Tight Bounds for HTN Planning

Ron Alford
ASEE/NRL Postdoctoral Fellow
Washington, DC, USA
ronald.alford.ctr@nrl.navy.mil

Pascal Bercher
Ulm University
Ulm, Germany
pascal.bercher@uni-ulm.de

David W. Aha
U.S. Naval Research Laboratory
Washington, DC, USA
david.aha@nrl.navy.mil

Abstract

Although HTN planning is in general undecidable, there are many syntactically identifiable sub-classes of HTN problems that can be decided. For these sub-classes, the decision procedures provide upper complexity bounds. Lower bounds were often not investigated in more detail, however. We generalize a propositional HTN formalization to one that is based upon a function-free first-order logic and provide tight upper and lower complexity results along three axes: whether variables are allowed in operator and method schemas, whether the initial task and methods must be totally ordered, and where recursion is allowed (arbitrary recursion, tail-recursion, and acyclic problems). Our findings have practical implications, both for the reuse of classical planning techniques for HTN planning, and for the design of efficient HTN algorithms.

1 Introduction

Hierarchical Task Network (HTN) planning (Ghallab, Nau, and Traverso 2004) is an automated planning formalism concerned with the completion of tasks (activities or processes). Tasks in HTN planning are either primitive, corresponding to an action that can be taken, or non-primitive. HTN problems have a set of methods that act as recipes on non-primitive tasks, decomposing them into a further set of subtasks for which to plan. A non-primitive task may even decompose into itself, either directly via a method, or indirectly via a sequence of decompositions.

This recursive structure, when combined with partially-ordered tasks, is powerful enough to encode semi-decidable problems (Erol, Hendler, and Nau 1994). However, any one of numerous restrictions on HTN planning are enough to make HTN planning decidable (Erol, Hendler, and Nau 1994; Geier and Bercher 2011; Alford et al. 2012; 2014; Höller et al. 2014). In this paper, we explore complexity and expressiveness that results from the interplay between syntactic restrictions on decomposition methods. Specifically, we examine all combinations of three characteristics: whether...

- there is no recursion, tail-recursion, or arbitrary recursion,
- the methods and initial task network are totally ordered,
- methods and operators are ground, and whether constants may be mixed with variables in method definitions.

Table 1: Complexity classes for HTN planning (only completeness results). The undecidability result ("semi-decidable") is from Erol, Hendler, and Nau (1994).

Hierarchy	Ordering	Variables	Complexity	Theorem
no recursion (acyclic)	total	no	PSPACE	4.1
	total	CFM	NEXPTIME	4.2
	total	yes	EXPSPACE	4.1
	partial	no	NEXPTIME	6.1
	partial	CFM	NEXPTIME	4.2
	partial	yes	2-NEXPTIME	6.1
tail-recursion	total	no	PSPACE	3.7
	total	CFM	EXPSPACE	3.7
	total	yes	EXPSPACE	3.7
	partial	no	EXPSPACE	6.1
	partial	CFM	EXPSPACE	3.7
	partial	yes	2-EXPSPACE	6.1
arbitrary recursion	total	no	EXPTIME	5.1
	total	CFM	2-EXPTIME	5.1
	total	yes	2-EXPTIME	5.1
	partial	—	semi-decidable	—

We find that even without variables, the restricted classes of HTN planning range in expressivity from PSPACE-complete to EXPSPACE-complete. Just as in classical planning, extending these problems to include variables in the method and operator schemas gives an exponential bump in complexity. However, we identify a new (yet commonly met) restriction on HTN structure, that of constant-free methods (CFM), which often mitigates the computational impact of planning with variables. Table 1 provides a summary of our results. Erol, Hendler, and Nau (1994) provide the semi-decidability result. The rest constitute new lower bounds, new upper bounds, or both.

2 Lifted HTN Planning

In this section, we present a lifted version of the HTN planning formalism of Geier and Bercher (2011), which we extend to introduce variables.

In HTN planning, *task names* represent activities to accomplish and are syntactically first-order atoms. Given a set of task names X, a *task network* is a tuple $tn = (T, \prec, \alpha)$ such that:

- T is a finite nonempty set of *task symbols*.
- $\prec$ is a strict partial order over T.
- $\alpha : T \to X$ maps from task symbols to task names.

The task symbols function as place holders for task names, allowing multiple instances of a task name to exist in a task network. We say a task network is *ground* if all task names occurring in it are variable-free.

An *HTN problem* is a tuple $(\mathcal{L}, \mathcal{O}, \mathcal{M}, s_I, tn_I)$, where:

- $\mathcal{L}$ is a function-free first order language with a finite set of relations and constants.
- $\mathcal{O}$ is a set of *operators*, where each operator o is a triple (n, χ, e), where n is a task name (referred to as $name(o)$) not occurring in $\mathcal{L}$, χ is a first-order logic formula called the *precondition* of o, and e is a conjunction of positive and negative literals in $\mathcal{L}$ called the *effects* of o. We refer to the set of task names in $\mathcal{O}$ as *primitive* task names.
- $\mathcal{M}$ is a set of *decomposition methods*, where each method m is a pair (c, tn), where c is a (non-primitive or compound) task name, called the method's *head*, not occurring in $\mathcal{O}$ or $\mathcal{L}$, and tn is a task network, called the method's *subtasks*, defined over the names in $\mathcal{O}$ and the method heads in $\mathcal{M}$.
- s_I is the (ground) initial state and tn_I is the initial task network that is defined over the names in $\mathcal{O}$.

