there is no action a for which $T(i,a,j) > \epsilon$, then j is not adjacent to i, and $C^0(i,j) = \infty$. Note that this does not mean that j cannot be reached from i; just that it cannot be reached in one step.

Now, for levels $l > 0$, we compute $C^l(i,j)$ for increasing values of l, by solving a deterministic shortest-path problem for each pair of level-l macro-states i and j. Exactly how we solve this problem depends on the structure of the MDP and will be discussed in sections 4.1.1 and 5.3.1.

3.2.3 Downward Pass

In the downward pass, the solver actually creates the policy. The solver starts at the highest level, L, and works down to level 1. The process is the same for all levels $l > 1$, and different for level 1.

The policy in macro-state i at any level $l > 1$ is a mapping π^l from each level $l-1$ macro-state $i' \in i$ to an adjacent level $l-1$ macro-state (which may or may not be in i).

When macro-state i is a goal at level l (that is, $i \in G^l$), then $\pi^{l+1}(i) = i$; that is, if we are in a state within macro-state i, we should attempt to stay within i and act to reach a primitive goal state. The single macro-state at the top level L will be treated this way, as well. Otherwise, when $\pi^{l+1}(i) = j \neq i$, then from macro-state i we must attempt to move to adjacent macro-state j.

To derive the policy for macro-state i at level l, we must specify a special set of level $l-1$ goal macro-states, G^i. This set is defined differently, depending on whether $\pi^{l+1}(i) = i$ or not. If $\pi^{l+1}(i) = i$, the objective in solving for a policy for macro-state i is to make transitions *within* macro-state i to goal states that are contained within it. So, in this case, the set of goal macro-states G^i, will simply be the level $l-1$ macro-states in i that have a primitive goal state as a descendant. When $\pi^{l+1}(i) \neq i$, the objective in solving for a policy for macro-state i will be to make transitions into macro-state $j^* = \pi^{l+1}(i)$. So, in this case, the set of goal macro-states G^i, will be all level $l-1$ macro-states $j' \in j^*$.

Now, in either case, we can solve a shortest-paths problem, over the macro-states in $i \cup G^i$. We define D^i to be the distance from each of these macro-states to the closest goal state; so $D^i(g) = 0$ for $g \in G^i$. How we compute $D^i(i')$ depends on the representation of the MDP, but, assuming for the moment that we can compute it, the policy is defined over all $i' \in i$ as:

$$\pi^l(i') = \begin{cases} i' & \text{if } i' \in G^{i'} \\ \arg\min_{j'} C^{l-1}(i',j') + D^i(j') & \text{otherwise;} \end{cases} \tag{3.3}$$

that is, it is the next macro-state on the shortest path to a local goal (based on the estimated costs C^{l-1}). Then, we can recursively solve for policies π^{l-1} at each

macro-state i'.

Finally, we reach a macro-state i at level 1. The goal set of primitive states, G^i, is determined as described above, depending on whether $\pi^2(i) = i$ or not. But at level 1, rather than using the expected costs to solve a shortest-paths problem, we wish to take the actual transition probabilities over the primitive states into account. In order to do this, we have to construct an MDP that represents the problem of moving from macro-state i to macro-state $\pi^2(i)$. Most of the transition probabilities and reward values have already been defined in the primitive MDP. We treat all local goal states in G^i as zero-reward absorbing states. In addition, we have to model the case in which the system would make a transition to a state that is neither a state in G^i nor a state in macro-state i itself. For the purposes of deriving a policy for this macro-state, we will represent all of these other states with a single special *out* state. Its transition probabilities are defined, for each primitive state $i' \in i$, and primitive action a, as

$$T(i', a, out) = 1 - \sum_{j' \in i \cup G^i} T(i', a, j'). \tag{3.4}$$

The reward associated with the *out* state is taken to be the negation of the weighted average of estimated distances to the goal, at level $L-1$, of the macro-states associated with the primitive states that comprise *out*. An additional "non-compliance" penalty (NCP) is also charged since our estimates of the distance-to-goal at level $L-1$ contain less information than we have about the primitive states in i and G^i. Without this penalty, an oscillation can occur where a primitive state in $i' \in i$ opts for the out state with an action that lands it with high probability in primitive state j while j also opts of the out state with an action that lands it, with high probability, back in i'. Solving this MDP will yield a value function and policy defined on all $i' \in i$.

When the downward traversal over the entire HMDP tree has been completed, we will have a policy π^1 defined on all primitive states s from the original MDP; we can use that policy to give us an approximate solution to the HMDP.

3.3 Creating HMDPs

The other point we need to address is how we actually create the hierarchical model of the MDP. Specifically, we discuss how we cluster primitive and macro- states to create the macro-states of the HMDP.

We use a clustering technique that guarantees that the hierarchical policy will not "strand" sub-states. Consider, for example, the situation shown in Figure 3-2 where the hierarchical policy requires that all sub-states of macro-state S_0^1 go through sub-states of S_1^1. One of the sub-states of S_0^1, however, is disconnected and cannot reach S_1^1. This sub-state is a "stranded" sub-state — one for which the solver would therefore conclude that there was no possible policy.

In essence, we want a clustering algorithm that guarantees that, if a state can reach a goal state under the optimal policy, it will also be able to reach a goal state under the solver's hierarchical policy. We do this by adopting the following principle:

Clustering Principle: There must exist some hierarchical policy on the output of the clustering that strands no sub-states.

The exact clustering algorithms we use to accomplish this principle depends on the representation of the MDP and are discussed in Sections 4.3 and 5.2.

3.4 Summary

In the next two chapters, we will expand this algorithm into an algorithm for enumerated-state MDPs and an algorithm for factored MDPs. For these algorithms to work with the outline we have given, we need to specify:

1. A method for solving for the cost matrix C^l at levels $l > 0$. We do this for enumerated-state MDPs in Section 4.1.1 and for factored MDPs in Section 5.3.1.

2. A method for solving the shortest path problems to find D^i, the distance from each macro-state to the closest goal macro-state. We do this for enumerated-state MDPs in Section 4.1.2 and for factored MDPs in Section 5.3.2.

3. A method for turning policy π^1 into a policy for the flat MDP. We do this for enumerated-state MDPs in Section 4.2 and for factored MDPs in Section 5.4.

4. The clustering algorithm. We do this for enumerated-state MDPs in Section 4.3 and for factored MDPs in Section 5.2.

Chapter 4

Enumerated-State MDPs: Algorithm and Results

In this chapter, we discuss the algorithms and results for enumerated-state MDPs. In Sections 4.1 and 4.3, we follow the framework outlined in Sections 3.2 and 3.3 respectively, but expand it to explain the algorithms we implemented. We also give a run-time analysis.

In Section 4.6, we do an empirical evaluation of the performance of our algorithm, HDet, by comparing it to other algorithms in the field. Because, for the small domains we are using, it is often possible, if slow, to run value iteration, in many cases we are able to compare to the optimal results. In addition, we discuss the possibility of using other, well-known clustering algorithms and show that those algorithms all result in stranded sub-states.

In Section 4.7 we give an analysis of the results. We focus our discussion on three issues: the trade-off between computation time and accuracy, the relationship between clustering time and solving time, and the number of levels in the hierarchy.

4.1 Solver

Recall from Section 3.2 that the solver works in two passes, an upward pass and a downward pass. In that section, however, we were unable to specify exactly how we do each of the necessary computations because we had not committed to the representation of the MDP. Here, we go into detail about the computation of the cost matrix and the hierarchical policy.

4.1.1 Upward Pass: Calculating the Cost Matrix

In the upward pass, we are attempting to construct a cost matrix giving the approximate cost of transitioning between macro-states. To do this, we use the assumption discussed in Section 3.2.2, giving the zero level costs shown in equation 3.2:

$$C^0(i,j) = \min_{a \in A} \frac{-R(i,a)}{T(i,a,j)}.$$

In Section 3.2.2 we did not specify how we create the cost function C^l for $l > 0$. For the enumerated-state MDPs, we compute $C^l(i, j)$ for increasing values of l by solving a deterministic shortest-path problem for each pair of level-l macro-states i and j. For each pair, i and j, we carry out the following procedure:

- For each level $l - 1$ macro-state $i' \in i$, initialize

$$d(i') = \min_{j' \in j} C^{l-1}(i', j') \tag{4.1}$$

  This is the one-step distance from each macro-state in i to its closest macro-state in j.

- Now run Dijkstra's algorithm to compute shortest-path distances for all macro-states $i' \in i$. This gives us an expected shortest-path distance, $d(i')$, from each macro-state i' in i to its closest macro-state in j.

- Return the average of the $d(i')$ over all $i' \in i$, as a measure of average expected distance from macro-state i to macro-state j.

Remember that equation 3.2 gives us the function C^0, allowing us to initialize this procedure.

At this point, we have pairwise costs between all macro-states at every level. These costs will be infinite for non-adjacent (ordered) pairs of macro-states. This is the complete cost matrix.

4.1.2 Downward Pass: Calculating Distance to Goal

In the downward pass we use the cost matrix to give us a hierarchical policy. This pass has been almost completely laid out in Section 3.2.3.

Recall that at each level $l > 1$, for each macro-state i at that level, we specify a goal macro-state G^i. At the next level down, we must solve the problem over the sub-states of $i \cup G^i$. To do this, we must construct for all $i' \in i$, the distance $D^i(i')$ from i' to the closest sub-state of G^i. In this case, constructing $D^i(i')$ is simple. We use the cost matrix found in the upward pass and run Dijkstra's algorithm from each goal macro-state. The value $D^i(i')$ is then the shortest path from i' to any goal macro-state. Following equation 3.3 gives us the policy for levels $l > 1$.

At level 1, as discussed in Section 3.2.3, we create not another shortest-paths problem, but a small sub-MDP problem. Because our representation of the MDP allows us to operate in time polynomial in the number of primitive states, we can use the original transition and reward matrices to almost completely define this sub-MDP problem. As explained in 3.2.3, however, the problem is not fully defined because, in some primitive states, we may have a non-zero probability of transitioning *out* of the sub-MDP. In this case, we use the strategy of an *out* state as discussed in 3.2.3.

4.2 Interpreting the Hierarchical Policy

For enumerated-state MDPs, our clustering algorithm (to be discussed in Section 4.3) will result in an HMDP with the property that, at each level in the tree, the macro-states represents *non-overlapping* sets of nodes at the lower levels. Therefore, each primitive state of the HMDP belongs to exactly one macro-state at each level and thus, the policy for the MDP is simply the union of the policies at level 1.

4.3 Clustering Algorithm

In this section we describe the clustering algorithm for the enumerated-state representation. Although many clustering algorithms exist for enumerated-state MDPs, as we will show in Section 4.6.2, these algorithms do not guarantee that we will not strand sub-states. We describe an agglomerative algorithm that adheres to the principle described in Section 3.3. We begin with a few more definitions.

4.3.1 Definitions

- **Adjacent**: Recall from Section 3.2.1 that we defined adjacency for primitive states. Here we extend that definition to macro-states. At level l, a macro-state i is *adjacent* to macro-state j if there exist level $l-1$ sub-states $i' \in i$ and $j' \in j$ such that i' is adjacent to j'.

- **Reachable**: Note that the definition of reachable given in Section 3.2.1 extends naturally to macro-states now that we have defined adjacency for them. A macro-state j is *reachable* from another macro-state i if j is adjacent to i, or if j is adjacent to some macro-state k that is reachable from macro-state i.

- **Exit State**: Sub-state i in macro-state c at level l is an *exit state* of c if there exists a level $l-1$ primitive or macro- state $j \notin c$ such that j is adjacent to i.

- **E-connected**: Macro-state c is *e-connected* if for all exit states $j \in c$, and for all sub-states $i \in c$ such that $i \neq j$, j is reachable from i. That is, if all sub-states in the macro-state can possibly transition to any of the exit states. Note that e-connectedness guarantees that all policies over the macro-states can be executed successfully; every sub-state in the macro-state can reach all the adjacent macro-states. Therefore, if our macro-states are e-connected they will certainly fulfill our clustering principle. A clustering has the property of being e-connected if all of the macro-states defined by the clustering are e-connected.

4.3.2 Basic Algorithm

Our goal is to create e-connected macro-states. The clustering algorithm is randomized, greedy, and heuristic. It takes as input an MDP and S_{max}, the maximum desired number of primitive states per macro-state, and C_{min}, the minimum number of macro-states. As we will see in Sections 4.4 and 4.7, both computation time and accuracy

are dependent on S_{max} and C_{min}. We describe how we create level 1 macro-states from the primitive states; the extension to creating hierarchies with more levels is clear.

The clustering algorithm operates by trying to find cycles of nodes within the transition graph and putting them in the same macro-state. We will show that it maintains the property of having an e-connected clustering at all times.

The initial clustering, C, puts each primitive state into a separate macro-state. This clustering is e-connected in a degenerate sense.

We start pre-processing by marking all macro-states containing primitive absorbing states (goal or not) as being *finished* and we will not try to grow them further. Until it is no longer possible, we find a primitive state s' and a macro-state c such that either

1. s' is adjacent to some primitive state $s \in c$ and has no non-self transitions to a primitive state outside c or

2. s' is adjacent to some primitive state $s \in c$ and there exists a primitive state $s'' \in c$ that is adjacent to s'

and merge s' into c.

Option 1 preserves e-connectedness, because it introduces no new exit states to c, and we know that s' can reach all exit states because it can reach s and s was already able to reach all exit states.

Option 2 adds a new exit state to c, s'. It preserves e-connectedness because: (1) s' can reach all other exit states because it can reach s, which was already able to reach all other exit states; and (2) all other primitive states can reach s' because they were already able to reach s'' since s'' was an exit state.

In some sense, we could see the preprocessing phase as finding cycles of length 2 and putting them into macro-states. Now we will work on finding larger cycles. Until we do not yet have fewer than C_{min} macro-states or we cannot form another macro-state without creating a macro-state larger than S_{max}, we:

1. Choose a random starting macro-state.

2. Find a cycle R of adjacent macro-states [Johnson, 1975] starting with that macro-state.

3. If the sum of all of the sub-states of the macro-states in R is smaller than S_{max}, merge all of their sub-states into a single macro-state.

4. Recompute the adjacency relationships.

Merging the macro-states around an entire cycle preserves e-connectedness because, at the macro level, it is possible to move from any macro-state to any other macro-state via the cycle, and, because the original macro-states were e-connected, the entire aggregate macro-state will be e-connected.

4.3.3 Relaxed Algorithm

The above clustering method fulfills our principle, but it is not practical for domains that have no cycles or only very long cycles. Consider, for example, a domain in which there is a resource that must be used up to reach the goal: there is no way of "un-using" the resource so this domain will have very few cycles. E-connectedness is also stricter than is necessary; we do not need to be able to execute *every* possible hierarchical policy, just one that allows all primitive states that can reach a primitive goal state to do so.

Therefore, in order to handle such domains effectively, we can relax our requirement for connectedness. We define a property on a whole clustering (as opposed to individual macro-states) that will still be sufficient for the correctness of the overall algorithm.

