An approach to generate MDPs using HTN representations

Felipe Meneguzzi and **Katia Sycara**
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{meneguzz, katia}@cs.cmu.edu

Yuqing Tang
Graduate Center
City University of New York
New York, NY, USA
ytang@cs.gc.cuny.edu

Simon Parsons
Brooklyn College
City University of New York
New York, NY, USA
parsons@sci.brooklyn.cuny.edu

Abstract

Planning for deterministic and probabilistic domains differ significantly in representations they require, the algorithms that solve them and the way in which results are represented. Hierarchical Task Networks (HTN) and Markov Decision Process (MDPs) are representative formalisms of, respectively, deterministic and stochastic planning. Stochastic domain specifications can easily become opaque to a human designer, especially as the domain size increases. Our research aims to develop algorithms for lossless and automatic mapping of HTN models that are easily intelligible to humans into MDPs. In this paper we develop algorithms to convert deterministic planning domains with HTN domain knowledge and an action error model into MDPs that can then be solved, while maintaining a bound on the number of MDP states.

1 Introduction

Deterministic planning domains are generally easy to visualize and understand, as the details of the meaning of transitions between states are clearly defined in the operators, and the resulting plans are intuitive and easily understood. One particular formalism for domain representation in deterministic planning is the hierarchical task network (HTN) [Erol *et al.*, 1994], which encodes not only STRIPS/PDDL actions with their preconditions and effects, but also domain knowledge in the form of a hierarchy of tasks that can be refined from a high-level objective into the actions required in the environment. Conversely, one of the most widely studied formalisms for planning under uncertainty is the Markov decision process (MDP) [Bellman, 2003], in which the evolution of the environment is modeled as a Markov chain, and the goals of the planner are *implicitly* represented in a function that defines, for each state, the reward of executing a certain action. The definition of stochastic planning problems quickly becomes unwieldy as the number of state variables increase.[1] As the number of states goes up, so does the size of the transition probability tables, with problems requiring one such table for each action in the domain. As a consequence, although MDPs are an elegant mathematical formalism for representing stochastic domains, it is not straightforward for non-specialists to model domains using this formalism.

Our goal is to use HTN models, which are more user-friendly, to automatically construct MDPs. In this paper we propose a step towards this overall aim, showing how to use HTNs to describe MDPs, thus allowing stochastic domains to be modeled using HTNs that are then translated into MDPs in order to be solved. Together with a simple model of action error, our conversion process allows efficient MEU planning over the state space induced by the HTN. The benefits of the approach are twofold: (a) reduction of the state space, and consequent reduction of the computational burden is beneficial since it enables the representation and solving of realistic planning problems, and (b) starting from a declarative representation makes planning more comprehensible to humans, while extending the representation to stochastic domains.

2 Background

2.1 MDPs

We consider an MDP to be a tuple $\Sigma = (S, A, Pr, u)$ where S is a finite set of states, A is a finite set of actions, Pr is a state-transition system and u is a reward (or utility) function [Ghallab *et al.*, 2004]. The state-transition system defines a probability distribution for each state transition. Here, given $\{s, s'\} \in S$ and $a \in A$, $Pr_a(s'|s)$ denotes the probability of transitioning from state s to state s' when executing action a. The solution of an MDP is a *policy*, which indicates the best action to take in each state. A policy π here is a total function $\pi : S \rightarrow A$ mapping states into actions. The analogous to goal states in MDPs are indirectly represented through *utility functions*, which typically assign a value $u(a_j, s_i)$ to the choices of actions a_j in states s_i. Such information makes it possible to compute the value of a given state under a particular policy π — it is the expected value of carrying out the policy from that state (where $a = \pi(s)$), given a discount factor.The *optimal* policy $\pi^*(s)$ is then one that maximizes this value, and can be found by various means.

2.2 HTNs

An HTN planning domain is a pair $\mathcal{D} = (\mathcal{A}, \mathcal{M})$ where $\mathcal{A}$ is a finite set of actions (or operators) and $\mathcal{M}$ is a finite set of methods, and an HTN is a pair $\mathcal{H} = (T, C)$ where T

[1] Of course the time and space complexity of solving these problems grows exponentially as well, but this is not our focus here.

