Exploiting Expert Knowledge in Factored POMDPs

Felix Müller and **Christian Späth** and **Thomas Geier** and **Susanne Biundo** [1]

Abstract. Decision support in real-world applications is often challenging because one has to deal with large and only partially observable domains. In case of full observability, large deterministic domains are successfully tackled by making use of expert knowledge and employing methods like Hierarchical Task Network (HTN) planning. In this paper, we present an approach that transfers the advantages of HTN planning to partially observable domains. Experimental results for two implemented algorithms, UCT and A* search, show that our approach significantly speeds up the generation of high-quality policies: the policies generated by our approach consistently outperform policies generated by Symbolic Perseus and can be computed in less than 10% of its runtime on average.

1 Introduction

Decision support in real-world applications, such as user assistance or the automated control of technical systems, requires advanced planning technology. In these settings, uncertainty often arises because state information depends on sensory input. Many such domains are large and also pose the problem that a solution needs to be found quickly. This is especially true in the case of user assistance: a user cannot be asked to wait for minutes before decision support is provided [2]. While factored Partially Observable Markov Decision Processes (POMDPs) can adequately represent such domains [3], solving them is difficult because scaling beyond medium-sized domains is problematic.

The challenge to get along with large planning domains is addressed by approaches that use (variants of) hierarchical task network planning [4]. HTNs provide the means to encode standard solutions to planning problems and thereby enable the exploitation of expert knowledge in solution discovery. Applications in many real-world domains rely on HTN-based systems [10]. However, HTN-based systems are aimed at fully observable deterministic domains.

In this paper, we present an expressive and scalable approach to exploit expert knowledge in partially observable domains by adapting HTN planning to POMDPs. We extend our earlier work [9] by introducing more expressive action abstraction. We apply two algorithms, UCT [7] and A* search, and provide data for a number of experiments on planning problems taken from the 2011 International Probabilistic Planning Competition (IPPC). Our experimental results show that we are able to generate high-quality policies quite quickly: on average, the policies generated by our approach consistently outperformed policies generated by Symbolic Perseus, while generating them took less than 10% of Symbolic Perseus' runtime on average.

The problem of planning in large and only partially observable environments has been addressed by various authors. On one hand, traditional non-hierarchical methods have been improved. Silver and Veness [13] present an adaptation of the UCT algorithm to POMDP planning, for example. On the other hand, some authors have begun leveraging the advantages of HTN planning in partially observable contexts. E.g., Kuter et al. [8] introduce an HTN-like approach in partially observable settings not based on POMDP semantics.

The rest of the paper is organized as follows. We review the relevant background for factored POMDPs and HTN planning in Section 2. Our HTN-like planning formalism for factored POMDPs and the application of UCT and A* are introduced in Sections 3 and 4. We present the experimental results in Section 5 and conclude with some final remarks in Section 6.

2 Background

Below, we present relevant background on POMDPs and HTNs.

2.1 POMDPs

A POMDP models a partially observable domain as a tuple $(S, A, T, R, O, Z, b_0, h)$, where S, A, and O are sets of system states, actions, and observations, respectively. The transition function $T(s, a, s') = P(s'|s, a)$ determines the probability of s' being the successor state after executing action a in state s. Analogously, $Z(s', a, o) = P(o|a, s')$ represents the probability of receiving observation o when executing action a resulted in state s'. The goals of the planning agent are given as non-positive rewards by $R(s, a) \in \mathbb{R}_0^-$. Where convenient, we will speak in terms of costs instead of negative rewards. To account for uncertainty in the initial knowledge about the system state, $b_0(s)$ defines a probability distribution over states, the so-called initial belief state. The time horizon h determines the number of time steps over which the agent's performance is measured.

A solution to a POMDP is a policy that maximizes the accumulated expected reward for acting over h time steps, starting in b_0. Finite state controllers are one possible policy representation [5]:

Definition 1 (Finite State Controller)**.** *A finite state controller is a tuple* (N, α, δ). *N is the finite set of controller nodes. Each node $n \in N$ is associated with an action $a \in A$ via the action association function $\alpha : N \to A$. Transitions are defined via the transition function $\delta : N \times N \to 2^O$.*

Executing a finite state controller works by iteratively executing the action associated with the current node, receiving an observation, and choosing the transition whose label includes the received observation, until the horizon is reached.

The node transitions in an FSC need to be well-defined, i.e., the outgoing transitions of each node n need to form a partition of O: for all observations o, (1) $o \in \delta(n, n')$ implies $o \notin \delta(n, n'')$ for all nodes $n' \neq n''$, and (2) there is a node n' such that $o \in \delta(n, n')$.[2]

[1] Institute of Artificial Intelligence, Ulm University, D-89069 Ulm, Germany

[2] Alternatively, one could define the transition function as $\delta : N \times O \to N$, which already guarantees well-defined transitions. However, we do not use this formulation because it does not generalize well to our definitions below.