A clustering C is *eg-connected* if

- There exists a policy $\pi : C \rightarrow C$ such that for all $c \in C$ where c has as a descendent at least one primitive state that can reach a primitive goal state, there exists a goal macro-state $g \in C$ such that $\pi(\pi(\ldots(\pi(c)))) = g$. That is, that the policy π can reach a goal macro-state from every macro-state containing a primitive state that could reach a primitive goal state.

- The clustering is π-connected. That is, for every $c \in C$ that has as a descendent a primitive state that can reach a primitive goal state, for every $i \in c$, there exists $j \in \pi(c)$ such that j is reachable from i.

This is a weaker condition than requiring that all macro-states be e-connected, because it does not require that all policies over the macro-states be executable, but just that there is some high-level policy that can reach a primitive goal state that can be executed from every primitive state within the macro-state.

We can extend the clustering algorithm to be more aggressive in its clustering; this change guarantees that the resulting clustering is eg-connected, but not necessarily e-connected. To do so, in the beginning we cluster all primitive goal states together into the same macro-state, which we will refer to as the *goal macro-state*, and each non-goal primitive state into its own macro-state. This clustering is eg-connected in a degenerate sense.

The pre-processing proceeds as in Section 4.3.2, except that we do not allow any new primitive states to be added to the goal macro-state. Before we begin the cycle-finding part of the phase, however, we temporarily add connections from the goal macro-state back to every other primitive state, and recompute the adjacency relationships among the macro-states from preprocessing. Now, when we find cycles, we may find one that goes through the goal macro-state, g. This requires us to be somewhat careful in our aggregation of the macro-states in the cycle, but it also allows us to be more precise in the size of the macro-states. For the method given in Section 4.3.2, when we found a cycle of macro-states we had to merge *all* of the macro-states in the cycle. By requiring only eg-connectedness however, whenever we generate a cycle that includes the goal macro-state, we can be more careful about

which macro-states we merge. Specifically consider the "cycle" $s_1, s_2, ..., s_p, g$. For any $i < p$, we can choose to merge s_i with as s_{i+1} because we know there is a path from this merged macro-state to the goal macro-state. In doing this, however, we know only that s_i can transition to s_{i+1}; it is possible that the primitive states of s_{i+1} *cannot* reach the primitive states of s_i. Therefore, after aggregating s_i and s_{i+1} to form macro-state A, we must mark A as adjacent to only those macro-states to which s_{i+1} was adjacent. This allows us to merge together as many or as few macro-states of the cycle as is need to keep the macro-state size below S_{max}.

The clustering algorithm with this step added yields an eg-connected clustering. This clustering guarantees that there exists an upper-level policy on its output with which every sub-state can comply and still reach the goal macro-state.

We have, in essence, just described the procedure for creating a 2-level hierarchy. To create a hierarchy with more levels, we simply re-run the algorithm on the macro-states we have just created. Because clustering takes a binary view of transitions, either a state can transition to another state or it cannot, we only need to specify an adjacency matrix for the macro-states. A more refined version of the clustering that takes into account the magnitude of the transition probabilities is left for future work.

4.4 Running Time Analysis

The running time of the MDP solver is the sum of the running times of many shortest-path algorithms plus the value iteration algorithms. Not surprisingly, this is dominated either by the time of the shortest path algorithm and value iteration in the largest macro-state or by the time of running the shortest path algorithm at level $L - 1$. Assuming a single-source shortest path algorithm that runs in $O(N^2)$ time where N is the number of nodes, therefore, the time bound on the solver for the upper levels is $O(S_{max}^2) + O(|S^{L-1}|^2)$ where S_{max} is the size of the largest macro-state.

We showed in Section 2.2, equation 2.23 that the run-time of value iteration is worst-case $O(|S|^3)$. However, note that in domains with short path lengths to goal states, value iteration can scale quadratically with the size of the state space (this will be the case for many of our domains). The time of the solver alone is then

$$Time(Solve) = O(S_{max}^2) + O(|S^{L-1}|^2) + O(S_{max}^3). \tag{4.2}$$

It is also necessary to take into account the time bound of the clustering algorithm. In the preprocessing step, for each macro-state formed, it takes a maximum of S_{max}^2 since each time we add a primitive state to the macro-state we may iterate through each of the primitive states already in the macro-state.

In the second step, cycles in the cluster graph are identified (note that there will be cycles because in this computation the goal macro-state is attached to every other macro-state). If most primitive states can reach a primitive goal state, we show that this can amortize to $O(N_C)$ where N_C is the number of macro-states in the graph after the preprocessing step. Let a fraction p of the primitive states be able to

Figure 4-1: Map of the grid world. The walls are shown in black and the goals in red. There are 1040 total primitive states, 800 of which are not walls.

reach a primitive goal state. Because we keep track of which macro-states we have explored, we will explore a maximum of $(1-p)N_C$ macro-states incorrectly for each correct macro-state, giving a running time of $p(1-p)N_C^2$. Therefore if only a small fraction of the primitive states cannot reach the goal, this computation is linear in N_C. Otherwise, the computation is order $O(N_C + E)$, which is upper bounded by $E = O(N_C^2)$. Therefore, at worst case (a dense MDP with few two-cycles and many nodes that cannot reach the goal), this clustering algorithm can be $O(|S|^2)$, but in many problems it should be $O(|S|)$ giving a clustering time of

$$Time(Cluster) = O(S_{max}^2) \tag{4.3}$$

Therefore, the running time of the full algorithm on an MDP with only a few primitive states that cannot reach the goal is

$$Time(HDet) = O(S_{max}^2) + O(|S^{L-1}|^2) + O(S_{max}^3). \tag{4.4}$$

4.5 Example

Before giving our results, we give an example of how the clustering and solving process work on a basic, easy-to-visualize domain: a grid world.

A grid world is a two-dimensional, discrete world in the form of a grid. The agent occupies one square of the grid and it has four actions: up, down, left, and right, allowing it to move to the adjacent squares. Squares can either be free so that the agent can move through them or they can be walls. The objective is to get to a goal square without hitting a wall. Our grid world consists of two rooms connected by a long hallway and ten goals chosen at random as shown in Figure 4-1. It has 1040 primitive states, 800 of which are non-wall states. For each action, the agent has an 85% chance of successfully reaching its intended square and a 15% chance of failing, causing movement into one of the three other adjacent squares (each with an equal probability of 5%). If the agent tries to move into a wall, it has an 85% chance of remaining where it is and a 5% chance of each of the other adjacent squares. Each action the agent takes incurs a reward of -1 unless it hits a wall, in which case it receives a reward of -10. The reward associated with the goal states is 0.

This world is very easy to cluster because all actions are immediately reversible in one step. Therefore, running only the preprocessing on the domain results in an e-

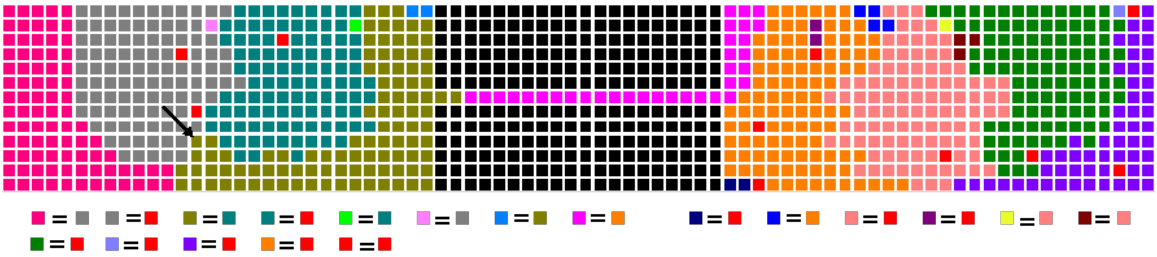

Figure 4-2: Clustering algorithm run on the grid world. Squares of the same color are in the same macro-states. Walls are black, goals are red (each wall is its own "macro-state", all goals are clustered together). The high level policy is shown under the figure. The equals sign to the right of a colored square indicates that the macro-state of that color transitions to the macro-state of the color on the right of the equals sign.

connected clustering with macro-states of size S_{max}. The clustering with $S_{max} = 100$ is shown in Figure 4-2.

Firstly notice that this clustering is certainly eg-connected. Each square of the grid world is adjacent to its 4-connected neighbors. Therefore the clustering is actually e-connected except that we put all the goals in the same macro-state.

One interesting point of this clustering is that there are a few very small (even single primitive state) macro-states. These happen when a primitive state becomes "squished" between two macro-states of maximum size so that neither macro-state can absorb it. This is an example of where a better stopping criterion for the clustering would be helpful.

We use this clustering as a two-level hierarchy for the solver. The solver first creates the determinized model; for primitive states i and j away from the wall, the cost model is uniformly $C^0(i,j) = \frac{1}{0.85} = 1.18$. We use this model to construct the costs at the upper level. For example, the cost of transitioning from the pink area on the far left to the grey area is the average cost for any pink square to reach any grey square. Note that this cost is smaller than, for example, the cost between the pink and gold regions since the grey region is, on average, closer to the pink. Since the grey region is also adjacent to a goal while the gold region is not, it's cost-to-goal is also smaller than that of the gold region. Therefore, the policy of the pink region is set as "grey". The full upper level policy is shown in Figure 4-2.

Once the upper level policy has been computed, the primitive level policy is solved for each pair of macro-states. For example, we construct an MDP of just the pink and grey primitive states, setting every grey state to be a goal state.

Since the clustering is a partition of the primitive state space, it is easy to construct a flat policy from the hierarchical policy: we just take the union of all the primitive-level policies.

4.6 Results

We implemented the algorithms described in Sections 4.1 and 4.3. In this section we show some of the results of running this algorithm. We leave discussion of some of the more surprising results to the analysis section, however.

4.6.1 Domains

We tested HDet on three standard domains:

- **Grid World**: The dynamics of this domain were described in detail in Section 4.5. Recall that there are 4 actions (UP, DOWN, LEFT, RIGHT) with an 85% chance of transitioning to the adjacent square corresponding to the action and a 5% chance of transitioning to any other adjacent square. We use two grid world domains: the domain described in Section 4.5 with 1040 primitive states and a large domain with 62,500 primitive states.

- **Factory**: The factory domain is a version of the common *Builder* factored MDP problem [Dearden and Boutilier, 1997]. Although our algorithm is not specifically designed to take advantage of a factored structure, we can solve factored domains by expanding out the primitive states. The factory domain has only 1024 states so expanding out the primitive states and specifying the flat transition matrix is possible. We use this domain because it allows us to move away from a two dimensional world and shows that the algorithm performs well even in domains that do not have a grid structure.

  In this domain, the agent is attempting to put together an object. There are two objects, each with 5 attributes (CLEANED, SHAPED, DRILLED, PAINTED, JOINED), for a total of 1024 states and 10 actions. Certain actions can affect more than one attribute; for example, the CLEAN action can make an object both CLEANED and ¬PAINTED. Actions are not necessarily reversible. The goal state is to have all 5 attributes true for both objects. Again, each step is -1 and the goal is 0. Note that this is a slightly modified version of the classic factory domain; we are not using additive rewards. We also modified the domain structure slightly so that, under an optimal policy, every primitive state can eventually reach the goal state. This allowed us to use this domain to compare clustering algorithms easily. A full description of transition matrix for the domain is given in Appendix A.1.1.

- **Mountain Car**: Lastly, we used a discretized, randomized version of the Mountain Car domain ([Sutton and Barto, 1998], section 8.4). This domain is not factorable, but has less structure than the grid world. This domain consists of a car on a mountain as shown in Figure 4-3. The object is to reach the top of the hill on the right, but the hill is so steep that the car cannot accelerate up it. Therefore, the correct policy is to back up the hill on the left and then accelerate down it and up the other hill. There are two variables: x_t, the

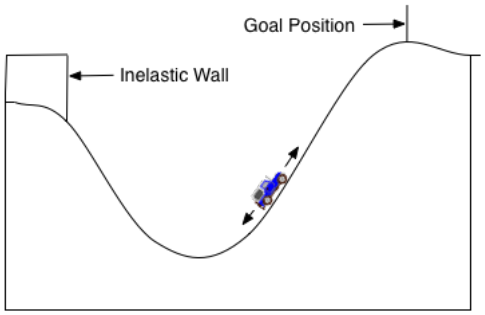

Figure 4-3: The mountain car domain. Figure taken from [RL-Library, 2009].

current x-axis position of the car and v_t, the car's current velocity. There are 3 actions: BACKWARDS, NEUTRAL, and FORWARDS and the transition matrix is defined by

$$\begin{aligned} x_{t+1} &= bound(x_t + \dot{x}_{t+1}) \\ \dot{x}_{t+1} &= bound(\dot{x}_t + 0.001a_t - 0.0025\cos(3x_t)) \end{aligned} \tag{4.5}$$

where $a_t = -1$ if the chosen action was BACKWARDS, 0 if the action was NEUTRAL, and +1 if the action was FORWARDS. The *bound* function just places the position and velocity in the closest bucket in our discrete representation. This is a typical definition of the mountain car domain.

We discretized the domain using 32 buckets on each axis and randomized it by giving a 5% chance to each of the squares adjacent to the expected square. In other words, the car might wind up slightly out of position or with a slightly different velocity than was predicted, but not both. This also had the effect of making no "dead" squares; because there is a possible drift, every square can eventually reach the goal. We used an x-axis from -1.2 to 0.6 and capped the velocity at ± 0.7. We again used rewards of -1 for each time step and 0 for reaching $x = 0.6$.

4.6.2 Comparison Against Other Clustering Techniques

We will show how HDet compares to other full algorithms in Section 4.6.3, but in this section we make a case for our clustering algorithm.

A common question about the HDet algorithm is why we chose to create a new clustering algorithm when there are already so many available. One answer is that eg-clustering is, in general, much faster than other techniques. Recall that we build a new hierarchy for every problem so that speed in clustering is important. Many clustering techniques assume the same hierarchy will be used for multiple problems so that extra time spent in clustering will be amortized out over many runs.

More importantly, however, many existing techniques do not satisfy our clustering principle and therefore suffer under a hierarchical policy. We show that in the Factory domain many common methods create stranded sub-states. Specifically we compare

Clustering Method	Hand-Built	Wavelets	N-Cuts	EG-Connected
Number of Stranded Sub-States (of 1024)	463	212	114	0
Clustering Time (s)	0	35	79	2

Table 4.1: Comparison of several clustering algorithms in the factory domain. All were implemented in Matlab and run on a 2.4 GHz Intel Core2 Q6600 Quad-Core.

against

- **Hand-Built Clustering**: We created by hand a clustering based on the factored properties of the domain. The clustering is shown in Figure 4-4. This clustering was an "intuitive" clustering in the sense that, if we start with two objects for which nothing is done, the "blank state", we want to traverse the hierarchy in reverse order. For instance, in the top level of the hierarchy, we differentiate primitive states by whether both objects are joined, only one object is joined, or neither object is joined; under the optimal policy from the blank state the last step we want to make is to join the objects. In the next level down, we differentiate on whether the objects are joined and painted — under the optimal policy painting is the second-to-last step. The problem with this clustering, however, is that we may *not* start with blank objects. Perhaps, for example, there has been in defect earlier in the manufacturing and the objects start as painted, but not cleaned, a state that is not reachable from the blank state. The clustering forces the hierarchical policy to treat all primitive states with PAINTA and PAINTB true in the same manner — there is no way to specify that if an object is painted but not cleaned it must be cleaned and then re-painted. Therefore, although this hierarchy appears to actually be guiding the solver to the correct answer it will not, in fact, work well for every primitive state because it is not eg-connected.