We define the semantics of lifted HTN planning through grounding. Given that $\mathcal{L}$ is function-free with a finite set of relations and constants, we can create a ground (or propositional) HTN planning problem $P = (\mathcal{L}, O, M, s_I, tn'_I)$, where O and M are variable-free: Let V be the set of all full assignments from variables in $\mathcal{L}$ to constants in $\mathcal{L}$. Then the ground methods M are given by $\bigcup_{v \in V, (c,tn) \in \mathcal{M}} \{(c[v], tn[v])\} \cup \bigcup_{v \in V} \{(c_{top}[v], tn_I[v])\}$, where the syntax $c[v]$ denotes syntactic variable substitution of the variables occurring in c with the matching term from v. Note that we introduced additional methods not present in $\mathcal{M}$ that are required to ground the initial task network. The symbol c_{top} not occurring in $\mathcal{O}$, $\mathcal{L}$, or in one of the methods of $\mathcal{M}$ is a new compound task name that decomposes into the possible groundings of the original initial task network tn_I. The initial task network tn'_I of the ground HTN problem P then consists of only this new task c_{top}. Let $\mathcal{O}'$ be the set of quantifier-free operators obtained from $\mathcal{O}$ by eliminating the variables with constants from $\mathcal{L}$ (Gazen and Knoblock 1997). Then the ground operators O are given by $\bigcup_{v \in V, (n,\chi,e) \in \mathcal{O}'} \{(n[v], \chi[v], e[v])\}$.

The ground operators O form an implicit *state-transition function* $\gamma : 2^{\mathcal{L}} \times O \to 2^{\mathcal{L}}$ for the problem, where:

- A state is any subset of the ground atoms in $\mathcal{L}$. The finite set of states in a problem is denoted by $2^{\mathcal{L}}$;
- o is *applicable* in a state s iff $s \models prec(o)$;
- $\gamma(s, o)$ is defined iff o is applicable in s; and
- $\gamma(s, o) = (s \setminus del(o)) \cup add(o)$.

A sequence of ground operators $\langle o_1, \ldots, o_n \rangle$ is *executable* in a state s_0 iff there exists a sequence of states $s_1, \ldots, s_n$ such that $\forall_{1 \le i \le n} \gamma(s_{i-1}, o_i) = s_i$. A ground task network $tn = (T, \prec, \alpha)$ is *primitive* iff it contains only task names from $\mathcal{O}$. tn is *executable* in a state s_0 iff tn is primitive and there exists a total ordering $t_1, \ldots, t_n$ of T consistent with $\prec$ such that $\langle \alpha(t_1), \ldots, \alpha(t_n) \rangle$ is executable starting in s_0.

For a ground HTN problem $P = (\mathcal{L}, O, M, s_I, tn_I)$, we can *decompose* the task network $tn_I = (T, \prec, \alpha)$ if there is a task $t \in T$ such that $\alpha(t)$ is a non-primitive task name and there is a corresponding method $m = (\alpha(t), (T_m, \prec_m, \alpha_m)) \in M$. Intuitively, decomposition is done by selecting a task with a non-primitive task name, and replacing the task in the network with the task network of a corresponding method. More formally, assume without loss of generality that $T \cap T_m = \emptyset$. Then the decomposition of t in tn by m into a task network tn' is given by:

$$\begin{aligned}
T' &:= (T \setminus \{t\}) \cup T_m \\
\prec' &:= \{(t, t') \in \prec \mid t, t' \in T'\} \cup \prec_m \\
&\quad \cup \{(t_1, t_2) \in T_m \times T \mid (t, t_2) \in \prec\} \\
&\quad \cup \{(t_1, t_2) \in T \times T_m \mid (t_1, t) \in \prec\} \\
\alpha' &:= \{(t, n) \in \alpha \mid t \in T'\} \cup \alpha_m \\
tn' &:= (T', \prec', \alpha')
\end{aligned}$$

A ground HTN problem P is *solvable* iff either tn is executable in s_I, or there is a sequence of decompositions of tn to a task network tn' such that tn' is executable in s_I. Checking whether an HTN problem has a solution is the plan-existence decision problem.

Current HTN planners either solve problems using decomposition directly, or by using *progression* (Nau et al. 2003; Alford et al. 2012). Progression consists of a choice of two operations on just the *unconstrained* tasks (those without a predecessor in the task network): decomposition, or *operator application*. Operator application takes an unconstrained primitive task t in the current task network tn such that $\alpha(t)$ is applicable in s, removes t from tn to form a new task network tn', and returns a new HTN problem with $\gamma(s, \alpha(t))$ as the initial state and tn' as the initial task network. Hence, progression interleaves decomposition and finding a total executable order over the primitive tasks. A problem's *progression bound* is the size of the largest task network reachable via any sequence of progressions.

3 Stratifications for propositional and constant-free method domains

Decomposition and progression respectively decide, roughly speaking, the class of *acyclic* HTN problems (HTN problems without recursion) and *tail-recursive* HTN problems (those problems where tasks can only recurse through the last task of any method). Alford et al. (2012) give syntactic tests for identifying ground method structures that are decided by decomposition and progression (identified by $\le_1$-stratification and $\le_r$-stratification, respectively).

In this section, we generalize the $\leq_1$- and $\leq_r$-stratification tests to identify all acyclic and tail-recursive lifted HTN problems whose methods are constant-free. Formally a *constant-free method* (CFM) HTN problem is one where only variables may occur as terms in the task names of the domain's methods (both in the head and the subtasks). Fully-ground domains can be trivially transformed into CFM domains by rewriting the task names with 0-arity predicates, and so we include both fully-ground and propositional problems in the class of constant-free method problems.

