MIT Libraries | DSpace@MIT

MIT Open Access Articles

DetH: Approximate Hierarchical Solution of Large Markov Decision Processes*

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Barry, Jennifer, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. "DetH*: Approximate Hierarchical Solution of Large Markov Decision Processes. In 22nd 2011 International Joint Conference on Artificial Intelligence, IJCAI-11, Barcelona, Catalonia, Spain, 16–22 July 2011. AAAI Press, (2011): p.1928-1935.

As Published: http://ijcai.org/papers11/Papers/IJCAI11-323.pdf

Publisher: AAAI Press/International Joint Conferences on Artificial Intelligence

Persistent URL: http://hdl.handle.net/1721.1/90898

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

DetH*: Approximate Hierarchical Solution of Large Markov Decision Processes*

Jennifer L. Barry, Leslie Pack Kaelbling, 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 approximately. It exploits factored representations to achieve compactness and efficiency and to discover connectivity properties of the domain. We provide a bound on the quality of the solutions and give asymptotic analysis of the runtimes; in addition we demonstrate performance on a collection of very large domains. Results show that the quality of resulting policies is very good and the total running times, for both creating and solving the hierarchy, are significantly less than for an optimal factored MDP solver.

1 Introduction

Our goal is to find good, though not necessarily optimal, solutions for large, factored Markov decision processes. We present an approximate algorithm, DetH*, which applies two types of leverage to the problem: it shortens the horizon using an automatically generated temporal hierarchy and it reduces the effective size of the state space through state aggregation.

DetH* uses connectivity heuristics to break the state space into a number of macro-states. It then assumes that transitions between these macro-states are deterministic, allowing it to quickly compute a top-level policy mapping macro-states to macro-states. Once this policy has been computed, DetH* solves for policies in each macro-state independently. We are able to *construct and solve* these hierarchies significantly faster than solving the original problem.

We represent factored MDPs compactly as algebraic decision diagrams (ADDs) as done by the SPUDD algorithm [Hoey *et al.*, 1999]. In order to maintain efficiency in large problems, our algorithm is composed of operations on ADDs and BDDs and so avoids ever enumerating the states in the domain. We make use of the independencies in the domain dynamics that are revealed by the structure of the ADD representation, and we use the SPUDD algorithm to solve the MDPs that constitute the individual macro-states. SPUDD naturally reduces the size of the effective state space by aggregating sets of states with the same value.

There are two important related lines of work, focusing either on factoring or on decomposition of large MDPs. Improvements on factored MDP solvers continue to treat the MDP as a single problem, but find more compact [Sanner and McAllester, 2005; Sanner *et al.*, 2010] or approximate [St-Aubin *et al.*, 2000] representations. Recent work on topological and focused topological value iteration [Dai and Goldsmith, 2007; Dai *et al.*, 2009] is similar to ours in that it decomposes a large MDP based on the connectivity of the states. However, TVI and FTVI cannot exploit a factored representation and, in a well-connected domain, they are not guaranteed to find any hierarchy at all. FTVI has been run successfully on some smaller factored problems, but requires knowledge of an initial state to reduce the state space size. Moreover, the size of the value function output by TVI or FTVI is necessarily the size of the optimal value function and, therefore, these cannot be run in domains where the representation of the optimal value function is prohibitively large. By relaxing the requirement that the algorithm output the optimal policy, we can find a good approximate policy even in some large problems where the optimal value function is not compactly representable. Our work is also similar to MAXQ [Dietterich, 1998; Mehta *et al.*, 2008] in that it uses a temporal hierarchy to reduce the size of the state space. However, unlike the original MAXQ algorithm, we are able to construct the hierarchy automatically. Although the work of Mehta *et al.* proposes a method for automatically creating MAXQ hierarchies, the hierarchy creation is costly and only worthwhile if it transfers to many domains.

We begin by introducing our hierarchical model of an MDP. We then discuss how DetH* creates and solves this model and conclude by presenting results on a set of large domains.

2 Hierarchical Model

In this paper, we restrict ourselves to creating and solving hierarchical models with only two levels and for simplicity explain only the two level hierarchy model. However, the extension to multi-level hierarchies is straightforward.

An MDP can be described as $M = \langle S, A, T, R \rangle$ where S is a finite set of states, A is a finite set of actions, T is

*This work was supported in part by ONR MURI grant N00014-09-1-1051 and in part by AFOSR grant AOARD-104135.

the transition model with $T(x, a, x')$ specifying the probability of a transition to state x' given that the system starts in state x and selects action a, and R is the reward model with $R(x)$ specifying the real-valued reward of state x. In addition, we assume a pre-specified set of *goal states*, $G \subset S$. Goal states are zero-reward absorbing states: for every $g \in G$ and $a \in A$, $T(g, a, g) = 1$ and $R(g) = 0$. Further, we assume that all other reward values are strictly negative. We solve this problem under the undiscounted total reward criterion, making it a 'negative stochastic shortest path' problem. Any discounted MDP can be transformed into an 'equivalent' negative stochastic shortest path problem, which can then be solved to produce the optimal policy for the original MDP [Bertsekas and Tsitsiklis, 1996; Bonet and Geffner, 2009].

A factored MDP is an MDP and a set of *state variables* $\mathbf{X} = \{X^1, ..., X^n\}$. For each variable X^i, we assume a finite set of possible values $\Omega_{X^i} = \{X^i_1, ..., X^i_{m_i}\}$. A state of the MDP is specified by a single assignment to each state variable; the value of variable X^i in state $x \in S$ is given by x^i. We represent the transition model using a collection of ADDs in the manner of SPUDD; for each action and for each value of each variable, X^i_j, an ADD specifies the probability that X^i will have value X^i_j on the next time-step. The reward is represented as a single ADD.