- **N-cuts Clustering**: The N-cuts clustering method is a common clustering technique pioneered by [Shi and Malik, 2000]. This is a divisive clustering technique that attempts to make "cuts" where there are only a few connections. We wrote our own implementation of N-cuts in Matlab, using k-means to cluster the eigenvectors.

- **Wavelets Clustering**: The wavelets clustering technique [Maggioni and Mahadevan, 2006b; 2006a; Coifman *et al.*, 2005] attempts to discover the underlying structure of the domain by finding a wavelet basis for it and using that to cluster. We used a pre-release Matlab implementation of this clustering obtained from the authors.

In Table 4.1, we show the number of primitive states each of these algorithms strand in the factory domain and the time each clustering took. Note that these are the number of primitive states stranded by the HDet solver algorithm. Clearly, these clustering algorithms might do better with another solver, but our point here is that, when using our solver, our algorithm is guaranteed not to strand sub-states. In

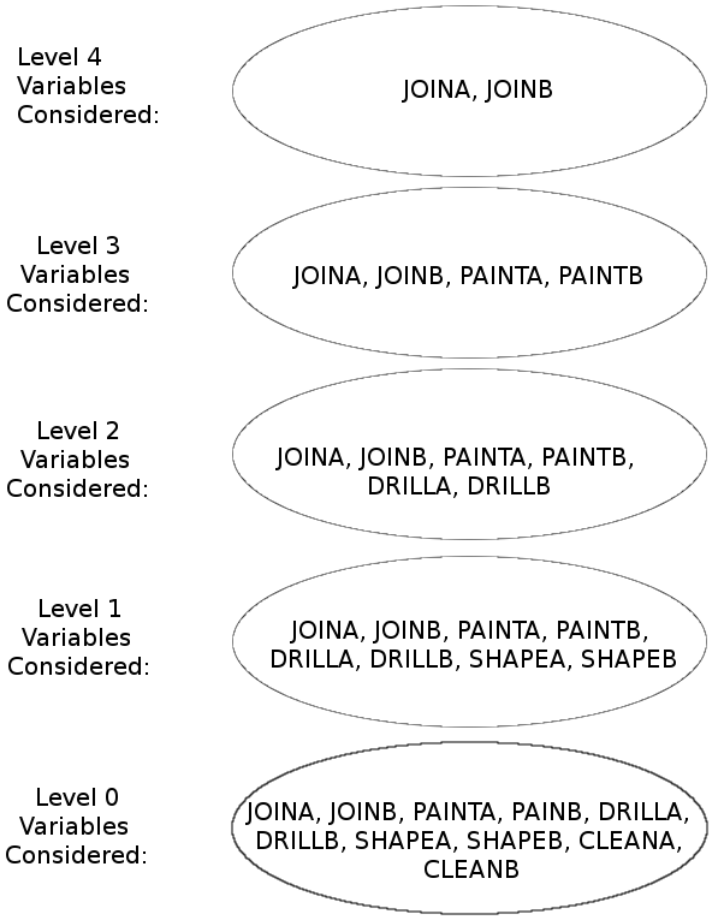

Figure 4-4: The hand-built 5-level hierarchy for the factory world. A and B refer to the two objects in the world. At each level, all primitive states with the same values of the variables considered are grouped together. For example, at level 4, there are four macro-states: primitive states with both objects joined, primitive states with just object A joined, primitive states with just object B joined, and primitive states with neither object joined.

addition, eg-clustering is also an order of magnitude faster than the other two methods because it does not require multiple restarts (N-cuts) nor costly mathematical operations (wavelets).

4.6.3 Comparison Against Other Algorithms

We compare HDet to other, contemporary MDP solvers on the domains listed in Section 4.6.1. Unless stated otherwise, we implemented and ran these algorithms on a 2.4 GHz Intel Core2 Q6600 Quad-Core. The algorithms we compare against are:

- **Value Iteration**: Most of the domains are small enough that they can be solved optimally using value iteration. This allows us to report the actual deviation of

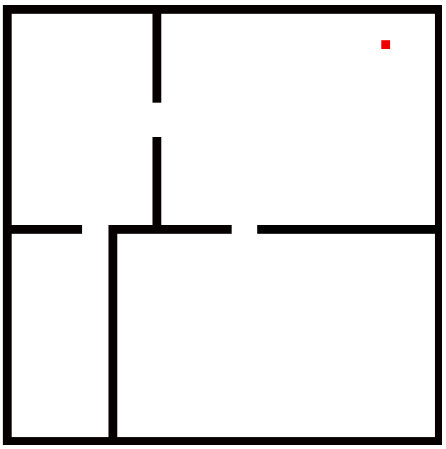

Figure 4-5: A grid world with 62500 primitive states. The red state is the goal state and the black states are wall states. There are 55710 non-wall states.

each algorithm's solution from the optimal solution.

- **HDet**: The HDet algorithm as described in Sections 4.1 and 4.3, with NCP, the out-cluster penalty, set to 10.
- **Det**: Det is a version of HDet, but without running the clustering algorithm and instead treating each primitive state as its own macro-state in a 2-level hierarchy. This strategy never solves any MDPs, but works entirely with the link cost estimates.
- **RPI:** Representation Policy Iteration (RPI) uses proto-value functions to decompose MDPs into hierarchies and solves the hierarchies using LSPI [Maggioni and Mahadevan, 2006b]. We do not implement this code but rather cite the results from [Maggioni and Mahadevan, 2006b].
- **HVI**: Hierarchical Value Iteration (HVI) was proposed by [Bakker *et al.*, 2005] as an extension of value iteration to hierarchical MDPs. They recommend using the normalized cuts clustering method (spectral clustering) to create the macro-states, but we found that using the eg-connected macro-states worked much better in general. We report results from HVI using spectral clustering as HVI(S) and results using eg-connected clustering as HVI(E).
- **VISA**: Variable Influence Structure Analysis (VISA) is an algorithm for solving factored MDP domains that takes advantage of the factored structure. We did not implement this code but cite the results from [Jonsson and Barto, 2006].

Of these algorithms, only value iteration actually gives a value for each primitive state. For the other algorithms, therefore, we compute the value function by running simulations. We run 1000 simulations of each policy starting from each primitive state in the domain and averaging together the total reward from the simulations to find a policy value for every primitive state in the domain. We report the average deviation

of these policy values from the optimal values. Since, for every domain, each action has a reward of -1, average deviation is a measure of, on average, how many more actions the hierarchical policy requires than the optimal policy. However, since there may be additional penalties, for example, for hitting a wall, this interpretation is only heuristic. We also report the "percent error", which is the average deviation divided by the average number of steps in the policy. The results are shown in Tables 4.2, 4.3, and 4.4.

We also ran one very large domain, a grid world of 62500 primitive states, of which 55710 states were not walls, shown in Figure 4-5, to show that the algorithm can handle large domains. We just ran Det and HDet in this world. We also attempted to run value iteration but the world is large enough that it did not converge in any reasonable time frame. In evaluating the policy, we ran 100 simulations from 343 randomly chosen starting primitive states. HDet found a policy that was on average 0.2 higher than the policy found by Det. Full results are given in Table 4.5.

We discuss these results in Section 4.7

4.7 Analysis

In this section, we analyze the results presented in Section 4.6.

4.7.1 Accuracy versus Running Time

Our runtime analysis indicates that clustering and solving time should both scale with the size of the clusters. Figure 4-6 shows this empirically in the factory domain. The solver time can be fit almost perfectly with a quadratic expression.

However, although small macro-states provide for a fast running time, they also cause a decrease in accuracy. This is shown empirically in Figure 4-7, but we also discuss a theoretical reason for it here.

A "cliff" is an area in which primitive states that can reach a primitive goal state in a small number of transitions are adjacent to primitive states that take many transitions to reach a primitive goal state or primitive states that incur a large negative reward. Consider for example, the mountain car domain. This domain has a number of metaphorical cliffs. For example, the situation in which the car has just enough velocity to reach the top provided the FORWARD action is chosen represents a cliff. If, in doing one of these FORWARD actions, the car is left with less velocity than anticipated, it will have to go back down the right hill and up the left hill before being able to reach the goal. Therefore, an optimal policy will try to avoid being left with *just* enough velocity. However, when we cluster primitive states together, we reduce the size of the MDPs we are solving. This increases the efficiency of the algorithm, but it decreases its accuracy because it decreases the ability to avoid primitive states on the edge of a cliff. Consider the extreme case of running Det on the Mountain Car domain. Det runs purely a shortest path algorithm and never solves an MDP. Therefore, because it never takes into account the uncertainty of the actions, it will never go further up the left hill than is far enough to acquire exactly the velocity

Algorithm	Number of Macro-states	Max States per Macro-State	Clustering Time (s)	Solver Time (s)	Total Time (s)	Average Deviation	Percent Error
HDet	17	100	0.72	0.69	1.41	0.48	5.80
Det	800	1	0	0.19	0.19	0.18	2.22
HVI (E)	17	100	0.72	9.94	10.66	0.84	10.16
HVI (S)	10	119	21.48	2.92	24.40	0.66	8.02
RPI	–	–	∼27	∼ 3	∼ 30	0	0
Value Iteration	1	1040	0	20.46	20.46	0	0

Table 4.2: Results in the grid world with 1040 states. The RPI results are taken from [Maggioni and Mahadevan, 2006a]. For the number of macro-states, we report only the number of non-wall macro-states (which is why we report only 800 macro-states for Det). Each wall state is its own macro-state.

Algorithm	Number of Macro-states	Max States per Macro-state	Clustering Time (s)	Solver Time (s)	Total Time (s)	Average Deviation	Percent Error
HDet	168	67	2.17	0.41	2.58	0.49	5.51
Det	1024	1	0	0.25	0.25	0.35	3.93
HVI (E)	168	67	2.17	38.02	40.09	0.62	6.97
HVI (S)	20	193	79.41	1.91	81.32	2.36 (114 states fail to reach goal)	27.39
VISA	–	–	∼5	∼20	∼25	0	0
Value Iteration	1	1024	0	25.22	25.22	0	0

Table 4.3: Results for the factory domain. The results for VISA given here were taken from [Jonsson and Barto, 2006].

Algorithm	Number of Macro-states	Max States per Macro-state	Clustering Time (s)	Solver Time (s)	Total Time (s)	Average Deviation	Percent Error
HDet	12	349	20.59	5.02	25.79	4.14	8.77
Det	1024	1	0	0.51	0.51	15.55	32.94
HVI (E)	12	349	20.59	58.35	78.94	12.94	27.41
HVI (S)	15	389	36.21	87.87	124.08	236.58	501.17
Value Iteration	1	1032	0	83.00	83.00	0	0

Table 4.4: Results for the mountain car domain.

Algorithm	Number of Macro-states	Max States per Macro-state	Clustering Time (s)	Solver Time (s)	Total Time (s)	Average Deviation (from HDet)	Percent Error (from HDet)
HDet	767	123	16.65	57.56	74.21	0	0
Det	55710	1	0	94.72	94.72	0.2	0.14

Table 4.5: Results for the grid world with 62500 primitive states. Since we do not have results for the optimal policy, the average deviation and percent error reported are the average deviation and percent error from the policy found by HDet. In other words, the value of the policy found by Det is, on average, 0.2 and 0.14% less than that found by HDet.

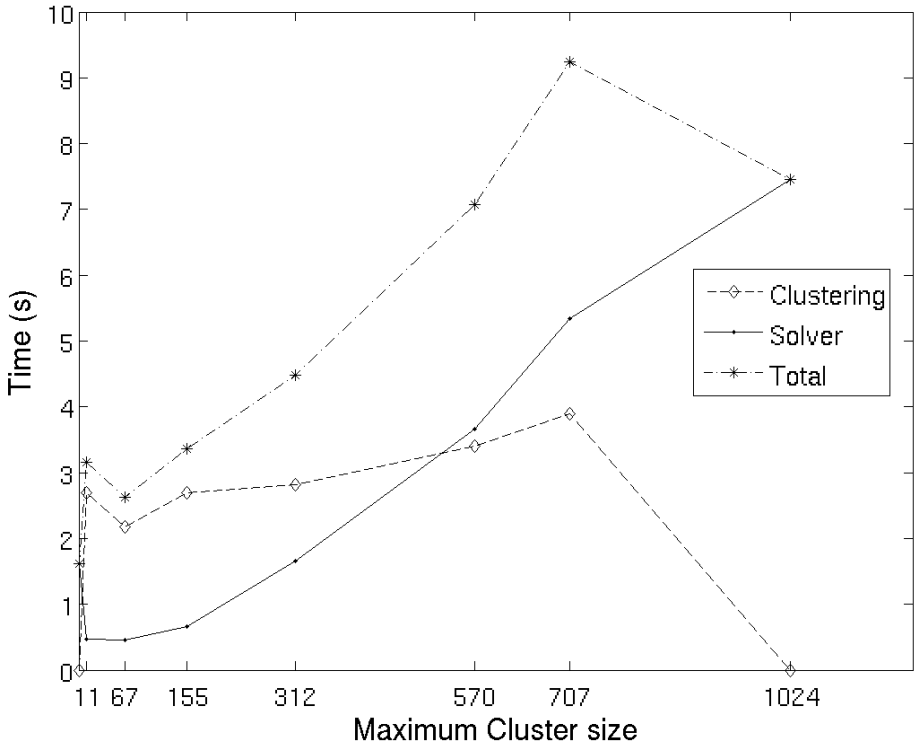

Figure 4-6: Running time as a function of macro-state size in the factory domain.

needed to crest the right hill. It is clear from Table 4.4 that Det performs much worse on the mountain car domain than HDet.

The grid world also has "cliffs" although these are less obvious. The cliffs in this domain correspond to the walls. An optimal policy will try to avoid squares near the walls since it is possible to accidentally crash into a wall. As the uncertainty in the actions increases, the optimal policy will go farther and farther out of the way to avoid the walls. Therefore, as non-determinism increases, clusterings with smaller macro-states will do increasingly worse. This is shown in Figure 4-8. As the non-determinism increases, Det, which has macro-states consisting of just a single primitive state does worse than HDet, which has macro-states of up to 100 primitive states, and the accuracy of both relative to the optimal policy decrease with increasing uncertainty. We will discuss later why Det does better than HDet with lower uncertainty.

Therefore, unsurprisingly, small macro-states usually lead to a more efficient algorithm, but less accuracy.

4.7.2 Clustering Time versus Solving Time

The relationship between clustering time and solving time is subtler than that between computation time and accuracy. For many domains, these in fact scale together because equation 4.2 is dominated by the S_{max} term.

