Hierarchical Solution of Large Markov Decision Processes

Jennifer Barry and **Leslie Pack Kaelbling** and **Tomás Lozano-Pérez**
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
{jbarry,lpk,tlp}@csail.mit.edu

Abstract

This paper presents an algorithm for finding approximately optimal policies in very large Markov decision processes by constructing a hierarchical model and then solving it. This strategy sacrifices optimality for the ability to address a large class of very large problems. Our algorithm works efficiently on enumerated-states and factored MDPs by constructing a hierarchical structure that is no larger than both the reduced model of the MDP and the regression tree for the goal in that MDP, and then using that structure to solve for a policy.

1 Introduction

Our goal is to solve a large class of very large Markov decision processes (MDPs), necessarily sacrificing optimality for feasibility. We apply two types of leverage to the problem: we shorten the horizon using an automatically generated temporal hierarchy and we reduce the size of the state space through state aggregation.

It is widely believed that hierarchical decomposition is a key to solving very large planning problems. However finding an appropriate hierarchy has proved challenging. Some approaches (Jonsson and Barto 2006; Mehta et al. 2008) operate over a relatively long time-frame to learn hierarchies that pay off over several different, but related problems. Others, as we will do, try to solve one large MDP quickly by aggregating together local states and assigning them the same sub-goal (Teichteil-Königsbuch and Fabiani 2005; Wu, Kayanam, and Givan 2008) or by aggregating together states that behave "similarly" under most actions (Givan, Dean, and Greig 2003; Kim and Dean 2002). Aggregating nearby states is most effective when a plan from a known starting state is needed: when trying to find a policy, it is often the case that states cannot reach their local sub-goal under the optimal policy, resulting in a policy in which some states cannot reach any goal state. Aggregating states with similar behavior can be difficult if there are few states in the domain that behave similarly: either the time taken to find the solution is very long or the solution is very inaccurate.

We combine the two approaches. We create potential cluster members, which we will call aggregate states, each of which is a set of primitive states, guided by a truncated regression algorithm. We ensure that each aggregate state contains a collection of primitive states that behave identically under the transitions to other members of the cluster. However, we do not require that primitive states contained in an aggregate state behave identically under *all* transitions, resulting in significantly smaller problem size than former approaches to this type of aggregation. We also require that all aggregate states in a cluster can transition easily among themselves, guaranteeing that any sub-goal in the cluster can be reached. Regressing factored MDPs has been used previously to approximate value functions (Boutilier, Dearden, and Goldszmidt 2000), but not to create a hierarchy.

Once we have the hierarchy, we use a deterministic approximation to solve the upper levels quickly by finding deterministic shortest paths. Different ways of determinizing MDPs have been explored (Lane and Kaelbling 2002; Yoon et al. 2008), although not for generating cost estimates between macro-states. At the bottom level, we treat each cluster as a small sub-MDP and run value iteration, generating a policy that is more robust than can be obtained from algorithms that use a purely deterministic approximation.

In this work, we focus on two types of MDPs: those that are specified by listing every state in the domain ("enumerated-states MDPs") and those that can be specified by a factored representation ("factored MDPs"). We begin by describing our conception of a hierarchical model of an MDP and how we can create and solve this model for enumerated-states MDPs. We then show how we can adapt the algorithm to the factored representation.

2 Hierarchical Model

A Markov decision process (MDP) is defined by $\langle S, A, T, R\rangle$, where S is a finite set of states, A is a finite set of actions, T is the transition model with $T(i', a, j')$ specifying the probability of a transition to i' given that the system starts in state j' and selects action a, and R is the reward model with $R(i', a)$ specifying the real-valued reward of taking action a in state i'. In addition, we assume a pre-specified set of *goal states*, $G \subset S$. Goal states are zero-reward absorbing states: for every $g \in G$, $T(g, a, g) = 1$ and $R(g, a) = 0$, for all a. Further, we assume that all other reward values are strictly negative. We solve this problem under the undiscounted total reward criterion, making it a 'stochastic shortest path' problem. Any MDP can be transformed into an 'equivalent' stochastic shortest path problem, which can then be solved to produce the optimal policy for the original MDP (Bertsekas and Tsitsiklis 1996).

Algorithm 1

Input: S^{l-1}: level $l-1$ states, A: primitive actions, T: transition function, G: primitive goal states

Output: A g-connected clustering of level l macro-states

ESCLUSTER(S^{l-1}, A, T, G)

1 $S^l \leftarrow \{\{i'\} \mid i' \in S^{l-1}\}$

2 *// create "goal macro-state" for level* 1
if $l = 1, g \leftarrow \{i' \mid i' \in G\}, S^1 \leftarrow \big(S^1 \setminus \{\{i'\} \mid i' \in G\}\big) \cup g$

3 **else** $g \leftarrow \{l-1$ goal macro-state$\}$ *// goal state already exists*

4 $Adj^l \leftarrow$ ADJMATRIX(S^l, A, T) *// adjacency defn in Sec. 2*

5 set g adjacent to every state

6 **while** $|S^l| >$ MINCLUST$_{ES}$ and $S^l_{\max} <$ MAXSIZE$_{ES}$

7 $\quad\{y_1, y_2, ..., y_n\} \leftarrow$ FINDCYCLE(S^l, Adj^l)

8 $\quad Y \leftarrow \{y_1, ..., y_n\} \setminus g$

9 $\quad$create new macro-state $u \leftarrow Y$

10 $\quad S^l \leftarrow \big(S^l \setminus Y\big) \cup u$, remove Y from Adj^l and add u

11 $\quad$**if** $g \notin \{y_1, ..., y_n\}$

12 $\quad\quad$**for** i adjacent to some $y \in Y$, set i adjacent to u

13 $\quad$**else** set only g adjacent to u

14 $\quad$**for** i s.t. $\exists y \in Y$ adjacent to i, set u adjacent to i

15 **return** S^l

From the input MDP, we construct and then solve a *hierarchically determinized MDP* (HDMDP). An HDMDP with L levels is given by a depth-L *tree*. The leaves of the tree, at level 0, are the states of the original MDP, referred to as *primitive states*. Internal nodes of the tree represent (possibly overlapping) sets of nodes at the lower levels. We refer to nodes of the HDMDP as macro-states. The set of macro-states at level l is represented by S^l.

