

Hierarchical Heuristic Forward Search in Stochastic Domains

Nicolas Meuleau*

NASA Ames Research Center
Mail Stop 269-3
Moffet Field, CA 94035-1000, USA
nmeuleau@email.arc.nasa.gov

Ronen I. Brafman

Department of Computer Science
Ben-Gurion University
Beer-Sheva 84105, Israel
brafman@cs.bgu.ac.il

Abstract

Many MDPs exhibit a hierarchical structure where the agent needs to perform various subtasks that are coupled only by a small sub-set of variables containing, notably, shared resources. Previous work has shown how this hierarchical structure can be exploited by solving several sub-MDPs representing the different sub-tasks in different calling contexts, and a root MDP responsible for sequencing and synchronizing the sub-tasks, instead of a huge MDP representing the whole problem. Another important idea used by efficient algorithms for solving flat MDPs, such as (L)AO* and (L)RTDP, is to exploit reachability information and an admissible heuristics in order to accelerate the search by pruning states that cannot be reached from a given starting state under an optimal policy. In this paper, we combine both ideas and develop a variant of the AO* algorithm for performing forward heuristic search in hierarchical models. This algorithm shows great performance improvements over hierarchical approaches using standard MDP solvers such as Value Iteration, as well as with respect to AO* applied to a flat representation of the problem. Moreover, it presents a general new method for accelerating AO* and other forward search algorithms. Substantial performance gains may be obtained in these algorithms by partitioning the set of search nodes, and solving a subset of nodes completely before propagating the results to other subsets.

In many decision-theoretic planning problems, the agent needs to perform various subtasks that are coupled only by a small sub-set of variables. A good example of this is our main application domain: planetary exploration. In this domain, the agent, an autonomous rover, must gather scientific data and perform experiments at different locations. The information gathering and experiment running task at each site is pretty much self-contained and independent of the other sites, except for two issues: the use of shared resources (such as time, energy and memory), and the state of some instruments that may be used at different locations (for instance, warmed-up or not). The rover has only limited resource for each plan execution phase and it must wisely allocate these resources to different tasks. In most problem instances, the set of tasks is not achievable jointly, and the agent must dynamically select a subset of achievable goals as a function of uncertain events outcome.

These problems are often called *oversubscribed planning problems*. They have a natural two-level hierarchical structure (Meuleau *et al.* 2006). At the lower level, we have the tasks of conducting experiments at each site. At the higher level, we have the task of selecting, sequencing and coordinating the sub-tasks (in the planetary rover domain, this includes the actions of tracking targets, navigating between locations, and warming-up instruments). This hierarchical structure is often obvious from the problem description, but it can also be recognized automatically using factoring methods such as that of (Amir and Englehardt 2003).

The potential benefit of hierarchical decomposition is clear. We might be able to solve a large problem by solving a number of smaller sub-problems, potentially gaining an exponential speed-up. Unfortunately, the existence of a hierarchical decomposition does not imply the existence of an optimal decomposable policy. Thus, one must settle for certain restricted forms of optimality, such as hierarchical optimality (Andre and Russell 2002), or consider domains in which compact optimal policies exist, such as those that satisfy the *reset assumption* of (Meuleau *et al.* 2006).

This paper formulates a hierarchical forward heuristic search algorithm for And-Or search spaces, which we refer to as *Hierarchical-AO** (HiAO*). This algorithm exploits a hierarchical partition of the domain to speed up standard AO* search (Nilsson 1980). It may be applied anywhere that AO* may be applied, that is, for all problems that can be represented as an acyclic And-Or graph, provided a hierarchical partition of the search space is defined. This could be MDPs or any other search problems.¹ Although there is no formal guarantee that it will result in actual performance improvements, our simulations with the rover domain show that it has a great potential for oversubscribed planning problems. Further work is required to show the relevance of this approach in a more general setting. However, this paper outlines directions that could benefit a wide variety of application domains, and other forward search algorithms such as A*.

The reason why forward heuristic search can be beneficial in hierarchical MDPs is pretty obvious. It is the well-

*QSS Group Inc.

¹Like standard AO*, HiAO* is not designed to work on problems that contain loops. A similar extension to that in (Hansen and Zilberstein 2001) is required to handle these problems.

known benefit of using reachability information and admissible heuristic. The most efficient algorithms for solving flat MDPs, such as (L)AO* (Hansen and Zilberstein 2001) and (L)RTDP (Barto *et al.* 1995; Bonet and Geffner 2003), accelerate the search by pruning states that cannot be reached from a given starting state under an optimal policy. The same benefit can be expected in a hierarchical representation of the problem. Hierarchical algorithms such as Abort-Update (Meuleau *et al.* 2006) must resolve similar sub-problems, but in different calling contexts. Implementations based on classical MDP solvers such as Value Iteration or Policy Iteration, which ignore the initial state, must consider all possible contexts. Forward search algorithms consider only reachable contexts, and with a good heuristic function, only a subset of these.