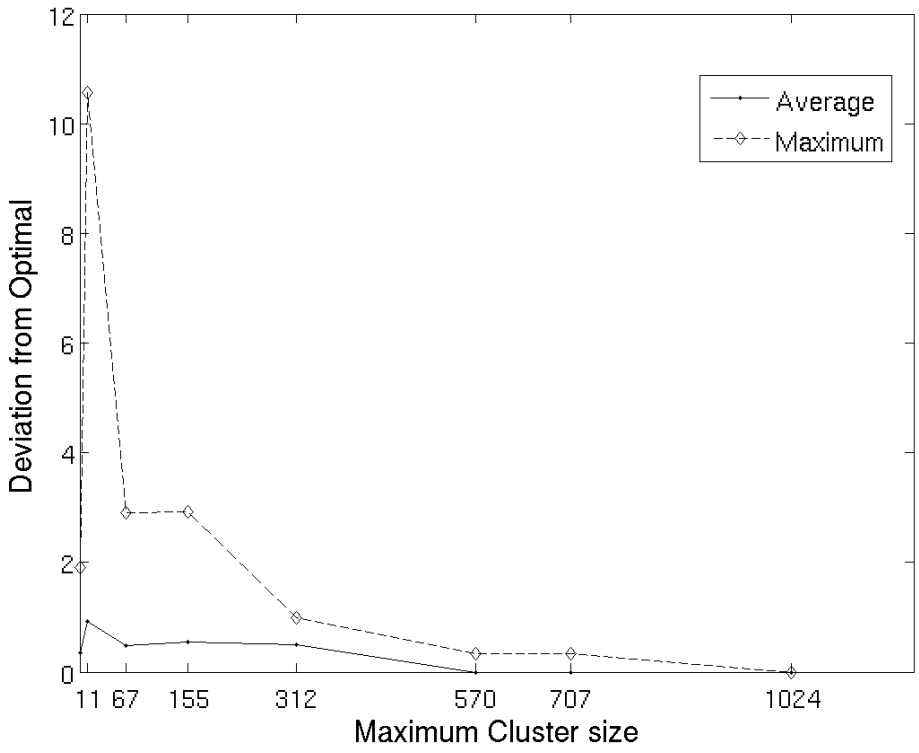

Figure 4-7: Average (over the primitive state values) and maximum deviation from the optimal policy as a function of macro-state size in the factory domain.

However, in some situations, equation 4.2 may be dominated by the $|S^{L-1}|$ term. In these situations, clustering time and solving time scale inversely, because the solver time is dominated by $|S^{L-1}|$ while the clustering is dominated by S_{max} (which is inversely related to $|S^{L-1}|$).

We can give an extreme example of this in the large grid world domain. In this problem, unlike in any of the others, Det was slower than HDet, although it takes no clustering time. This arises from that fact that Det solves the shortest path problem over the entire primitive state space, which does, in fact, scale as $O(|S|^2)$. Therefore, in this case, spending time in clustering actually reduced the time spent in solving and, in fact, the overall runtime.

Thus, as the size of the primitive state space increases, a clustering can yield both higher accuracy and lower runtime. Balancing the time gained from fewer number of macro-states at level $L-1$ with the time lost in clustering and solving these macro-states is a tricky optimization problem that we leave to future work.

4.7.3 Number of Levels in the Hierarchy

In the results we have given here, all of our hierarchies contained only 2 levels, because this was the appropriate number of levels for problems of the size we presented. In

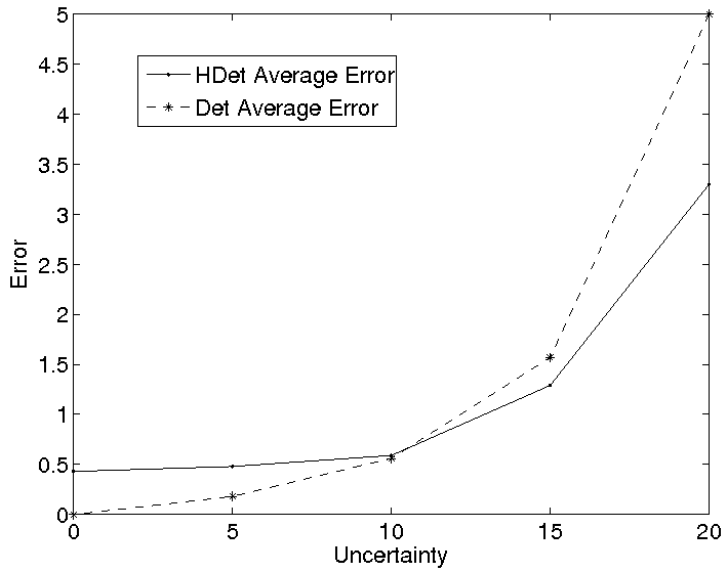

Figure 4-8: Average deviation from the optimal policy as a function of uncertainty in the grid world domain. Here $x\%$ uncertainty refers to the probability an action transitions to a wrong square. The probability the action will transition to the correct square is 1 - $\frac{3x}{100}$. For the HDet point at 20% we used an out-cluster penalty (NCP) of 50 to avoid oscillations. The number of macro-states used for HDet was 17 with a maximum of 100 primitive states per macro-state. The clustering is shown in Figure 4-2.

this section, we discuss reasons why we might or might not want more levels in the hierarchy.

We firstly again consider runtime. Equation 4.4 scales with the size of the largest macro-state but it *also* scales with the number of macro-states at level $L-1$. Therefore, in a very large domain, we may want a large number of macro-states at level 1 in order to have relatively small macro-states. However, as we discussed in Section 4.7.2 solving a shortest path over all of the level 1 macro-states may be too expensive; in this case, we want a 3-level hierarchy. For small domains, though, the extra time taken to create a 3-level hierarchy is not enough to offset the possible advantage in solving time. In all of the domains presented here, even the large grid world, a 2-level hierarchy was the correct choice.

From the discussion in Section 4.7.1, it may appear that if we created a 3-level hierarchy and a 2-level hierarchy with the same clustering at level 1, the accuracy of the two solutions would be identical because they have the same ability to avoid "cliffs". In actuality, however, there is another factor in determining accuracy.

Because our solver operates in a top-down fashion, an upper level policy is imposed at the lower levels. Consider again the clustering of the grid world shown in Figure 4-2 and note the state, s, pointed to by an arrow in the gold macro-state. Clearly the closest goal state to s is the state 2 squares above it. However, to reach this goal state, s must pass through the grey macro-state and the top level policy of the

gold macro-state requires that its sub-states go through the turquoise macro-state. Therefore, the policy given by HDet for s requires two more steps than the optimal policy.

The penalty for following the upper level policy is most apparent in a comparison of Det and HDet. Because Det has "macro-states" of size 1, it does not suffer from the imposition of an upper-level policy. Therefore, in highly-deterministic domains with few cliffs, Det will actually perform better than HDet. This is apparent both in Tables 4.2 and 4.3 and in Figure 4-8.

We can also see the penalty for following the upper level policy if we compare HVI(E) with HDet. Although these have the same clustering, HVI(E) generally does worse than HDet. This is because with HDet, we are able to "guess" the value of deviating, at the lower level, from the upper level policy to some accuracy using our *out* state. With HVI we cannot do this because we have no meaningful value for the upper level macro-states. Therefore, HVI enforces the upper level policy strictly while HDet occasionally allows deviations from it and, as a result, HVI returns less accurate solutions.

Thus, although for large domains a multi-level hierarchy may be necessary for a reasonable runtime, adding levels to the hierarchy is likely to reduce the accuracy.

In this section, we have presented our algorithms and results for enumerated-state MDPs, including a clustering algorithm that does not strand sub-states. We have shown that HDet performs as well or better than current work in the field and discussed the trade-offs that must be made in creating and solving a hierarchical MDP.

In the next chapter, we will discuss how these algorithms can be modified to work with factored MDPs.

Chapter 5

Factored MDPs: Algorithm and Results

In this chapter, we discuss the algorithms for factored MDPs. Working with the factored representation of an MDP tends to be more difficult than working with the enumerated-state representation because the primitive state space is exponential in the input size. Therefore, our algorithms must be polynomial in the number of state variables and logarithmic in the state space size. We follow the general outline of chapter 3, but expand and modify it for this representation.

5.1 Input to the Algorithm

We begin with a description of the input we expect for our algorithm. We work with domains that can be specified using the Probabilistic Planning Domain Description Language (P-PDDL), which is the standard for the ICAPS Planning Competitions [ICAPS, 2009]. We chose to use this input specification because there are a large number of domains and problems available in it.

P-PDDL is fully documented in [Younes and Littman, 2004], but for completeness we give a description here. For example domains and problems see Appendix A.2.

P-PDDL has two types of input files: domains and problems. Domains specify the world dynamics while problems specify the exact properties of the world, a goal condition and a probability distribution over initial conditions. For example, in a grid world, the domain would describe the dynamics of moving from square to square while the problem would specify the size of the grid, a probability distribution over starting squares, and the goal squares. Many problem instances may correspond to one domain.

A domain consists of:

- *Types*: The types of variables. Actions can use types in the preconditions since the domain file does not necessarily list all of the state variables.
- *Predicates*: A predicate is a function. The predicate and a specific parameter list together make an *atom* or *state variable* that can be either true or false.

The possible parameters to the predicates are specified in the problem file. For example, in the grid world, we may specify the type `grid-square` and the predicate `(at ?s - grid-square)` where the notation of the predicate indicates that `?s` is a variable of type `grid-square`. The problem file might then list 9 `grid-square`s (for a 3x3 grid). The result would be a problem with 9 atoms. However, with the *same* domain file, we could also create a 4x4 grid with a problem file that listed 16 `grid-square`s.

- *Actions*: An action consists of two parts:
  - *Precondition*: A condition that specifies in what states the action is "active". An attempt to call an action from a state in which the precondition is not met results in an error. Currently, we require that preconditions not be disjunctive or require quantified operators. Both are "requirements" flags to PDDL and there are many domains that do not require them. We hope to expand the algorithm to take these requirements in future work.
  - *Effect*: A description of the effect of the action. There are four types of effects
    1. *Atomic*: An effect that causes a single atom to become true or false.
    2. *Conjunctive*: A conjunction of effects.
    3. *Probabilistic*: An effect that assigns a certain probability to each effect in a list. Probabilities should sum to one.
    4. *Conditional*: An effect that only takes place when a certain condition is met. Note that this is different from the precondition to the whole action — an action can be called even if it has conditions on the effects that are not met. For example, in a grid world if moving left is always a valid action but only results in a move left if the agent is not already at the far left edge, the effect of the action is conditional. However, if move left is only *valid* when the agent is not already at the far left edge, then "not-at far left edge" is a precondition to the action. For the moment, we require that conditions on effects not be disjunctive or involve quantified operators.

The problem consists of

- *Atoms*: Boolean variables that can be either true or false. A single "state" of the MDP is an assignment to all of the atoms. We may also refer to atoms as *state variables*.
- *Initial State Distribution*: A list of the domain atoms along with the probability of how likely each atom is to be true in the starting state. When a simulation is run, it picks a starting state according to this probability distribution.
- *Goal*: A condition specifying the goal states. We require this condition to be conjunctive.

Currently, our planning algorithm cannot take the requirement flags `:disjunctive-preconditions, :existential-preconditions, :universal-preconditions, :rewards, :fluents`, or any requirements that imply these requirements. At the moment, we assume that for all MDPs the goal reward is 0 and each step has a reward of -1.

5.2 Clustering

We begin by describing our clustering algorithm for factored MDPs. For the factored representation, because we cannot look at every primitive state, we take the view of clustering as "divisive" rather than agglomerative as we did in the enumerated-state representation, meaning that we start with all primitive states in one macro-state and "divide" this macro-state to create more. The resulting clustering will exhaustively cover the state space, but it may not be a partition — some macro-states may overlap.

In essence, in a factored domain we now work with sets of primitive states instead of individual primitive states. Rather than try to keep the clustering principle true for every primitive state in the domain, we keep it true across sets of states. We begin with some definitions. For completeness, we repeat some of the definitions given in Section 3.1.

5.2.1 Definitions

- **Primitive state**: A full assignment to the domain atoms.
- **Goal state**: A primitive state that satisfies the goal condition.
- **Adjacent**: A primitive state j is *adjacent* to primitive state i if there exists an action a such that there is more than an ϵ probability that taking action a in state i will result in a transition to state j.
- **Reachable**: Primitive state j is *reachable* from primitive state i if j is adjacent to i, or if j is adjacent to some primitive state k that is reachable from i.
- **f-State**: A set of primitive states.
- **Goal f-state**: A set of primitive goal states.
- **Macro-state**: A level 1 macro-state is a set of f-states. A level $l > 1$ macro-state is a set of level $l - 1$ macro-states.
- **Sub-state**: A sub-state of a macro-state is any direct descendent of the macro-state. Sub-states may be either f-states or macro-states.
- **Goal macro-state**: A goal macro-state is a macro-state that has a goal f-state as a descendent.

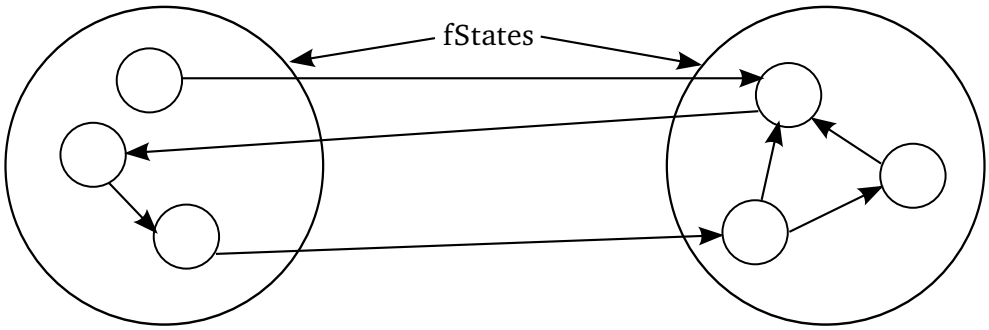

Figure 5-1: An example of an fe-connected macro-state with two f-states. Although the macro-state is not connected (not every sub-state can reach every other sub-state), it is *fe-connected* because all primitive states of the f-state on the left can reach some primitive state of the f-state on the right and vice-versa.

- **Macro-action**: Macro-actions represent an attempt to transition between macro-states. For any non-leaf level of the tree with n nodes, we have n^2 macro-actions, one for each pair of macro-states.

- **f-Adjacent**: An f-state f_2 is *f-adjacent* to an f-state f_1 if, for every primitive state s_1 in f_1 there exists some primitive state s_2 in f_2 such that s_2 is adjacent to s_1 or $s_1 = s_2$. A macro-state m_2 is f-adjacent to a macro-state m_1 if there exists sub-states $m_2' \in m_2$ and $m_1' \in m_1$ such that m_2' is f-adjacent to m_1'.

- **f-Reachable**: An f-state f_2 is *f-reachable* from an f-state f_1 if f_2 is f-adjacent to f_1 or if f_2 is f-adjacent to some f-state f_3 that is f-reachable from f_1. A macro-state m_2 is *f-reachable* from an macro-state m_1 if m_2 is f-adjacent to m_1 or if m_2 is f-adjacent to some macro-state m_3 that is f-reachable from m_1.