The expected value of executing a controller C starting at node n in system state s over a finite time horizon h, denoted $V_h(C,n,s)$, can be calculated by iteratively computing a t-step-to-go value function [5]:

$$\begin{aligned} V_0(C,n,s) =& R(s,\alpha(n)) \\ V_t(C,n,s) =& R(s,\alpha(n)) + \\ & \sum_{n',s'} T(s,\alpha(n),s') \sum_{o\in\delta(n,n')} Z(s',\alpha(n),o) V_{t-1}(C,n',s') \end{aligned} \quad (1)$$

The value for a belief state b is defined as $V_t(C,n,b) = \sum_s V_t(C,n,s)b(s)$. The controller execution is started in the node n_0 that maximizes the expected value for b_0, i.e., $n_0 = \text{argmax}_n\{V_h(C,n,b_0)\}$.

2.2 Factorization

In a propositionally factored POMDP, the state space S is the cross product of k propositional state variables, i.e., $S = S_1 \times \cdots \times S_k$, so that states $s = (s_1,\ldots,s_k)$ are interpretations of state variables. Similarly, the observation space $O = O_1 \times \cdots \times O_l$ is factored into l observation variables. This allows transition probabilities, observation probabilities, and rewards to be compactly represented by, e.g., algebraic decision diagrams (ADDs) [1], which represent real-valued functions in boolean or finite domain variables as graphs. An advantage of ADDs is that operations such as addition, point-wise multiplication, or summing over variables (also called *sum out*) can be implemented efficiently on ADDs, i.e., without explicitly enumerating all possible assignments. We use *dual action diagrams* as introduced by Hoey et al. [6] to represent the state transition function: for each action a, there is a dual action diagram $T^a_{S'_i}(S'_i, S_1,\ldots,S_k)$ for each primed state variable S'_i, denoting $P(S'_i = s'_i | S_1 = s_1,\ldots,S_k = s_k)$ under execution of a. Analogously, dual observation diagrams $Z^a_{O_j}(O_j, S'_1,\ldots,S'_k)$ represent $P(O_j = o_j | S'_1 = s'_1,\ldots,S'_k = s'_k)$ for action a and observation variable O_j. Rewards are represented as ADDs $R^a(S_1,\ldots,S_k)$. The initial belief state is represented as an ADD $b_0(S_1,\ldots,S_k)$.

2.3 Logical FSCs

For POMDPs with factored observations, the representation of the transition function δ of a controller as defined in Definition 1 is very large, since all observations must be enumerated. We therefore adopt a compact representation of finite state controllers that makes use of the fact that observations are factored [9]:

Definition 2 (Logical Finite State Controller)**.** *A* logical finite state controller (LFSC) *is defined in the same way as in Definition 1, except for the transition function: now, $\delta : N \times N \to \mathcal{F}(O_1,\ldots,O_l)$ is a function that maps pairs of controller nodes to propositional formulas over observation variables.*

To determine whether a transition from n to n' is chosen in the execution of an LFSC, it now needs to be checked whether the received observation o fulfills the respective transition formula, i.e., whether $o \models \delta(n,n')$. Transitions still need to be well-defined. Now, this is captured by requiring for all nodes n and observations o, (1) whenever $o \models \delta(n,n')$ it holds that $o \not\models \delta(n,n'')$ for all nodes $n' \neq n''$, and (2) there exists a node n' such that $o \models \delta(n,n')$.

As noted in Equation 1, the value of a flat FSC can be computed as a sequence of value functions. This transfers to LFSCs by letting the second summation range over $o \models \delta(n,n')$ instead of $o \in \delta(n,n')$ to account for the fact that transitions are governed by formulas.

2.4 Hierarchical Task Networks

HTN planning [4] is an approach to planning in fully observable deterministic domains. World states are described in a similar manner as shown above, yet actions are deterministic and the (single) initial world state is completely known. Hence, no observations are needed, because the evolution of the world can be predicted with certainty.

Apart from the domain dynamics, the HTN planning problem definition includes a hierarchy of actions: in addition to normal actions there are also abstract actions that cannot be executed directly. Instead, one or more implementations, called decomposition methods, are provided for each abstract action. Methods are specified by domain experts and represent known possible solutions to subproblems given by the abstract actions, therefore encoding a domain expert's knowledge about typical problem-solving "recipes". In its simplest form, a method is a sequence of primitive or abstract actions. At planning time, abstract actions are iteratively eliminated from an initial sequence of abstract actions by replacing them with implementing methods until a plan is found that contains only primitive actions.

3 Hierarchical Factored POMDPs

We now introduce action abstraction into POMDPs. For this, we first augment a factored POMDP $P = (S,A,T,R,O,Z,b_0,h)$ with a set of abstract actions A^a, $A^a \cap A = \emptyset$. To complement the abstract actions, we also define a set of m abstract observation variables $O^a = \{O^a_1,\ldots,O^a_m\}$, $O^a \cap \{O_1,\ldots,O_l\} = \emptyset$. Just like normal observations do for primitive actions, abstract observations represent information about the outcome of executing an abstract action.