More interesting are the reasons why HiAO* can outperform standard heuristic search over flat-domains, as implemented in standard AO*. The first reason has to do with the order and efficiency of the value propagation step, which is known to be the costliest part of AO*. HiAO* embodies a form of lazy approach to value updates: the update work is concentrated within a particular sub-domain each time, and work on other domains is postponed until their value is needed.

The second benefit of the hierarchical partitioning of the domain is the possibility to use *macro-connectors*. When the optimal solution over a subset of states is known, it can be represented as a macro-connector similar to the *temporally abstract actions* or *macro-actions* used in Reinforcement Learning (Sutton *et al.* 1999). Macro-connectors are then used while solving other subsets of states, to accelerate the traverse of the graph.

The third benefit of forward heuristic search in a hierarchical representation is the possibility of using a known solution of a sub-task to seed the algorithm in charge of solving similar sub-tasks. As will become apparent later, the dynamics within a sub-problem is not influenced by the calling context. The latter influence the value of nodes, and so, the optimal policy, but not the search space structure. Therefore, we can reuse expanded solutions for different calling context, using them as starting points. Although farther expansion will be required, we are likely to benefit from much of this work.

Finally, with a decomposed domain, we believe that one can formulate more accurate heuristic functions that are appropriate for each of the sub-domains. A simple example would be the assessment of the value of a sub-domain under new calling contexts. We can simply use the value computed for similar contexts. A more sophisticated example would be the use heuristics specifically designed for particular sub-domains.

In this paper, we focus on explaining and testing the first three points above. We provide an abstract, and quite general interpretation of the use of any hierarchical decomposition in AO* in terms of (1) propagation order, (2) extra level by-passing edges, and (3) local solution re-use. These three properties emerge naturally when one attempts to search hierarchically, and they can be used given an arbitrary partitioning of the state space (that is, beyond the application

to oversubscription planning). This formulation, its abstract interpretation, and its evaluation in the rover domain are the main contributions of this paper.

In the next section, we explain how we model hierarchical MDPs. Then, we review the AO* algorithm, we explain the new HiAO* algorithm, and we show how it can be applied to oversubscribed planning problems. Finally, we describe our empirical evaluation of HiAO* and conclude.

Hierarchical MDPs

A Markov Decision Process (MDP) (Puterman 1994) is a four-tuple $\langle S, A, T, R \rangle$, where S is a set of states, A is a set of actions, $T : S \times A \times S \rightarrow [0, 1]$ is the transition function which specifies for every two states $s, s' \in S$ and action $a \in A$ the probability of making a transition from s to s' when a is executed, and $R : S \times A \times S \rightarrow \mathbb{R}$ is the reward function. We augment the above description with a concrete initial state $s^{\text{init}} \in S$, obtaining a quintuple $\langle S, A, T, R, s^{\text{init}} \rangle$. In this paper, we focus on bounded-horizon MDPs (in which the length of each trajectory is bounded by a finite number) and we use the undiscounted expected reward criterion.

We shall concentrate on factored MDPs which have the form $\langle X, A, T, R, s^{\text{init}} \rangle$. Here X is a set of state variables, and A, T, R, s^{init} are as before. The variables in X induce a state space, consisting of the Cartesian product of their domains. To simplify notations, we assume all variables are boolean, that is, they are *fluents*. Typically, it is assumed that the transition function T is also described in a compact manner that utilizes the special form of the state space, such as dynamic Bayes net (Dean and Kanazawa 1993) or probabilistic STRIPS rules (Hanks and McDermott 1994). In this paper, we do not commit to a concrete action description language, but we expect it to be variable-based, such as the above methods, as we assume that it is easy to identify the relevant variables with respect to an action $a \in A$. This is the set of variables whose value can change when a is executed, as well as those variables that affect the probability by which these variables change their value and the immediate reward received for executing a .

An hierarchical decompositions of an MDP consists of a set of smaller factored MDPs. In its definition, the notion of a projection of T and R plays a central role. Let X' be some subset of the variables in X . The projection of T over X' w.r.t. action $a \in A$ is well defined if a does not affect the value of the variables in $X \setminus X'$ and the transition probabilities and reward of a do not depend on the value of these variables. In that case, the actual projection can be obtained by fixing some arbitrary value to the variables in $X \setminus X'$.