Extending the stratification tests to CFM problems will yield upper complexity bounds for acyclic and tail-recursive CFM problems (NEXPTIME and EXPSPACE, respectively). Since propositional HTN planning is equivalent to HTN planning with all unary predicates (and thus constant-free), these are also upper bounds for acyclic and tail-recursive propositional problems.

Upper bounds for mostly-acyclic HTN problems

When the method structure of a problem is acyclic, every sequence of decompositions is finite, and so almost any decomposition-based algorithm is a decision-procedure for the problem (Erol, Hendler, and Nau 1994). Alford et al. (2012) extend the class of ayclic problems to include those whose methods only allow recursion when it does not increase the size of the task network, and define a syntactic test called $\leq_1$-stratifiability to recognize ground instances of these problems. Here we extend $\leq_1$-stratification to CFM HTN problems.

A CFM HTN problem $\mathcal{P}$ is $\leq_1$*-stratifiable* if there exists a total preorder (a relation that is both reflexive and transitive) $\leq_1$ on the task names in $\mathcal{P}$ such that:

- For any task names c_1 and c_2 in $\mathcal{P}$, if there are variable substitutions v_1, v_2 such that $c_1 [v_1] = c_2 [v_2]$, then $c_1 \leq_1 c_2 \leq_1 c_1$.
- For every method $(c, (T, \prec, \alpha))$ in $\mathcal{P}$:
  - If $|T| > 1$, then $\forall_{t_i \in T} \alpha(t_i) <_1 c$
  - If $T = \{t\}$, then $\alpha(t) \leq_1 c$

The above conditions ensure that any decomposition in $\leq_1$-stratifiable problems either replaces a task with a single task from the same stratum, or replaces a task with one or more tasks from lower strata. We can determine $\leq_1$-stratification in polynomial time with any algorithm for finding strongly connected components in a directed graph, such as Tarjan's algorithm.

By extending $\leq_1$-stratification to CFM HTN problems, we can show that there is a strict correspondence between the stratification of a CFM problem and its grounding:

Lemma 3.1. *A CFM HTN problem $\mathcal{P}$ is $\leq_1$-stratifiable if and only if there exists a $\leq_1$-stratification of the grounding of $\mathcal{P}$ of the same height.*

Proof. Let $\mathcal{L}$ be the language of $\mathcal{P}$ and P be the grounding of $\mathcal{P}$. If $\mathcal{P}$ is $\leq_1$-stratifiable, then grounding the task names of each level of a $\leq_1$-stratification for $\mathcal{P}$ is a stratification of P of the same height.

So assume $\mathcal{P}$ is not $\leq_1$ stratifiable. Then by the negation of $\leq_1$-stratifiability, there are methods $(c_1, tn_1), \ldots, (c_k, tn_k)$ and variable assignments $v_1, \ldots, v_k$ such that tn_1 has more than one subtask, $tn_k [v_k]$ contains the subtask c_1, and each $tn_i [v_i]$ contains the subtask c_{i+1}.

Let a be an arbitrary constant in $\mathcal{L}$ and v_a be the assignment that maps all variables in $\mathcal{L}$ to a. Then each $(c_i [v_a], tn_i [v_a])$ is a ground method in P. Each $tn_i [v_a]$ contains the task name $c_{i+1} [v_a]$, and $tn_k [v_a]$ contains $c_1 [v_a]$ as a subtask, so P is also not $\leq_1$-stratifiable. □

We call HTN problems *mostly-acyclic* if either they are CFM and $\leq_1$-stratifiable or they are non-CFM and their grounding is $\leq_1$-stratifiable. If all sequences of decompositions of a given problem are finite, we call that problem *acyclic*. Section 4 contains examples of both CFM and non-CFM acyclic method structures.

We can transform any propositional $\leq_1$-stratifiable problem into an acyclic problem in polynomial time as follows: Let $P = (\mathcal{L}, O, M, s_I, tn_I)$ be any $\leq_1$-stratifiable propositional HTN problem, and let S be the *maximal* $\leq_1$-stratification in the following sense: if c_1 and c_2 are task names on a stratum of S, then $c_1 \leq_1 c_2 \leq_1 c_1$ in any $\leq_1$-stratification of P. So each task name is on a stratum by itself, or the stratum consists of a set of task names $c_1, \ldots, c_k$ such that $c_1 \leq_1 \ldots \leq_1 c_k \leq_1 c_1$. Let $M_r \subseteq M$ be the methods responsible for these later constraints (each having some c_i as a task head and a task network with a singular task of some c_j), and let $M_a \subseteq M$ be the methods leading to strictly lower strata. Then $(M \setminus M_r) \cup \bigcup_{1 \leq i \leq k} \{(c_i, tn_a) \mid (c_a, tn_a) \in M_a\}$ eliminates recursion at this stratum while still admitting the same set of primitive decompositions as M. Repeating this process on each strata eliminates recursion from the problem at the cost of a polynomial increase in size.

This leads to an upper bound for $\leq_1$-stratifiable CFM HTN problems:

Corollary 3.2. *Plan-existence for CFM (and propositional) mostly-acyclic HTN problems is in* NEXPTIME.

Proof. Let $\mathcal{P} = (\mathcal{L}, \mathcal{O}, \mathcal{M}, s_I, tn_I)$ be a $\leq_1$-stratifiable CFM HTN problem, and let $\mathcal{S}$ be $\mathcal{P}$'s maximal $\leq_1$-stratification. Let m be the maximum number of tasks in tn_I or any method, and let k be the number of task names occurring in $\mathcal{M}$.