The solution process computes a *hierarchical policy* π with L levels, each of which prescribes behavior for each level l state. At levels $l > 0$, the policy π^l maps each level l macro-state i to some other level l macro-state j, signifying that when the system is in a primitive state contained in i it should attempt to move to some primitive state in j. At level 0, the policy π^0 is a standard MDP policy mapping the primitive states to primitive actions.

At the primitive level, a state i' is *adjacent* to a state j' if there is some action a such that $T(j', a, i') > 0$. At levels $l > 0$, a macro-state i is adjacent to a macro-state j if there is some $i' \in i$ and $j' \in j$ such that i' is adjacent to j'. A state j is *reachable* from a state i if j is adjacent to i or j is adjacent to some state k which is reachable from i. If $i' \in i$ is a level $l-1$ sub-state of i then a level $l-1$ state j' is reachable from i' if j' is adjacent to some state $k' \in i$ and k' is reachable from i' without leaving i.

3 Enumerated-States MDPs

3.1 Clustering Algorithm

We begin by discussing how we create and solve the hierarchical model for an enumerated-states MDP. We view creating the hierarchical model as clustering: macro-states at level l of the tree are clusters of level $l-1$ states. There are many plausible criteria for clustering states of an MDP, but we base our algorithm on one tenet: we want a structure in which every state that could reach a goal state in the flat MDP can reach a goal state under some hierarchical policy.

This criterion is *not* guaranteed by an arbitrary hierarchy and the type of hierarchical policy described in Section 2. That policy requires all sub-states of macro-state i at level l to find a path through i to some sub-state of $\pi^l(i)$. In a hierarchy where there is no level l state reachable from all sub-states of i, there is no hierarchical policy under which every sub-state of i can reach a goal state. To avoid such hierarchies, we require that, at each level, all macro-states be *g-connected*. A set U of macro-states with goal macro-state g is g-connected if there exists a policy $\pi : U \rightarrow U$ such that: (1) $\forall i \in U$, i can reach g under π, and (2) $\forall i \in U$, for each $i' \in i$ that can reach a goal state in the flat MDP, there exists $j' \in \pi(i)$ s.t. j' is reachable from i'.

To create g-connected macro-states at level l from a set of $l-1$ macro-states, we run ESCLUSTER shown in Algorithm 1, which creates macro-states consisting of cycles of level $l-1$ states after setting the level l goal macro-state adjacent to all other level l macro-states. Setting the goal macro-state adjacent to all other states allows domains that contain few cycles to still be viewed hierarchically by grouping sets of states that are close together and lead to the goal.

Theorem 1: ESCLUSTER creates a g-connected clustering.

Proof Sketch: Each level l macro-state i is composed of a cycle of level $l-1$ states. If this is a true cycle, then all sub-states of i can reach all other sub-states in i and therefore a sub-state in any level l macro-state adjacent to i. Thus, in this case, all sub-states of i can comply with any policy that maps i to an adjacent macro-state. If i is composed of a "cycle" that goes through the goal macro-state g, all sub-states of i will be able to reach g. In this case, all sub-states of i will be able to comply with a policy that maps i to g. There is one subtlety: if i is composed of a cycle that goes through g, all sub-states of i can reach g, but may not be able to reach all macro-states adjacent to i. We acknowledge this in line 13 by marking only g as adjacent to i. For a detailed proof of this and all other theorems in this paper see (Barry, Kaelbling, and Lozano-Pérez 2010).

The complexity of ESCLUSTER is dominated by finding cycles, which is worst-case quadratic, but can be linear in domains where many states can reach a goal state.

Theorem 2: If a fraction p of the states in the MDP can reach a goal state, ESCLUSTER terminates in time $O(p|S| + (1-p)p|S|^2)$ where $|S|$ is the size of the state space.

ESCLUSTER relies on two parameters, MINCLUST$_{ES}$ and MAXSIZE$_{ES}$, defining the minimum number of macro-states allowed and the maximum size of those macro-states. These parameters can be set to control the time/accuracy trade-off of the algorithm, as we will discuss in Section 3.3.

3.2 Solver

The hierarchical model created in ESCLUSTER is input for a solver that uses the g-connectedness to quickly find an approximate solution for the MDP. In solving, we approximate the cost of transitioning between upper-level macro-states as deterministic. This allows us to find policies for $l > 0$ quickly using a deterministic shortest path algorithm.