Formally, we define a *hierarchical decomposition* of a factored MDP $M = \langle X, A, T, R, s^{\text{init}} \rangle$ as a tree of factored MDPs, $M_H = \{M_0, \dots, M_n\}$, where $M_i = \langle X_i, A_i^+, T_i, R_i, s_i^{\text{init}} \rangle$. Each M_i is just a factored MDP. The only difference is that its set of actions, A_i^+ contains a number of special control actions which signify the passing of control to its parent or one of its children. These actions encode the hierarchy, too. More specifically, $X_i \subseteq X$; A_i^+ is the union of some subset $A_i \subseteq A$ together with two types

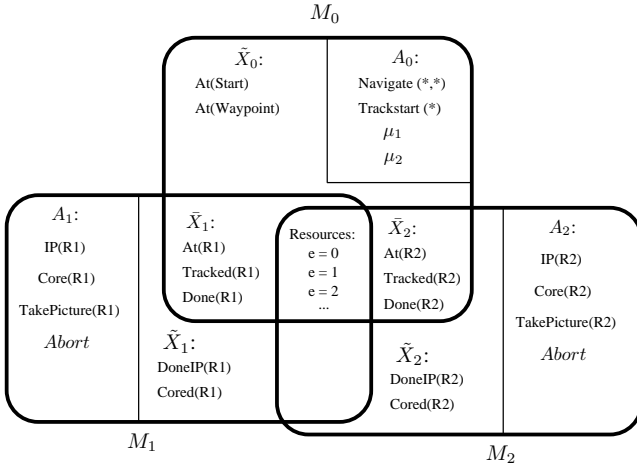


Figure 1: Decomposition of a simple rover problem: a root process M_0 navigates among two rocks, R1 and R2, and schedules two corresponding sub-processes M_1 and M_2 (IP stands for “Instrument Placement”). \tilde{X}_i is the set of private variables of process i , and \tilde{X}_i is the separation between sub-task i and the root process.

of control actions: μ_j corresponds to passing control to a child process M_j , and *Abort* signifies the passing of control back to the parent process; T_i is the projection of T over X_i , such that T_i is well-defined; R_i depends on X_i and A_i only; and s_i^{init} is the projection of s^{init} over X_i . In addition, $\bigcup_{i=1,\dots,n} X_i = X$, $\bigcup_{i=1,\dots,n} A_i = A$, $A_i \cap A_j = \emptyset$ for every $i \neq j$, and whenever $x \in X_i \cap X_j$ then $x \in X_k$ for every k such that M_k is on the path between M_i and M_j (which is known as the *running intersection* property). See (Meuleau *et al.* 2006) for a more detailed description of this model. A decomposition of M into M_H can be obtained automatically by a straight forward extension of the methods described in (Amir and Englehardt 2003) for deterministic planning domains.

As an illustration, Fig. 1 describes a decomposition of a simple planetary rover domain into three sub-domains. M_0 is the top-level domain in which we control the position of the rover and the choice of which subtask to perform next (these are the μ_i actions). M_1 and M_2 represent the subtasks associated with experimenting on one of two rocks. M_0 shares some variables with its children, and the resource level is a variable shared by all.

Hierarchical AO*

We start with a short review of the AO* algorithm. Next, we show how a hierarchical partition of the search nodes may be exploited to accelerate the algorithm, and we provide the pseudo-code of a new general algorithm called *Hierarchical-AO** (HiAO*).

The AO* Algorithm

AO* (Nilsson 1980) is a generalization of A* to And-Or graphs. As such, it can be applied to the problem of gen-

- 1: Initialize the explicit graph G to the start state s^{init} .
- 2: Mark s^{init} as *open*.
- 3: **while** the greedy solution graph contains some *open* nodes **do**
- 4: $s :=$ any *open* node of the greedy graph.
- 5: Unmark s as *open*.
- 6: EXPAND(s). // *Expand the greedy graph*
- 7: UPDATE(s). // *Update state values and mark best actions*
- 8: Return the greedy solution graph.

Algorithm 1: Standard AO*.

- 1: **for** each $a \in A$ **do**
- 2: **for** each $s' \in S$ such that $T(s, a, s') > 0$ **do**
- 3: **if** $s' \notin G$ (s' is not already present in G) **then**
- 4: Add s' to G .
- 5: **if** s' is a terminal state **then**
- 6: $V(s') := 0$.
- 7: **else**
- 8: $V(s') := H(s')$.
- 9: Mark s' as *open*.
- 10: Add the connector (s, a, \cdot) to G .

Algorithm 2: EXPAND(s).

erating a policy for an MDP given an initial state, assuming there are no loops in the transition graph (Hansen and Zilberstein 2001). In MDPs, an OR node corresponds to a choice of an action at a state, and an AND node corresponds to the set of possible states resulting from each such choice. The possible AND nodes are annotated by their probability.

AO* maintains two basic structures: the *explicit graph* and the *greedy graph*. The explicit graph simply depicts the portion of the search space that has been expanded so far. The greedy graph is a subgraph of the explicit graph which represents those nodes that are reachable by the greedy policy. To obtain the greedy graph, one needs to perform dynamic programming in the explicit graph, propagating values bottom-up from the leaf nodes (using their heuristic values) to the root node. The value of an AND node is the expected value of its children, and the value of an OR node is the maximum over the value of its children. If we note the choice that yielded this maximum, we can now generate the greedy graph by simply following these choices starting at the root node. The pseudo-code for AO* is described in Algorithms 1 - 3.