Let P be the grounding of $\mathcal{P}$ (taking EXPTIME) with a $\leq_1$-stratification S of the same height as $\mathcal{S}$. By the above process, we can create P' as an acyclic version of P, and since that process preserves any stratification, S is also a $\leq_1$-stratification of P'.

By the construction of S from $\mathcal{S}$, any decomposition of a task results in a set of tasks from strictly lower strata. This gives a tree-like structure to the decomposition hierarchy, with a maximum branching factor of m and a max depth of k. So m^k is a bound on the length of any sequence of decompositions of the initial task network. Thus the following is a decision procedure for $\mathcal{P}$:

Pick and apply a sequence of decompositions of tn_I of length m^k or less (NEXPTIME). Guess a total ordering of the resulting network and check if it is executable in s. □

Grounding an HTN problem produces a worst-case size blowup that is exponential in the arity of task names and predicates. Thus, if $b(x)$ is an upper space or time bound for a class of ground HTN problems B, then $O\left(2^{b(x)}\right)$ is an upper bound (space or time, respectively) for problems whose groundings are in B. So Corollary 3.2 implies a 2-NEXPTIME upper bound for non-CFM mostly-acyclic HTN problems. Section 5 provides matching lower bounds.

Upper bounds for tail-recursive HTN problems

Many problems are structured so that tasks can only recurse through the last task of any of its associated methods. These problems are guaranteed to have a finite progression bound, and thus are decided by simple progression-based algorithms. Alford et al. (2012) introduced a syntactic test called $\leq_r$-stratifiability to identify all sets of propositional methods that are guaranteed to have a finite progression bound. Here we extend the definition of $\leq_r$-stratifiability to include CFM HTN problems. We then prove upper space bounds on the size of task networks under progression for $\leq_r$-stratifiable problems: an exponential bound for $\leq_r$-stratifiable CFM problems, and a polynomial bound for totally-ordered propositional $\leq_r$-stratifiable problems.

A CFM HTN problem $\mathcal{P}$ is *$\leq_r$-stratifiable* if there exists a total preorder $\leq_r$ on the task names in $\mathcal{P}$ such that:

- For each pair of task names c_1 and c_2 in $\mathcal{P}$, if there are variable substitutions v_1, v_2 such that $c_1[v_1] = c_2[v_2]$, then $c_1 \leq_r c_2 \leq_r c_1$.
- For every method $(c, (T, \prec, \alpha))$ in $\mathcal{P}$:
  - If there is a task $t_r \in T$ such that all other tasks are predecessors ($\forall_{t \in T, t \neq t_r} t \prec t_r$), then $\alpha(t_r) \leq_r c$. We call t_r the *last task* of $(T, \prec, \alpha)$.
  - For all non-last tasks $t \in T$, $\alpha(t) <_r c$.

The above conditions ensure that methods in $\leq_r$-stratifiable problems can only recurse through their last task, yet it still allows hierarchies of tasks. This is a strict generalization of $\leq_1$-stratifiability from the previous section, and of *regular* HTN problems from Erol, Hendler, and Nau (1994), which only allow at most one non-primitive task in every method (and the initial task network) that also has to occur as the last task in the respective method (the initial task network, respectively). As with $\leq_1$-stratifiability, there is a strict correspondence between a CFM problem's $\leq_r$-stratifiability and its grounding's $\leq_r$-stratibiablity:

Lemma 3.3. *A CFM HTN problem $\mathcal{P}$ is $\leq_r$-stratifiable if and only if there exists a $\leq_r$-stratification of the grounding of $\mathcal{P}$ of the same height.*

We omit a formal proof, since it is structurally identical to the one of Lemma 3.1. We call both $\leq_r$-stratifiable CFM problems and non-CFM problems whose groundings are $\leq_r$-stratifiable *tail-recursive*.

This allows us to calculate a bound on the size of task networks reachable under progression in a bottom-up fashion. Let $\mathcal{P} = (\mathcal{L}, \mathcal{O}, \mathcal{M}, s_I, tn_I)$ be a tail-recursive CFM HTN problem where $S_1, \ldots, S_n$ is a $\leq_r$-stratification of $\mathcal{P}$. WLOG, assume that S_1 contains only primitive task names. The contribution of any primitive task in tn (those with names in S_1) to the progression bound is 1. The contribution of tasks of tn with names in S_2 is then bounded by the number of tasks in the largest method corresponding to a task name in S_2. Upper strata are bounded by the size of the largest corresponding task network multiplied by the bound for the next lower stratum. This gives an exponential worst-case progression bound on $\leq_r$-stratifiable problems:

Lemma 3.4. *If $\mathcal{P}$ is a tail-recursive CFM HTN problem with k initial tasks, r is the largest number of tasks in any method in $\mathcal{P}$ and h the height of $\mathcal{P}$'s $\leq_r$-stratification, then $k \cdot r^h$ is a progression bound for $\mathcal{P}$.*

This implies upper bounds for all tail-recursive problems:

Corollary 3.5. *Plan-existence for propositional and CFM HTN $\leq_r$-stratifiable problems is in* EXPSPACE*. Plan-existence for non-CFM HTN problems whose groundings are $\leq_r$-stratifiable is in 2-*EXPSPACE*.*

Consider the case that each task network in $\mathcal{P}$ is totally-ordered. Then any progression of $\mathcal{P}$ leaves the initial task network totally ordered. Moreover, decomposition of the first task in the network can only grow the list with tasks from lower strata. This gives a progression bound for totally-ordered tail-recursive problems:

Lemma 3.6. *If $\mathcal{P}$ is a totally-ordered tail-recursive CFM HTN problem with k initial tasks, r is the largest number of tasks in any method in $\mathcal{P}$, and h the height of $\mathcal{P}$'s $\leq_r$-stratification, then $k + r \cdot h$ is a progression bound for $\mathcal{P}$.*

This gives a PSPACE upper bound for propositional totally-ordered tail-recursive problems. CFM $\leq_r$-stratifiable problems, whether ordered or not, are dominated by the size of the state and not the task network, and so have an EXPSPACE upper bound. Via grounding, non-CFM totally-ordered tail-recursive problems (those whose groundings are $\leq_r$-stratifiable) have an exponential progression bound, which matches their worst-case state size, and so are also in EXPSPACE.