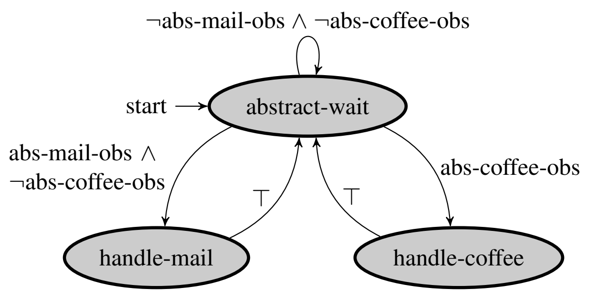
As an example, consider a simplified variant of the robot domain introduced by Boutilier et al. [3], where a robot has to deliver mail and coffee in an office environment. The robot can deterministically move clockwise and counterclockwise through the rooms, e.g., executing *move-clk* moves the robot to the mail room when it was in the office before. The problem is partially observable, because the robot can only observe whether there is a coffee request when it is in the office, and can only detect whether mail has arrived when it is in the mail room. We introduce three abstract actions, *abstract-wait*, *handle-coffee*, and *handle-mail*, and two abstract observation variables, *abs-mail-obs* and *abs-coffee-obs*. The intention is that *abstract-wait* detects both whether there is a coffee request and whether mail has arrived, and the *handle-x* actions handle coffee and mail requests.

3.1 Methods

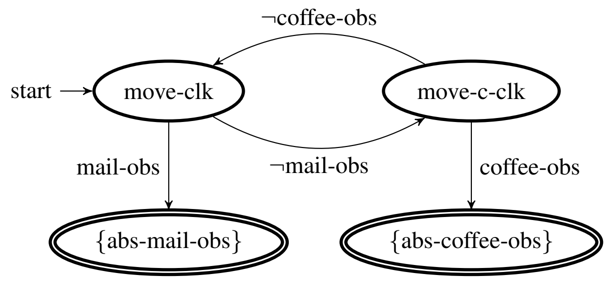
We allow abstract actions to occur in LFSCs, i.e., we modify the action association function α so that it maps to $A \cup A^a$ and the transition function δ so that it maps to formulas over O^a when the source node is labeled with an abstract action. In the course of this paper, we will call nodes labeled with primitive or abstract actions primitive or abstract nodes, respectively.

Since abstract actions represent high-level activities that are not directly executable, a *partially abstract* LFSC such as the one shown in Figure 1a is not directly executable either. To get an executable controller, we must specify how abstract actions are executed. For this purpose, each abstract action a is associated with one or more *decomposition methods*. A decomposition method $m = (a, C)$ consists of the respective abstract action a and an implementation C – a certain kind of LFSC that has start and end points, and can return an abstract observation (e.g., whether mail has arrived). In our example, a possible implementation of *abstract-wait* is to move back and forth between office and mail room, looking whether there is a coffee request or mail has arrived (Figure 1b). More precisely, such a *method FSC* is defined as follows:

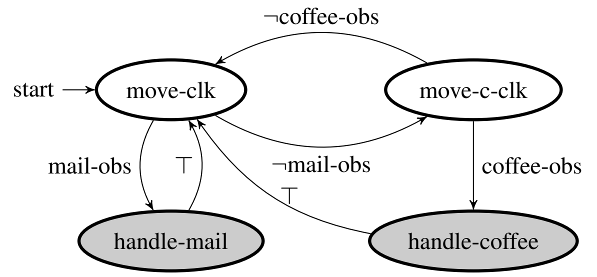
Definition 3 (Method FSC)**.** *A* method FSC (MFSC) *is a partially abstract LFSC, augmented with a set of terminal nodes N_t, $N_t \cap N = \emptyset$, and a labeling function $L : N_t \to 2^{O^a}$ that maps terminal nodes to abstract observations, i.e., interpretations of the abstract observation variables in O^a. Terminal nodes may occur as the target nodes of δ. Complementary to terminal nodes, there is a single initial node $n_0 \in N$ that defines where the execution of the MFSC starts.*


¬abs-mail-obs ∧ ¬abs-coffee-obs
start
abstract-wait
abs-mail-obs ∧ ¬abs-coffee-obs
⊤
⊤
abs-coffee-obs
handle-mail
handle-coffee

(a) A partially abstract LFSC for the robot domain.



(b) An MFSC for the abstract-wait action. The start node is identified by the *start* marker. Doubly bordered nodes are terminal nodes.



(c) The result of applying the method to the partially abstract LFSC.

Figure 1: Visualization of the method application procedure. A gray node background identifies abstract nodes. For the sake of readability, we omit edges with unsatisfiable transition conditions.