Algorithm 2
Input: $S^0, ..., S^{L-1}$: hierarchical model, A: primitive actions, T: transition function, R: reward function, G: primitive goal states
Output: A hierarchical policy for $S^0, ..., S^{L-1}$

UPWARDPASS($S^0, ..., S^{L-1}, A, T, R$)
1 **for** $l = 0$ **to** $L - 1$
2 **for** $i \in S^l$
3 **for** j adjacent to i
4 **if** $l = 0$, $C^0(i,j) \leftarrow \min_{a \in A} -\frac{R(i,a)}{T(i,a,j)}$
5 **else** $C^l(i,j) \leftarrow$
$\frac{1}{|i|} \sum_{i' \in i} \min_{j' \in j} [\text{DIJKSTRA}(i', j', C^{l-1})]$
6 **return** C

DOWNWARDPASS($S^0, ..., S^{L-1}, A, T, R, G, C$)
1 **for** $l = L - 1$ **to** 1
2 **for** $i' \in S^l$ contained in $i \in S^{l+1}$
3 **if** $l = L - 1$, $g \leftarrow$ level $L - 1$ goal macro-state
4 **else** $g \leftarrow \pi^{l+1}(i)$
5 $\pi^l(i') \leftarrow$
$\arg\min_{j' \in S^l} C^l(i', j') + \text{DIJKSTRA}(j', g, C^l)$
6 **for** $i \in S^1$
7 $M \leftarrow \text{CREATEMDP}(i, \pi^1(i), A, T, R, \Delta)$
8 $\pi_i^0 \leftarrow \text{VALUEITERATION}(M)$
9 **for** $i' \in S^0$, $\pi^0(i') \leftarrow \pi^0_{\arg\min_{\{i \in S^1 | i' \in i\}} D^1(i)}(i')$
10 **return** π

We run the algorithm in two passes as shown in Algorithm 2. In UPWARDPASS, we compute an approximation for the cost of transitioning between two macro-states. We assume that, at the primitive level, any action a taken in state i with the goal of ending in state j does make a transition to j with probability $T(i, a, j)$; but that with the remaining probability mass, it stays in state i. Executing such an action a repeatedly will, in expectation, take $1/T(i, a, j)$ steps to move to state j, each of which costs $-R(i, a)$. We can select whichever action would minimize this cost, yielding the cost estimate shown on line 4 of UPWARDPASS. Once we have level 0 costs, we can solve deterministic shortest path problems to compute costs at all levels.

Next, we execute DOWNWARDPASS to find the hierarchical policy π. At the top levels, we use a deterministic shortest path algorithm to assign the policy. At level 0, rather than using the expected costs to solve a shortest-paths problem, we take the actual transition probabilities over the primitive states into account. In order to do this, we construct an MDP that represents the problem of moving from a macro-state i to a macro-state $\pi^1(i)$. Most of the transition probabilities and reward values have already been defined in the original MDP model. We treat all states in $\pi^1(i)$ as zero-reward absorbing local goal states. We model transitions to states that are neither in $\pi^1(i)$ nor in node i itself as going to a single special *out* state and incurring a fixed, large, negative penalty Δ. We use value iteration to solve for the policy.

The cost of solving at the upper levels is dominated by a quadratic deterministic shortest path algorithm. The time at the bottom level is dominated by value iteration, which Bertsekas (1995) showed is cubic in the size of the state space for stochastic shortest path problems.

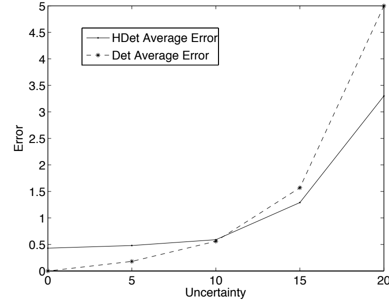


Figure 1: Average deviation from the optimal policy as a function of uncertainty in the grid world domain. Here $x\%$ uncertainty refers to the probability an action transitions to a wrong square. The probability of transitioning to the correct square is $1 - 0.03x$.

Theorem 3: Algorithm 2 has time complexity quadratic in the size of the largest macro-state and the number of $L - 1$ macro-states and cubic in the size of the largest level 1 macro-state.

3.3 Results

We tested the algorithm described above (consisting first of clustering and then of solving), called *HDet* for *hierarchically determinized*, on several different enumerated-states domains, and compared its performance to that of value iteration and HVI (Bakker, Zivkovic, and Krose 2005). HVI originally used spectral clustering, reported as HVI (S); we also tried it with g-connected clustering, reported as HVI (G). We also tried a version of HDet, Det, which does not run the clustering algorithm at all but instead treats each state as its own cluster. Det never solves any MDPs.

We used three experimental domains. *Grid world* is a typical grid world with four actions each with an 85% chance of transitioning to the expected square and a 5% chance of transitioning to each of the other adjacent squares. The *Grid World* had 1040 states and the *Large Grid World* had 62500 states. *Factory* is a version of the common *Builder* factored MDP problem (Dearden and Boutilier 1997), run on the fully enumerated state space of 1024 states. *Mountain Car* is a discretized, randomized version of the Mountain Car domain ((Sutton and Barto 1998), section 8.2) with 1024 states. For full explanations of these domains see (Barry 2009).

We evaluated the policies in each domain by running 1000 simulations of each policy starting from each state in the domain and averaging together the total reward from the simulations to find a policy value for every state in the domain. We report the average deviation of these policy values from the optimal values. Results on the algorithms on each of the domains are shown in Table 1.

Running time vs. accuracy The results show that HDet is substantially faster than value iteration with a modest decrease in the quality of the solution. HDet also substantially outperforms HVI. The closest competitor to HDet is, in fact, Det, the purely deterministic, non-hierarchical, version of

	Grid World		Large Grid World		Factory		Mountain Car	
Algorithm	Run Time (s)	Avg. Dev.	Run Time (s)	Avg. Dev.	Run Time (s)	Avg. Dev.	Run Time (s)	Avg. Dev.
Value Iteration	20.46	0	$> 10^4$	–	25.22	0	83.00	0
HDet	1.41	0.48	74.21	0	2.58	0.49	25.79	4.14
Det	0.19	0.18	94.73	0.2	0.25	0.35	0.51	15.55
HVI (G)	10.66	0.84	$> 10^4$	–	40.72	0.62	78.94	12.94
HVI (S)	24.40	0.66	$> 10^4$	–	81.32	2.36	124.08	236.58

Table 1: Results for three domains. *Run time* gives the total running times, which for HDet and HVI includes clustering time as well as solution time. *Avg. Dev.* is the deviation from the reward of the optimal policy divided by the length of the corresponding path. HVI and value iteration did not converge on the large grid world so we report average deviation from the policy found by HDet, which had the highest value. All algorithms were implemented by us and run on the same computer.