is a finite set of tasks to be accomplished and C is a set of partial ordering constraints on tasks in T [Kuter *et al.*, 2009]. Constraints specify the *order* in which certain tasks must be executed and are represented by the *precedes* relation, where $t_i \prec t_j$ means that task t_i must be executed *before* t_j. The set of tasks contains primitive and non-primitive tasks. All tasks have *preconditions*, specifying a state that must be valid before the task can be carried out, and primitive tasks (associated with actions that are executed in the environment) have *effects*, specifying changes in the state that was valid before the action was executed. Preconditions and effects are propositions p_i, so each task t_i has a set of preconditions $preconds(t_i) = \{p_1, \ldots, p_n\}$, and each primitive task/action has a set of effects $effects(t_i) = \{p_1, \ldots, p_m\}$. In the planning literature effects are generally represented as add and delete lists of true facts about the world. Here we modify this concept and specify effects as a list of positive and negated propositions. Thus, if the positive version of a proposition is present in the state prior to the execution of an action with a negated effect, that proposition is removed from the resulting state in order to preserve consistency, while if a negated proposition is present in a state and its positive version is present in the effects of an action then that proposition becomes true in the resulting state. For example, if a state $s_i = \{p_1, \neg p_2, p_3\}$ is valid before an action a_i with $effects(a_i) = \{\neg p_1, p_2\}$ is executed, the state resulting from the execution of a_i is $s_{i+1} = \{\neg p_1, p_2, p_3\}$. Primitive tasks are action instances from $a \in \mathcal{A}$ (formalized in Definition 1), while non-primitive tasks denote tasks that can be decomposed using an appropriate method $m \in \mathcal{M}$.

Definition 1 (Action execution)**.** *Let s and s' be two states in the environment and a an action. An action a is executable in state s if $s \models preconds(a)$, and the execution of a in state s, denoted $\gamma(s, a)$ is a new state s' such that $s' = (s - neg(effects(a))) \cup pos(effects(a))$, where pos and neg denote, respectively, the positive and negative atoms in a set, and $s' \not\models \bot$.*

Methods describe how non-primitive tasks can be decomposed into subtasks. We represent methods as tuples $m = (S, t, H')$, where S is a *precondition* corresponding to a set of states that satisfy this precondition, also denoted by $preconds(m)$, specifying what must hold in the current state for a task t (also denoted $task(m)$) to be refined into $H' = (T', C')$ (also denoted $network(m)$), which decomposes t into new tasks $t_i \in T'$ with constraints $c_j \in C'$. Decomposing an HTN through a method m consists of removing task $task(m)$ from the network into which m is applied, adding the tasks and constraints from the method network $\mathcal{H}'$ while linking the constraints that refer to the task being decomposed to the new tasks from $\mathcal{H}'$. This is formalized in in Definition 2 (adapted from [Ghallab *et al.*, 2004]). To express the constraints to be changed in the HTN being decomposed, we denote the set of tasks that have no predecessors in a network $\mathcal{H} = (T, C)$ as $first(\mathcal{H})$, whereas the set of tasks that have no successors are represented by $last(\mathcal{H})$, or formally $first(H) = \{t | t' \prec t \notin C\}$ and $last(H) = \{t | t \prec t' \notin C\}$

Definition 2 (Method Application)**.** *Let $m_t = (S, t, H')$ be a method in a domain $\mathcal{D}$ for refining a task t_i and $\mathcal{H} = (T, C)$ be an HTN, where $t_i \in T$ is a task to be refined, $C_t \subseteq C$ are all the constraints referring to t_i. Furthermore, let all tasks $t_j \in T$ such that $t_j \prec t_i$ form a sequence $[t_0, \ldots, t_i]$, and s_i be the state resulting from executing $s_i = \gamma(\gamma(s_{i-1}, t_{i-1}), t_i)$ recursively until $\gamma(s_0, t_0)$ is executed. A method m_t is applicable in $\mathcal{H}$ if and only if $s_i \models preconds(m_t)$ and $t \cdot \sigma = t_i \cdot \sigma$ under some unifier σ, and the application of m_t to an HTN $\mathcal{H} = (T, C)$, denoted $\delta(\mathcal{H}, t_i, m_t, \sigma)$, results in a new HTN $\mathcal{H}'' = (T'', C'')$ where:*

- $T'' = (T - \{t_i\}) \cup T'$; *and*
- $C'' = (C - C_t) \cup (C'_t \cup C')$

Here C_t is the set of constraints containing the decomposed task t_i (*i.e.* $\{c \in C | c = t_j \prec t_i \vee (c = t_i \prec t_j)\}$), and C'_t is a new set of constraints created by replicating each previously removed constraint (*i.e.* the constraints of C_t) with each element of T' replacing t_i. For example if HTN $\mathcal{H}$ contains a task t_u and a task t_i and one constraint including t_u, $t_u \prec t_i$, and a method $m = (\top, t_u, \{a_1, a_2\}, \{a_1 \prec a_2\})$ is applied, the resulting set of constraints will be $\{a_1 \prec t_i, a_2 \prec t_i, a_1 \prec a_2\}$. All non-primitive tasks must be fully decomposed into primitive tasks before a full plan can be derived from the HTN, *i.e.* it must contain no non-primitive tasks.