Applying a method transforms a partially abstract LFSC into a more concrete version by replacing the abstract action with the implementation specified by the method, as visualized in Figure 1c. Intuitively, the node sets are merged together, throwing out the decomposed node and the terminal nodes of the MFSC. The new transition function preserves the internal transitions of the original controllers, and introduces new transitions for wiring them together: transitions to the decomposed node are redirected to the initial node of the MFSC. Analogously, transitions to terminal nodes of the MFSC are rerouted by evaluating terminal node labels with respect to whether they fulfill outgoing abstract transition formulas of the decomposed node: in Figure 1a, *abstract-wait* has an outgoing transition with condition *abs-coffee-obs*, and the label of the right terminal node in Figure 1b assigns *true* to *abs-coffee-obs*, therefore there is a transition from *move-c-clk* to *handle-coffee* in Figure 1c.

For a formal definition of method application, let $\mathrm{pd}_C(n)$ denote the set of predecessors of a node n in an LFSC C with node set N and transition function δ, i.e., $\mathrm{pd}_C(n) = \{n' \in N | \delta(n', n) \not\equiv \perp\}$ and $\mathrm{sc}_C(n)$ the set of successors of n, i.e., $\mathrm{sc}_C(n) = \{n' \in N | \delta(n, n') \not\equiv \perp\}$. We use superscripts to disambiguate the members of the different FSCs involved, e.g., we write N^1 for the node set of the LFSC C^1.

Definition 4 (Method Application)**.** *Let C^1 be an LFSC containing a node n_a^1 with $a = \alpha^1(n_a^1)$ being an abstract action, $m = (a, C)$ a method for a, and C^2 an isomorphic copy of C with $N^1 \cap N^2 = \emptyset$. Applying m to n_a^1 results in a new LFSC $C^3 = (N^3, \alpha^3, \delta^3)$ with:*

- $N^3 := (N^1 \cup N^2) \setminus (\{n_a^1\} \cup N_t^2)$; $\alpha^3(n) := \begin{cases} \alpha^1(n), & n \in N^1 \\ \alpha^2(n), & n \in N^2 \end{cases}$
- $\delta^3(n, n') :=$
$$\begin{cases} \delta^1(n, n'), & n, n' \in N^1 \\ \delta^2(n, n'), & n, n' \in N^2 \\ \delta^1(n, n_a^1), & n \in \mathrm{pd}_{C^1}(n_a^1), n' = n_0^2 \\ \bigvee_t \delta^2(n, t), & t \in N_t^2, n \in \mathrm{pd}_{C^2}(t), n' \in N^1, L(t) \models \delta^1(n_a^1, n') \\ \bigvee_t \delta^2(n, t), & t \in N_t^2, n \in \mathrm{pd}_{C^2}(t), n' = n_0^2, L(t) \models \delta^1(n_a^1, n_a^1) \\ \perp, & \textit{else} \end{cases}$$

Applying methods maintains the well-definedness property:

Theorem 1. *Let the transitions in C^1 and C^2 be well-defined. It follows that the transitions in C^3 are well-defined.*

Proof sketch. We need to prove that outgoing transitions are well-defined for each node $n \in N^3$. For all nodes that originated from N^1, outgoing transitions either stay as they are or are only redirected to n_0^2. Thus, they remain well-defined.

For n's that originate from N^2, we first prove that for all o, there is an n' such that $o \models \delta(n, n')$. For an observation o, the transition target either was an inner node of C^2, in which case it still is, or a terminal node t. Because transitions in C^1 are well-defined, there is a node n' such that $L(t) \models \delta^1(n_a^1, n')$. It follows that $o \models \delta^3(n, n')$. To show that $o \models \delta(n, n')$ implies $o \not\models \delta(n, n'')$, observe that the original transition conditions in C^2 are combined disjunctively, and that $o \not\models \delta^2(n, t)$ and $o \not\models \delta^2(n, t')$ implies $o \not\models \delta^2(n, t) \vee \delta^2(n, t')$. $\square$

3.2 Planning Problem

MFSCs can in turn contain abstract nodes, so a set of methods defines an action hierarchy. With a given initial partially abstract controller, we can now fully define hierarchical factored POMDPs:

Definition 5 (Hierarchical Factored POMDP)**.** *A hierarchical factored POMDP is a tuple $(P, A^a, O^a, M, C^{\mathrm{init}}, n_0^{\mathrm{init}})$, where*

- *P is the underlying factored POMDP $(S, A, T, R, O, Z, b_0, h)$,*
- *A^a is the finite set of abstract actions,*
- *O^a is the finite set of abstract observation variables,*
- *M is the finite set of methods,*
- *C^{init} is the initial controller, a partially abstract LFSC, and*
- *$n_0^{\mathrm{init}} \in N^{\mathrm{init}}$ is the initial node of C^{init}.*

The fact that an initial node is defined for the initial controller can be seen as an additional piece of expert knowledge: in our example, we define that we always start with *abstract-wait*.