Erol, Hendler, and Nau (1994) give encodings of both propositional and lifted classical planning into *regular* HTN problems, where every method has at most one non-primitive task, and that task must be the last task in the method. The lifted encoding uses no constants in the methods, so both are $\leq_r$-stratifiable. Since propositional planning is PSPACE-complete (Bylander 1994) and lifted planning is EXPSPACE-complete (Erol, Nau, and Subrahmanian 1995), this gives our first completeness results of the paper:

Theorem 3.7. *Plan-existence for propositional totally-ordered tail-recursive problems is* PSPACE*-complete. Plan-existence for tail-recursive CFM HTN problems and totally-ordered non-CFM tail-recursive problems is* EXPSPACE*-complete.*

4 Hierarchies and counting in HTNs

Whereas tail-recursive HTN problems allow us to express tasks that may repeat an arbitrary number of times, the number of repeats is fixed in advance for acyclic and mostly-acyclic problems. In this section, we will show how to express in polynomial space tasks that occur in exponential and double-exponential numbers of times in any solution. This will give us immediate lower bounds for totally-ordered acyclic problems, and will also be used in Section 6 for the lower bounds of partially-ordered problems.

First, we show how to repeat a task an exponential number of times with a set of propositional $\leq_1$-stratifiable methods: Let $k \geq 0$ and let o_0 be some task name. Let $M_{ok} = \{(o_1, tn_1), \ldots, (o_k, tn_k)\}$ be task names such that each $tn_i = (T, \prec, \alpha)$ contains two tasks t_1, t_2, such that $t_1 \prec t_2$ and $\alpha(t_1) = \alpha(t_2) = o_{i-1}$. Thus o_1 decomposes into two copies of o_0, o_2 decomposes into four, and so on until o_k decomposes into 2^k copies of o_0. With M_{ok}, any sequence of o_1 of length $2^{k+1} - 1$ or less can be expressed in a task network by taking the appropriate subset of $\{o_0, \ldots, o_k\}$.

We can also encode doubly-exponential repeats of tasks with non-CFM $\leq_1$-stratifiable methods: Let k be a positive integer and let o be some task name, and let $0, 1$ be arbitrary, distinct constants from $\mathcal{L}$. Given a new k-arity predicate oe, we will give a set of task names and methods that form a counter from $oe(1, \ldots, 1)$ down to $oe(0, \ldots, 0)$. Let $oe_1, \ldots, oe_k$ be task names such that each oe_i has the form $oe(v_k, \ldots, v_{i+1}, 1, 0, \ldots, 0)$ where each v_m is a variable. So $oe_1 = oe(v_k, \ldots, v_2, 1)$, $oe_2 = oe(v_k, \ldots, v_3, 1, 0)$, $oe_{k-1} = oe(v_k, 1, 0, \ldots, 0)$, and $oe_k = oe(1, 0, \ldots, 0)$. Similarly, let $oe'_1, \ldots, oe'_k$ be task names of the form $oe(v_k, \ldots, v_{i+1}, 0, 1, \ldots, 1)$, so $oe'_1 = oe(v_k, \ldots, v_2, 0)$, $oe'_2 = oe(v_k, \ldots, v_3, 0, 1)$, $oe'_{k-1} = oe(v_k, 0, 1, \ldots, 1)$, and $oe'_k = oe(0, 1, \ldots, 1)$. So if v is an assignment of $v_1, \ldots, v_k$ to $\{0, 1\}$ and we view $oe_i[v]$ as a binary number j, then $oe'_i[v]$ is $j - 1$.

Let $\mathcal{M}_{oek} = \{(oe_0, tn_{o0}), \ldots, (oe_k, tn_{oek})\}$, where: $oe_0 = oe(0, \ldots, 0)$, tn_{oe0} has two copies of o as subtasks, and each tn_{oei} has two ordered copies of oe'_i. Grounding $\mathcal{M}_{oek}$ to $\{0, 1\}$ produces a set of ground methods with a $\leq_1$-stratification of height 2^{k+1} (including o). Groundings that include other constants are essentially truncated counters with no methods that lead to oe_0 (though they are still $\leq_1$-stratifiable and disjoint with the grounding to $\{0, 1\}$).

Thus, $oe(0, \ldots, 0)$ decomposes into two copies of o, $oe(0, \ldots, 0, 1)$ decomposes first into two copies of $oe(0, \ldots, 0)$ (and then four of o), and so on, until $oe(1, \ldots, 1)$ has $2^{2^{k+1}-1}$ copies of o.

Let K be any polynomially-bounded sum of integers of the form 2^i and 2^{2^i} for $i < k$. The above two counting results allow us to express K repetitions of a task o in space polynomial in k. We will use the shorthand $K \cdot o$ as the task name that decomposes into such a sequence, along with implying the existence of the supporting methods.