HDet. The speed of execution of Det on most of the problems is due to the relatively small size of these problems, chosen to enable value iteration to terminate in reasonable time. In the larger Grid World problem, Det required more time than HDet. We expect HDet's advantage to increase with the size of the problem.

Similarly, as the non-determinism in the domain increases, we expect the accuracy of both Det and HDet to suffer, but the average deviation of Det increases faster than that of HDet, so that when there is only a 40% chance of ending up in the intended square, the average deviation of Det is close to 5, but that of HDet is closer to 3 (Figure 1). We can control how accurate HDet is by setting the parameters MINCLUST and MAXSIZE. With fewer and larger clusters HDet will be more accurate, but slower.

Thus, when run on enumerated-states MDPs, HDet finds good approximate solutions and has total running times (for clustering and solving) that improve substantially on competing methods. It is important to note that the time taken to create the hierarchy need not be amortized over several problem instances: it pays off on a single instance.

Algorithm 3

Input: X: state variables, A: primitive actions, T: transition function, G: primitive goal states

Output: A compact regression graph (CRG)

CREATECRG(X, A, T, G)

1. *// Initialize the CRG with the goal node*
   $\Upsilon \leftarrow \{G\}$, $\Upsilon_0 \leftarrow \{G\}$, $k \leftarrow 0$
2. **repeat**
3. $\quad k \leftarrow k+1$, $\Upsilon_k \leftarrow \emptyset$
4. $\quad$ **for** $\eta \in \Upsilon_{k-1}$
5. $\quad\quad$ **for** $a \in A$
6. $\quad\quad\quad$ node $\nu \leftarrow$ {states to which η is adjacent under a}
7. $\quad\quad\quad$ **if** all states in ν are in some node in Υ
8. $\quad\quad\quad\quad$ enable a in smallest set of nodes $\Lambda \subseteq \Upsilon$ s.t. all states in ν now have a enabled
9. $\quad\quad\quad$ **else** $\Upsilon \leftarrow \Upsilon \cup \nu$, $\Upsilon_k \leftarrow \Upsilon_k \cup \nu$
10. **until** no new nodes were added to Υ
11. **return** Υ

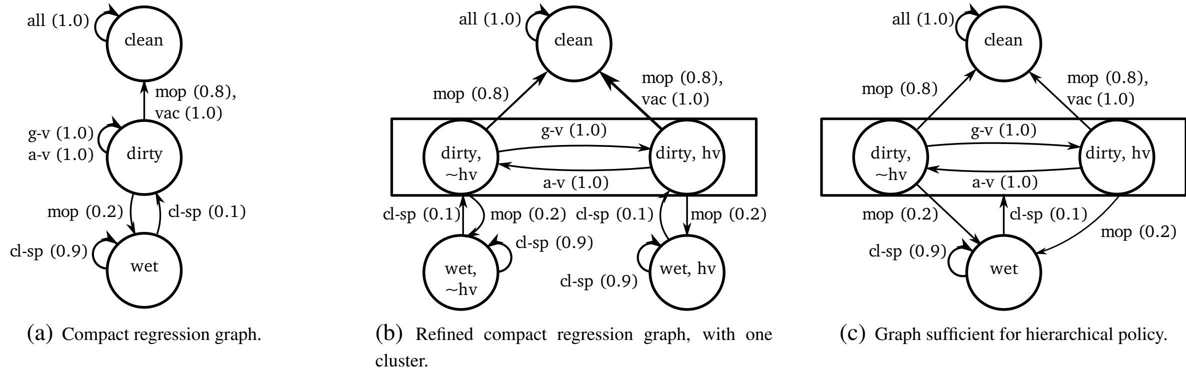
4 Factored MDPs

4.1 Clustering Algorithm

A factored MDP is defined by $\langle X, A, T, R, G\rangle$ where X is a finite set of discrete state variables. The state space S of the MDP can be obtained from X; a state of the MDP, $i' \in S$, is an assignment to all state variables. Components A, T, R, and G are as described in Section 2. We modify HDet (FHDet) to take advantage of the factored representation.

To create clusters for a factored MDP, we first run CREATECRG (Algorithm 3) and then FCLUSTER (Algorithm 4). HDet relies on examining every state, which is clearly not practical in the factored MDP case. Therefore, we begin our clustering algorithm using goal regression (Russell and Norvig 2003) to encode every action each primitive state could take that could possibly lead to the goal. However, CREATECRG, which creates a *compact regression graph* (CRG) terminates much earlier than standard regression. The CRG consists of nodes that represent possibly overlapping sets of states. Each node η has a set of actions $A(\eta)$ that are *enabled* for it. If there is no node η in the compact regression graph containing a primitive MDP state i' with action a enabled, then i' cannot reach a goal by taking action a. For the rest of this section, *node* will refer to a formula defining a set of states, such as the nodes in the CRG, and *macro-state* will refer to a collection of nodes.