A (two level) hierarchical model of a factored MDP consists of a factored MDP and a set of macro-states, U, which partition the state space. Each macro-state in U is a set of states of the original MDP (*primitive states*), represented as a binary decision diagram (BDD). U must contain a single macro-state $g = G$, called the *goal macro-state*, which is exactly the set of primitive goal states. Given a macro-state u, we say that a variable value $X^i_j \in u$ if there exists some primitive state $x \in u$ with $x^i = X^i_j$. A *hierarchical policy* $\pi = \{\pi_U, \pi_S\}$ consists of an upper-level policy π_U and a lower-level policy π_S. The policy π_U maps each macro-state $u \in U$ to another macro-state $u' \in U$, signifying that when the system is in a primitive state contained in u it should attempt to move to some primitive state in u'. The policy π_S is a standard MDP policy, represented as an ADD, mapping the primitive states to primitive actions.

3 DetH* Overview

The DetH* algorithm takes as input an MDP represented using ADDs and outputs a hierarchical policy. The input representation is the same that of SPUDD. The algorithm works in three stages:

1. It computes a set of macro-states, U, for the MDP that partition the state space. We discuss the clustering algorithm used to create these macro-states in Section 5.
2. It computes an upper-level policy π_U mapping each macro-state in U to another macro-state in U. We explain this algorithm in Section 4.
3. It uses π_U to break the original, large MDP into many smaller sub-MDPs. Specifically, for each macro-state $u \in U$, it creates a sub-MDP by setting all states in $\pi_U(u)$ to have the fixed value 0 and setting all states not in u or $\pi_U(u)$ to have a fixed value $-\Delta$ where Δ is large. No other modifications are made to the original transition and reward model of the MDP and any factored MDP solver can be used to solve for a policy in this sub-MDP. Since the macro-states partition the state space, π_S is just the union of the policies of the sub-MDPs.

We begin by describing step 2 of DetH* as it will motivate the choices we make in designing the clustering algorithm.

4 Solving the Hierarchical Factored MDP

The solver takes as input a hierarchical model of a Markov decision process (MDP) and outputs a policy, π_U on the macro-states of the hierarchy. We will discuss how to find the hierarchical model in Section 5.

4.1 Algorithm

To find π_U, we approximate the transitions between macro-states as being deterministic. Given a function $C(u, u')$, representing the cost to transition from macro-state $u \in U$ to macro-state $u' \in U$, DetH* uses Dijkstra's algorithm to find a shortest path, $\langle u, v_1, v_2, ..., g\rangle$, for each macro-state $u \in U$ to g. It sets $\pi_U(u) = v_1$.

Cost Model

To model the cost $C(u, u')$ of transitioning from macro-state u to u', we assume that, at the primitive level, any action $a \in A$ taken in state $x \in S$ with goal of ending in state $x' \in S$ does make a transition to x' with probability $T(x, a, x')$, but that with the remaining probability mass, it stays in state x. Executing such an action a repeatedly will, in expectation, take $1/T(x, a, x')$ steps to move to state x', each of which incurs cost $-R(x)$. We can select whichever action would minimize this cost, yielding a deterministic cost estimate at the primitive level of

$$C(x, x') = \min_{a \in A} -\frac{R(x)}{T(x, a, x')}.$$

If there is no action that causes a transition from x to x' then the cost of transitioning between the two is infinite. This cost model is unlikely to reflect the exact dynamics of the domain, but it allows us to incorporate both the transition probabilities and reward into the cost calculation in an efficiently computable manner.

Let $U(x)$ be the macro-state containing x (since the macro-states are a partition, U is a function). We define $d(x, x')$ as the shortest distance, using C as a cost measure, from x to x' considering only states in $U(x)$ and $U(x')$. For macro-state u', let $d^{u'}(x) = \min_{x' \in u'} d(x, x')$, the shortest distance from some primitive state x to some $x' \in u'$ considering only states in $U(x)$ and u'. We define $C(u, u')$ as the average over the $d^{u'}(x)$ for $x \in u$:

$$C(u, u') = \frac{1}{|u|} \sum_{x \in u} d^{u'}(x).$$

Given two macro-states, u and u', we can calculate $C(u, u')$ using Dijkstra's algorithm to find $d^{u'}(x)$. Because of the factored representation, $d^{u'}(x)$ is likely to be identical for many

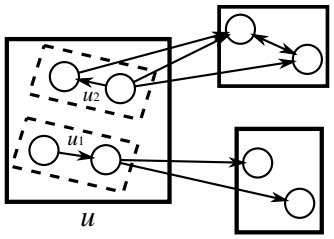


Figure 1: A hierarchical model of an MDP. The bold rectangles depict the macro-states and the small circles are the primitive states. An arrow is drawn from primitive state x to primitive state x' if some action can transition x to x'. For macro-state u, regardless of which macro-state is chosen as $\pi_U(u)$, either the states in u_1 or the states in u_2 will not be able to reach $\pi_U(u)$.

$x \in u$ and we can use ADD operations to treat all states with the same $d^{u'}(x)$ compactly.

Approximate Cost Model

Even using ADDs, the representation of the cost C tends to grow exponentially quickly, so we approximate C by assigning the same distance value to all primitive states within u that can reach $\pi_U(u)$ in the same minimum number of actions.