A solution to a hierarchical POMDP is the best primitive controller that can be generated from a hierarchical POMDP. When we require that n_0^{init} is primitive, a solution can be defined as follows:

Definition 6 (Solution)**.** *A solution for a hierarchical factored POMDP is a controller C^* such that*

- *C^* can be generated from C^{init} via repeated method application,*
- *C^* does not contain abstract nodes, and*
- *there does not exist a controller C^+ that satisfies the above criteria and also satisfies $V_h(C^+, n_0^{\text{init}}, b_0) > V_h(C^*, n_0^{\text{init}}, b_0)$.*

Since n_0^{init} is primitive, all decompositions of C^{init} also contain n_0^{init}. In practice, requiring n_0^{init} to be primitive would be somewhat inconvenient, and is in fact not the case in our example. Therefore, when n_0^{init} is abstract, we take the initial node of the method applied to n_0^{init} as the new initial node of the decomposed controller (cf. the nodes labeled with *abstract-wait* and *move-clk* in Figure 1a and 1c, respectively). Note that a solution as defined in Definition 6 is in general not an optimal policy for the underlying POMDP, as it is constrained to be a policy that can be generated via decomposition.

We now take a look at how difficult planning with Hierarchical Factored POMDPs is for the special case where initial nodes of MFSCs have to be primitive:

Theorem 2. *Let H be a Hierarchical Factored POMDP where the initial nodes of all MFSCs are primitive and let the action hierarchy allow for termination, i.e., for each abstract action, there exists a finite sequence of decompositions so that the result of applying the decompositions is fully primitive. Then a solution for H can be computed in finite time.*

Proof sketch. We prove that the maximum number of decompositions that have to be applied to C^{init} to receive a primitive controller is finite. With the fact that the number of methods is finite, it follows that only finitely many primitive controllers with different value can be generated from C^{init}.

Take N_d to be the set of abstract nodes with minimum distance d from the initial node, where distance is measured as the minimum number of nodes in a path between two nodes. Decomposing a node from N_d removes one element from N_d and does not introduce new elements to N_d, because the first node of the used MFSC is primitive. After $|N_d|$ such steps, N_d will be empty. We proceed by decomposing nodes from N_{d+1} until it is empty, and so on. When N_h is empty, the controller is fully primitive within the horizon h and hence, its value is fully determined. The remaining abstract nodes can then be arbitrarily decomposed until the controller is primitive. $\square$

Requiring primitive initial nodes in all MFSCs may seem overly restrictive. To avoid this, the requirement can be weakened to allow finitely many (instead of only one) decomposition steps to occur before a decomposition with a primitive initial node is applied. This does not invalidate the above result and suffices for a broad range of applications.

4 Algorithms

We apply two algorithms to hierarchical factored POMDPs: A* search and the UCT algorithm. For both, we conduct a search in the space of plans, where search nodes correspond to partially abstract LFSCs. The search is started with C^{init}. Modifications applicable to a search node correspond to all methods applicable to an abstract controller node chosen by some scheme, e.g., one that has minimum distance d to the initial node. Terminal search nodes contain primitive controllers. The specifics for each algorithm are detailed below.

4.1 A* Search

For A*, we need to define a cost function g and a heuristic function h, each of which is used to evaluate partially abstract LFSCs. We will first show how to compute the value of a primitive LFSC, before showing how this is used in the computation of g and h.

Because the state and observation spaces of the POMDP are propositionally factored, computing the value by explicitly enumerating all states and observations takes a prohibitively large amount of time. Therefore, we will now show how the calculation of controller value can be implemented with ADDs. For this, we need to map the elements of Equation 1 to ADDs and operations on ADDs, avoiding the explicit enumeration of states or observations.

Since the LFSC nodes determine which action is executed via α, we introduce a multivalued node variable N and combine action-dependent transition, observation, and reward ADDs into single ADDs, yielding $T_{S'_i}(S'_i, N, S_1, \ldots, S_k)$, $Z_{O_j}(O_j, N, S'_1, \ldots, S'_k)$, and $R(N, S_1, \ldots, S_k)$, respectively. Next, we need to address the node transition formulas δ. Since node transition formulas are propositional, they can be represented by zero/one-valued ADDs. We also combine these ADDs with controller node information, but in this case we need two node variables N and N': one for the source node and one for the target node of the transition. This yields an ADD $\delta(N', N, O_1, \ldots, O_l)$. Finally, we augment the initial belief state ADD b_0 with the initial node of the controller.

With the available operations on ADDs, we can directly implement Equation 1 similarly to the SPUDD algorithm [6]. Algorithm 1 shows the complete algorithm for computing the value of an LFSC.

```
Input  : ADDs $T_{S'_i}$, $Z_{O_j}$, $R$, $\delta$, $b_0$ for LFSC $C$; horizon $h$
Output : Value of $C$ in $b_0$ over $h$
1 $Z := \delta$
2 for $j := 1 \ldots l$ do $Z := (Z * Z_{O_j}).\text{sumOut}(O_j)$
3 $V_0 := R$
4 for $t := 1 \ldots h$ do
5     $V_t := \text{primeVariables}(V_{t-1})$
6     $V_t := (V_t * Z).\text{sumOut}(N')$
7     for $i := 1 \ldots k$ do $V_t := (V_t * T_{S'_i}).\text{sumOut}(S'_i)$
8     $V_t := V_t + R$
9 return $(V_t * b_0).\text{sumOut}(N, S_1, \ldots, S_k)$
```

Algorithm 1: The ADD-based LFSC value calculation algorithm. Note that transition probabilities between controller nodes are precomputed by summing out observation variables in the ADD Z.