3 Using HTNs to represent MDPs

Given an HTN and an MDP that represent the same domain, using the same set of actions and the same set of states, there must be some relationship between them. They both, after all, capture the same information. The MDP model of a domain can be visualized as a directed hypergraph where the nodes are states and the edges that connect states represent actions – where an arc connects one state to many other states, that action can lead to those other states. A trajectory through the state-space is the result of a single outcome of a specific action in each state, and is close to the notion of a plan in an HTN [Simari and Parsons, 2006].

3.1 States in a fully expanded HTN

From an HTN domain $\mathcal{D}$ and an HTN $\mathcal{H}$, it is possible to induce a directed hypergraph by considering all possible decompositions of each non-primitive task, obtaining a structure that models a multitude of possible states depending on user choices and error rates, which in turn can model MDP states and rewards. Here, each primitive task t_i in the HTN associated with an action $a \in \mathcal{A}$ will map to a state representing the world state achieved immediately after the execution of a. Moreover, state transitions will be determined via the ordering constraints in the HTN, so that there will be a non-zero probability of transitioning from a state t_i to a state t_j in the MDP if and only if it can be determined that t_i immediately precedes t_j in any potential plan generated by the HTN. We represent this immediate sequential precedence relation as $t_i \prec_1 t_j$.We shall use these relations in our algorithm to derive the transition functions for each action in the MDP.

In order to create a fully decomposed HTN (containing all possible method decompositions), instead of choosing one method to apply to each non-primitive task, we apply all applicable methods and collect all constraints resulting from

all possible method applications.[2] We create a fully decomposed HTN by slightly modifying the task decomposition algorithms used in HTN planning. First, we create a new method application function δ^* that takes a set of methods applicable to a task and adds all possible expansions of it to the HTN, making sure that the precedence relations are only added when the last primitive task of the preceding networks supports the preconditions of the method being applied. Verification that a certain primitive task t supports the preconditions of a method requires the evaluation of the possible states that could be reached by the time task t is executed, which in turn depends on the possible sequences of primitive tasks that could be generated before t, given the constraints on a task network. Thus, we use function $pState(s_0, t, \mathcal{H})$ to generate one possible state for a primitive task t in network $\mathcal{H}$ from an initial state s_0, defined recursively as follows:

$$pState(s_0, t, \mathcal{H}) = \begin{cases} \gamma(s_0, t) & \text{if } t \in \mathit{first}(\mathcal{H}) \\ \gamma(pState(s_0, t_i, \mathcal{H}), t) & \text{if } (t_i \prec_1 t) \in C \end{cases}$$

We say that a primitive task t_i in a network $\mathcal{H}$ can support a certain state s_i, given an initial state s_0, and denoted $t_i \models_{s_0,\mathcal{H}} s_i$, if and only if $\exists s_i . s_i = pState(s_0, t_i, \mathcal{H})$. Finally, we restrict the addition of constraints in the fully expanded HTN deriving from tasks t_i preceding task t_j being expanded by only adding constraints referring to methods that can potentially be supported by t_i. We denote the set of methods that are potentially applicable given a previous task t_i as $M'_{t_i,t_j} = \{m \in M_{t_j} \wedge t_i \models_{s_0,\mathcal{H}} t_j\}$. Given the notion of possible support for a state, we define the application of multiple methods for the full expansion of an MDP in Definition 3.

Definition 3 (Multi-Method Application)**.** *Let $M_t = \{m_t^1, \ldots, m_t^n\}$ be a set of methods in the form $m_t^i = (S, t, \mathcal{H}'_i)$ in a domain $\mathcal{D}$ for refining a task t and $\mathcal{H} = (T, C)$ be an HTN, where $t \in T$ is a task to be refined and $C_t \subseteq C$ are all the constraints referring to t. Furthermore, let s_i be the state resulting from the execution of all primitive tasks from the current decomposition of all tasks preceding t in $\mathcal{H}$. The application of a set of methods M_t, denoted $\delta^*(\mathcal{H}, t, M_t, \sigma)$, under some substitution σ to an HTN $\mathcal{H} = (T, C)$ results in a new fully expanded HTN $\mathcal{H}^* = (T^*, C^*)$ where:*