Specifically, we break u into the set of sub-macro-states $\{u_1, ..., u_{z(u,u')}\}$ such that for $x \in u_j$, x can reach some sub-state of u' with non-zero probability in j actions and no fewer. The quantity $z(u, u')$ is the smallest j such that all sub-states in u that can reach some sub-state in u' with non-zero probability can do so in j actions or fewer. If some sub-states of u cannot reach a sub-state of u', they are not included in the sub-macro-states. We approximate the cost $\overline{C}$ of transitioning from sub-macro-state u_j to u_{j-1} as the average cost of transitioning from a state in u_j to a state in u_{j-1}:

$$\overline{C}(u_j, u_{j-1}) = \frac{1}{|u_j|} \sum_{x \in u_j} \min_{x' \in u_{j-1}} C(x, x'),$$

which can be calculated efficiently using ADD operations. Note that since all $x \in u_j$ can reach some state in u' in j actions and no fewer and all $x' \in u_{j-1}$ can reach some state in u' in $j-1$ actions and no fewer, every state in u_j has a non-zero transition probability to some state in u_{j-1} and $\overline{C}(u_j, u_{j-1})$ is finite. Letting $u_0 = u'$, this gives us the approximation of C, C':

$$C'(u, u') = \frac{1}{z(u, u')} \sum_{i=1}^{z(u,u')} \sum_{j=1}^{i} \overline{C}(u_j, u_{j-1}).$$

Solver

Given C', we could simply use Dijkstra's algorithm on the macro-states to find π_U. However, consider the case of the hierarchy shown in Figure 1. In this hierarchy, there is a macro-state $u \in U$ that is made up of two distinct sets of primitive states, u_1 and u_2, such that there is no macro-state $u' \in U$ such that all sub-states of u can reach some sub-state of u'. Therefore, whatever the value of $\pi_U(u)$, there will be some primitive state $x \in u$ such that x cannot reach a sub-state of $\pi_U(u)$. This could result in an infinite negative value for x under π_S even if the optimal value function is finite everywhere. To avoid this, when calculating $C'(u, u')$, we calculate C' for only those sub-states $u_R \subseteq u$ that can reach some sub-state of u'. If, after the cost calculation, u' is on the current path-to-goal for u_R, we split u into u_R and $u \setminus u_R$. This allows us to guarantee the following:

Theorem 1: Let π^* be the optimal policy for an MDP M and let π_S be the policy found by DetH* on any hierarchy and V_{π_S} be the associated value function. Provided that for all $x \in S$, $V_{\pi_S}(x) > -\Delta$, then for any $x \in S$, if x has a non-zero probability of reaching some goal state under π^*, then x will also have a non-zero probability of reaching some goal state under π_S.

Proof: Proof of all theorems in this paper can be found in the associated technical report [Barry *et al.*, 2011].

Corollary 2: Let V^* be the optimal value function for an MDP M and let π_S be the policy found by DetH* on any hierarchy and V_{π_S} be the associated value function. Provided that for all $x \in S$, $V_{\pi_S}(s) > -\Delta$, if V^* is finite for all $x \in S$, V_{π_S} will be finite for all $x \in S$.

We can also give an asymptotic upper bound on the running time of the solver. We do this in terms of the set of macro-states H, which is the set of macro-states *after* running the solver. For state x, we define $T(x) = \pi_U(U(x))$ to be the target macro-state from $U(x)$. The bound relies on three quantities: $|H|$, the number of macro-states after running the solver, $Z = \max_{h \in H} z(h, T(h))$, the maximum number of actions required for any primitive state x to move from $U(x)$ to its target macro-state $T(x)$, and B, the size of the largest BDD representation of a set of states seen while creating or solving the hierarchy.

Theorem 3: The running time of the upper level solver is $O(|H|^2 Z B^2)$.

Proof Sketch: The running time of the upper level solver is dominated by $|H|^2$ calculations of C'. Calculating C' requires $O(|Z|)$ calls to $\overline{C}$, each of which takes $O(B^2)$.

4.2 Accuracy

Dietterich (1998) introduced the terms *hierarchically* and *recursively optimal* for discussing the polices associated with a hierarchy. These criteria assume a fixed hierarchy with known sub-goals at lower levels, which allows the problem to be solved bottom-up, with higher levels guaranteeing optimality subject to the quality of solutions at levels below. In DetH*, we do not assume that appropriate subgoals are known in advance; we must determine them, top-down, and so the resulting policies will not generally be hierarchically or recursively optimal. However, we can still bound the sub-optimality of the resulting policy.

We define $O(x) = \{x' | x' \notin (U(x) \cup T(x))\}$ to be the "out" set of primitive states neither in the macro-state of x nor its target. Let $B^*_{x,x'}$ be the probability that the first primitive state not in $U(x)$ reached under π^* is x'. Similarly $B^{\pi_S}_{x,x'}$ is the probability that the first primitive state not in $U(x)$ reached under π_S is x'.

Algorithm 1 *Input*: g: the goal macro-state *Output*: A set of g-connected macro-states.