Erol, Hendler, and Nau (1994) use encoding of classical planning into regular HTN problems (a subset of tail-recursive problems, but not acyclic) to achieve lower bounds of PSPACE- and EXPSPACE-complete for propositional and lifted regular HTN problems, respectively. Here we sketch how to adapt this proof to acyclic problems:

Theorem 4.1. *Plan-existence for totally-ordered mostly-acyclic propositional HTN problems is* PSPACE*-complete. For non-CFM totally-ordered mostly-acyclic problems, plan-existence is* EXPSPACE*-complete.*

Proof. The upper bounds of PSPACE and EXPSPACE, respectively, are established by Theorem 3.7, since acyclic problems are by definition tail-recursive.

Let $\mathcal{P}_C = (\mathcal{L}, \mathcal{O}, s, \mathcal{G})$ be a classical planning problem where $\mathcal{G}$ is a closed formula describing all goal states, and the rest are defined as in HTN planning. Any executable sequence of operators leading to a state satisfying the goal is a solution. The length of the shortest solution in classical planning is bound by the number of possible states, which in problems with variables is $2^{p \cdot c^a}$, where p is the number of relations, c is the number of constants, and a is the max arity of any relation. Let $k = \lceil \log_2 p + a \cdot \log_2 c \rceil$. Then we can encode $\mathcal{P}_C$ as follows:

Let $\mathcal{P}_H = (\mathcal{L}, \mathcal{O}_H, \mathcal{M}, s, tn_I)$ be an HTN problem where $\mathcal{L}$ and s are the same as in $\mathcal{P}_C$. Let $\mathcal{O}_H = \mathcal{O} \cup \{skip, g\}$, where $skip$ is an operator without preconditions or effects, and g is an operator with the precondition of $\mathcal{G}$.

Let $\mathcal{M}$ contain a new task name any, along with methods for each operator $o \in \mathcal{O} \cup \{skip\}$ that decomposes any into o. $\mathcal{M}$ should also contain the necessary method for implementing $2^{2^k} \cdot any$. Let the initial task network tn_I contain two tasks t_1, t_2 with $t_1 \prec t_2$, such that $\alpha(t_1) = 2^{2^k} \cdot any$ and $\alpha(t_2) = g$.

Then $\mathcal{P}_H$ is a totally-ordered acyclic problem, and tn can decompose into any sequence of 2^{2^k} or less (ignoring skips) followed by an operator g which checks that the goal condition holds. Thus any solution to $\mathcal{P}_H$ can be trivially transformed into a solution to $\mathcal{P}_C$, and any solution to $\mathcal{P}_C$ can be padded with $skip$ operators to form a solution to $\mathcal{P}_H$. So $\mathcal{P}_H$ is solvable if and only if $\mathcal{P}_C$ is solvable, making totally-ordered acyclic HTN planning EXPSPACE hard.

If, instead, $\mathcal{P}$ was ground, the number of possible states (and thus the length of the shortest solution) is bound by 2^p, where p is the number of propositions in $\mathcal{L}$. As $2^p \cdot any$ can be represented in polynomial space in a propositional HTN problem, the above translation is a polynomial encoding of propositional classical planning into propositional totally-ordered acyclic HTN planning. Thus totally-ordered acyclic propositional HTN planning is PSPACE hard. □

Acyclic CFM problems constitute a middle ground between propositional and lifted HTN planning. Here we adapt the encoding of an EXPSPACE-bounded Turing machine into classical planning (Erol, Nau, and Subrahmanian 1991). However, we will only run it an exponential number of steps, giving us a NEXPTIME lower bound.

Theorem 4.2. *For CFM mostly-acyclic HTN problems, regardless of ordering, plan-existence is* NEXPTIME*-complete.*

Proof. The upper bound for CFM acyclic problems is established by Corollary 3.2.

Let M be a nondeterministic Turing machine (TM). We will give an encoding of M that simulates 2^k steps of M. Erol, Nau, and Subrahmanian (1991) describe an encoding of an EXPSPACE-bounded TM into a classical planning problem with an initial state s and a set of operators $\mathcal{O}_{init}$, $\mathcal{O}_{step}$, and $\mathcal{O}_{done}$, where:

- Operators from $\mathcal{O}_{init}$ initialize the 'tape' (a set of *cell* relations indexed with a binary counter)
- Each operator $o \in \mathcal{O}_{step}$ mimics a single transition of M.
- Each operator $o \in \mathcal{O}_{done}$ adds the literal $done()$ to the state whenever the machine is in an accepting state.

Let $accepted$ be an operator which has a precondition of '$done()$', and let $\mathcal{O} = \mathcal{O}_{init} \cup \mathcal{O}_{step} \cup \mathcal{O}_{done} \cup \{accepted\}$.

We define $\mathcal{M}$ to be a set of methods such that $sim()$ is a non-primitive task with methods that decompose it into any operator in $\mathcal{O}_{step} \cup \mathcal{O}_{done}$, and $\mathcal{M}$ contains methods for implementing $2^k \cdot init$ and $2^k \cdot sim$. Let tn_I be the initial task network which contains three tasks, $t_1 \prec t_2 \prec t_3$ such that $\alpha(t_1) = 2^k \cdot init$, $\alpha(t_2) = 2^k \cdot sim$, and $\alpha(t_3) = accepted$. Then the acyclic CFM problem $\mathcal{P} = (\mathcal{L}, \mathcal{O}, \mathcal{M}, s, tn_I)$ is solvable if and only if there is a run of M that finds an excepting state within 2^k steps. Thus, acyclic CFM planning is NEXPTIME-hard. □

5 Alternating Turing machines for totally-ordered problems

Erol, Hendler, and Nau (1994) show that while arbitrary recursion when combined with partially-ordered tasks is undecidable, arbitrary recursion with totally ordered tasks in EXPTIME for propositional problems and 2-EXPTIME otherwise. Here we show that those bounds are tight by encoding space-bounded alternating Turing machines.