- **fe-connected**: A macro-state is *fe-connected* if every f-state in the macro-state is f-reachable from every other state. An example of an fe-connected macro-state is shown in Figure 5-1.

Using these definitions for macro-states and macro-actions, we define our HMDP as discussed in Section 3.1.

5.2.2 Operations

In this section we define a set of operations that we can perform on a macro-state or a set of macro-states. Recall that our algorithm is a divisive clustering method so that we begin with only one initial macro-state. The operations we will need to perform are

- **Split**: We can *split a level l macro-state C on a set S* where S is an arbitrary set of level $l-1$ f- or macro- states. The result of a split is two new macro-states

C_S and $C_{\setminus S}$:

$$C_S = C \cap S \tag{5.1}$$
$$C_{\setminus S} = C \setminus S \tag{5.2}$$

- **Insert**: We can insert a level l macro-state N into a level l macro-state C, to create a new level l macro-state denoted $C \leftarrow N$. For a set S of level l states let $R_{\rightarrow S}$ be all the level $l-1$ f- or macro- states that can f-reach some sub-state of S and $R_{\leftarrow S}$ be all the level $l-1$ primitive or macro- states that are f-reachable from some sub-state of S. Then

$$C \leftarrow N = (C \cap R_{\rightarrow N} \cap R_{\leftarrow N}) \cup (N \cap R_{\rightarrow C} \cap R_{\leftarrow C}). \tag{5.3}$$

  The insertion operation takes all of parts of the sub-states of C that can reach and be reached by some parts of the sub-states of N and combines those with all parts of the sub-states of N that can reach or be reached by some parts of the sub-states of C. Therefore, insertion is a method for combining states of C and N while preserving connectedness. The macro-state $C \leftarrow N$ has $|C| + |N|$ sub-states, although these sub-states may describe fewer primitive states than they did originally. Note that insertion is a commutative operation.

- **Prune**: Given a list of macro-states, *pruning* is the process of removing any empty macro-states and any macro-states that are subsets of another cluster in the list.

5.2.3 Time Bounds

Before describing our clustering algorithm, we discuss the time required to perform each of the clustering operations in order to give a sense of how we represent and manipulate macro-states and f-states.

We describe f-states using boolean formulas of the atoms. Therefore, macro-states are also described by boolean formulas (the disjunction of the formulas describing all of the sub-states of the macro-state). The time taken by each operation is

- *Set Union*: The union of two sets S_1 and S_2, described by formulas F_1 and F_2 respectively, can be done in constant time by creating $S_3 = S_1 \cup S_2$ described by $F_1 \vee F_2$.

- *Set Intersection*: The intersection of two sets S_1 and S_2, described by formulas F_1 and F_2 respectively, can be done in constant time by creating $S_3 = S_1 \cap S_2$ described by $F_1 \wedge F_2$.

- *Set Difference*: The difference of two sets S_1 and S_2, described by formulas F_1 and F_2 respectively, can be done in constant time by creating $S_3 = S_1 \setminus S_2$ described by $F_1 \wedge \neg F_2$.

- *Creating* $R_{\leftarrow S}$: For a set S described by the formula F the process of creating $R_{\leftarrow S}$ is linear in the number of actions. For each action A, we must test if the precondition P of A satisfies F. In the general case, this may be NP, but, because we require conjunctive preconditions, P can actually be treated as a partial assignment to domain atoms. Therefore, if F has $|F|$ nested conditions and no more than F_{max} terms in a condition, evaluating if P satisfies F, is $O(|F|F_{max})$. If P does satisfy F then generating the possible next states of the action is linear in the number of effects of the action. Therefore, creating $R_{\leftarrow S}$ is $O(|A||F|F_{max}|E|_{max})$ where $|E|_{max}$ is the largest number of effects any action has. Note that we are working in domains with small action spaces so that this is a reasonable runtime.

- *Creating* $C \cap R_{\rightarrow S}$: Finding all of the states that can transition into a set S can be difficult. However, we never actually need $R_{\rightarrow S}$, only the intersection of $R_{\rightarrow S}$ and some other set C. Note that $C \cap R_{\rightarrow S}$ can be represented as $R_{\leftarrow C} \cap S$ (all of the sub-states of C that can reach S). Since intersection is a constant time operation $C \cap R_{\rightarrow S}$ can also be found in $O(|A||F|F_{max}|E|_{max})$ where $|F|$ and F_{max} refer to the condition describing C.

- *Split*: Split consists of one intersection operation and one difference operation and can therefore be performed in constant time.

- *Insert*: Combining the above run times, it is clear that the insert operation requires time $O(|A||F|F_{max}|E|_{max})$.

- *Prune*: The prune operation is clearly in NP since discovering empty macro-states requires deciding if there is a solution to an arbitrary boolean equation. However, this is a problem that can generally be solved quickly using a heuristic. Our heuristic is simply to generate a large number of primitive states and mark the clusters to which those states belong. If a state falls into multiple clusters, we mark the one with the largest number of f-states. Any cluster that is unmarked at the end of this process is pruned. As we will discuss, accuracy is dependent on the number of f-states in a macro-state. Therefore, if we prune a macro-state with few f-states that was not, in fact, redundant, little accuracy will be lost.

Therefore, in general, the clustering can be done in time polynomial in the number of domain atoms and actions. We now describe the actual algorithm.

5.2.4 Clustering Algorithm

The input to this algorithm is a maximum cluster size S_{max}, where size refers to the number of f-states, a P-PDDL domain file, and a P-PDDL problem file. We describe how we create a 2 level hierarchy and then extend that to creating hierarchies with multiple levels.

To initialize this algorithm, we read in the P-PDDL domain and problem files and then create a single macro-state. This macro-state consists of a single f-state described by the formula TRUE.

Now assume that at some point in the algorithm we have a list of p fe-connected macro-states. The algorithm proceeds as follows.

We first generate an fe-connected macro-state N. We pick a starting f-state, in a manner we will describe in Section 5.2.6, and, from that f-state create a cycle of f-states using a simple heuristic search.

Given N, we then run through the list of macro-states and, for each macro-state C_i, we create a macro-state $C_i' = C_i \leftarrow N$. If the number of f-states in C_i' is smaller than S_{max}, we add it to the list of macro-states. For example, let C_i be represented by the formula A and N be represented by the formula B (in general these formulas will be much more complicated, but this is just for the purposes of illustration), and assume we have only two actions: action 1 that takes $A \wedge D$ to $B \wedge D$ and action 2 that takes B to A regardless of the value of the other state variables. Then $C_i \leftarrow N$ consists of two f-states, $A \wedge D$ and $B \wedge D$. Note that this is more restrictive than the union of C_i and N because we have no way of transitioning simply from A to B. Instead, we need to restrict the f-state A to be $A \wedge D$ to be certain that there is a transition to some primitive state of B. Similarly, we restrict the primitive states of B to be the ones that we can transition to from $A \wedge D$. Action 2 allows the transition from $B \wedge D$ back to $A \wedge D$.

After we have created all of the C_i', we also add N to the set of macro-states for a total of at most $2p + 1$ macro-states in our list of macro-states. Note that, by the properties of insertion, each macro-state in the list is still fe-connected.

Because we never merge macro-states, this has the possibility to grow exponentially. However, in most cases at least some of the $2p + 1$ macro-states will be empty or a subset of another, larger macro-state. Therefore, we finish the algorithm by pruning the macro-state list.

5.2.5 Relaxed Algorithm

As with the enumerated state clustering, we may have domains for which it is difficult to generate cycles of f-states. However, again, we do not actually need fe-connected macro-states. We simply need macro-states that guarantee a connection to the goal. Therefore we define the property of *fg-connectedness*:

fg-connected: A macro-state is *fg-connected* if one of the following holds

- The macro-state has as a one or more sub-states $g_1, ..., g_p$ each of which contain, as a descendent, a goal f-state, and each sub-state of the macro-state can f-reach at least one of $g_1, ..., g_p$.

- The macro-state is fe-connected.

We modify our clustering algorithm to create fg-connected macro-states as follows:

When we first begin the clustering, we create two macro-states: *goal* described by the goal condition, and $\neg goal$, described by the negation of the goal condition.

When creating N, the new macro-state we are going to insert, we find either a cycle of f-states *or* a chain of f-states that culminates in a goal f-state. Now N is fg-connected but may not be fe-connected.

When doing the insert operation $C_i \leftarrow N$, if C_i contains a goal f-state, we define $R_{\leftarrow C_i} = \emptyset$. Similarly, if N contains a goal state, we define $R_{\leftarrow N} = \emptyset$. This results in lists of fg-connected macro-states.

As before, we have essentially described an algorithm that results in a 2-level hierarchy. A hierarchy with more levels can be created by starting with each of the macro-states of the 2-level hierarchy, rather than the macro-state TRUE, and dividing it.

5.2.6 Relation to Minimized-State MDPs

Recall from Section 2.5 that we can create a minimized model of the MDP that acts as an MDP in its own right. Here, we discuss the relationship between our clustering algorithm and minimized models. We aim to show that we are clustering the minimized states of the reduced MDP. In other words, we want to show that our clustering algorithm makes no "unnecessary" distinctions; ie, no distinctions that would not be made in creating a reduced MDP.

Theorem: If the domain is such that the actions have no disjunctive preconditions or disjunctive conditions on conditional effects (where a condition is also considered "disjunctive" if multiple conditions lead to the same effect even if this is expressed without explicitly using an "or"), then, for any clustering created using the algorithm described in Sections 5.2.4 and 5.2.5 where all "start" f-states are conjunctive conditions, there exists a partition P of the state space such that every f-state in the clustering is a set of the minimized states of P. In other words, the clustering makes no distinctions that would not have to be made in $MM(P)$.

Lemma A: Let P be a partition. Then any homogeneous refinement of P is a refinement of $MM(P)$.

Proof: This follows directly from Corollary 8.2 of [Givan *et al.*, 2003].

Lemma B: If p is a refinement of P then $MM(p)$ is a refinement of $MM(P)$.

Proof: Since $MM(p)$ is a homogeneous refinement of p, it is also a homogeneous refinement of P. Therefore, by Lemma A it must be a refinement of $MM(P)$.

Lemma C: Let $f_0, f_1, ..., f_{m-1}$ be a set of fg-connected states such that f_i is f-adjacent to f_{i+1} and $f_i \neq f_{i+1 \bmod m}$ and either f_{m-1} is a goal f-state or f_{m-1} is adjacent to f_0. Let f_i transition to $f_{i+1 \bmod m}$ with action a_i and probability t_i. If we started with f_0 and, in creating $f_1, ..., f_{m-1}$ made no distinctions that were not

required to enforce f-adjacency, then $f_0, ..., f_{m-1}$ are all sets of the minimized states of the partition $P_s = \{f_0, \neg f_0\}$.

Proof: We proceed by induction.

Base Case: Clearly f_0 is a set of minimized states of P_s since it is a block of the original partition. If f_{m-1} is not a goal f-state then the rest of this proof holds considering f_0 to be the base case. If f_{m-1} *is* a goal f-state then it describes all goal states reachable from f_0 by this chain. Since in creating these states, no distinctions were made that were not required for f-adjacency, these goal states therefore have different dynamics than any other states in the domain and thus are themselves a set of minimized states of P_s.

Inductive Step: Assume $f_{m-1}, f_{m-2}, ..., f_{k+1}$ are all sets of the minimized states of P_s and consider f_k. Because we make no distinctions not required for f-adjacency, the f-state f_k represents all states that can transition to f_{k+1} with probability t_k under action a_k. Let $f_{k+1} = m_1 \cup m_2 \cup ... \cup m_p$ where the m_i are the minimized states composing f_{k+1}. Now assume f_k is not composed of minimized states of P_s. Let Q be the set of states such that $f_k \cup Q = l_1 \cup l_2 \cup ... \cup l_q$ where the l_i are minimized states and $f_k \cap Q = \emptyset$. Then, since f_k was not a set of minimized states there must be some l_i such that l_i includes states both from f_k and from Q. Let $r \in l_i$ be described by f_k and $u \in l_i$ be described by Q. Then, since $r \in f_k$ there must exist some j such that r can transition to m_j under action a_k with probability t_k. However, since $u \notin f_k$, u cannot transition to m_j with probability t_k under action a_k. Therefore l_i is not stable with respect to m_j and action a_k contradicting that l_i and m_j are minimized states. Thus, by contradiction, f_k must also be composed of minimized states.

Proof of Theorem: We prove our theorem by constructing P and then showing by induction that any distinctions made by the clustering would also need to be made in creating $MM(P)$.

To create P, consider that during the clustering process we generate a number of possibly overlapping but non-identical "start" f-states and that we assume that these f-states are conjunctive conditions. Since we begin with the partition, $\{goal, \neg goal\}$, we let $s_0 = goal$. We can therefore partition the state space into 2^{k+1} blocks where the condition s_j holds in the ith block if the jth bit of i is 1. It is possible that some blocks of this partition are empty.

Note that at any point in the clustering such a partition exists. Therefore let P_i be the partition at the ith step in clustering. We show by induction that at any step i in the clustering, all f-states described in the clustering are sets of the minimized states of P_i.

Base Case: Partition P_0 is the partition $\{goal, \neg goal\}$, while the clustering at step 0 is also the partition $\{goal, \neg goal\}$. Clearly, therefore, since $MM(P_0)$ is a refinement of P_0, this clustering is a set of the minimized state of P_0.

Inductive Step: Consider step k of the clustering algorithm with partition P_k. At step $k+1$ we generate f-state s_{k+1} as a starting state creating partition P_{k+1}.

We first create the set of fg-connected f-states $f_0, f_1, ..., f_{m-1}$ where $f_0 = s_{k+1}$ such that for f_i is f-adjacent to f_{i+1} and $f_i \neq f_{i+1 \bmod m}$ and either f_{m-1} is a goal f-state or f_{m-1} is adjacent to f_0. In creating $f_1 ... f_{m-1}$ we made no distinctions that were not required to enforce f-adjacency. Therefore, by Lemma C, the f_i are all composed of minimized states of $MM(\{s_{k+1}, \neg s_{k+1}\})$. Note that since P_{k+1} is a refinement of $MM(\{s_{k+1}, \neg s_{k+1}\})$, this implies by Lemma B that the f_i are also composed of minimized states of P_{k+1}.

Now consider "merging" the new macro-state into the existing macro-states. Firstly, note that any f-states that remain untouched are sets of the minimized states of P_{k+1} since they were sets of the minimized states of P_k and P_{k+1} is a refinement of P_k. Therefore, consider a macro-state created by inserting the new macro-state N into macro-state C_i. In C_i, we split each f-state into an f-state that can f-reach s_{k+1} and one that cannot (and keep only the one that can f-reach s_{k+1}). Clearly, in $MM(\{s_{k+1}, \neg s_{k+1}\})$, those states that can reach some sub-state of s_{k+1} must be in a different block from those that cannot. Therefore, this split is one that would be made to create $MM(\{s_{k+1}, \neg s_{k+1}\})$ and the f-states of $C_i \leftarrow N$ originally belonging to C_i are sets of the minimized states of $\{s_{k+1}, \neg s_{k+1}\}$.