SIMPLE-CLUSTER(g)
$U \leftarrow \{g\}$
while $\exists$ unclustered states that can reach the goal
  for each macro-state u created on the last iteration
    // all unclustered states *that reach u in 1 step*
    $n \leftarrow$ REGRESS(u, unclustered, 1)
    $U \leftarrow U \cup n$
$U \leftarrow U \cup$ macro-state consisting of any unclustered states
return U

Dai and Goldsmith (2007) showed that dividing a domain into macro-states and solving each macro-state separately can result in an optimal solution. The error that DetH* incurs arises from two differences between the set of macro-states it uses and the topologically sorted set proposed by Dai and Goldsmith. *Drift* measures error due to coercing the process to stay within a macro-state until exiting to the target macro-state, rather than moving to an out state:

$$\delta(x) = \sum_{x' \in O(x)} B^*_{x,x'} \left(V^*(x') + \Delta\right). \quad (1)$$

Heterogeneity measures the error incurred from having a sub-optimal arrival distribution over primitive states in the target macro-state:

$$h(x) = \sum_{x' \in T(x)} \left(B^*_{x,x'} - B^{\pi_S}_{x,x'}\right) V^*(x'). \quad (2)$$

Theorem 4: Provided that for all $x \in S$, $V_{\pi_S}(x) > -\Delta$,

$$E(x) = V^*(x) - V_{\pi_S}(x) \leq \delta(x) + h(x) + \sum_{x' \in T(x)} B^{\pi_S}_{x,x'} E(x').$$

Because for any x in G, $E(x) = 0$, error in the other primitive states can be characterized by backward induction on the graph defined by π_U.

In constructing a hierarchical model, we can attempt to reduce the drift term by creating macro-states that the optimal policy would prefer not to exit, or to exit by chance. We can attempt to reduce the heterogeneity term by creating macro-states whose primitive states have similar values.

5 Creating the Hierarchical Model

Given a factored MDP, we create a hierarchical model with the goal that it be solvable both accurately and efficiently.

g-Connected Clustering

We wish to create a set of macro-states U which require very little splitting during solution, which will occur if there already exists a policy on U for which Theorem 1 holds. To this end we define the property of *g-connectedness*:

Definition 5: A set of macro-states U with goal macro-state g is *g-connected under policy* π_U if

- For all $u \in U$, if u contains any primitive state that can reach a goal state under the optimal policy, u can reach g under π_U.
- For all $u \in U$, every sub-state of u can reach some sub-state of $\pi_U(u)$ without leaving u.

U is *g-connected* if there exists some policy under which it is g-connected.

For example, the set of macro-states shown in Figure 1 is not g-connected because, to fulfill the second condition of the definition, a policy must map u to u and this cannot fulfill the first condition.

Algorithm 2 *Input*: g: the goal macro-state, Υ: the maximum number of macro-states *Output*: A hierarchy for use with the DetH* solver.

CLUSTER(g, Υ)
// First create a large number of initial macro-states
$U, r \leftarrow$ SPLIT($g, \Upsilon, 1$)
while $\exists$ unclustered states that can reach the goal
  for each macro-state u created on the last iteration
    // all unclustered states *that reach u in $\leq r$ steps*
    $n \leftarrow$ REGRESS(u, unclustered, r)
    for $X^i \in \mathbf{X}$
      // Reduce drift by adding states
      $n \leftarrow$ GROW(n, unclustered, X^i)
    $U \leftarrow U \cup n$
$U \leftarrow U \cup$ macro-state containing any unclustered states
return U

SPLIT(g, Υ, r)
$U \leftarrow \{S \setminus g\}$, split $\leftarrow$ True
while split **and** $|U| < \Upsilon$
  split $\leftarrow$ False, $U' \leftarrow \{\}$
  for $u \in U$
    // the subset of u that can reach g in $\leq r$ steps
    $N \leftarrow \{$REGRESS(g, u, r)$\}$
    for $X^i \in \mathbf{X}$
      $N' \leftarrow \{\}$
      for $n \in N$
        $P \leftarrow$ partition of n s.t. for $X^i_j \in n_l$, all X^i_k i-reachable by X^i_j through n are in n_l.
        if $\forall n_j, n_k \in P$ s.t. $\forall X^i_m \in n_j, X^i_p \in n_k$, X^i_m cannot i-reach X^i_p
          // Exogenous variable.
          $P \leftarrow \{n\}$
        if $|P| > \Upsilon$, **return** SPLIT($g, \Upsilon, r+1$)
        if $|P| > 1$, split $\leftarrow$ True
        for each $n_j \in P$
          $n_j \leftarrow$ GROW(n_j, unclustered, X^i)
          $N' \leftarrow N' \cup \{n_j\}$
      $N \leftarrow N'$
    $U' \leftarrow U' \cup N$
  $U \leftarrow U'$
return $U \cup \{g\}, r$

GROW(u, Q, X^i)
$u' \leftarrow$ empty macro-state
for $x \in u$
  $u' \leftarrow u' \cup \{x' \in Q | x'^i$ can i-reach x^i through $Q \cup u$ in less than $\max_j |\Omega_{X^j}|/2$ steps, $x'^i \in u$, and for $j \neq i$, $x'^j = x^j\}$
return u'

The function REGRESS(u_g, u_a, i) returns all states in u_a that can reach some state in u_g in i steps or fewer. Therefore, a simple method for creating g-connected clusters is to repeatedly call REGRESS as shown in Algorithm 1.

Theorem 6: SIMPLE-CLUSTER creates g-connected macro-states.

Proof Sketch: By induction over U_k, the set of macro-states after the kth call to REGRESS.

Base Case: $U_0 = \{g\}$ is trivially g-connected.