- $T^* = (T - \{t\}) \cup T'$*; and*
- $C^* = (C - C_t) \cup (C_t^* \cup C^*)$.

with the sets C_t and C_t^* denoting, respectively, the sets of constraints to be removed and added to the expanded HTN, as follows:

$$\begin{aligned} C_t &= \{t \prec t_i | t \prec t_i \in C\} \cup \{t_i \prec t | t_i \prec t \in C\} \\ C_t^* &= \{t \prec t_j | (t \prec t_j \in C) \wedge (t_j \in \bigcup\nolimits_{\forall m_t^i \in M'_{t_i,t_j}} \mathit{first}(\mathcal{H}'_i))\} \\ &\cup \{t_j \prec t | (t_j \prec t \in C) \wedge (t_j \in \bigcup\nolimits_{\forall m_t^i \in M'_{t,t_j}} \mathit{last}(\mathcal{H}')_i)\} \end{aligned}$$

Using the expanded the notion of method application of Definition 3 we can generate a fully expanded HTN $\mathcal{H}^*$ with Algorithm 1. This expansion differs from the traditional HTN expansion by the addition of an initial state to the task network in the form of a new primitive task (in Line 2) and ensuring that this state is connected by precedence constraints in the fully expanded HTN (Line 3). Moreover, instead of using a recursive approach, with the help of the notion of possible state induced by a task and the new multiple method application function, we define an iterative approach that tries to expand tasks until only primitive tasks are left in Lines 5 through 9. We expand the HTN from left to right in order to make sure that function $pState$ only finds primitive tasks in the path to the initial state, which is accomplished in Line 6. Finally, if we have relevant methods M_t in Line 7, we can use function δ^* to refine a task into all its possible expansions in Line 8, otherwise return failure as the resulting HTN contains unexpandable non-primitive tasks. Notice that the condition for this algorithm to fail is exactly the same as the one for the HTN planning algorithm, since if an HTN is not solvable the resulting MDP cannot lead to any of the desired solutions.

Algorithm 1 Expanding HTN.

```
1: function FULLDECOMPOSITION(s_0,H = (T, C),A,M)
2:     T* ← T ∪ {s_0}      // Add a primitive task as initial state
3:     C* ← C ∪ {s_0 ≺ t_i | t_i ∈ first(H)}
4:     H* ← (T*, C*)
5:     while T* has non-primitive tasks do
6:         t_u ← any non-primitive t ∈ T, s. t. ∄t_v ∈ A, t_v ≺ t_u
7:         M_{t_u} ← {m ∈ M | task(m) unifies with t_u with σ}
8:         if M_{t_u} ≠ ∅ then H* = δ*(H*, t_u, M_{t_u}, σ)
9:         else return failure
10:    return H*
```

3.2 Generating transition functions

The collection of constraints resulting from the full decomposition of the original HTN can then be used to construct the transition functions for each action. We represent a fully decomposed HTN as $\mathcal{H}^* = (T^*, C^*)$, and consider that each task $t_i \in T^*$ (*i.e.* the set of primitive tasks in $\mathcal{H}^*$) is labelled with an action $a \in \mathcal{A}$. For ease of presentation, we assume the ability to make inferences to determine $\prec_1$ on the sequential constraints in C^*.[3] We denote the $label(t_i)$ of a primitive task t_i to be the name of the action associated with that task.

We start by defining the set of states in the MDP. From Definition 3, we have that each primitive task in the fully expanded HTN leads to a *macro* state comprising the states resulting from all possible execution paths that lead to the execution of that task. Such a macro state is captured by function $mState$ in Definition 4 below, and is defined in terms of the multiple possible states, each of which is created by $pState$.

Definition 4 (Possible States at Task)**.** *Let $t_i \in T^*$ be a task in a fully expanded HTN $\mathcal{H}^* = (T^*, C^*)$ and s_0 an initial state. The possible MDP state $mState(s_0, t_i, \mathcal{H}^*)$ at task is the formula corresponding to the disjunction of all possible states $pState(s_0, t_i, \mathcal{H}^*)$ resulting from the execution of primitive tasks before t_i starting at s_0, so*

$$mState(s_0, t_i, \mathcal{H}^*) = \bigvee\nolimits_{\forall t_i \in T^*} pState(s_0, t_i, \mathcal{H}^*)$$

[2] In the propositional case, the MDP resulting from an HTN $\mathcal{H}$ has the same number of states as the number of non-primitive task nodes in the fully decomposed HTN plus an initial state.