Now consider the f-states of $C_i \leftarrow N$ that are derived from the f-states of N. We split these f-states into those states that can f-reach the f-states of C_i and those that cannot. Since the f-states of C_i were originally sets of the minimized states of P_k, any division between them represents necessary divisions in forming $MM(P_k)$. Therefore the distinction between being able to f-reach and not f-reach (or be f-reached or not f-reached by) these states is a distinction that must be made in forming $MM(P_k)$ and the f-states of $C_i \leftarrow N$ that originally belonged to N are sets of the minimized states of P_k.

Therefore no splits are made that would not be made in creating either $MM(\{s_{k+1}, \neg s_{k+1}\})$ or $MM(P_k)$. Since, by Lemma B, $MM(P_{k+1})$ is a refinement of both of these, no splits are made that would not be made in creating $MM(P_{k+1})$. Thus all f-states in the clustering at step $k+1$ are sets of the minimized states of P_{k+1}.

Corollary: There exists a method for picking start states such that $MM(P_k) = MM(\{goal, \neg goal\})$ for any k. Since we consider only MDPs for which the reward is -1 for non-goal states and 0 for goal states, $MM(\{goal, \neg goal\})$ is the state space of the reduced MDP.

Proof: Consider the method for choosing start states such that at time step k, we pick a start state s_k such that s_k is stable with respect to some s_i for $i < k$, $goal \cap s_k = \emptyset$, and $s_k \neq s_i$ for $i < k$. Then by induction we can show that $MM(\{goal, \neg goal\})$ is a refinement of P_k. Recall that $s_0 = goal$.

Base Case: For $k = 0$, P_k is the partition $\{goal, \neg goal\}$. By definition, therefore $MM(\{goal, \neg goal\})$ is a refinement of P_0.

Inductive Step: Assume at step $k-1$ all s_i have been chosen in the manner described above and $MM(\{goal, \neg goal\})$ is a refinement of P_{k-1}. Now consider choosing s_k such that s_k is stable with respect to s_i. Let s_k describe all states that transition to s_i under action a with probability t. Let $B_1, B_2, ...B_{2^{k-1}}$ be the blocks of P_{k-1} and order them so that $B_1, B_2, ..., B_{2^{k-2}}$ contain only states described by s_i and $B_{2^{k-2}+1}, ..., B_{2^{k-1}}$ contain no states described by s_i. Now let B_j be any block of P_{k-1}. In creating P_k we split B_j into those states that can transition to s_i with probability t under action a and those that cannot. Therefore, if, in creating P_k we split B_j that implies that there is at least one state $p \in B_j$ such that p can transition to some B_v where $v \leq 2^{k-2}$ with probability t under action a and one state q such that q *cannot* transition to B_v with probability t and action a. Therefore, we only split B_j in creating P_k if there was some state $B_v \in P_{k-1}$ such that B_j was not stable with respect to B_v and action a. Therefore, any split we make in creating P_k is also made in creating $MM(P_{k-1})$ and, since the order of the splits does not matter [Dean and Givan, 1997], $MM(P_k) = MM(P_{k-1})$. By our inductive hypothesis this implies that $MM(P_k) = MM(\{goal, \neg goal\})$.

Therefore, if the starting states are picked in the above manner, any f-state in the clustering is a set of the minimized states of $\{goal, \neg goal\}$.

This also gives us a stopping point for the clustering; the clustering should stop when we can no longer generate any more unique starting states. Note that this does not mean that this clustering algorithm will produce the same clustering every time - the clustering will also depend on the S_{max} parameter.

5.2.7 Macro-State Structure

How we do bookkeeping within a macro-state is important to the solver algorithm. Therefore, we briefly discuss it here.

When a new macro-state N is inserted into an existing macro-state C, we modify each sub-state of N and C as described in Section 5.2.4 so as to create the fg-connected macro-state $C \leftarrow N$. In doing this, we also keep track of the possible transitions between sub-states. Specifically, each sub-state has a "to-adjacency" list, listing which sub-states in the macro-state can transition to it and a "from-adjacency" list listing the sub-states to which it can transition. An entry in an adjacency list for sub-state f has the following parts:

- *sub-state pointer:* A pointer to the sub-state to or from which f can transition.
- *Condition:* A conjunctive condition that must be true of f in order to allow this transition.

We require a condition in the adjacency lists because sub-states may be disjunctions. Therefore, although our clustering algorithm guarantees that a sub-state can transition to *some* other sub-state in the macro-state, *which* other sub-state may depend on the current conditions. We know, however, that the condition will be conjunctive because we require that preconditions to actions be conjunctive.

We will be associating values with conditions, which means we will often want to find, for a given state or partial assignment, which values apply. In doing this, we will use the idea of a "maximally matching" condition:

Maximal match: Given a list of conjunctive conditions $c_1, c_2, ..., c_k$, and a partial assignment of the domain atoms p, a condition c_i is a *maximal match* for p if c_i mentions at least as many of the variables assigned by p as any other condition in the list. A list may contain more than one maximally matching condition.

5.3 Solver

The solver takes as input an fg-connected hierarchy of f-states, the domain and problem files and outputs a hierarchical policy.

Before beginning the solver algorithm, for each f- and macro- state in the hierarchy we create a list, the *condition list* of the state, of all of the unique conditions in its adjacency lists. We will associate with each condition in this list a cost and next macro-state (if the state is a macro-state, ie level greater than 0) or a value and action (if the state is an f-state, ie level 0).

The solver still follows the general framework as described in Section 3.2. Here, we fill in those parts of the algorithm that depend on representation. We begin by describing how we compute the cost matrix.

5.3.1 Upward Pass: Calculating the Cost Matrix

Recall that in the upward pass, we require a method for estimating the expected cost to transition between two macro-states. We proceed as follows.

Because we now cluster sets of states, it is possible that a macro-state is adjacent to only *part* of another macro-state. Therefore, when calculating the cost of going from macro-state C_1 to macro-state C_2, in reality we only calculate the cost of going from $C_1 \cap R_{\rightarrow C_2}$ to C_2. We take this into account in the later part of the algorithm and represent this by adding C_1 to the condition and to-adjacency lists of all sub-states in C_2 f-adjacent to a sub-state in C_1 and C_2 to the condition and from-adjacency lists of all f-states in C_1 that can transition to a sub-state in C_2.

To get an exact estimate of the cost of going from C_1 to C_2, we would need to treat each condition in each sub-state's condition list as a different sub-state. However, this could result in an exponential unfolding of the macro-state so, instead, we consider the cost of each sub-state to be the weighted average of the cost of each condition in its condition list. The weighting is done by assuming a uniform distribution over the states in the sub-state. Specifically, since the conditions are conjunctive, the weight of any condition is $\frac{1}{2^a}$ where a is the number of distinct atoms mentioned in the condition. This is proportional to the probability that a primitive state picked at random from the primitive states described by the f- or macro-state would fulfill that condition. If the cost of a condition has yet to be assigned in the algorithm, we assume it has to go through the whole macro-state and assign it that cost.

To calculate the distance from one macro-state to another, we use Dijkstra's algorithm. When updating the cost of a successor to the current sub-state, we update the costs of only the conditions that are fulfilled by the current sub-state. The distance between the two macro-states is the average cost of the sub-states of the macro-state.

5.3.2 Downward Pass: Calculating Distance to Goal

During the downward pass, we must be able to calculate the distance D^i of all nodes from a goal node. As with the enumerated states representation, we use Dijkstra's algorithm. However, here it is not so straightforward because transitions between macro-states may be conditional. Therefore, when generating successors of a macro-state while running Dijkstra's algorithm, we must generate only those parts of macro-states that can transition to the current macro-state.

In finding successors to a current macro-state $currClust$, we iterate through each of the original macro-states C_i, referred to as "parent macro-states", and create a new macro-state $C_i' = C_i \cap R_{\rightarrow currClust}$. We then use the cost matrix created as described in Section 5.3.1 to assign C_i' a distance from $currClust$. Since $currClust$ has a distance from the goal assigned, this gives us a distance-to-goal estimate for C_i'.

Now if this were all we did, we could never reach a stopping point because we cannot identify empty macro-states (in this case, pruning is not accurate enough). However, we *can* keep track of whether macro-states are enclosed by other macro-states by keeping track of the conditions with which we have intersected the macro-state. Then, given two macro-states C_1 and C_2, C_1 is a sub-macro-state of C_2 if both of the following are true:

- C_1 and C_2 have the same parent macro-state
- The conditions with which we have intersected C_2 are a subset of the conditions with which we have intersected C_1.

Therefore, when we generate a macro-state C_1 that is a sub-macro-state of another macro-state C_2, we only keep it if the distance-to-goal assigned to C_1 is smaller than the distance assigned to C_2. Similarly, when we generate a super-macro-state C_1 of a macro-state C_2, if the distance assigned to C_1 is smaller than that assigned to C_2, we replace C_2 with C_1. This allows us to terminate the algorithm.

5.3.3 Value Iteration

Recall that at the lowest level of the hierarchy, we run a value iteration algorithm. However, now we are running value iteration not on primitive states, but on f-states. Because f-states have conditional transitions, a traditional value iteration algorithm does not work.

We modify value iteration similarly to how we modified Dijkstra's algorithm. We associate with every condition in the f-state's condition list a different value, but consider the "value" of the f-state to be the weighted average value of these conditions.

We then iterate through every f-state in the typical value iteration fashion. However, for each f-state, we calculate values *separately* for each condition in the condition list. Before discussing the full algorithm, we explain how we deal with one potential complication.

Consider an f-state s and condition c, describing the state set $s \wedge c$. It is possible that $s \wedge c$ may define a set for which those states that can transition to some f-state n

are a proper subset. In this case, technically we should split $s \wedge c$ on $R_{\rightarrow n}$. However, this can result in an exponential explosion of the number of conditions. Therefore, we allow each condition c to be split only once. To do this, we create two copies of the condition list. One of the copies we mark as "volatile" and one we mark as "constant". When we calculate the value for the volatile condition, if the best option is to take an action that is applicable for only a subset of the states, we allow that addition to the condition. When we calculate the value of a condition in the constant list, however, we require that actions be applicable to all states.

Now consider calculating the value associated with a transition from f-state s and condition c to some f-state n. Recall, however, that n also has a list of conditions with values associated with them. To calculate the value of transitioning to n, therefore, we take a weighted average of those conditions that maximally match c. Note that if c is TRUE, this is simply the value of n. As before, for the weighted average, we assume a uniform distribution over all primitive states.

In addition, when doing value iteration here, it is much harder to know if an f-state can transition to an f-state in another, non-goal macro-state. Therefore, we assign the "out state" a uniform reward rather than trying to estimate the distance of this state from the goal.

Using these procedures to calculate cost, distance to goal and value iteration, the full solver algorithm then proceeds as outlined in Section 3.2.

5.4 Interpreting the Hierarchical Policy

The clustering for the factored representation is not a partition. Therefore, a state may have different actions specified in the hierarchical policy. We choose which action to take for a given state s as follows:

1. Locate the macro-state including the state that has the shortest distance to goal. Let N be the macro-state that is specified as the next macro-state of this macro-state.

2. Identify macro-states $C_1, ..., C_M$ such that $s \in C_i$ and the next macro-state of C_i is N for all i.

3. For each macro-state C_i, find the conditions that maximally match s and choose the highest value/lowest cost condition. Let c be the condition with highest value/lowest cost of all of the $C_1, ..., C_M$.

4. If the level is greater than one, repeat this process from Step 2 with N set equal to the next macro-state of c. Else, return the action associated with c.

5.5 Preliminary Results

In studying the solver, one question that arises is why we bother clustering the states beforehand. After all, given a "clustering" that clusters all states into one macro-state described by the condition TRUE, the solver will produce a solution. However, similarly to the Det algorithm, this solution ignores the non-determinism of the domain. In this section we show that, in general, the clustering improves the accuracy of the solution.

5.5.1 Domains

We use two domains: the coffee domain and the tireworld domain. The P-PDDL descriptions of these domains are shown in A.2, but we give a brief description of each domain here:

- Coffee Domain: In the coffee domain, a robot is trying to bring coffee to a human. This domain has six variables, two of which (`has-umbrella` and `is-wet`) are actually irrelevant to the task.
- Tireworld Domain: In this domain a robotic car is trying to drive from point A to point B without getting a flat tire. The car has room to carry one spare tire and some locations have spare tires at them. At these locations, if the car is not carrying a spare, it can pick up one. The number of variables in this domain corresponds to the number of possible locations, which is specified in the problem files.

We use both of these domains because they represent different ways of looking at a factored MDP. In the coffee domain, the state variables refer to different characteristics of the robot. Any true/false combination of these variables is a valid state. In the tireworld domain, however, the state variables refer to the placement of a car. The only valid states are states where only *one* location is true. However, this is not explicitly stated anywhere in the domain; instead the solver does actually solve for cases of multiple vehicles and has to be able to pick out, from that solution, the best policy for the single vehicle.

Problem files for the coffee domain and tireworld domain are given in A.2. We use one coffee problem file and two tireworld problem files. The coffee problem file has 6 variables and 4 actions. The small tireworld file has 12 variables and 14 actions. The big tireworld file has 40 variables and 100 actions.

5.5.2 Results

In the coffee domain, the solver algorithm with no clustering is able to reach the optimal policy with an average reward of -1.71. However, in this domain clustering can actually hurts us. We created a clustering of 7 macro-states, but because the clustering is constrained to follow the upper level policy, from a few starting states, the agent will unnecessarily `get-umbrella` giving the clustering a value of -1.85.

In the tireworld domain, however, solving without clustering is generally worse than solving with the clustering. This is because, if the car does not have a spare tire, an optimal policy generally involves trying to get a spare tire. However, without the clustering, the solver never takes into account the probability that the car might lose a tire. In the small world, with a clustering of 14 macro-states, the solver finds the optimal policy from the specified start state and the car is able to reach the goal 83.3% of the time while, without, the car is able to reach the goal only 35.8% of the time.

In the larger problem, the optimal solution from the specified starting state is found by solving both with a clustering of 3 macro-states and without since the policy is a trivial one-step policy. However, in looking more carefully at these policies it is clear that if the car was to start, for example, in location n17, the policy with clustering specifies the action `loadtire` while the policy without clustering just has the car move. Therefore, in this case, clustering would give an advantage.

In this chapter, we have presented our algorithms for creating and solving an HMDP when the MDP was originally specified in factored form. We gave time bounds on the clustering algorithm and showed that it forms clusters of the minimized states of the reduced MDP. We also gave some preliminary results of our implementation, including a run on a world with over one trillion primitive states (40 state variables).

Chapter 6

Conclusions

6.1 Future Work

There are many directions in which we may take this work. We discuss some of them here.