Induction Step: Let $\pi_{U_{k-1}}$ be a policy under which U_{k-1} is g-connected. Assume we regress macro-state u_{k-1} on iteration k creating macro-state u_k. For $v \in U_{k-1}$, let $\pi_{U_k}(v) = \pi_{U_{k-1}}(v)$. Set $\pi_{U_k}(u_k) = u_{k-1}$. Since all sub-states of u_k can reach some sub-state of u_{k-1} by definition of REGRESS, it follows by induction that π_{U_k} is a policy under which U_k is g-connected.

Improving Clustering

SIMPLE-CLUSTER usually returns a small number of large macro-states so that solving each macro-state is costly. In addition, although the macro-states are g-connected, once π_U is determined, each macro-state may still have a significant heterogeneity and drift error, decreasing the accuracy of the solution. To improve efficiency, we would like to make more macro-states; to improve accuracy, we would like to decrease their drift and heterogeneity.

However, attempting to decrease drift and heterogeneity while using exact reachability on primitive states is very expensive in large domains. We use the factored structure of the transition model to define an efficiently-computable approximate notion of connectivity by considering the connectivity of each state variable separately.

Define a variable value X_j^i to be *adjacent* to a variable value X_k^i if $X_j^i = X_k^i$ or there exist states $x, x' \in S$ with $x^i = X_k^i$ and $x'^i = X_j^i$ and action $a \in A$ such that $T(x, a, x') > \epsilon$ where ϵ is a small (possibly zero) user-defined constant. Only variable values of the same variable can be adjacent.

Definition 7: Primitive state $x' \in S$ is *i-adjacent* (independently adjacent) to state $x \in S$ if, for all $X^i \in \mathbf{X}$, x'^i is adjacent to x^i. A macro-state $u' \in U$ is *i-adjacent* to a macro-state $u \in U$ if there exist primitive states $x' \in u'$ and $x \in u$ such that x' is i-adjacent to x. An element (variable value or state) j is *i-reachable from* an element i if j is i-adjacent to i or j is i-reachable from some element k that is i-adjacent to i. A set of macro-states is *ig-connected* if they are g-connected under i-adjacency.

Note that the existence of $a \in A$ such that $T(x, a, x') > \epsilon$ is sufficient for x' to be i-adjacent to x, but not necessary. In most domains many more states will be i-adjacent than are actually adjacent. While the solver will be most efficient when ig-connectedness implies actual g-connectedness, all of the results of Section 4 hold for any hierarchy input to the solver.

Pseudo-code for creating improved ig-connected clusters is shown in Algorithm 2. CLUSTER addresses the problems of few macro-states, drift, and heterogeneity in two ways.

Improving Drift: The GROW function adds to each macro-state a large number of primitive states that can easily i-reach the macro-state u in an attempt to reduce the drift error in later created macro-states by ensuring that states that can easily i-reach u are part of u. The limit of $\max_i |\Omega_{X^i}|/2$ is in place on line 3 of GROW to keep all states from being placed into the same macro-state in well connected domains.

Improving Heterogeneity: The SPLIT function creates several initial macro-states, which addresses the problem of few macro-states and heterogeneity. Since each macro-state created in SPLIT will be regressed separately on line 4 of CLUSTER, the splits made in this function will propagate, creating a much larger number of macro-states than if we simply called REGRESS repeatedly.[1]

The SPLIT function essentially splits a macro-state based on i-reachability so that the resulting macro-states contain only those primitive states which can i-reach each other. Primitive states that can easily i-reach one another are likely to have similar values, addressing the issue of heterogeneity. Because we use i-reachability, we can consider each variable separately, which allows this computation to be performed efficiently. The procedure for splitting a macro-state into partitions containing only primitive states that can i-reach each other is shown on lines 6-9 of SPLIT.

However, we found that splitting macro-states only on i-reachability created far too many macro-states if there were many variables in the domain whose values were not well-connected. We use the check on line 10 to identify variables with large numbers of values that cannot reach each other. The partition formed on line 9 of SPLIT only considers reachability *through* the current macro-state n. Line 10 checks if there is a path between two partitions through the entire domain, although there may not be one in n. If the partitions could never reach one another then we do not split the macro-state. This check is an attempt to identify *exogenous variables*, variables over which the agent has no control. These variables often have values set a priori that never change so splitting on them creates a large number of macro-states. Moreover, for many values of an exogenous variable, the value function is the same over a short horizon. Therefore, there is no need to split on the values of an exogenous variable; it will neither improve the homogeneity of the macro-state nor the efficiency of the solution at the bottom level.

Even attempting to avoid splitting on exogenous variables, it is possible that SPLIT creates too many macro-states, In this case, we redo SPLIT on line 12 by forming a macro-state that contains all primitive states that can reach a goal state in two steps rather than one. This larger macro-state is likely to split into fewer partitions.

Because we consider each variable individually, after partitioning and growing the macro-states, it is possible that re-

[1] In a very large domain, it is possible that better clusters could be created by splitting later macro-states as well, but, in general, this will create so many macro-states that the factors of $|H|$ in the running time of the solver will have a noticeable impact on running time. We found that just splitting the first macro-state worked well on all domains we tried.

peating the process will give us an even more refined set of macro-states. Therefore, we repeat the process until convergence is reached or we have a maximum number of macro-states.

Theorem 8: The macro-states created by CLUSTER are ig-connected and partition the state space.

Proof Sketch: In SPLIT, we initially call REGRESS on the full set of non-goal states. Subsequently, we partition the states and then only add to them states that have not yet been added to a macro-state. Similarly, during the main loop of CLUSTER, we only consider the set of states not yet clustered when we REGRESS and GROW macro-states. Therefore, the resulting macro-states partition the state space.