We will eventually want to treat the CRG as an MDP where each "primitive state" of that MDP is a node. Therefore, we wish to be able to specify the transition probabilities between nodes. Before we can do that, however, we must be able to specify the transition of a primitive state i' into a node η under an MDP action a. We might wish to consider that the transition probability is the sum of the probabilities that i' can transition under a to any primitive state in η. However, because nodes do not partition the state space, this would double count some primitive states. Therefore, if a primitive state j' belongs to two nodes, the algorithm may choose which node to place it in for the purposes of transition probability. Let the set of nodes be Υ. A "graph action", $\alpha = \langle a, \tau\rangle$, consists of an MDP action a and a partitioning of primitive states into nodes, $\tau : \Upsilon \rightarrow S^0$. For a node η, $\tau(\eta) \subseteq \eta$ defines which primitive states in η we consider as contributing probability to transitions into η under α. To avoid double counting states, we impose the restriction on τ that for any primitive state j' present in the CRG, there is exactly one node $\eta_{j'}$ such that $j' \in \tau(\eta_{j'})$. The reward $R(i', \alpha)$ and transition probability $T(i', \alpha, \eta)$ from a primi-



(a) Compact regression graph.

(b) Refined compact regression graph, with one cluster.

(c) Graph sufficient for hierarchical policy.

Figure 2: An example of clustering in domain in which a robot is trying to clean a room. There are two state variables in this domain: the room-state variable can take on three values, *clean*, *dirty*, and *wet*; the *hv* variable, indicating whether the robot has the vacuum, can be either true (*hv*) or false (∼*hv*). To achieve the goal of a clean room, the robot can either mop it using *mop* or, if *hv* is true, vacuum it using *vac*. The vacuum can be obtained using *g-v* and put down using *a-v*. Vacuuming cleans the room with probability 1, but mopping has a 20% chance of spilling water, transitioning from *dirty* to *wet*. In *wet*, the robot must try to clean up the spill using *cl-sp* before it can attempt to clean the room again. All rewards in the domain are -1. Directed edges represent graph actions.

Algorithm 4

Input: Υ^0: set of nodes, A: primitive actions, T: transition function, R: reward function, G: primitive goal states

Output: A g-connected, locally stable clustering and a refinement of Υ^0

FCLUSTER(Υ^0, A, T, R)

1 $\Upsilon^1 \leftarrow \Upsilon^0$, $Adj \leftarrow$ ADJMATRIX(Υ^1, A, T)
2 **while** $|\Upsilon^1| >$ MINCLUST$_F$ and $\Upsilon^1_{\max} <$ MAXSIZE$_F$
3 $\quad \Lambda^1 \leftarrow$ FINDCYCLE(Υ^1, Adj)
4 $\quad \Lambda^0 \leftarrow$ nodes in Λ^1 // Λ^1 *is level 1 macro-states*
5 $\quad \Theta^0 \leftarrow$ REFINE(Λ^0, A, T, R)
6 $\quad \Theta^1 \leftarrow$ ESCLUSTER(Θ^0, A, T, G)
7 $\quad \Upsilon^1 \leftarrow (\Upsilon^1 \setminus \Lambda^1) \cup \Theta^1, \Upsilon^0 \leftarrow (\Upsilon^0 \setminus \Lambda^0) \cup \Theta^0$
8 $\quad Adj \leftarrow$ ADJMATRIX(Υ^1, A, T)
9 **return** Υ^1, Υ^0

tive state i' into a node η via graph action $\alpha = \langle a, \tau \rangle$ is

$$R(i', \alpha) = R(i', a) \quad (1)$$

$$T(i', \alpha, \eta) = \sum_{j' \in \tau(\eta)} T(i', a, j'). \quad (2)$$

A node $\eta \in \Upsilon$ is *stable w.r.t. a graph action* α if

$$\exists r, t \in \mathbb{R} \text{ s.t. } \forall i' \in \eta,\ \forall \eta' \in \Upsilon$$
$$T(i', \alpha, \eta') = t \text{ and } R(s, \alpha) = r. \quad (3)$$

The CRG may have nodes that are unstable with respect to graph actions. For example, in Figure 2(a), there is an edge from *(dirty)* to *(clean)* annotated with *vac* (1.0), but only the state *(dirty, hv)* can transition to *(clean)* using *vac* with probability 1.0. To remove instabilities from a set of nodes Υ we run REFINE (Algorithm 5). If Υ does not span all the primitive states so that some actions transition a node out of Υ, we treat all of those transitions as terminating in the same node, which we will refer to as the *out* node. The result of

Algorithm 5

Input: Υ: set of nodes, A: primitive actions, T: transition function, R: reward function

Output: A set of nodes Υ' s.t. Υ' contains the primitive states in Υ' and all nodes in Υ are stable w.r.t. each other and the *out* node.

REFINE(Υ, A, T, R)

1 *out* $\leftarrow$ new node, $\Gamma \leftarrow$ GRAPHACTIONS(Υ, A, T)
2 **while** $\exists\ \eta \in \Upsilon,\ \alpha = \langle a, \tau \rangle \in \Gamma,\ \nu \in \Upsilon^0$
$\quad\quad$ s.t. η is unstable w.r.t. α and ν
3 $\quad \{\eta_1, ..., \eta_n\} \leftarrow$ partition of η s.t. each η_i is max size to be stable w.r.t. α and ν if transitions out of Υ go to *out*
4 $\quad$ **for** $i = 1$ **to** n
5 $\quad\quad$ **if** all states in η_i are in some node in $\Upsilon \setminus \eta$
6 $\quad\quad\quad$ enable a in smallest set of nodes $\Lambda \subseteq (\Upsilon \setminus \eta)$ s.t. all states in ν now have a enabled
7 $\quad\quad$ **else** $\Upsilon \leftarrow \Upsilon \cup \eta_i$
8 $\quad \Upsilon \leftarrow \Upsilon \setminus \eta, \Gamma \leftarrow$ GRAPHACTIONS(Υ, A, T)
9 **return** Υ

refining the entire CRG in the cleaning domain is shown in Figure 2(b) (ignore the rectangle).