6.1.1 Clustering Improvements

Our current clustering algorithms consider "adjacent" to be a binary value: either two states are connected or they are not. This can result in macro-states where a "connection" between two sub-states actually depends on an unlikely transition. Instead, we would like to cluster with a bias towards clustering together states with higher transition probability.

In the enumerated-state clustering, we may be able to do this in two ways. In the preprocessing step, we often have more than one primitive state that can be added to a macro-state at a time. By ordering these primitive states by their transition probability into the macro-state, we could create more tightly connected macro-states. Similarly, when calculating cycles of macro-states, any time there are two macro-states that the algorithm would consider equivalent, we can order them by transition probability.

In the factored case, we can introduce transition probabilities when picking initial start f-states by picking a start f-state with high transition probability to whichever previous start f-state with respect to which it is stable. As with the enumerated-state clustering, we could also use transition probabilities as a tie-breaker in creating the initial fg-connected cycles.

In addition, the "stopping point" for the enumerated-state clustering may not be best described by a number of macro-states or a maximum size of macro-states. For example, in the grid world clustering shown in Figure 4-2, there are a few squares that belong to singleton or very small clusters. This occurs because all of the surrounding clusters have reached maximal size, but, really, we would prefer that they be clustered with one of the nearby large macro-states. Future work could include research into a better condition for stopping the clustering algorithm.

6.1.2 Theoretical Work

At the moment, we have no theoretical error bounds on either the factored or the enumerated-state algorithm, but we believe that it should be possible to attain those bounds.

In addition, we do not have a good algorithmic method for setting the algorithmic parameters, namely S_{max}, C_{min}, and the NCP. In this work, we have used empirical methods for setting the parameters, but an explicit equation relating them to time and accuracy would allow for much better parameter selection.

6.1.3 Improvements for Factored Algorithms

In this thesis, we include no results comparing our factored algorithm to other algorithms. We need to optimize the factored algorithms code so that it can be compared against current factored algorithms.

We also could improve our factored algorithms to allow more interesting reward functions than $R(s, a) = -1$. The theoretical basis for this is present; it is a matter of implementing the lower-level cost function to deal with non-uniform rewards.

We may also be able to extend the factored solver algorithm to deal with disjunctive preconditions. Currently, we use the conjunctive preconditions to avoid satisfiability issues in deciding if a set of states matches a precondition for an action. However, it may be possible to overcome this restriction by use of a heuristic.

Lastly, we could improve the factored clustering algorithm using the concept of "exit f-states" defined similarly to primitive exit states. Then, rather than require all f-states of a macro-state to be connected, we could just require that all f-states reach an exit f-state.

6.1.4 Real-World Problems

All of our current results, for both enumerated-state MDPs and factored MDPs,, are in simulation. While simulation works well for comparing our algorithms to other current work in the field, it often does not give a good idea of how well the algorithm will work in the real world. Therefore, in future work, we would like to implement this algorithm outside of simulation.

In addition, for large problems, the algorithm is still too slow to be real-time algorithms. However, because we use a hierarchy, we could possibly change this simply by being clever about the order in which we solve our hierarchy. The clustering and the top-most level solution would still need to be done first. However, once the top-most level solution was computed, we could solve all the way down the hierarchy for the first step and begin carrying out that step while solving the rest of the problem. This may allow the algorithm to be fast enough that it could be done on-the-fly rather than having to compute all of the solutions off-line.

6.1.5 Extension to POMDPs

In working with MDPs, we have made the assumption that the world is fully observable. In general, especially in robotics, this is a poor assumption. Usually, there is uncertainty not only in the actions, but also in the state space. If we incorporate uncertainty about the state, but keep the Markovian assumption, we can model the world using a partially observable Markov decision process (POMDP). POMDPs are, in general, a much better model of the world than MDPs, with a much larger class of corresponding problems. However they are also much harder to solve than MDPs.

One method for solving a POMDP is to convert it to a "belief-state MDP". A belief-state MDP is an MDP where the states of the MDP are the *belief-states* of the POMDP. A belief state is actually a probability distribution over the states of the POMDP, representing the current possible states. How a belief state will change when a certain action is taken is deterministic. Therefore, by converting from POMDP states to belief states, we can convert from a POMDP to an MDP.

However, belief-state MDPs have, even for small POMDPs, extremely large, continuous state spaces. Therefore, most classic algorithms for solving MDPs do not work on belief-state MDPs.

This gives us two possible directions for extending our work to POMDPs: we could attempt to extend the algorithms to explicitly work with POMDPs or we could try to work with the belief state MDPs. Either one presents possibilities, as well as difficulties.

6.2 Conclusion

In this thesis we have presented methods for creating and solving Markov decision processes. We outlined a method for solving hierarchical MDPs based on a deterministic assumption and then introduced a clustering principle that guarantees that if a state can reach a goal state under an optimal policy, it can reach a goal state under the hierarchical policy.

We discussed our clustering and solving algorithms for enumerated-state and factored MDPs. We used the enumerated-state MDPs to compare against other clustering methods and show that, for our purposes, our clustering method works better than other common methods. We also showed that our algorithm introduces a significant speed-up over other algorithms with only a small reduction in optimality.

We have also implemented the algorithms for the factored MDP representation and we showed that in this case, it is possible to run our algorithm on very large state spaces.

Appendix A

Domains

A.1 Enumerated State Domains

.

A.1.1 Factory Domain

The factory domain consists of two objects A and B each of which has 5 attributes. There are also 10 actions corresponding to each of these attributes. The attributes/actions are CLEANA, CLEANB, SHAPEA, SHAPEB, DRILLA, DRILLB, PAINTA, PAINTB, JOINA, JOINB. The goal is to have all attributes true.

The dynamics of the actions for this domain are shown in Table A.1.

Action	Precondition	Effect (Probability)
CleanA	–	CLEANA (0.9) and ¬PAINTA, ¬CLEANA and ¬PAINTA (0.1)
CleanB	–	CLEANB (0.9) and ¬PAINTB, ¬CLEANB and ¬PAINTB (0.1)
ShapeA	¬PAINTA	SHAPEA (0.8), ¬SHAPEA (0.2)
ShapeB	¬PAINTB	SHAPEB (0.8), ¬SHAPEB
DrillA	SHAPEA	DRILLA (0.8), ¬DRILLA and ¬SHAPEA (0.2)
DrillB	SHAPEB	DRILLB (0.8), ¬DRILLB and ¬SHAPEB(0.2)
PaintA	CLEANA	PAINTA (0.8), ¬PAINTA and ¬CLEANA (0.2)
PaintB	CLEANB	PAINTB (0.8), ¬PAINTB and ¬CLEANB (0.2)
JoinA	SHAPEA and DRILLA and CLEANA	JOINA (0.7), ¬JOINA and ¬CLEANA (0.3)
JoinB	SHAPEB and DRILLB and CLEANB	JOINB (0.7), ¬JOINB and ¬CLEANB (0.3)

Table A.1: The dynamics of the factory domain

A.2 Factored Domains

P-PDDL descriptions for the factored domains.

A.2.1 Coffee

Coffee domain:

```
;; -*-lisp-*-
;; Coffee domain:
;;
;;   Dearden Richard, and Craig Boutilier.  1997.  Abstraction and
;;   approximate decision-theoretic planning.  Artificial
;;   Intelligence, 89(1-2):219-283.
;;   Modified by Jennifer Barry to remove additive rewards and
;;   disjunctive preconditions.


(define (domain coffee)
  (:requirements :negative-preconditions
                 :conditional-effects :probabilistic-effects :rewards)
  (:predicates (in-office) (raining) (has-umbrella) (is-wet)
               (has-coffee) (user-has-coffee))
  (:action move
           :effect (and (when (in-office)
                          (probabilistic 0.9 (not (in-office))))
                        (when (not (in-office))
                          (probabilistic 0.9 (in-office)))
                        (when (and (raining) (not (has-umbrella)))
                          (probabilistic 0.9 (is-wet)))
                        (decrease (reward) 1)))
  (:action buy-coffee
           :effect (and (when (not (in-office))
                          (probabilistic 0.8 (has-coffee)))
                        (decrease (reward) 1)))
  (:action get-umbrella
           :effect (and (when (in-office)
                          (probabilistic 0.9 (has-umbrella)))
                        (decrease (reward) 1)))
  (:action deliver-coffee
           :effect (and (when (and (in-office) (has-coffee))
                          (probabilistic 0.8 (and (user-has-coffee)
                                                  (not (has-coffee)))
                                         0.2 (and (probabilistic
                                                   0.5 (not (has-coffee))))))))
```

```
                        (when (and (not (in-office)) (has-coffee))
                          (and (probabilistic 0.8 (not (has-coffee)))))
                        (decrease (reward) 1))))
```

Coffee problem:

```
(define (problem coffee)
        (:domain coffee)
        (:init (probabilistic 0.5 (in-office))
               (probabilistic 0.5 (raining))
               (probabilistic 0.5 (has-umbrella))
               (probabilistic 0.5 (is-wet))
               (probabilistic 0.5 (has-coffee))
               (probabilistic 0.5 (user-has-coffee)))
        (:goal (user-has-coffee)))
```

A.2.2 Tireworld

Tireworld domain:

```
;; -*_lisp-*-
;;  Authors: Michael Littman and David Weissman
;;  Modified: Blai Bonet for IPC 2006 ;;;

(define (domain tire)
  (:requirements :typing :strips :equality :probabilistic-effects :fluents :rewards)
  (:types location)
  (:predicates (vehicle-at ?loc - location) (spare-in ?loc - location)
               (road ?from - location ?to - location) (not-flattire) (hasspare))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicle-at ?from) (road ?from ?to) (not-flattire))
    :effect (and (vehicle-at ?to) (not (vehicle-at ?from))
                 (probabilistic 2/5 (not (not-flattire)))
                 (decrease (reward) 1))
  )
  (:action loadtire
    :parameters (?loc - location)
    :precondition (and (vehicle-at ?loc) (spare-in ?loc))
    :effect (and (hasspare)
                 (not (spare-in ?loc))
                 (decrease (reward) 1))
  )
  (:action changetire
    :precondition (hasspare)
    :effect (and (probabilistic 1/2 (and (not (hasspare)) (not-flattire)))
                 (decrease (reward) 1))
  )
)
```

Small problem:

```
(define (problem tire_small)
        (:domain tire)
        (:objects n0 n1 n2 n3 n4 - location)
        (:init (vehicle-at n1)
               (road n0 n1) (road n1 n0)
               (road n1 n2) (road n2 n1)
               (road n2 n3) (road n3 n2)
               (road n3 n4) (road n4 n3)
               (spare-in n1)
               (not-flattire)
```

```
        )
        (:goal (vehicle-at n4))
)
```

Large problem

```
(define (problem tire_19_0_28845)
        (:domain tire)
        (:objects n0 n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14
                  n15 n16 n17 n18 - location)
         (:init (vehicle-at n12)
                (road n0 n8) (road n8 n0)
                (road n1 n2) (road n2 n1)
                (road n1 n3) (road n3 n1)
                (road n1 n6) (road n6 n1)
                (road n1 n7) (road n7 n1)
                (road n1 n11) (road n11 n1)
                (road n1 n13) (road n13 n1)
                (road n2 n18) (road n18 n2)
                (road n3 n7) (road n7 n3)
                (road n3 n9) (road n9 n3)
                (road n3 n12) (road n12 n3)
                (road n3 n18) (road n18 n3)
                (road n4 n9) (road n9 n4)
                (road n5 n7) (road n7 n5)
                (road n6 n8) (road n8 n6)
                (road n6 n17) (road n17 n6)
                (road n7 n13) (road n13 n7)
                (road n7 n16) (road n16 n7)
                (road n8 n10) (road n10 n8)
                (road n8 n15) (road n15 n8)
                (road n8 n17) (road n17 n8)
                (road n8 n18) (road n18 n8)
                (road n9 n15) (road n15 n9)
                (road n9 n16) (road n16 n9)
                (road n10 n12) (road n12 n10)
                (road n10 n16) (road n16 n10)
                (road n11 n13) (road n13 n11)
                (road n12 n15) (road n15 n12)
                (road n12 n17) (road n17 n12)
                (road n12 n18) (road n18 n12)
                (road n13 n14) (road n14 n13)
                (road n13 n16) (road n16 n13)
                (road n13 n18) (road n18 n13)
                (road n14 n15) (road n15 n14)
```

```
                (road n14 n16) (road n16 n14)
                (road n14 n17) (road n17 n14)
                (road n15 n17) (road n17 n15)
                (road n16 n17) (road n17 n16)
                (road n16 n18) (road n18 n16)
                (road n17 n18) (road n18 n17)
                (spare-in n4)
                (spare-in n5)
                (spare-in n6)
                (spare-in n9)
                (spare-in n10)
                (spare-in n11)
                (spare-in n12)
                (spare-in n13)
                (spare-in n17)
                (spare-in n18)
                (not-flattire)
        )
        (:goal (vehicle-at n3))
)
```

Appendix B

Theoretical Analysis of the Deterministic Assumption for Enumerated-State MDPs

We have done a simple analysis of how much our cost estimates we find using our deterministic assumption differ from the actual cost of moving from one primitive state to another. We hope this will lead to a bound on the deviation between the value of the policies found by our algorithm and the optimal policy.

We begin by bounding the difference between our estimated cost C^0 and the actual cost of transitioning between two states..

Costs First, let $C(i, j)$ be the expected cost to get from primitive state i to primitive state j under the optimal policy for going from i to j. To compute this, we let j be an absorbing goal state and solve for the optimal policy π_j. Then $C(i, j) = V^{\pi_j}(i)$.

Now, our approximate costs. This is the expected cost to get from i to j, if (1) we take the action a that is most likely to get us there, and (2) whenever that action fails to take us to j, it leaves us at i:

$$C^0(i, j) = \min_a \frac{-R(i, a)}{T(i, a, j)}. \tag{B.1}$$

We want to give a bound on $|C(i, j) - C^0(i, j)|$. The first thing is that we do not know whether $C^0(i, j)$ is greater or less than $C(i, j)$. So we need both an upper and lower bound on $C(i, j)$. The lower bound is, in fact, trivial:

$$C(i, j) \geq -R_{max}. \tag{B.2}$$

(Recall that for negative MDPs, R_{max} will be a small negative number, making $-R_{max}$ a *lower* bound).

To get an upper bound, let M_j be the MDP considering j to be a goal state and assume π is *any* policy defined on M_j. Then, since $C(i, j)$ is defined using the *optimal*

policy to go from i to j we have that

$$C(i,j) \leq -V^{\pi}(i). \tag{B.3}$$

Now let $\mathbf{Q}_{\pi}$ be the transient transition matrix of M_j under π. A transition matrix for a policy π is an $|S| \times |S|$ matrix where the ijth entry corresponds to the probability of transitioning from primitive state i to primitive state j using action $\pi(i)$. The rows of a transition matrix sum to 1. A "transient transition matrix" is an $|S| - |G| \times |S| - |G|$ matrix formed from the transition matrix by removing the gth row and column of the transition matrix if g is an absorbing goal state. Note that the rows of the transient transition matrix may not sum to 1.