By Theorem 6, repeatedly calling REGRESS creates g-connected clusters. Now consider the GROW(u, Q, X^i) function where u is some macro-state belonging to a g-connected set of macro-states U and let π_U be some policy under which U is g-connected. The call to GROW creates a new set of macro-states $U' = (U \setminus u) \cup u'$ where $u' =$ GROW(u, Q, X^i), which we claim is ig-connected. For $v \in (U \setminus u)$, let $\pi_{U'}(v) = \pi_U(v)$ if $\pi_U(v) \neq u$ and u' otherwise. Set $\pi_{U'}(u') = \pi_U(u)$. Since $u \subseteq u'$, for macro-state $v \in U$, all sub-states of v can i-reach some sub-state of $\pi_{U'}(v)$ without leaving v. Now consider u'. For each $x \in u'$, either $x \in u$ or x can i-reach some $x' \in u$ without leaving u' by line 3 of GROW. Therefore, for all $x \in u'$, x can i-reach some sub-state of $\pi_{U'}(u')$ and U' is ig-connected under $\pi_{U'}$.

The proof that SPLIT creates ig-connected macro-states is similar. An induction over the calls to REGRESS, SPLIT, and GROW completes the full proof.□

In practice, to save computation time, we break the ig-connectivity slightly: we always let $Q = S$ for the GROW function and compute the partition afterwards by subtracting earlier created macro-states from later ones. We also estimate reachability for the variable values by assuming that all values can only reach one sink and therefore that all values that reach the same sink should be put in the same partition.

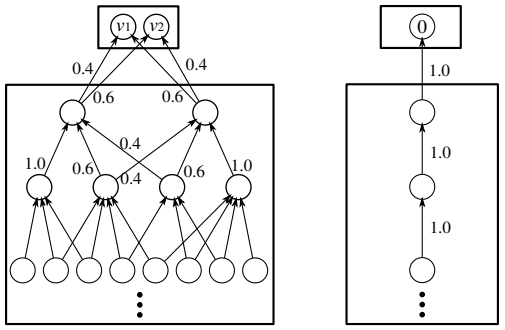
Theorem 9: The running time of CLUSTER is $O(|H|ZB^2)$.

Let $|V_H|$ be the size of the largest value function, in terms of the ADD representation, that SPUDD constructs during the solution of any macro-state in stage 3 of DetH*, and I_H be the largest number of iterations SPUDD requires in any macro-state. Then

Theorem 10: The total running time for DetH* is bounded by $O(|H|^2ZB^2) + O(|H|I_HB|V_H|)$.

Proof Sketch: The running time of CLUSTER is dominated by the running time of the upper level solver. For each macro-state, each iteration of SPUDD is bounded by $O(B|V_H|)$ so stage 3 is bounded by $O(|H|I_HB|V_H|)$.

Efficiency Advantage of DetH*: The effect of DetH* is to break the solution of one large SPUDD problem into many smaller problems. Although the number of iterations, I_H, that SPUDD requires is not easily evaluated in the infinite horizon case, intuitively it should be approximately upper bounded by the number of states with different values, which can be upper bounded $|V_H|$, giving us a running time of $O(|V_H|^2)$ for



(a) Solving for the optimal value function.

(b) Solving the bottom macro-state as a sub-MDP.

Figure 2: An MDP in which $|V_H| \ll |V^*|$. Here sets of primitive states with different values are shown as small circles and macro-states are shown by the rectangles. Actions are shown as arrows with the probability of transition written on the edge (to avoid clutter, we assume all rewards are -1). To find the optimal solution, two distinct values of the top macro-state must be considered, as shown in (a), which leads to an exponential explosion in the number of different values in the bottom macro-state. Therefore, both the SPUDD and (F)TVI would be unable to solve this problem. However, DetH* solves the bottom macro-state by assuming all values in the top macro-state are zero as shown in (b). Therefore, it is able to aggregate many more states together, resulting in a much smaller representation of the value function in the bottom macro-state.

SPUDD. Therefore, we expect that the divide-and-conquer approach of DetH* will show an improvement in the same manner as topological value iteration improves value iteration. However, because of the manner in which we create sub-MDPs, the running time improvement is likely to be even more significant. For each sub-MDP, we only allow the values of the states in that sub-MDP to change, fixing the rest at 0 or $-\Delta$. This can exponentially decrease the size of the resulting value function as shown in Figure 2. Therefore, in almost all cases, $|V_H| \ll |V|$, the largest value function SPUDD considers in solving the flat problem, and DetH* can solve problems for which the representation of the optimal value function (or even some intermediate value function) is too large to be machine-representable. Unfortunately, there are pathological cases in which, by forcing states in u to choose actions that will push them towards $\pi_U(u)$, we can have $|V_H| > |V|$, but we expect that these cases are very rare in practice. An example of such a pathological case can be found in Barry *et al.*, 2011.