An alternating Turing machine (ATM) is syntactically identical to a nondeterministic Turing machine (NTM). However, where an NTM accepts if any run of the machine accepts, an ATM accepts only if all runs of the machine accept. The classes of problems that run in polynomial or exponential space on an ATM are APSPACE and AEXPSPACE, respectively. Since APSPACE=EXPTIME and AEXPSPACE=2-EXPTIME (Chandra, Kozen, and Stockmeyer 1981), an encoding of a space-bounded ATM gives lower time bounds for totally-ordered HTN planning:

Theorem 5.1. *Propositional totally-ordered HTN planning is* EXPTIME*-complete. CFM and non-CFM HTN planning is* 2-EXPTIME*-complete.*

Proof. Erol, Hendler, and Nau (1994) established the EXPTIME and 2-EXPTIME upper bounds. Alford et al. (2012) confirm these upper bounds with a set-theoretic formulation of HTN planning.

Let A be an ATM, denoted by $A = (S, \Sigma, \Gamma, \delta, q_0, F)$, where K is a finite state of state symbols, $F \subseteq S$ is the set of accepting states, Γ is the set of tape symbols with $\Sigma \subset \Gamma$ being the allowable input symbols, $q_0 \in S$ is the initial state, and δ is the transition function, mapping from $S \times \Gamma$ to $\mathcal{P}(S \times \Gamma \times \{\text{Left, Right}\})$.

We will use the same initial state s and operators $\mathcal{O}_{init}$ and $\mathcal{O}_{step}$ that were used in the proof of Theorem 4.2. We will use the operators $\mathcal{O}_{done}$ with the modification that they have no effect, just the precondition of test whether the machine is in an accepting state. To this, we add a set of inverting operators $\mathcal{O}_{step}^{-1}$, such that for each $o \in \mathcal{O}_{step}$, there is a $o^{-1} \in \mathcal{O}_{step}^{-1}$ such that $\gamma\left(\gamma(s, o), o^{-1}\right) = s$ for every state s in which o is applicable.

Let $\mathcal{O} = \mathcal{O}_{init} \cup \mathcal{O}_{done} \cup \mathcal{O}_{step} \cup \mathcal{O}_{step}^{-1}$. The set of methods $\mathcal{M}$ is defined as follows:

- For each operator $o \in \mathcal{O}_{init}$, we have a method $(init, tn)$, where tn contains just the task o.
- For each operator $o \in \mathcal{O}_{done}$, we have a method (sim, tn), where tn contains just the task o.
- For each state s and tape symbol c, we have a method $(sim, tn) \in \mathcal{M}$. Let $T = \delta(s, c)$. T denotes a set of transitions, and A must halt on each of these. So let $o_1, \ldots, o_n$ be the operators from $\mathcal{O}_{step}$ associated with the transitions in T. Then tn is the tasks network with the totally-ordered tasks o_1, sim, o_1^{-1} o_2, sim, o_2^{-1}, …, o_n, sim, o_n^{-1}.

Let tn_I be the initial task network which contains the totally-ordered tasks $2^k \cdot init$ and sim. Let $\mathcal{P}$ be the totally-ordered CFM HTN problem given by $\mathcal{P} = (\mathcal{L}, \mathcal{O}, \mathcal{M}, s, tn_I)$. Notice that, although the methods for the sim task change the state, they always revert it before they're done: If the machine is already in an accepting state, the first set of methods leave the state unchanged. Otherwise, if there is a transition in δ for the current state and tape symbol, then there is a corresponding method in $\mathcal{M}$. The method applies an operator, runs the sim task to conclusion, reverts the operators, and so on until each operator has been applied and the sim tasks run. So all possible runs of A (which we bound to run in AEXPSPACE) are verified. Since A was arbitrary, totally-ordered CFM planning is 2-EXPTIME-hard.

The encoding from Erol, Nau, and Subrahmanian (1991) uses predicates of logarithmic arity in the size of the bound. The only use of variables in our encoding was for these operators, so if we had chosen a polynomial space bound, the grounding of our encoding would have been polynomial in size. Thus, the same proof works with the polynomial bound to encode an APSPACE machine, so totally-ordered propositional planning is EXPTIME-hard. □

6 Interactions with partial-orders and hierarchies

In Section 4, we described acyclic counting techniques that generated large numbers of tasks. When the methods and initial tasks were ordered, progression-based algorithms had to deal with only a small portion of those tasks at a time. However, when the tasks are partially ordered, tasks can interact with each other in intricate ways. Here we adapt the EXPSPACE-completeness proof of reachability for communicating hierarchical state machines from Alur, Kannan, and Yannakakis (1999) to obtain lower bounds for partially-ordered HTN problems that match the upper bounds we provide in Section 3. Since the proofs will be nearly identical to each other, we collapse them into one theorem and prove only the completeness bound for the partially-ordered lifted acyclic HTN problems:

Theorem 6.1. *Plan-existence for partially-ordered propositional acyclic HTN planning is* NEXPTIME*-complete; for partially-ordered propositional tail-recursive HTN planning is* EXPSPACE*-complete; for partially-ordered lifted acyclic*

*HTN planning is 2-*NEXPTIME*-complete; and for partially-ordered lifted tail-recursive HTN planning is 2-*EXPSPACE*-complete.*

Proof. Upper bounds were established by Lemma 3.1.