If we start at i, the expected number of transitions $S^{\pi}(i)$ before entering a goal state is

$$S^{\pi}(i) = \sum_{n=0}^{\infty} \sum_{j} [\mathbf{Q}_{\pi}^{n}]_{ij}. \tag{B.4}$$

Therefore, the expected cost of starting at i must be upper bounded by

$$V^{\pi}(i) \leq -R_{min} S^{\pi}(i) = -R_{min} \sum_{n=0}^{\infty} \sum_{j} [\mathbf{Q}_{\pi}^{k}]_{ij}. \tag{B.5}$$

Now let the spectral decomposition of $\mathbf{Q}_{\pi}$ be

$$\mathbf{Q}_{\pi} = \mathbf{A}\mathbf{D}\mathbf{A}^{-1}. \tag{B.6}$$

where $\mathbf{D}$ is diagonal and, without loss of generality, arranged in descending order by absolute value (so that $|\mathbf{D}_{00}|$ is the eigenvalue of $\mathbf{Q}_{\pi}$ with greatest magnitude). Then

$$\begin{aligned} \sum_{j} [\mathbf{Q}_{\pi}^{n}]_{ij} &= \sum_{j} [\mathbf{A}\mathbf{D}^{n}\mathbf{A}^{-1}]_{ij} \\ &= \sum_{j} \sum_{k} \mathbf{A}_{ik} \sum_{m} [\mathbf{D}]_{km}^{n} \mathbf{A}_{mj}^{-1}. \end{aligned} \tag{B.7}$$

Using that fact that $\mathbf{D}$ is diagonal, we find

$$\begin{aligned} \sum_{j} [\mathbf{Q}_{\pi}^{n}]_{ij} &= \sum_{j} \sum_{k} \mathbf{A}_{ik} \mathbf{D}_{kk}^{n} \mathbf{A}_{kj}^{-1} \\ &\leq |\mathbf{D}_{00}|^{n} \sum_{j} \sum_{k} \mathbf{A}_{ik} \mathbf{A}_{kj}^{-1} \\ &= |\mathbf{D}_{00}^{n}| \sum_{j} [AA^{-1}]_{ij} \\ &= |\mathbf{D}_{00}^{n}|. \end{aligned} \tag{B.8}$$

Therefore, we find an upper bound on $C(i,j)$ of

$$C(i,j) \leq -R_{min} \sum_{n=0}^{\infty} |\mathbf{D}_{00}|^n = \frac{-R_{min}}{1-|\mathbf{D}_{00}|}. \tag{B.9}$$

Thus we can bound $|C(i,j) - C^0(i,j)|$ as

$$|C(i,j) - C^0(i,j)| \leq \max\left(\frac{-R_{min}}{1-|\mathbf{D}_{00}|} - C^0(i,j), C^0(i,j) - -R_{max}\right) \tag{B.10}$$

where the bound holds if $|\mathbf{D}_{00}|$ is the greatest magnitude eigenvalue of the transient transition matrix of M_j under some policy.

Distances Once we have these costs, we can define shortest paths from all states to a goal state. Let $D(i)$ be the shortest distance to a goal state from primitive state i, using the costs $C(i,j)$, and let $\hat{D}(i)$ be the shortest distance to a goal state from primitive state i using the approximate costs $C^0(i,j)$. We denote the policy that acts greedily with respect to $-D$ as $\hat{\pi}_{-D}$ and the one that acts greedily with respect to $-\hat{D}$ as $\hat{\pi}_{-\hat{D}}$. We can find a bound on $|D(i) - \hat{D}(i)|$.

Again, we need to separate out the cases where $\hat{D}(i) > D(i)$ and $\hat{D}(i) < D(i)$. We will first consider bounding $D(i) - \hat{D}(i)$.

When dealing with these types of hierarchical deterministic policies, we have a well-defined idea of the next state. Therefore, for a hierarchical policy $\hat{\pi}$ we define a next state function $N_{\hat{\pi}}$ such that $N_{\hat{\pi}}(i) = j$ if j is the next specified state of i. We also let $N^k_{\hat{\pi}}(i)$ be the state k steps after i. So $N^0_{\hat{\pi}}(i) = i$, $\hat{N}^1_{\hat{\pi}}(i) = N_{\hat{\pi}}(i) = j$, $N^2_{\hat{\pi}}(i) = N_{\hat{\pi}}(N_{\hat{\pi}}(i)) = N_{\hat{\pi}}(j)$, etc. In addition, we let $L_{\hat{\pi}}(i)$ be the number of steps to the goal from state i following policy $\hat{\pi}$. Then

$$\begin{aligned} D(i) - \hat{D}(i) = &\sum_{j=0}^{L_{\hat{\pi}_{-D}}(i)-1} C(N^j_{\hat{\pi}_{-D}}(i), N^{j+1}_{\hat{\pi}_{-D}}(i)) - \\ &\sum_{k=0}^{L_{\hat{\pi}_{-\hat{D}}}(i)-1} C^0(N^k_{\hat{\pi}_{-\hat{D}}}(i), N^{k+1}_{\hat{\pi}_{-\hat{D}}}(i)) \end{aligned} \tag{B.11}$$

and we want an upper bound on this difference. Note that, since $\hat{\pi}_{-D}$ is a shortest path

$$\sum_{j=0}^{L_{\hat{\pi}_{-D}}(i)-1} C(N^j_{\hat{\pi}_{-D}}(i), N^{j+1}_{\hat{\pi}_{-D}}(i)) \leq \sum_{k=0}^{L_{\hat{\pi}_{-\hat{D}}}(i)-1} C(N^k_{\hat{\pi}_{-\hat{D}}}(i), N^{k+1}_{\hat{\pi}_{-\hat{D}}}(i)). \tag{B.12}$$

Therefore

$$\begin{aligned} D(i) - \hat{D}(i) \leq & \sum_{k=0}^{L_{\hat{\pi}_{-\hat{D}}}(i)-1} C(N^k_{\hat{\pi}_{-\hat{D}}}(i), N^{k+1}_{\hat{\pi}_{-\hat{D}}}(i)) - C^0(N^k_{\hat{\pi}_{-\hat{D}}}(i), N^{k+1}_{\hat{\pi}_{-\hat{D}}}(i)) \\ & \leq L_{\hat{\pi}_{-\hat{D}}}(i) \max_k \left[C(N^k_{\hat{\pi}_{-\hat{D}}}(i), N^{k+1}_{\hat{\pi}_{-\hat{D}}}(i)) - C^0(N^k_{\hat{\pi}_{-\hat{D}}}(i), N^{k+1}_{\hat{\pi}_{-\hat{D}}}(i)) \right] \\ & \leq L_{\hat{\pi}_{-\hat{D}}}(i) \left(\frac{-R_{min}}{1 - |\mathbf{D}_{00}|} - \min_{s,s'} C^0(s, s') \right). \end{aligned} \tag{B.13}$$

Note that our solver solves for $L_{\hat{\pi}_{-\hat{D}}}$, but it can also be bounded above by the leading eigenvalue of the transient transition matrix under $\hat{\pi}_{-\hat{D}}$.

The case of $\hat{D}(i) - D(i)$ can be approached similarly:

$$\begin{aligned} \hat{D}(i) - D(i) = & \sum_{k=0}^{L_{\hat{\pi}_{-\hat{D}}}(i)-1} C^0(N^k_{\hat{\pi}_{-\hat{D}}}(i), N^{k+1}_{\hat{\pi}_{-\hat{D}}}(i)) - \sum_{j=0}^{L_{\hat{\pi}_{-D}}(i)-1} C(N^j_{\hat{\pi}_{-D}}(i), N^{j+1}_{\hat{\pi}_{-D}}(i)) \\ \leq & \sum_{j=0}^{L_{\hat{\pi}_{-D}}(i)-1} C^0(N^j_{\hat{\pi}_{-D}}(i), N^{j+1}_{\hat{\pi}_{-D}}(i)) - \sum_{j=0}^{L_{\hat{\pi}_{-D}}(i)-1} C(N^j_{\hat{\pi}_{-D}}(i), N^{j+1}_{\hat{\pi}_{-D}}(i)) \\ & \leq L_{\hat{\pi}_{-D}}(i) \max_j C(N^j_{\hat{\pi}_{-D}}(i), N^{j+1}_{\hat{\pi}_{-D}}(i)) - C^0(N^j_{\hat{\pi}_{-D}}(i), N^{j+1}_{\hat{\pi}_{-D}}(i)) \\ & \leq |S| \left(\max_{s,s'} C^0(s, s') - -R_{max} \right). \end{aligned} \tag{B.14}$$

The bound is then

$$\begin{aligned} |D(i) - \hat{D}(i)| \leq \max \{ & L_{\hat{\pi}_{-\hat{D}}}(i) \left(\frac{-R_{min}}{1 - |\mathbf{D}_{00}|} - \min_{s,s'} C^0(s, s') \right), \\ & |S| \left(\max_{s,s'} C^0(s, s') - -R_{max} \right) \Big\}. \end{aligned} \tag{B.15}$$

Bibliography

[Adams, 1979] Douglas Adams. *The Hitchhiker's Guide to the Galaxy.* Pan Books, London, 1979.

[Bakker *et al.*, 2005] Bram Bakker, Zoran Zivkovic, and Ben Krose. Hierarchical Dynamic Programming for Robot Path Planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3720–3725, 2005.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, Massachusetts, 1996.

[Bertsekas, 1995] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, Belmont, Massachusetts, 1995. Volumes 1 and 2.

[Bonet and Geffner, 2009] Blai Bonet and Hector Geffner. Solving POMDPs: RTDP-Bel vs. Point-based Algorithms. In *21st Int. Joint. Conf. on Artificial Intelligence (IJCAI)*, Pasadena, California, 2009. To appear.

[Boutilier *et al.*, 1995] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Exploiting Structure in Policy Construction. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, pages 1104–1113, 1995.

[Coifman *et al.*, 2005] R. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, B. Nadler, F. Warner, and S. W. Zucker. Geometric Diffusions as a Tool for Harmonic Analysis and Structure Definition of Data: Multiscale Methods. *PNAS*, 102(21):7432–7437, MAY 2005.

[Contributors, 2008] Wikipedia Contributors. Observable Universe. Wikipedia, http://en.wikipedia.org/wiki/Observable_universe, February 2008.

[Dean and Givan, 1997] Thomas Dean and Robert Givan. Model Minimization in Markov Decision Processes. In *AAAI*, pages 106–111, Cambridge, MA, 1997.

[Dean *et al.*, 1997] Thomas Dean, Robert Givan, and Sonia Leach. Model Reduction Techniques for Computing Approximately Optimal Solutions for Markov decision processes. In *Proc. of the 13th Conference on Uncertainty in Artificial Intelligence*, pages 124–131, San Francisco, CA, 1997.

[Dean *et al.*, 1998] Thomas Dean, Thomas Dean, Kee eung Kim, Kee eung Kim, Robert Givan, and Robert Givan. Solving stochastic planning problems with large state and action spaces. In *In Proc. Fourth International Conference on Artificial Intelligence Planning Systems*, pages 102–110. AAAI Press, 1998.

[Dearden and Boutilier, 1997] Richard Dearden and Craig Boutilier. Abstraction and Approximate Decision-Theoretic Planning. *Artificial Intelligence*, 89:219–283, January 1997.

[Dietterich, 1998] Thomas G. Dietterich. The MAXQ Method for Hierarchical Reinforcement Learning. In *ICML*, pages 118–126, San Francisco, 1998.

[Digney, 1996] Bruce L. Digney. Emergent Hierarchical Control Structures: Learning Reactive / Hierarchical Relationships in Reinforcement Environments. In *Proceedings of the Fourth Conference on the Simulation of Adaptive Behavior: SAB 98*, 1996.

[Givan *et al.*, 2003] Robert Givan, Thomas Dean, and Matthew Greig. Equivalence Notions and Model Minimization in Markov Decision Processes. *Artificial Intelligence*, 142(1-2):163–223, 2003.

[Hauskrecht *et al.*, 1998] Milos Hauskrecht, Nicolas Meuleau, Craig Boutilier, Leslie Pack Kaelbling, and Thomas Dean. Hierarchical solution of Markov decision processes using macroactions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, Madison, Wisconsin, 1998.

[ICAPS, 2009] ICAPS. International Conference on Automated Planning and Scheduling. http://www.icaps-conference.org/, 2009.

[Johnson, 1975] Donald B. Johnson. Finding All the Elementary Circuits if a Directed Graph. *SIAM J. Comput.*, 4(1), MARCH 1975.

[Jonsson and Barto, 2006] Anders Jonsson and Andrew Barto. Causal Graph Based Decomposition of Factored MDPs. *Journal of Machine Learning Research*, 7:2259–2301, 2006.

[Kim and Dean, 2001] Kee-Eung Kim and Thomas Dean. Solving Factored MDPs Using Non-Homogenous Partitions. In *Proc. International Joint Conference on Artificial Intelligence*, pages 683–689, San Francisco, CA, 2001. Morgan Kaufmann Publishers.

[Kim and Dean, 2002] Kee-Eung Kim and Thomas Dean. Solving Factored MDPs with Large Action Space Using Algebraic Decision Diagrams. In *Proc. of the 7th Pacific Rim International Conference on Artificial Intelligence*, pages 80–89. Springer, 2002.

[Lane and Kaelbling, 2002] Terran Lane and Leslie Pack Kaelbling. Nearly Deterministic Abstractions of Markov Decision Processes. In *AAAI*, Edmonton, 2002.

[Maggioni and Mahadevan, 2006a] Mauro Maggioni and Sridhar Mahadevan. A Multiscale Framework for Markov Decision Processes using Diffusion Wavelets. Technical Report 2006-36, University of Massachusetts, July 2006.

[Maggioni and Mahadevan, 2006b] Mauro Maggioni and Sridhar Mahadevan. Fast Direct Policy Evaluation using Multiscale Analysis of Markov Diffusion Process. In *ICML*, Pittsburgh, 2006.

[Mahadevan, 2008] Sridhar Mahadevan. *Representation Discovery Using Harmonic Analysis*. Morgan and Claypool Publishers, 2008.

[McGovern and Barto, 2001] A. McGovern and A. Barto. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. *ICML*, pages 361–368, 2001.

[Mehta *et al.*, 2008] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. Automatic Discovery and Transfer of MAXQ Hierarchies. In *Proceedings of the 25th International Conference on Machine Learning*, Helinski, Finland, 2008.

[Parr and Russell, 1997] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Neural Information Processing Systems*, 1997.

[Puterman, 1994] Martin L. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, 1994.

[Russell and Norvig, 2003] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, New Jersey, second edition, 2003.

[Shi and Malik, 2000] Jianbo Shi and Jitendra Malik. Normalized Cuts and Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2000.

[Simsek *et al.*, 2005] Ozgur Simsek, Alicia P. Wolfe, and Andrew G. Barto. Identifying Useful Subgoals in Reinforcement Learning by Graph Partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*, Bonn, Germany, 2005.

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

[Sutton *et al.*, 1999] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, 1999.

[Younes and Littman, 2004] Hakan L. S. Younes and Michael L. Littman. PPDDL 1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical Report CMU-CS-04-167, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2004.