We can now define the cost function g for a partially abstract LFSC C by transforming the partially abstract LFSC into a primitive *guaranteed-costs LFSC C_g* with the property that the cost incurred by executing C_g is guaranteed to be incurred by executing any decomposition of C. This cost can therefore be interpreted as the "path costs" of reaching C. To construct C_g, we redirect all transitions to abstract controller nodes to a newly introduced controller node with a sling labeled with $\top$ and label it with a zero-cost action, i.e., we ignore costs incurred by abstract actions. We define $g(C) := V_h(C_g, n_0^{\text{init}}, b_0)$.

The heuristic function $h(C)$ estimates the cost incurred by the abstract nodes in an admissible manner. For this, we need to estimate both the probability and the cost of reaching each abstract node n_a. Since the cost incurred by executing $\alpha(n_a)$ crucially depends on how much time is left, we need to do this estimation separately for each time step in which the node can be reached.

Let $P_t(C, n_a)$ denote the probability of reaching n_a with t steps to go through any fully primitive path from n_0 to n_a in C. This is not the true probability of reaching n_a with t time steps left, because decomposing other abstract nodes might add new primitive paths from n_0 to n_a. Still, the true probability cannot be lower than $P_t(C, n_a)$.

The cost of n_a is given by the minimum cost of its primitive decompositions. If we define $\tilde{V}_t(C, n_a)$ to be a state-independent underestimation of the true value of $V_t(C, n_a, s)$ then $\tilde{V}(C, n_a) := \sum_{t=1}^{h} P_t(C, n_a)\tilde{V}_t(C, n_a)$ is an underestimation of the expected cost of n_a. The heuristic value can then be defined as $h(C) := \sum_{n_a} \tilde{V}(C, n_a)$. To receive a state-independent underestimation $\tilde{V}_t(C, n_a)$, we conduct a fixed-depth relaxed search in plan space. We only consider minimal costs for each executed primitive action $\tilde{R}(a) = \min_s R(s, a)$ and for traversing LFSCs, we allow the agent to "choose" the best transitions without respecting observations.

For our implementation, we use the decision diagram library provided with Symbolic Perseus [11], since it supports multi-valued variables. We heuristically choose the decomposition depth for the computation of h to be 2.

4.2 UCT

UCT (Upper Confidence Bound applied to Trees) is an instance of Monte Carlo Tree Search (MCTS), a family of algorithms that incrementally expand a search tree in memory by interacting with a domain simulator [7]. A concrete MCTS algorithm requires the choice of a tree policy for selecting actions to traverse the search tree and a rollout policy for generating information about the search space. UCT uses a special kind of tree policy, which is governed by an adaptation of the UCB algorithm for the bandit problem.

Applied to searching in the space of LFSCs, the tree policy chooses a method m^* to apply to a partially abstract controller C:

$$m^* = \operatorname*{argmin}_m \left(Q(C, m) + c\sqrt{(\ln n(C))/n(C, m)} \right), \quad (2)$$

where $n(C)$ is the number of times C has been encountered while traversing the tree, $n(C, m)$ is the number of times method m was selected when choosing a method to apply to C, and $Q(C, m)$ is the average of the values sampled from primitive LFSCs generated from C. We heuristically pick c to be equal to the planning horizon of the underlying POMDP, i.e., h. As the rollout policy, we choose simple uniformly random rollouts.

It is noteworthy that applying methods is deterministic and does not incur a cost unless the resulting LFSC is fully primitive, i.e., there are only terminal rewards. In this case, the cost function of the primitive LFSC is sampled by simulating its execution within the underlying POMDP once. Intuitively, a primitive controller corresponds to a bandit in the context of the original UCB algorithm.

To implement the required LFSC execution simulation, we directly employ the functionality provided via the rddlsim tool used in the 2011 IPPC by letting our LFSC data structure implement the `Policy` interface and running it via the provided `Simulator`.

5 Evaluation

To evaluate our approach, we modeled action hierarchies for the POMDP variants of the Elevator, Navigation, and Skill Teaching domains from the 2011 IPPC. Hierarchies were specified in a lifted manner using a hierarchical extension of the RDDL language [12] and then grounded by our implementation to fit our definitions above. Lifted decomposition methods may contain variables both for parameters of primitive and abstract actions and inside the formulas used as edge labels. Thus, we could model a single hierarchy per domain and instance-specific initial controllers. We quickly sketch how we defined action hierarchies for the domains before we present experimental results.

Elevators. In the Elevators domain, our hierarchy encodes the choice between several high-level strategies for controlling elevator movement. One strategy is to let the elevator repeatedly go from the bottom to the top floor and back, opening the door at floors with waiting persons. A second strategy is to let the elevator hover, waiting for passengers. and as soon as there is one, get them and deliver them to the top or to the bottom floor. We restricted our analysis to problem instances with one elevator (i.e., IPPC instances 1, 4, 7, and 10), since our approach currently does not support concurrent actions.