If an *admissible* heuristic function is used in AO*, then the value of each node may only decrease at each iteration.² When we update the value of a node outside the greedy graph, the greedy graph may not be affected because the update can only reduce the value of the node. Thus, a choice that was sub-optimal cannot become optimal. Consequently, new node values need only be propagated up edges marked as *optimal*, i.e., denoting the optimal choice in that state.

In most cases, the most expensive parts of AO* are: (1)

²In a reward-maximization framework, an admissible heuristic is an heuristic that never underestimates the value of a state.

- 1: Create set Z containing only the state s .
- 2: **repeat**
- 3: Remove from Z a state s' such that no descendent of s' in G occurs in Z .
- 4: Set $V(s') := \max_{a \in A} \left[\sum_{s'' \in G} T(s', a, s'') (R(s', a, s'') + V(s'')) \right]$ and mark the connector associated with the best action in s' as *optimal*.
- 5: **if** $V(s')$ decreased at previous step (it cannot increase) **then**
- 6: Add all parents of s' along connectors marked as *optimal* to Z .
- 7: **until** Z is empty.

Algorithm 3: UPDATE(s).

- 1: Initialize the explicit graph G to the start state s^{init} .
- 2: Mark s^{init} as *open*.
- 3: SOLVE($\mathcal{S}_0, s^{\text{init}}$).
- 4: Return the greedy solution graph starting at s^{init} .

Algorithm 4: Hierarchical AO*.

the update stage, where we propagate value changes up the explicit graph (Alg. 3); (2) the computation of the *fringe*, that is, the set of open nodes in the greedy graph (line 2 of Alg. 1). We show below how a partition of the states expanded by AO* may be exploited to accelerate these two operations.

The HiAO* Algorithm

We now assume that the state space S is partitioned into subsets $\mathcal{S}_0, \dots, \mathcal{S}_k$. Furthermore, we assume that these subsets are organized in a hierarchy in the form of tree,³ so that: (1) the starting state s^{init} belongs to the root subset denoted \mathcal{S}_0 ; (2) state transitions are possible only between states in the same subsets, or between two states belonging to two subsets one being the parent of the other. These choices are motivated by the hierarchical MDP framework presented above, where a child subset represents a sub-process that can be called from the current subset/process (therefore, we sometimes use the term “subprocess” to designate a child subset). However, our formalism is general enough for it can be applied to any applications of AO* where a hierarchy of nodes can be defined.

The HiAO* algorithm is presented in Alg. 4 - 6. The algorithm also uses the same function EXPAND() as standard AO* (Alg. 2). The basic principle of this algorithm is the following: each time that the optimal action in a state leads to a child subset (in other words, each time it appears optimal to call a sub-process from the current state), we recursively call the HiAO* algorithm to solve this child subset completely before continuing solution of the current level. We explain below how this simple mechanism may benefit the algorithm.

³The algorithm can be generalized to handle hierarchies represented by a directed acyclic graph.

- 1: **while** the greedy solution graph starting at s contains some *open* nodes in \mathcal{S}_i **do**
- 2: $s' :=$ any *open* node of the greedy graph starting at s in \mathcal{S}_i .
- 3: Unmark s' as *open*.
- 4: EXPAND(s').
- 5: Mark all nodes created at previous step and that belong to a child of \mathcal{S}_i as *outdated*.
- 6: HIERARCHICALUPDATE(\mathcal{S}_i, s').

Algorithm 5: Function SOLVE(\mathcal{S}_i, s).

Delaying the propagation of new values: The main principle implemented by HiAO* is to delay the propagation of new node values in between states belonging to different subsets. Let us first consider why delaying updates might be useful. Suppose that we have expanded node u and then node v , and following their expansion, their value has changed. In standard AO*, following the change in value in u we would propagate this information upwards along marked edges. Then, we would repeat the process for v . However, it is quite possible that u and v have common ancestors, and thus we will update these ancestors twice. Sometimes, such an update is important because it influences our choice of which node to expand. Sometimes, such an update can be delayed, and we can save the repeated update of shared ancestors. HiAO* tries to exploit the second case, assuming it happens more often than the first case.

The general scheme we propose for delaying updates is as follows. We associate to each subset \mathcal{S}_i a set of nodes to update Z_i similar to the set Z used in Alg. 3. At any particular point, the algorithm has a single subset of nodes in focus and performs the standard AO* operations only inside of that subset. Whenever a state $s \in \mathcal{S}_i$ is expanded and thus needs to be evaluated, we insert it into Z_i and enter a loop that will propagate its value up into the graph by emptying and re-filling the set Z_i , as in standard AO* (Alg. 3). However, each time that a new value needs to be propagated to a state s' belonging to another subset $\mathcal{S}_j, j \neq i$ (\mathcal{S}_j is necessarily the parent or a child of \mathcal{S}_i), s' is added to Z_j and not to Z_i (line 18 of Alg. 6), and so, it is excluded from the loop that works only with Z_i . In short, the propagation is limited to the subset \mathcal{S}_i , and nodes that would need to be updated but belong to another subset \mathcal{S}_j are just stored in Z_j , but they are not re-evaluated yet and their new value is not propagated yet.