[3] In practice, we keep track of $\mathit{first}(\mathcal{H})$ and $\mathit{last}(\mathcal{H})$ when applying methods using δ^* so that all constraints are $\prec_1$.

Algorithm 2 Creating MDP states.

1: **function** CREATEMDPSTATES(s_0,$\mathcal{H}^* = (T^*, C^*)$)
2: **for all** Tasks $t_i \in \mathcal{H}^*$ in some fixed order **do**
3: $S_i \leftarrow mState(s_0, t_i, \mathcal{H}^*) - \bigcup_{j=1}^{i-1} S_j$
4: $states[t_i] \leftarrow S_i$
5: **for all** $\{t_j \in \mathcal{H}^* | t_j \prec t_i\}$ in some fixed order **do**
6: **if** $mState(s_0, t_i, \mathcal{H}^*) \cap S_j \neq \emptyset$ **then**
7: $states[t_i] \leftarrow states[t_i] \cup \{S_j\}$ // Keep track of which MDP states t_i has effects onto
8: **return** $states$

Algorithm 3 Creating transition function.

1: **function** CREATETRANSITIONTABLES($\mathcal{A}$,$\mathcal{H}^*$,$states$,ϵ)
2: Create $|\mathcal{A}|$ square matrices of size $|T^*|$, call each matrix P_a
3: **for all** $a \in \mathcal{A}$ **do**
4: Initialize P_a with 0s
5: **for all** $c = (t_i \prec_1 t_j) \in C^*$, where $label(t_j) = a$ **do**
6: **for all** $S_i \in states[t_i]$ and $S_j \in states[t_j]$ **do**
7: $P_a(S_i, S_j) \leftarrow P_a(S_i, S_j) + 1/_{|states[t_j]|}$
8: **for all** S_i **do**
9: $$P_a(S_i, S_j) \leftarrow \begin{cases} \frac{P_a(S_i,S_j)-\epsilon(a)}{\sum_{j=0}^{n} P_a(S_i,S_j)}, & if S_i \neq S_j \\ \epsilon(a), & if S_i = S_j \end{cases}$$

It is easy to see that such macro states are likely to intersect as the number of possible paths that lead to a certain primitive task in the fully expanded HTN increases. In an HTN that has single methods to decompose each task, each primitive task in $\mathcal{H}^*$ could be made to correspond to a single state in the resulting MDP. However, when there are alternative paths to a certain state, we must devise a way to uniquely identify the possible ways in which a macro state is formed. This unique identification is necessary to allow an agent planning with the resulting MDP to identify, through its perceptions, which state it is in. Thus, we develop Algorithm 2 to detect overlapping states and partition them so that there is no ambiguity over the current state of an agent planning with the resulting MDP. Algorithm 2 creates a table of the MDP states $states[t_i]$ corresponding to each primitive task $t_i \in T^*$ by iterating over these tasks in a fixed order (preferably increasing complexity of the state) to ensure that the partition of states is consistent. If one macro S_i state overlaps with another, the resulting set of MDP states is mapped into one state consisting of the non-overlapping part of the original macro state (Line 4) as well as all the states S_j that originally overlapped with S_i (Line 7).

Each primitive task in the fully decomposed HTN is considered to represent the state achieved immediately after executing the action associated with it. So Algorithm 3 computes, for each action in the domain with an error rate ϵ, the transition probabilities between the states achieved by primitive tasks in $\mathcal{H}^*$, given the state overlaps and correspondences computed by Algorithm 2. Notice from Line 7 that, as the probability is conditional on (S_i, a), there is no need to factor the probability by $1/_{|states[t_i]|}$. Finally, in order to account for some uncertainty in the world, we introduce an error rate function ϵ, that, given an action a, returns the probability that this action will fail and the agent remains in the same state. In consequence, the resulting transition tables uniformly distribute the probability of transitioning from all possible immediately preceding states into those resulting from the task currently under consideration. For example, consider an HTN $\mathcal{H}_1 = (\{t_1, t_2, t_3\}, \{t_1 \prec t_2, t_2 \prec t_3\})$, with a domain $\mathcal{D} = (\{a_1, a_2, a_3, a_4\}, \{t_1, t_2, t_3\}, \{m_1, m_2, m_3\})$, in which:

- $m_1 = (\top, t_1, (\{a_1, a_2\}, \{a_1 \prec a_2\}))$
- $m_2 = (\top, t_1, (\{a_1, a_3\}, \{a_1 \prec a_3\}))$
- $m_3 = (\top, t_2, (\{a_4, a_1\}, \{a_4 \prec a_1\}))$

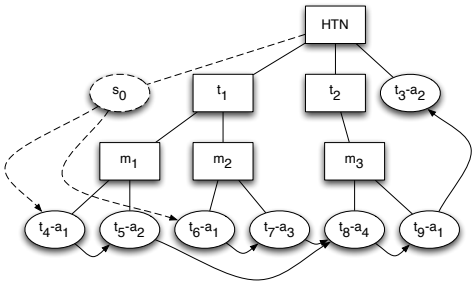


Figure 1: Fully decomposed HTN. Rectangles denote non-primitive tasks while ellipses denote primitive tasks. Dashed elements are added to the HTN for the MDP conversion.

where t_1 and t_2 are non-primitive tasks and t_3–t_9 are primitive task to execute actions a_1 to a_4.

The full decomposition of $\mathcal{H}_1$ is shown in Figure 1. The resulting MDP has eight states: the seven primitive tasks t_3, t_4, t_5, t_6, t_7, t_8 and t_9, plus the initial state s_0. Furthermore the transition function for action a_1 would have 0.5 probability of transitioning from s_0 to either t_4 or t_6, and probability 1 of transitioning from t_8 to t_9; while a_2 would have probability 1 for transitioning from t_4 to t_5, as well as for t_9 to t_3.

Proposition 1. *If a plan $\Delta = [a_1, \ldots, a_n]$ possible in $\mathcal{D}$ for an HTN $\mathcal{H}$ and a starting state s_0 induces a sequence of states $s1, \ldots, s_n$, then the probability $P_\Delta = \prod_{i=1}^{n} P_{a_n}(s_{n-1}, s_n)$ induced from the transition function generated from Algorithm 3 is greater than zero.*

Proof. Given that all sequences of action executions will be captured through $\prec$ constraints in $\mathcal{H}^*$ generated from Algorithm 1, and the corresponding states induced from these actions are captured in the mapping of Algorithm 2. Now, if there is any sequence of two tasks t_i, tj in $\mathcal{H}^*$, then the probability of a transition $P_{label(t_i)}(S_i, S_j)$ will necessarily be greater than zero due to Line 7 of Algorithm 3. Thus the multiplication of the proposition must be greater than zero. □

3.3 Generating a reward function

With the transition function in place, the only element missing from a full MDP description is the reward function. The purpose of a reward function in an MDP is to induce the solver to select actions that lead an agent towards certain desirable states. Correspondingly, in an HTN, the sequence of ordered tasks is also intended to lead an agent towards certain desirable outcomes. A similar analogy was made in earlier efforts [Simari and Parsons, 2006] regarding the conversion of Belief-Desire-Intention agent plans (so called *i-plans*) into

Algorithm 4 Creating reward values.

```
1: function CREATEREWARDFUNCTION(A,H* = (T*, C*))
2:     initialize all rewards u(a_j, S_i) to zero
3:     for all c = (t_i ≺_1 t_j) ∈ C* do
4:         S_i, S_j ← states in MDP corresponding to t_i, tj ∈ T
5:         a_j ← label(t_j) ∈ A
6:         κ ← maxpath(t_i, first(H*))
7:         for all s_k ∈ S_i do u(a_j, s_k) ← κ · Σ_{s_l ∈ S_j} u(s_l)
8:     return u
```

reward functions. Such a conversion consists of assigning an increasing reward value to the states resulting from actions that occur later in order of the steps in an i-plan. The same correspondence is leveraged in our work, as it has been shown by Sardiña *et al.*[Sardiña and Padgham, 2011] that i-plans and HTN methods are formally related in the sense that the ordering of tasks within the HTN used to decompose a task through a method has the same purpose to the action sequences in i-plans. As a consequence, we adapt the conversion algorithm in [Simari and Parsons, 2006] to generate a reward function from a fully expanded HTN $\mathcal{H}^*$ as Algorithm 4.