6 Results

We compared DetH* against SPUDD 3.6.2, which is a state-of-the-art optimal solver for factored domains. In order to illustrate our point about using any solver at the bottom level, we used two different variable orderings of primed variables relative to unprimed variables. The SPUDD default order-

Domain			SPUDD				SPUDD^O				DetH*		
Name	Variables	Actions	Order1		Order2		Order1		Order2		Order1	Order2	Value
			Time	Value	Time	Value	Time	Value	Time	Value	Time	Time	
factory	25	15	156.6 (50)	-17.2	41.3 (50)	-17.2	119.2 (40)	–	31.7 (40)	–	19.0	7.9	-18.5
tire50	25	166	34.8 (6)	0.77	23.8 (50)	0.83	1.5 (5)	0.75	0.2 (6)	0.77	27.7	12.0	0.75
tire80	27	254	127.3 (6)	0.64	268.9 (50)	0.73	1.0 (5)	0.62	0.3 (6)	0.64	52.8	32.2	0.63
tire90	33	291	1556.0 (6)	0.70	2063.9 (50)	0.76	12.2 (5)	0.67	0.5 (6)	0.70	85.3	48.4	0.69
tire110	45	364	– (6)	–	$> 10^5$ (50)	–	6415.8 (5)	0.74	0.6 (5)	0.74	266.2	126.8	0.75
tire120	45	391	– (6)	–	$> 10^5$ (50)	–	38.6 (5)	0.67	0.5 (5)	0.67	7328.8	108.0	0.66
elev10-8	20	3	57.3	-18.6	18.3	-18.6	31.4 (25)	-18.6	10.6 (25)	-18.6	7.8	7.7	-18.9
elev10-10	24	3	485.2	-20.1	256.2	-20.1	219.7 (25)	-20.1	71.3 (25)	-20.1	8.7	6.2	-20.8
elev10-12	28	3	3921.1	-24.9	923.3	-24.9	1614.4 (30)	–	390.8 (30)	–	104.9	95.0	-25.7
trash3-40	51	4	236.5	-15.4	2.3	-15.4	231.0 (29)	–	2.1 (29)	–	2.0	1.2	-16.8
trash3-50	61	4	344.3	-17.9	3.2	-17.9	324.7 (30)	–	2.6 (30)	–	3.1	1.9	-19.9
trash3-60	71	4	421.6	-19.9	4.4	-19.9	378.5 (30)	–	2.7 (30)	–	3.6	2.4	-23.0
trash3-90	102	4	823.5	-27.7	7.8	-27.7	791.6 (50)	–	7.4 (50)	–	10.5	5.8	-33.0
trash5-40	51	4	19496.8	-17.2	23.5	-17.2	19254.3 (30)	–	16.5 (30)	–	2.1	1.4	-18.0
trash5-60	71	4	35642.7	-22.3	37.4	-22.3	34751.1 (30)	–	16.8 (30)	–	4.2	2.7	-24.4
surv3-3	17	11	54.9	-7.2	18.9	-7.2	0.2 (4)	-7.5	0.05 (4)	-7.5	1.9	1.9	-10.0
surv3-5	24	25	–	–	–	–	100.4 (4)	-5.5	1.4 (4)	-5.5	57.6	101.0	-8.3
surv4-5	29	28	–	–	–	–	4458.7 (4)	-5.6	2.7 (4)	-5.6	129.8	217.0	-8.4
survA2-9	29	13	–	–	–	–	– (6)	–	1228.3 (6)	-5.7	1868.2	2294.0	-6.3
survA2-10	29	13	–	–	–	–	– (6)	–	– (6)	–	6876.6	9393.5	-6.4

Table 1: Results for SPUDD, SPUDD^O, and DetH*. All times are in seconds. Note that the number of variables reported is the number *binary* variables used to express the domain. Since $|\Omega_{X^i}|$ might not be a power of two for all X^i, the actual domain size may be less than two to this number. If a horizon was set, the horizon is shown in parentheses after the time. A – under time indicates that more than 2 GB of memory was used and the process was killed. A – under value indicates that at least one of the tested states could not reach the goal under the policy even though it could under the optimal policy. We used SPUDD as the factored MDP solver required by DetH*. All SPUDD algorithms were run to a convergence of 0.1 with no pre-multiplication of ADDs. We set $\epsilon = 0.1$, $\Delta = 100$, and $\Upsilon = 100$. Order1 is an ordering that places all primed variables above unprimed variables. Order2 is an ordering that interleaves primed and unprimed variables. For **tire**, the fraction of time the policy could reach a goal state is reported rather than value.

ing is shown as Order2, while an ordering which places all primed variables above unprimed variables is shown as Order1. Since one of DetH*'s major advantages over SPUDD is shortening the horizon, we also compared to SPUDD^O, an idealized version of SPUDD with a horizon oracle that knew in advance the best time/accuracy trade-off for running SPUDD. Because we solve using an undiscounted model, in domains where not all states can reach a goal state, SPUDD will never terminate. In these domains, we attempted to pick a horizon that gave the optimal policy (listed under SPUDD) and then tried to find the shortest horizon that gave a reasonable approximate policy (listed under SPUDD^O). In some domains we showed that running SPUDD to anything less than convergence resulted in infinitely bad policies since a state that could reach a goal state under the optimal policy could no longer reach a goal state under the approximate policy. In these domains, we report that SPUDD^O was unable to find a good policy. We ran the algorithms on instances of five domains:

1. **factory**: A version of the builder domain distributed with SPUDD modified very slightly to have negative rewards and goal states. Many states cannot reach a goal state, but all states that can reach a goal state have a finite optimal value.
2. **tire**: A version of the PPDDL Tireworld domain [Bonet and Givan, 2006] adapted to the SPUDD language. We chose roads mostly randomly but in such a way that we guaranteed a path from every location to the goal location. Every location had a 25% chance of having a spare tire. Almost all states can reach some state that cannot reach the goal. Therefore, comparing values is not very informative and, instead, we compare the fraction of the time the policy was able to reach a goal state. The size of this domain is approximately controlled by the number of locations; this is shown as tire*#locations*.
3. **elev**: The elevator domain distributed with SPUDD scaled up and slightly modified to have negative rewards and goal states. The size of this domain is controlled by the number of floors and the number of people waiting for an elevator; this is shown as elev*#floors-#people*.
4. **trash**: An example domain involving a robot moving trash bags either by hand or with a cart [Barry *et al.*, 2011]. The size of this domain is controlled by the number of intervening locations between the bags and the cart and the number of bags; this is shown as trash*#locations-#bags*.
5. **surv/survA**: Surveillance domains in which an autonomous agent is trying to take a picture of an interesting location [Barry *et al.*, 2011]. Locations become interesting or not interesting non-deterministically, partly depending on the current location of the agent. The world is divided into neighborhoods, which are far apart, and locations within a neighborhood, which are close together. Once an agent leaves a neighborhood it cannot return. The size of these domains is controlled by the number of neighborhoods and the number locations; this is shown as surv*#neighborhoods-#locations*.

Evaluation: For each problem, we randomly chose 1000 transient states and, for each algorithm, averaged the values of 1000 trials of the policy starting at each of the states. A

state was determined not to be able to reach a goal state if, during policy evaluation, its value fell below -500.

As shown in Table 1, DetH* runs significantly faster than SPUDD on all domains and never runs out of memory. In addition, it is clear that, as claimed, using a faster solver on the macro-states also decreases the time DetH* requires. In all domains where SPUDD was able to run to completion, Order2 was faster than Order1 for both SPUDD and DetH*.

DetH* also runs significantly faster than SPUDD^O in all big domains except **surv** using Order2 and **tire**. In both of these domains, a correct guess of a good short horizon can give an accurate answer to the whole MDP so quickly that the overhead of solving many sub-MDPs is unjustified. However, finding this horizon is not possible in general. In the **tire** domain, the first few iterations are very fast so it would be possible to test a large number of them and find that 5 or 6 iterations gives a good performance faster than DetH* can solve the problem. However, in the **surv** domains, the increase in running time tends to be very non-linear. Four or five iterations may be run very quickly and then the sixth may take a very long time. Therefore, it is not clear how to find a good horizon in this domain. As the rest of Table 1 attests, simply running SPUDD for a fixed amount of time, but not to convergence will not always give good performance (for example, in the **elev** domain).

7 Conclusion

There have been many adaptations to SPUDD, from the suggestions of optimization in the original paper [Hoey *et al.*, 1999], to approximating value functions [St-Aubin *et al.*, 2000], to using affine ADDs [Sanner *et al.*, 2010]. All of these adaptations can be applied to DetH* since it can use *any* solver for the sub-MDPs. In addition, the COST and REGRESS functions used in clustering and solving require the same ADD operations as a SPUDD Bellman backup. Thus any optimization, such as pre-multiplying action ADDs or using affine ADDs, that decrease the running time of those operations will also be expected to decrease the time required for DetH*'s clustering and upper-level solving. The real power of DetH* lies in its ability to break the problem of solving a large MDP into many smaller MDPs. Any advance in the solving of those small MDPs will be reflected directly in the running time of DetH*.

Planning in very large MDPs is difficult because of long horizons, uncertainty in outcomes and large state spaces. The DetH* algorithm addresses each of these. By constructing a temporal hierarchy, DetH* reduces 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. DetH* makes 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. A combination of the shortened horizon and the use of ADD representations significantly reduces the effective state-space size of the remaining MDPs to be solved. The combination of these strategies results in accurate, but still efficient, decisions and we have shown empirically that DetH* can find good approximate policies very quickly.

References

[Barry *et al.*, 2011] J. Barry, L. Kaelbling, and T. Lozano-Pérez. Hierarchical Solution of Large Markov Decision Processes. Technical report, Massachusetts Institute of Technology, 2011.

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

[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.

[Bonet and Givan, 2006] Blai Bonet and Bob Givan. Non-Deterministic Planning Track of the 2006 International Planning Competition. `www.ldc.usb.ve/%7ebonet/ipc5`, 2006.

[Dai and Goldsmith, 2007] Peng Dai and Judy Goldsmith. Topological value iteration. In *Twentieth International Joint Conference on Artificial Intelligence*, pages 1860–65, Hyderabad, India, January 2007.

[Dai *et al.*, 2009] Peng Dai, Mausam, and Daniel S. Weld. Focused topological value iteration. In *Nineteenth International Conference on Automated Planning and Scheduling*, pages 82–89, Thessaloniki, Greece, September 2009.

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

[Hoey *et al.*, 1999] Jesse Hoey, Robert St-Aubin, Alan J Hu, and Craig Boutilier. Spudd: Stochastic planning using decision diagrams. In *Uncertainty in Artificial Intelligence*, Stockholm, Sweden, 1999.

[Mehta *et al.*, 2008] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. Automatic Discovery and Transfer of MAXQ Hierarchies. In *ICML-08*, Helinski, Finland, 2008.

[Sanner and McAllester, 2005] S. Sanner and D. McAllester. Affine algebraic decision diagrams and their application to structured probabilistic inference. In *Nineteenth International Joint Conference on Artificial Intelligence*, 2005.

[Sanner *et al.*, 2010] S. Sanner, W. Uther, and K. V. Delgado. Approximate dynamic programming with affine adds. In *Ninth International Conference on Autonomous Agents and Multiagent Systems*, Toronto, Canada, 2010.

[St-Aubin *et al.*, 2000] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. Apricodd: Approximate policy construction using decision diagrams. In *Neural Information Processing Systems*, Denver, CO, 2000.