So, when are the nodes in $\mathcal{S}_j, j \neq i$ updated? The answer depends on the hierarchical relation between \mathcal{S}_j on \mathcal{S}_i . If \mathcal{S}_j is the parent of \mathcal{S}_i , then the states that have been put in Z_j during the solution of \mathcal{S}_i are updated when the algorithm moves focus from \mathcal{S}_i back to its parent subset \mathcal{S}_j . If \mathcal{S}_j is child of \mathcal{S}_i , then the states in Z_j are updated at the latest possible time, that is, when we want to evaluate a state outside of \mathcal{S}_j and that may lead into \mathcal{S}_j under the optimal action. This may never happen, if calling sub-process \mathcal{S}_j is the optimal decision in none of the states visited by the algorithm later on, in which case we save all the work of updating states in \mathcal{S}_j . In practice, this is implemented through

```

1:  $Z_i := Z_i \cup \{s\}$ .
2: while  $Z_i \neq \emptyset$  do
3:   Remove from  $Z_i$  a state  $s'$  such that no descendent of  $s'$  in  $G$  occurs in  $Z_i$ .
4:   Set  $V(s') := \max_{a \in A} [\sum_{s'' \in G} T(s', a, s'') (R(s', a, s'') + V(s''))]$ , call  $a^*(s')$  the optimal action in  $s'$ , and mark the associated connector as optimal.
5:   while executing  $a^*(s')$  in  $s'$  may lead to states that are not in  $\mathcal{S}_i$  and that are marked as outdated do
6:     for each child  $\mathcal{S}_j$  of  $\mathcal{S}_i$  do
7:       for each  $s'' \in \mathcal{S}_j$  such that  $T(s', a^*(s'), s'') > 0$  and  $s''$  is marked as outdated do
8:         Unmark  $s''$  as outdated.
9:         HIERARCHICALUPDATE( $\mathcal{S}_j$ , NULL).
10:        SOLVE( $\mathcal{S}_j$ ,  $s''$ ).
11:        Set  $V(s') := \max_{a \in A} [\sum_{s'' \in G} T(s', a, s'') (R(s', a, s'') + V(s''))]$ , call  $a^*(s')$  the optimal action in  $s'$ , and mark the associated connector as optimal.
12:   if  $V(s')$  decreased at previous step (it cannot increase) then
13:     for each state  $s''$  that is a parent of  $s'$  along a connector marked as optimal do
14:       if  $s'' \in \mathcal{S}_i$  then
15:         Add  $s''$  to  $Z_i$ .
16:       else
17:         Call  $\mathcal{S}_j$ ,  $j \neq i$ , the subset containing  $s''$ .
18:         Add  $s''$  to  $Z_j$ .
19:       if  $\mathcal{S}_j$  is a child of  $\mathcal{S}_i$  then
20:         Mark  $s''$ , and all its predecessors in  $\mathcal{S}_j$  along connectors marked as optimal, as outdated.

```

Algorithm 6: Function HIERARCHICALUPDATE(\mathcal{S}_i, s).

the use of *outdated* markers, as explained below.

One of the basic principle of the HiAO* algorithm is that, when we are done updating a node in \mathcal{S}_i that can possibly lead to a node s in a child subset \mathcal{S}_j ($j \neq i$) under the optimal action, then we must know with accuracy the set of optimal decisions that leads from s either to terminal states, or back into \mathcal{S}_i . A node of \mathcal{S}_j is marked as *outdated* if the greedy subgraph starting at that node has a chance of not being accurate or complete. Suppose that the algorithm is currently working in subset \mathcal{S}_i and needs to propagate a new value to a state s belonging to a child \mathcal{S}_j of \mathcal{S}_i . Then, as explained above, the update of s is delayed and s is just added to Z_j (line 18 of Alg. 6). Moreover, all nodes of \mathcal{S}_j “above” s in the greedy graph are marked as *outdated* (line 19 and 20 of Alg. 6). Later, if we want to evaluate a node outside of \mathcal{S}_j that can lead to a state $s' \in \mathcal{S}_j$, the value of s' may not be accurate, because we have delayed propagation of new values in \mathcal{S}_j . If this is the case, then s' must be marked as *outdated* and, consequently, two operations are performed: (1) we purge the set Z_j and update all the nodes in \mathcal{S}_j that need to be re-evaluated using a recursive call to HIERARCHICALUPDATE() (line 9 of Alg. 6). This may change the

greedy policy in \mathcal{S}_j below some of the entry nodes of \mathcal{S}_j , including node s' ; (2) the greedy graph starting at s' in \mathcal{S}_j is updated through a call to SOLVE() (line 10 of Alg. 6). So, the *outdated* marker in s' may be deleted (line 8 of Alg. 6), but the markers of all other entry nodes are kept.⁴ If we later get to update another state outside of \mathcal{S}_j and that can lead to $s'' \in \mathcal{S}_j$, then the outdated marker of s'' will indicate that the greedy graph below s'' in \mathcal{S}_j needs to be updated, even if no new node has been added to Z_j in the mean time.