Theorem 4: Solving the refined CRG as an MDP gives the optimal solution to the original MDP.

Theorem 5: The number of nodes in the refined CRG for an MDP is upper bounded by the number of states in the reduced MDP proposed by Givan et al. (2003).

Proof Sketch: Each node in the refined CRG contains some primitive state not contained in any other node in the graph and is a union of states of the reduced MDP. Since the states of the reduced MDP partition the primitive state space, the theorem follows from the pigeon-hole principle.

We could create a hierarchical MDP model by running ESCLUSTER on the nodes of the refined CRG. However, this is impractical, since there is no guarantee that the refined CRG has significantly fewer nodes than the original

MDP. It is also more work than generally necessary. Consider the solving process at level 0 of the hierarchy: each level 1 macro-state i is solved as its own small MDP. We treat any transition to $\pi^1(i)$ as going to a zero-reward goal state and any transition to any other macro-state as going to the *out* state. Thus, more states behave equivalently according to the hierarchy than do according to the flat MDP, as illustrated in Figure 2(c). Therefore rather than refine and *then* cluster, we refine as we cluster. The algorithm is given in FCLUSTER and fulfills two basic criteria:

Theorem 6: The set of macro-states output by FCLUSTER is g-connected.

Proof Sketch: The CRG is g-connected because each macro-state contains only one node. All of the new macro-states we create are created using ESCLUSTER. Therefore, by Theorem 1, the set remains g-connected.

Theorem 7: The set of macro-states output by FCLUSTER is *locally stable*. A macro-state is locally stable if all nodes within the macro-state are stable with respect to all other nodes in the macro-state assuming that any transitions out of the macro-state go to an *out* node.

Proof Sketch: Each time we create a new macro-state, we refine it until it is locally stable.

At termination of the clustering, we have locally stable, g-connected macro-states. However transitions *among* the macro-states may be unstable because, in creating these macro-states, we ignored any transitions terminating outside the macro-state. Since the solver produces a deterministic policy at the upper levels, we do not need to refine all macro-states with respect to each other now; instead the solver will choose which macro-states to refine further. We have described the process building a two-level hierarchy; more levels can be built using a repeated application of this process.

4.2 Solver

The solver takes as input the output of FCLUSTER and outputs a hierarchical policy and a set of g-connected, locally stable macro-states that may be further refined. The FHDet solver is similar to the HDet solver, but rather than executing the strict two-pass algorithm we described in Section 3.2, we interweave the computations of the costs and the hierarchical policy with each other and with refinement of the lowest level state space. In fact, we do not run UPWARDPASS at all, but begin with DOWNWARDPASS. We can run DOWNWARDPASS almost exactly as in the enumerated states case, except in line 5 where we require a cost measure between macro-states. To compute this cost measure, we run FACSHORTESTPATH shown in Algorithm 6. The reason for this change is that eventually we will need to construct an MDP from a macro-state i. In this MDP we require that all states be locally stable with respect to each other and a *goal node* representing transitions to $\pi^1(i)$. Since we wish to avoid refining nodes in i with respect to nodes in every other macro-state, we wait until we have a candidate for $\pi^1(i)$ before refining. Otherwise, the algorithm is unchanged from DOWNWARDPASS.

Algorithm 6

Input: i, j: level 1 macro-states, A: primitive actions, T: transition function, R: reward function

Output: The distance from i to j and a partition of the nodes of i that is locally stable with respect to j.

FACSHORTESTPATH(i, j, A, T, R)

```
1   pq ← PriorityQueue, d(i) ← ∞
2   pq.PUSH(j, 0)
3   while pq.NOTEMPTY()
4       η ← pq.POP()
5       for ν ∈ PREDECESSORS(η)
6           if ∃μ ∈ i s.t. ν ⊆ μ and d(μ) < d(η), continue
7           Λ ← REFINE(ν ∪ η, A, T, R)
8           for λ ∈ Λ
9               d(λ) ← min_{a∈A} −T(λ,a,η)/R(λ,a) + d(η)
10              if ∀μ ∈ i s.t. λ ⊆ μ, d(μ) > d(λ)
11                  pq.PUSH(λ, d(λ)))
12          i ← (i \ ν) ∪ Λ
13  // |i| = number states in i, |η| = number states in η
    return (1/|i|) Σ_{η∈i} |η|d(η), i
```

Theorem 8: Let H be a hierarchy of nodes with policy π found by running DOWNWARDPASS and FACSHORTESTPATH on an arbitrary hierarchy. If we created a hierarchy of primitive states H' by expanding each node in H, Algorithm 2 run on H' outputs π.

Proof Sketch: Consider a level 1 macro-state i. This macro-state is its own refined CRG (all of its nodes are stable w.r.t. all other nodes in the macro-state, a goal node representing $\pi^1(i)$, and an *out* node). Therefore, the policy we find for the MDP defined by i and $\pi^1(i)$ is optimal by Theorem 4, and refining i any further would not change the policy.

The problems of partitioning, stability checking, and set cover that must be solved in CREATECRG and REFINE are worst case exponential in the number of state variables, $|X|$. However, all three of these may be approximated with heuristics. Since the running time will then depend on the implementation chosen, we characterize it in terms of the number of calls to partition, stability check, and set cover.

Theorem 9: In terms of the size of the largest macro-state, running CREATECRG and FCLUSTER followed by DOWNWARDPASS and FACSHORTESTPATH will terminate after running a linear number of partitions and a quadratic number of set cover problems and stability checks.