Navigation. In the Navigation domain, the task is to reach the goal cell on the bottom right of a grid. Starting from a random cell in the top row, the agent needs to traverse rows of cells that are safe on the left side and get increasingly more dangerous to the right. Since the agent can only observe when it hits a corner, a simple (and the safest) policy is to move left until the top left corner is hit, move down until the bottom left corner is hit, and then move right until the bottom right corner is hit, which is also the goal. Alternatives include taking a more direct route towards the goal once the top left corner is hit (at which point the agent fully knows all aspects of the state), or simply ignoring the dangers in the center rows completely and taking the most direct route right away.

Skill Teaching. Skill Teaching is about teaching a set of interdependent skills to a student using hints and multiple choice questions. For this domain, we chose a very light-weight hierarchy. There is an abstract action *teachAll* with a noop implementation and a recursive implementation parameterized with a skill *s*, the latter of which essentially consists of teaching the skill *s* followed by *teachAll*. Teaching a single skill is an abstract action, parameterized with a skill, and with two possibilities for teaching that skill as implementations, namely giving a hint and asking questions until the user answered correctly once. The purpose of this hierarchy is to measure performance for the case where little expert knowledge is available.

5.1 Results

Our experiments address two questions: first, we want to know how good the polices generated with our approach are. Second, we want to determine how quickly policies can be generated.

We compare computed policy quality and running times of both our A* and UCT adaptations against Symbolic Perseus and a blind policy. For the blind policy, we take the maximum of executing a noop policy and a random policy that randomly chooses an action in each time step. Since UCT is an anytime algorithm, we make experiments with running times of 0.1s, 1s, and 10s.

We compare policy quality using the simulation feature of rddlsim, reporting the average of 1000 runs. All experiments were conducted on a 24-core Intel Xeon machine running at 2.4GHz. All planners are written in Java, the virtual machine was given 4GB of memory. Table 1 shows our results. The time limit was 2 hours.

Table 1: The results of our experiments. In a cell, the upper value is the runtime in milliseconds and the lower value the corresponding quality of the computed policy, rounded to 3 decimal places. *Timeout* or *memout* means that the planner ran out of time or memory, respectively. The low results of Symbolic Perseus for the Navigation domain seem to be due to a bug in its policy simulation code. Therefore, we show the values it itself reported in parentheses.

Instance	A*	UCT 0.1s	UCT 1s	UCT 10s	Sym. Perseus	Blind
Elevators						
instance 1	timeout n/a	1,187 -32.580	1,999 -32.580	10,871 -32.580	651,885 -35.418	n/a -44.363
instance 4	timeout n/a	2,278 -74.256	2,938 -74.256	12,184 -74.256	timeout n/a	n/a -88.982
instance 7	timeout n/a	4,116 -112.874	4,571 -112.949	14,486 -113.247	timeout n/a	n/a -133.809
instance 10	timeout n/a	14,783 -154.429	15,105 -168.112	23,926 -148,802	timeout n/a	n/a -177.855
Navigation						
instance 1	5,597 -10.165	1,123 -10.165	1,929 -10.165	11,669 -10.165	104,042 -40 (-12,355)	n/a -38.605
instance 2	15,936 -10.781	1,159 -10.781	3,607 -10.781	12,269 -10.781	302,217 -40 (-11,696)	n/a -39.178
instance 3	258,135 -12.457	2,076 -12.457	2,445 -12.457	11,799 -12.457	5,824,802 -40 (-14,314)	n/a -39.833
instance 4	memout n/a	1,356 -15.745	2,524 -15.745	11,361 -15.745	memout n/a	n/a -39.991
instance 5	timeout n/a	4,004 -17.924	4,763 -17.924	12,752 -17.924	memout n/a	n/a -39.839
instance 6	memout n/a	3,854 -19.978	4,291 -19.978	13,746 -19.978	memout n/a	n/a -39.995
instance 7	memout n/a	2,390 -22.385	3,215 -22.385	12,057 -22.385	memout n/a	n/a -39.996
instance 8	memout n/a	2,498 -32.042	3,817 -32.042	14,789 -32.042	memout n/a	n/a -39.87
instance 9	memout n/a	6,002 -33.228	5,731 -33.228	13,709 -33.228	memout n/a	n/a -39.997
instance 10	memout n/a	5,214 -33.511	5,989 -33.511	14,968 -33.511	memout n/a	n/a -40
Skill Teaching						
instance 1	timeout n/a	736 39.319	1,590 49.888	12,559 48.699	26,943 -16.039	n/a 25.937
instance 2	timeout n/a	1,160 42.344	1,634 48.559	10,616 51.774	25,906 20.193	n/a 26.551
instance 3	timeout n/a	1,388 9.185	2,209 9.185	11,181 28.263	timeout n/a	n/a -86.6
instance 4	timeout n/a	3,358 -22.914	4,020 -48.606	12,855 -3.646	timeout n/a	n/a -111.645
instance 5	timeout n/a	3,704 -118.919	5,766 -116.301	14,019 -46.2	memout n/a	n/a -247.745
instance 6	timeout n/a	4,047 -165.705	5,374 -203.779	14,812 -106.799	memout n/a	n/a -292.613
instance 7	timeout n/a	9,318 -572.92	10,482 -394.459	18,693 -181.782	memout n/a	n/a -589.307
instance 8	timeout n/a	6,584 -487.102	7,641 -287.325	16,516 -202.218	memout n/a	n/a -537.33
instance 9	timeout n/a	9,142 -401.814	10,037 -303.297	19,070 -270.603	memout n/a	n/a -602.724
instance 10	timeout n/a	15,208 -437.534	15,556 -360.025	25,135 -307.424	memout n/a	n/a -542.868