Convergence: As long as the problem contains no loop, Hierarchical AO* is guaranteed to terminate in finite time and to return the optimal solution:

Theorem 1 *If the heuristic H is admissible, then HiAO* returns the optimal policy in finite time.*

HiAO* may not always be efficient, particularly if the number of connections between subsets is large. However, we claim that, in the case of oversubscription planning, the state partition induced by the hierarchy is efficient and allows leveraging the general principle presented above. The simulation results presented in this paper support this claim.

We now present two acceleration techniques that can be implemented in HiAO* to get further leverage from the hierarchy, but are not used in the pseudo-code of Alg. 4 - 6.

Optimizing the algorithm: The HiAO* algorithm presented above exhibits the following inefficiency. If an action a leading with certainty to a state $s' \in \mathcal{S}_j$ appears optimal in state $s \in \mathcal{S}_i$ ($i \neq j$), then the algorithm will SOLVE completely the sub-problem \mathcal{S}_j starting in s' before returning to state s . In the process of solving \mathcal{S}_j , the Q-value of action a in s may only decrease, so that it may happen that, before we are done solving \mathcal{S}_j , a does not appear as optimal in s anymore. The algorithm presented above will not detect this and single-mindedly continue solving \mathcal{S}_j until the greedy policy starting in s' is known, which can be highly inefficient.

This issue can easily be addressed under an additional hypothesis that is satisfied in the hierarchical planning context described above and that could easily be relaxed to deal with a more general case. We assume that each action either does not change the partition subset (in the case of hierarchical planning, this is a primitive actions $a \in A$), or it leads with certainty to a single state belonging to another subset than the current (this is the case of control passing actions μ_i and *Abort*). Then, we can modify the algorithm by adding a threshold parameter τ to the function SOLVE(\mathcal{S}, s) so that, when the value of s falls below τ during the solution of \mathcal{S} , the function exits. In the initial call to SOLVE (line 3 of Alg. 4), the threshold parameter is set to $-\infty$, so that optimization will be pursued until its end. Later calls (line 10 of Alg. 6) are performed in the following way: while computing the value of a state (lines 4 and 11 of Alg. 6), we record the Q-value of the best action that does not change the partition subset (primitive action), the Q-value of the best action that induce a change of subset (control-passing action), and

⁴A straightforward improvement to the algorithm is to delete the *outdated* marker in all nodes of \mathcal{S}_j belonging to the greedy graph starting in s' after recomputing this graph.

the Q-value of the second best control-passing action. Then, if a control-passing action appears optimal and we enter the loop in lines 5 to 11, then the threshold parameter τ used for the call to SOLVE() in line 10 is set to the Q-value of the best primitive action, or the second best control-passing action, depending on which is greater. In this way, SOLVE aborts the solution of a child subset as soon as the action leading into it does not appear optimal anymore. Note that the outdated marker is removed (line 8 of Alg. 6) only if SOLVE completed.

By-passing levels: Two of the operations of standard AO* require it to traverse up or down a large part of the explicit graph. The function EXPAND() selects a node on the fringe of the greedy graph. This requires computing the fringe, which is done by following marked edges down from the root node. This operation is often one of the most costly.⁵ The function UPDATE() also propagates information up the graph, when determining the greedy parents of a recently updated nodes (line 6 of Alg. 3). Because HiAO* performs the standard AO* operations only inside of a subset of nodes, it saves time on both operations. First, the search graph projected to each subset is smaller, so that the size of the fringe for that subset and the time to compute it is likely to be (much) smaller. Of course, by doing this, we expand the most promising fringe node in a given subset, and not in the entire greedy graph as AO* does. So, we influence the order in which nodes are selected for expansion, which may have good or bad consequences. Second, as explained before, HiAO* postpones the propagation of some values from a subset to another other, and thus limits the propagation of information up in the graph.