4.3 Results

The purpose of FHDet is to produce approximate solutions on very large domains. Domains previously used to test policy solvers such as in Givan et al. (2003), Jonsson and Barto (2006), and Kim and Dean (2002) are small and don't test the strengths of FHDet. Thus we first present results on a large domain ($> 2^{100}$ primitive states) of our own devising.

FHDet works best in well-connected domains that have small CRGs and no irrecoverable outcomes. We created a domain, House-Clean, with these properties. In this domain, a robot is trying to clean a series of rooms. It can always mop, but might have a costly mop water spill. In rooms with

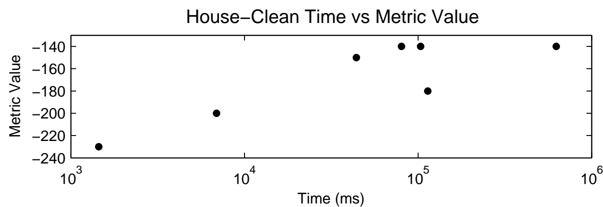


Figure 3: Time taken in the House-Clean domain versus the metric value.

a supply closet, the robot can pick up a vacuum or cleaning solution. Getting the vacuum out of the closet is more costly, but the robot can take it between rooms. The cleaning solution once used cannot be used again. The supply closet also contains another robot to which the agent can delegate the room. Walking between rooms is costly unless the robot takes a cart out of the supply closet. To transport the vacuum quickly, the robot also has to take an extension to the cart. It can abandon the vacuum, but once it does so, it cannot pick it up again. If a robot re-enters a room, it must clean it again.

Results for a domain with 40 rooms and 20 supply closets (over 2^{100} primitive states), are shown in Figure 3. Selecting different values for MINCLUST and MAXSIZE in the clustering results in different points in the time/accuracy trade-off as shown. We can solve this problem with *many* fewer abstract states than primitive states. By varying the parameters, we control the number of states in our representation. Det solves a representation using 377 states, obtaining a metric value of -230. With 562 states, FHDet finds a policy in which it picks up the vacuum at the first room, but abandons it later (value -200). At 1207 states—which actually takes less time to solve—FHDet finds a policy in which it takes the vacuum from the first room and later abandons it, but not quite as early as before (value -150). At 1197 states, FHDet finds the policy where it keeps the vacuum all the way through (value -140). None of these points represents the optimal policy, which takes the cart and the extension, as well as the vacuum, from the first. Finding this policy requires enough nodes that it cannot be stored in RAM on a desktop PC. However, finding the optimal policy is not our goal; we want to be able to find approximate policies quickly. We have shown that FHDet can efficiently compute policies that are a significant improvement over the completely deterministic approximation, in a problem whose size puts it out of range for other MDP solution methods.

We also wished to do a comparison of our algorithm with other state-of-the-art MDP solvers. However, the problem of finding a full policy for an MDP is so difficult that there exist very few solvers that attempt it. Rather, most of the results we could find on large problems, solved the simpler problem of finding a good solution *given* an initial starting state ahead of time. In many domains, this allows them to ignore a large portion of the state space. Thus, these planners can often quickly find a good solution for a given initial state, but, if the initial state changes, must begin all over again. Therefore, a policy that gives a good solution for *every possible* initial state and can be found in approximately the same amount of time or a little slower is much preferable.

We tested FHDet and FDet on two domains written in PPDDL, tireworld and exploding blocksworld, from the ICAPs (Bonet and Givan 2006; Bryce and Buffet 2008) competitions. We used a two level hierarchy for all problems, but set the parameters for FHDet empirically for each problem. The other algorithms, with the exception of FOALP (Sanner and Boutilier 2006), took advantage of the provided initial state information. Thus we are comparing the time it takes for us and FOALP to solve for a good policy over *all* states to the time it takes the other planners to solve for *one* initial state.

The results on the ICAPS domains are shown in Figure 4. The time shown for the planners that use the provided initial state is the average time taken for *one* run; the time shown for FDet, FHDet, and FOALP is the time to find an entire policy. In the problems from 2006, the time for FHDet and FDet to find a complete policy is comparable to the times the other planners require for a single execution. FHDet finds policies at least as good as the other planners, except on the largest blocksworld instances. In the large blocksworld instances, the CRGs were too large, and so our implementation of FHDet could not run to completion. This is also the reason that FHDet, which is faster than FOALP on the tireworld, is slower than FOALP in the blocksworld. We are working on a version of FHDet that does not require a CRG.

In 2008, the ICAPS problem instances were specifically created so that the paths with the shortest expected length were the worst possible policies that could still reach the goal. Thus, they are impossible for FDet and very hard even for FHDet; even so, FHDet found policies comparable in performance to those of many other planners in times only a factor of 10-50 times longer than it took other planners to solve a single instance (FOALP was not run in 2008).

5 Conclusion

Planning in very large MDPs is difficult because of long horizons, uncertainty in outcomes and large state spaces. The HDet and FHDet algorithms address each of these. By constructing a temporal hierarchy, HDet and FHDet reduce a single long-horizon planning problem to several planning problems with much shorter horizons, considerably reducing planning time. Uncertain planning problems can always be simplified by making a deterministic approximation, but sometimes this simplification comes at a great cost. HDet and FHDet make only a limited deterministic approximation, taking care to model stochasticity in the domain in the short horizon by solving MDPs at the leaf nodes of the hierarchy, but summarizing the cost of longer-term decisions with their expectations. This strategy results in more accurate, but still efficient, decisions.