Since A* is an optimal algorithm, the values computed by A* are solutions in the sense of Definition 6, while UCT computes approximations in an anytime fashion. UCT quickly converges towards good policies and is far superior to our A* adaptation in terms of speed. UCT always quickly returns a policy that is significantly better than a blind policy and one generated by Symbolic Perseus, which can only solve small instances. On average, UCT10s requires only 8.4% of the runtime needed by Symbolic Perseus, while producing better policies. The need to exactly compute a controller's value poses the biggest obstacle for A*. UCT, on the other hand, limits the effort spent on determining the value of suboptimal controllers.

Navigation is a domain where it is very easy to come up with good quality expert knowledge: the described safest policy is always a very good policy. As a result, our UCT adaptation computes better policies faster than Symbolic Perseus across all instances. For the elevator domain, the policies generated by our approach are only slightly better than policies found by Symbolic Perseus. We conjecture that better performance would be possible with more elaborate expert knowledge, which could be gained by, e.g., looking at commercial elevator systems. In the skill teaching domain, running UCT for a short period of time already results in policies that outperform both blind policies and policies generated by Symbolic Perseus, which means that UCT can generate good policies even with light-weight hierarchies. Running UCT for a longer period of time drastically improves the found policies, even beyond the values shown here: in our further experiments, we observed that, e.g., UCT was able to produce a policy with a value of -89.013 for instance 10 after 10^3s.

6 Conclusion

We introduced an adaptation of HTN planning to the POMDP setting and two algorithms, UCT and A*, to deal with these hierarchical POMDPs. Our experiments on problems taken from the 2011 IPPC showed a significant speed up in policy generation: on average, UCT10s needs only 8.4% of the runtime required by Symbolic Perseus to produce an even better solution than Symbolic Perseus.

Acknowledgments. This work is done within the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems" funded by the German Research Foundation (DFG).

REFERENCES

[1] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, 'Algebraic decision diagrams and their applications', in *ICCAD*, pp. 188 –191, (1993).

[2] J. Boger, P. Poupart, J. Hoey, C. Boutilier, G. Fernie, and A. Mihailidis, 'A decision-theoretic approach to task assistance for persons with dementia', in *IJCAI*, pp. 1293–1299, (2005).

[3] C. Boutilier, T. Dean, and S. Hanks, 'Decision-theoretic planning: Structural assumptions and computational leverage', *JAIR*, **11**, 1–94, (1999).

[4] K. Erol, J. Hendler, and D. S. Nau, 'UMCP: A sound and complete procedure for hierarchical task-network planning', in *AIPS*, pp. 249–254, (1994).

[5] E. A. Hansen, 'Solving POMDPs by searching in policy space', in *UAI*, pp. 211–219, (1998).

[6] J. Hoey, R. St-aubin, A. Hu, and C. Boutilier, 'SPUDD: Stochastic planning using decision diagrams', in *UAI*, pp. 279–288, (1999).

[7] Levente Kocsis and Csaba Szepesvári, 'Bandit based monte-carlo planning', in *ECML*, pp. 282–293, (2006).

[8] U. Kuter, D. Nau, E. Reisner, and R. Goldman, 'Conditionalization: Adapting forward-chaining planners to partially observable environments', in *PlanEx*, (2007).

[9] F. Müller and S. Biundo, 'HTN-style planning in relational POMDPs using first-order FSCs', in *KI*, pp. 216–227, (2011).

[10] D. Nau, T.-Ch. Au, O. Ilghami, U. Kuter, H. Muñoz-Avila, J. W. Murdock, D. Wu, and F. Yaman, 'Applications of SHOP and SHOP2', in *IEEE Intelligent Systems*, (2004).

[11] P. Poupart, *Exploiting structure to efficiently solve large scale partially observable Markov decision processes*, Ph.D. dissertation, University of Toronto, 2005.

[12] Scott Sanner, 'Relational dynamic influence diagram language (RDDL): Language description'. http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf, 2010.

[13] D. Silver and J. Veness, 'Monte-carlo planning in large POMDPs', in *NIPS*, pp. 2164–2172, (2010).