To gain more leverage, HiAO* may be augmented with *macro-connectors* that represent the effect of applying the greedy solution for a subset of nodes. Macro-connectors are then used when solving other subsets, to accelerate the traverse of the graph. Imagine, for instance, that u belongs to \mathcal{S}_i , and applying some action a leads from u to $u' \in \mathcal{S}_j$, $j \neq i$, and then applying a' leads from u' to $u'' \in \mathcal{S}_i$. Then, we add an explicit edge between u and u'' and we use it when solving \mathcal{S}_i . By doing this, we build, in effect, a projection of the greedy graph on \mathcal{S}_i . The new edge between u and u'' helps us during the solution of \mathcal{S}_i in two circumstances: when we compute the fringe, we can jump over a large number of states that flat AO* would have considered individually; and similarly when we determine the greedy parents of a recently updated node. Creating and maintaining macro-connectors increases the complexity of HiAO* slightly, but the efficiency gained compensates for this extra cost. Indeed, our simulations showed that this is one of the major causes of the good performances exhibited by our algorithm.

Application to Oversubscription Planning

We now focus on the class of problems that motivates this work, that is, oversubscribed planning problems. Follow-

⁵It is possible to store the fringe and update it as we progress, but it turns out that because the structure we are searching is a directed acyclic graph, not a tree, this is a costly operation.

ing (Meuleau *et al.* 2006), these problems are naturally modelled in the hierarchical planning framework presented above with a two-level hierarchy. There is a single root process M_0 and multiple leaf/child processes M_i , $i = 1, \dots, n$ representing the different subtasks the agent can select to perform. For each sub-process M_i , we define:

- $\bar{X}_i = X_i \cap X_0$ are the separator (shared) variables between M_0 and M_i . The shared variables include, but are not limited to, the shared resource levels. They may also include, for instance, the state of instruments used to perform several subtasks.
- $\tilde{X}_i = X_i - X_0$ are the private variables of M_i . Typically, they represent the state of advancement of the subtask.
- $X_{0-i} = X_0 - X_i$ is the difference of M_0 and M_i .

The set of private variables of M_0 is defined as $\tilde{X}_0 = X_0 - \bigcup_{i=1}^n X_i = \bigcap_{i=1}^n X_{0-i}$. We also define various classes of states: $S = 2^X$, $S_i = 2^{X_i}$, $\bar{S}_i = 2^{\bar{X}_i}$, $\tilde{S}_i = 2^{\tilde{X}_i}$, $S_{0-i} = 2^{X_{0-i}}$.

We restrict our attention to domain satisfying the *reset assumption* of (Meuleau *et al.* 2006). The *reset assumption* states that whenever the process moves into a child sub-MDP, the state of the private variables of this sub-MDP is the same. It holds true in the rover domain because the rover must have its arm stowed and all of its instruments parked prior to any movement, and so, this is its configuration when it arrives at a new rock/sub-task. Moreover, actions of preparing the rock for a measurement, such as coring the rock, have to be redone if we abort this rock before completing the measure, because it is not possible to put the rover arm exactly at the place it was when the rock was cored. So, all intermediate work towards the goal is lost once we move to another rock. Note that variables that are not private to the child sub-MDP, such as resource levels, are not reset to their initial value when we abort the sub-task. Note also that this assumption is not as restrictive as it may look, since the user may modify the sets \bar{X}_I if needed: if some intermediate work may be preserved when we abort a sub-task M_i before its completion, then the fluent representing this intermediate state should be moved from \tilde{X}_i to \bar{X}_i . However, HiAO*, as well as the algorithms in (Meuleau *et al.* 2006) work better if the separation sets are as small as possible.

Under the *reset assumption*, the space of reachable states is naturally partitioned into a hierarchy of subsets. The root subset S_0 represents the states where all subtasks are in their initial condition. For each subtask i and each state $s_{0-i} \in S_{0-i}$, there is a child subset $S_i[s_{0-i}]$ containing all states where (1) M_i is not in its initial condition, (2) the private variable of every other subtasks M_j , $j \neq i$, have their initial value, and (3) the variables in X_{0-i} are equal to s_{0-i} , and so, s_{0-i} represents the context in which subtask i is called. All other states, for instance, states where two subtask are not in their initial state, are not reachable. The Abort-Update algorithm of (Meuleau *et al.* 2006) is an instance of Asynchronous Policy Iteration (Bertsekas and Tsitsiklis 1997) that exploits this property. Here, we stress that this hierarchical partition may serve as the basis to an

implementation of HiAO*. In a sense, HiAO* is to Abort-Update what standard AO* is to Policy Iteration.

Reusing macro-actions: As explained above, when we apply HiAO* in to an oversubscribed planning problem, there is a subset $\mathcal{S}_i[s_{0-i}]$, for each subtask M_i and each context s_{0-i} in which M_i may be started. For a fixed i , the dynamics within $\mathcal{S}_i[s_{0-i}]$ do not depend on the context s_{0-i} . The latter influences the value of states where we transit when we abort the subtask (the *exit values*), and so, the optimal policy, but not the structure of the explicit graph representing $\mathcal{S}_i[s_{0-i}]$. Therefore, if we have computed part of the explicit graph for a given context, we can reuse/copy it when we want to solve the same subtask in another context. We might need to expand this graph further, because different exit values may induce different policies. Similarly, this seed may contain some nodes that we would not have created if we started the solution of subtask i in the new context from scratch. However, we could benefit from this because because much of the work of expanding the graph is already done.