For our modified algorithm, we consider that each state s_i in the MDP (that was created from a primitive task t_i from $\mathcal{H}^*$) has some base utility $u(s_i)$, which can either be supplied by the user, or assumed to be 1. In order to obtain this gradient of rewards from the first possible tasks to the last ones in a plan, we use the ordering constraints in the fully expanded HTN to determine the maximum possible length of a plan before a certain state and action combination for which a reward is needed. Formally, the maximum length of plan that reaches a state s_i derived from a fully expanded HTN $\mathcal{H}^*$, denoted $maxpath(s_i, first(\mathcal{H}^*))$, is computed by following constraints $path_{t_n} = [t_0 \prec_1 t_1, \ldots, t_{n-1} \prec_1 t_n]$ in C^* such that t_n is the task in $\mathcal{H}^*$ that generated state s_i and $t_0 \in first(\mathcal{H}^*)$. Thus, the MDP reward function $u(a_j, s_i)$ for choosing an action a_j at a state s_i is generated by multiplying the base utility of s_j by the length of longest path between the state s_j being considered and the states $first(\mathcal{H}^*)$ that have no predecessors in the constraints for $\mathcal{H}^*$.

4 Experiments

In order to evaluate the computational demands of our conversion algorithm, as well as the efficiency of the resulting MDPs, we created an implementation of the conversion process and used it to create MDPs for an adaptation of the "Blogohar" scenario from [Sycara *et al.*, 2010]. In this scenario the objective is to plan the best allocation of resources and routing (from HQ) to neutralize various target insurgent strongholds using a set of resources, which consists of a combination of military vehicles of various types, including Humvees, Armored Personnel Carriers (APCs), and attack helicopters. Each vehicle has a certain probability of succeeding in its actions (both attack and movement) can use multiple routes to reach its target, and once a vehicle is committed to a target, it cannot be used to attack another target.[4]

[4]Due to space constraints, we shall not go into further detail about the scenario.

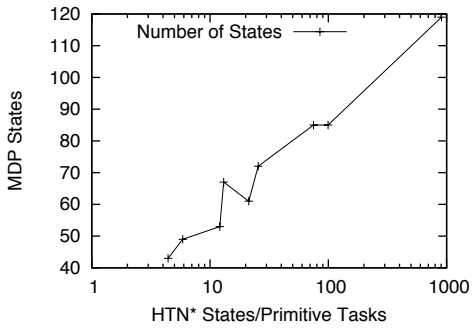
Our experiments consisted of an increasing number of vehicles and targets, which result in much larger fully expanded HTNs. The number of vehicles varied from 7 to 11 while the number of targets varied from 1 to 3, with at least two routes between HQ and each target. The domain encoding we used for this planning problem resulted in a Herbrand base that varied in size from about 15 to 25 thousand predicates for each planning instance. First, we measured the number of primitive tasks generated in the fully expanded HTN for each problem, and compared this to the number of unique states (in terms of true and false predicates) used in the resulting MDP, which is illustrated in the graph of Figure 2a. This shows that, while the number of primitive tasks in the fully expanded HTN explodes quite quickly, the MDP states resulting from Algorithm 2 grow much more slowly (notice the logarithmic scale of the graph). Second we measured the runtime of the conversion process, as well as the computation of an optimal MDP policy using policy iteration with a discount factor of 1. The conversion process as well as the MDP solver were implemented in Java, and the experiments were conducted in a $2.53Ghz$ 64-bit Intel processor running *Mac OS X* with a $2GB$ limit on VM memory usage.[5] The runtimes in the graph of Figure 2b show that, while the conversion process is rather expensive due to the huge size of the HTN state space, the reduced state space led to very efficient MDP encodings, which were solved by policy iteration always in under $100ms$.

5 Increasing Efficiency

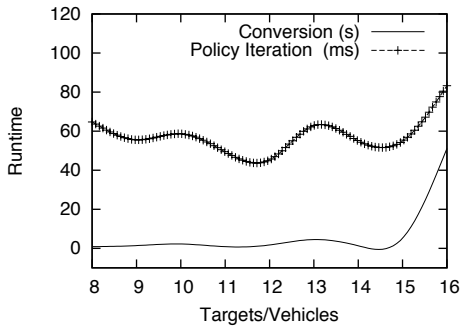
The calculation of the precedence relations for the fully expanded HTN in function δ^* requires verification for the possible states to support a method's precondition performed by the relation $\models_{s_0,\mathcal{H}}$, but has a complexity that grows exponentially with the branching factor of the expanded HTN tree (that is, the maximum number of methods applicable to any task). We can significantly increase the efficiency of the verification of method preconditions by adapting a technique used in the Yoyo planner [Kuter *et al.*, 2009], whereby states and actions are represented as BDDs. BDDs are directed acyclic graphs that represent logical formulas in a canonical form in which nodes correspond to propositions, each of which has two outgoing edges representing whether the proposition is true or not. The terminal nodes of a BDD represent either true or false, indicating whether the path chosen for traversing the graph (*i.e.* the truth value assigned to each proposition along the path) is true or false. Logical operations of conjunction, disjunction, negation and quantifiers can be implemented fairly efficiently among BDDs as long as good variable orderings are found for the results.