There is an important synergy between hierarchy and state-space abstraction: sub-problems in a hierarchical decomposition can often be circumscribed in such a way that allow special-purpose state-space abstraction. Thus, even in domains where there is no opportunity for abstraction over the whole state space, there may be considerable opportunity for abstraction within each subproblem. FHDet con-

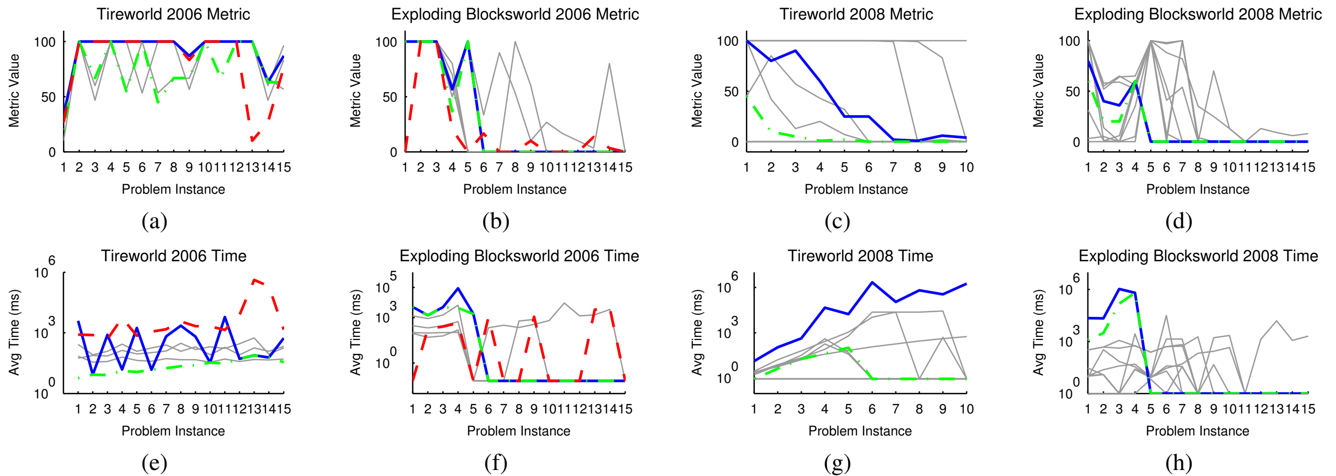


Figure 4: Results in the tireworld and exploding blocksworld domain for the problem instances from ICAPs 2006 and 2008. Our algorithm is FHDet (solid blue line). For comparison, we also run FDet, which is FHDet without any clustering: all clusters are size 1 (dash-dot green line). FOALP, a first order MDP policy solver run only in 2006, is shown as a red dashed line. All other algorithms run in the ICAPs competitions are shown as solid grey lines. Plots (a) - (d) show percentage of the time the planner was able to reach the goal in each domain. Plots (e) - (h) show running times in these domains.

structs appropriate abstractions at the same time as it creates the hierarchy, to maximize this synergy.

As a result, FHDet can find complete policies for many domains in the time it takes other methods to solve a single instance. This policy can be re-used for other starting states and the same goal set. In addition, for other goals in the same domain, much of the work of building the hierarchy can be re-used to generate new policies extremely efficiently.

References

Bakker, B.; Zivkovic, Z.; and Krose, B. 2005. Hierarchical Dynamic Programming for Robot Path Planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3720–3725.

Barry, J.; Kaelbling, L.; and Lozano-Pérez, T. 2010. Hierarchical Solution of Large Markov Decision Processes. Technical report, Massachusetts Institute of Technology.

Barry, J. 2009. Fast Approximate Hierarchical Solution of MDPs. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Belmont, Massachusetts: Athena Scientific.

Bertsekas, D. P. 1995. *Dynamic Programming and Optimal Control*. Belmont, Massachusetts: Athena Scientific.

Bonet, B., and Givan, B. 2006. Non-Deterministic Planning Track of the 2006 International Planning Competition. `www.ldc.usb.ve/~bonet/ipc5/`.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 2000. Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence* 121:49–107.

Bryce, D., and Buffet, O. 2008. The Uncertainty Part of the 6th International Planning Competition 2008. `ippc-2008.loria.fr/wiki/index.php/Main_Page`.

Dearden, R., and Boutilier, C. 1997. Abstraction and Approximate Decision-Theoretic Planning. *Artificial Intelligence* 89:219–283.

Givan, R.; Dean, T.; and Greig, M. 2003. Equivalence Notions and Model Minimization in Markov Decision Processes. *Artificial Intelligence* 142(1-2):163–223.

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

Kim, K.-E., and Dean, T. 2002. Solving Factored MDPs with Large Action Space Using Algebraic Decision Diagrams. In *PRICAI-02*, 80–89. Springer.

Lane, T., and Kaelbling, L. P. 2002. Nearly Deterministic Abstractions of Markov Decision Processes. In *AAAI-02*.

Mehta, N.; Ray, S.; Tadepalli, P.; and Dietterich, T. 2008. Automatic Discovery and Transfer of MAXQ Hierarchies. In *ICML-08*.

Russell, S. J., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. New Jersey: Pearson Education, second edition.

Sanner, S., and Boutilier, C. 2006. Practical linear value-approximation techniques for first-order mdps. In *UAI-06*.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Teichteil-Königsbuch, F., and Fabiani, P. 2005. Symbolic Heuristic Policy Iteration Algorithms for Structured Decision-Theoretic Exploration Problems. In *Workshop on Planning Under Uncertainty for Autonomous Systems at ICAPS-05*, 66–74.

Wu, J.-H.; Kayanam, R.; and Givan, R. 2008. Planning using stochastic enforced hill-climbing. In *IPPC-08*.

Yoon, S.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *AAAI-08*, 1010–1016. AAAI Press.