Optimality: HiAO* finds the optimal solution within the predefined hierarchy, which is the definition of hierarchical optimality (Andre and Russell 2002). As shown in (Meuleau *et al.* 2006), if the problem satisfies the *reset assumption*, then there is an absolute optimal policy that can be encoded in the hierarchy defined above. Therefore, HiAO* finds the optimal policy if the *reset assumption* holds. If the *reset assumption* does not hold, HiAO* based on the hierarchy defined above finds a hierarchical optimal policy, which may of lesser value than an absolute optimal policy. However, we can use a different hierarchical representation of the problem that would guarantee absolute optimality. This is achieved by defining a subset $\mathcal{S}_i[s_{-i}]$ for each sub-task i and each state $s_{-i} \in S_{-i} = 2^{X \setminus X_i}$ (note that, now, the context also include the private variable of sub-tasks $j \neq i$).

Experimental Evaluation

One of the goals of our empirical evaluation is to show that the fundamental structural changes of HiAO*, inherited from Partitioned-AO*, are beneficial. Figure 2 presents a typical performance curve obtained with a complex, real-size, rover domain. It shows the evolution of the solution time of standard AO* and HiAO* as a function of the initial resource available to the rover. Although the complexity of the problem increase badly with the initial resource, this increase is much less in HiAO* due to the savings during value updates and graph traversal. Very similar results were obtained with nearly all problem instances tried (the exception being tiny problem instances).

We also wish to show that reachability information is crucial to scaling up the hierarchical approach to over-subscribed planning problem. In fact, there is no need to run any simulation to establish this point. In the problem instance used to produce Fig. 2, with an initial resource of 100, the root MDP contains 142 fluents (and so, 2^{142} states), and each sub-MDP contains 114 fluents (2^{114} states). So, this problem is far out of the range of hierarchical algorithms

such as Abort-Update (Meuleau *et al.* 2006) that are based on a standard MDP solver such as Value Iteration.

In future work, we shall examine the additional possible benefits of hierarchical decompositions: the reuse of sub-trees between similar contexts, and the ability to formulate specialized and more accurate heuristic functions. We note that initial experiments have not yielded any advantage to the reuse of sub-trees, but a more serious assessment of this aspect is still required.

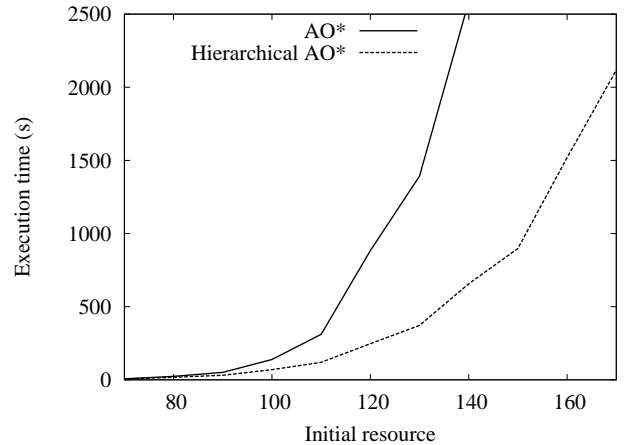


Figure 2: Execution time on a sample rover problem.

Conclusions

We showed how AO* can be modified to make use of hierarchical decompositions of a domain. This leads to the first forward search algorithm that explicitly deals with hierarchical domains. We analyzed the relationship between standard AO* and the hierarchical version in terms of its effect on the order of propagation and the introduction of shortcuts in the form of edges that by-pass levels. Our experimental evaluation of these differences indicate that they are, indeed beneficial. In future work we will take a closer look at the additional likely benefits of an hierarchical approach: the reuse of work within each sub-domains, and the use of more specialized heuristics for sub-domains.

Acknowledgements

This work was supported by NASA Intelligent Systems program under grant NRA2-38169. This work was performed while Ronen Brafman was visiting NASA Ames Research Center as contractor from the Research Institute for Advanced Computer Science. Ronen Brafman was supported in part by The Paul Ivanier Center for Robotics Research and Production Management.

References

- E. Amir and B. Englehardt. Factored planning. In *IJCAI'03*, pages 929–935, 2003.
- D. Andre and S. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI'02*, 2002.

- A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 672(1-2):81–138, 1995.
- D.P. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1997.
- B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS'03*, 2003.
- T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1993.
- S. Hanks and D. V. McDermott. Modeling a dynamic and uncertain world I: Symbolic probabilistic reasoning about change. *Artificial Intelligence*, 66(1):1–55, 1994.
- E. A. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.
- N. Meuleau, R. Brafman, and E. Benazera. Stochastic over-subscription planning using hierarchies of MDPs. In *ICAPS'06*, 2006.
- N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.
- M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, NY, 1994.
- R.S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.