Using these properties of BDDs, it is possible to encode the execution of an action $s' = \gamma(s, a)$ Definition 1 as a BDD representing the conjunction of the logical formula for the propositions that hold at state s, the propositions that represents action a and the propositions that represent the resulting state s'. More importantly, the execution of an action over a set of states can be encoded even more efficiently using BDDs in order to mitigate the complexity of the $mState$ operation from Definition 4. We denote this new operation

[5]Implementation available at: `http://goo.gl/5SZRc`



(a) States in the HTN and corresponding MDP.



(b) Runtime results for our prototype.

Figure 2: Results of the experiments for the Blogohar scenario.

as $S' = \gamma^*(S, a)$, whereby S encodes the disjunction of the BDDs representing all possible previous states $s \in S$ and S' encodes the disjunction of all states $s' \in S'$ that are possibly true after a is executed. Using this encoding of states and actions as BDDs, we can also significantly increase the efficiency with which the $t_i \models_{s_0,\mathcal{H}} t_j$ relation can be computed, as the entailment of two BDD encoded (sets of) states possible for two primitive tasks t_i and t_j can be checked with complexity $O(|t_i| * |t_j|)$, where$|t_i|$ and $|t_j|$ are the size of the BDDs representing the models of t_i and t_j.

6 Conclusions and future work

The complexity of planning in the real world is best captured by non-deterministic planning, but the available representations are cumbersome and hard to use — adopting them may therefore increase rather than decrease the burden on the planners. However, as we describe here, it is possible to specify the planning problem as if it were a deterministic problem, and then convert the representation into a non-deterministic one. The main technical challenge has been to develop a conversion scheme that maintains the properties of the plan paths described by the HTN. In this paper we have shown how this conversion can be carried out. This conversion process has been implemented and tested for finite domains and initial results show that the conversion can be done fairly efficiently.

The techniques that perform the conversion are initial efforts with inherent limitations. The main limitation is that the MDP probabilities automatically generated by our conversion process refer mainly to the uncertainty in the planning process, with a simple action error model providing the uncertainty from the world, but we envision richer ways of having this information supplied with the input. For example, subjective probabilities can be annotated in the HTN methods and then used to calculate state transition probabilities in the resulting MDP. Alternatively, probabilities on the effect propositions of each action can also be supplied. Many techniques for expressing non-deterministic planning problems in a more compact form (e.g. factored MDPs [Boutilier *et al.*, 2000]) or in a more user-friendly way (e.g. PPDDL [Younes *et al.*, 2005]) have been created. These techniques provide significant improvements of performance for the planning algorithms, or readability of the planning domain, but not necessarily both at the same time. We claim that the technique developed in this paper comprises the initial steps for a compromise between user-friendliness and algorithmic efficiency. Moreover, our technique can be applied with little modification in traditional BDI-based agent languages [Sardiña and Padgham, 2011] to perform decision-theoretic planning.

Acknowledgement. This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[Bellman, 2003] R. E. Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[Boutilier *et al.*, 2000] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artif. Intell.*, 121:49–107, August 2000.

[Erol *et al.*, 1994] K. Erol, J. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[Ghallab *et al.*, 2004] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

[Kuter *et al.*, 2009] U. Kuter, D. Nau, M. Pistore, and P. Traverso. Task decomposition on abstract states, for planning under nondeterminism. *Artificial Intelligence*, 173(5-6):669 – 695, 2009.

[Sardiña and Padgham, 2011] Sebastian Sardiña and Lin Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.

[Simari and Parsons, 2006] G. I. Simari and S. Parsons. On the relationship between MDPs and the BDI architecture. In *Proc. 5th Intl. Joint Conf. on Auton. Agents and Multiagent Systems*, 2006.

[Sycara *et al.*, 2010] K. Sycara, T. J. Norman, J. A. Giampapa, M. J. Kollingbaum, C. Burnett, D. Masato, M. McCallum, and M. H. Strub. Agent support for policy-driven collaborative mission planning. *The Computer Journal*, 53(5):528–540, 2010.

[Younes *et al.*, 2005] Håkan L. S. Younes, Michael L. Littman, David Weissman, and John Asmuth. The first probabilistic track of the international planning competition. *J. Artif. Intell. Res. (JAIR)*, 24:851–887, 2005.