For the lower bound, let $N = (S, \Sigma, \Gamma, \delta, q_0, F)$ be a non-deterministic Turing machine (NTM). Given a positive integer k, we will encode N into a lifted acyclic HTN planning problem $\mathcal{P}$ such that $\mathcal{P}$ is solvable if and only if there is a run of N that is in an accepting state after 2^{2^k} steps.

Let $K = 2^{2^k}$. Since N has a K size space bound, we can view a configuration of N as the position of the head, the state, and a string w over Γ of length K representing the tape. Then, if $w_0, \ldots, w_K$ are the tape configurations of an accepting run of N, then the string $W = \#w_0\#w_1\#\ldots\#w_K$, where $\#$ is a separator, represents a checkable proof that N halts on this input in K steps. To check the proof, we need to make sure that each w_i follows from the w_{i-1} before it, which we will check character by character. Specifically, if we are checking the j^{th} character of w_i, then the j^{th} character of w_{i+1} (or exactly $K+1$ characters later in W) is either the same as it was in w_i, or the head was over the j^{th} position and the j^{th} character of w_{i+1} follows from some legal transition from δ. Without loss of generality, we will assume that Γ contains the character $\#$ used for separation, and that δ defines no transitions for it. We also assume that N always has a transition from any accepting state back to an accepting state. Further, we assume that the tape is initially blank (WLOG, since there is a polynomial transformation from an NTM with input to one with a blank input). Let 0 be the default tape character.

Our encoding of this check will have an entirely propositional state language $\mathcal{L}$:

- There is a set of propositions for each state in S. Only one of these propositions will be true at any point in time, encoding the state of N for configuration w_i up until we check the character underneath the head, when we switch to the state for the next tape configuration, w_{i+1}.
- There is a set of propositions for each tape symbol in Γ. Only one will be true at a time.
- There are three pairs of propositions used to synchronize tasks: $\mathit{head\_step}$, $\mathit{head\_stepped}$, $\mathit{check\_step}$, $\mathit{check\_stepped}$, $\mathit{sync\_step}$, and $\mathit{sync\_stepped}$. Of each pair, at most one will be true at a time. We will describe how these are used shortly.

Let $\mathcal{O}$ be the set of propositional operators defined below:

- For each character $c \in \Gamma$ there are two operators: assert_c and check_c, where assert_c adds c to the state while retracting every other character in Γ, and check_c, which has c as precondition and no effects.
- For all transitions $(s', c', \text{Left}) \in \delta(s, c)$, there is an operator $\mathit{step\_s\_s'\_c\_l}$, which has a precondition of $\mathit{head} \wedge s \wedge c$ and has an effect of retracting k and asserting k'. $\mathit{step\_s\_s'\_c\_r}$ is defined similarly for transitions that move the head right.
- There is an operator done, which has a precondition of $\bigvee_{k \in F} k$ and no effects.
- There is an operator $\mathit{no\_head}$, which has $\neg \mathit{head}$ as a precondition and no effects, as well as operators $\mathit{assert\_head}$ and $\mathit{retract\_head}$ for asserting and retracting the head proposition.
- For the *head_step*/*head_stepped* propositions, we define four operators: *call_head* which has no precondition, asserts *head_step*; *start_head* has the precondition of *head_step* which it retracts; *respond_head* has no precondition and asserts *head_stepped*; and $\mathit{wait\_head}$ has *head_stepped* as its precondition which it retracts. Operators for the other *step*/*stepped* pairs are defined similarly.

Let $\mathcal{M}$ be the following set of methods:

- For the $\mathit{call\_head}/\mathit{wait\_head}$ operators, we introduce a method $(\mathit{step\_head}, \mathit{tn})$, where tn contains the totally-ordered tasks $\mathit{call\_head}$ and $\mathit{head\_wait}$. step tasks are defined similarly for the rest of the *call*/*wait* operators.
- For each $c \in \Gamma$, a method $(\mathit{produce}, \mathit{tn})$, where tn contains the totally ordered tasks: assert_c, $\mathit{step\_head}$, $2K \cdot \mathit{step\_check}$, $2K \cdot \mathit{step\_sync}$. We will define the consumers for the step tasks below. We also have two specialized versions of the method, $(\mathit{produce}_\#, \mathit{tn})$ and $(\mathit{produce}_0, \mathit{tn})$ specifically for asserting $\#$ and 0, but are otherwise the same.
- We add a method $(\mathit{produce}, \mathit{tn})$ where tn contains $\mathit{check\_done}$, which will only be applicable if a previous $\mathit{produce}$ gave a valid character that moved the machine into an accepting state.
- We add a method $(\mathit{config\_producers}, \mathit{tn})$ which contains the task $(K+1)\cdot\mathit{produce}$, which decomposes into enough $\mathit{produce}$ tasks to ensure that the configuration w_{i+1} is a valid successor of the current configuration w_i. We also add $(\mathit{config\_producers}_{\mathit{init}}, \mathit{tn}_{\mathit{init}})$ to encode the initial tape configuration. $\mathit{tn}_{\mathit{init}}$ has two ordered subtasks: $\mathit{produce}_\#$ and $K \cdot \mathit{produce}_0$.
- A method $(\mathit{head}, \mathit{tn})$ for asserting the head, where tn contains the tasks $\mathit{start\_head}$, $\mathit{assert\_head}$, and $\mathit{respond\_head}$. A method for the task $\mathit{no\_head}$ is defined similarly with $\mathit{retract\_head}$.
- A method $(\mathit{heads}, \mathit{tn})$, where tn contains the totally-ordered tasks $\mathit{no\_head}$, head, $(K+2) \cdot \mathit{no\_head}$. This will place the head on the second character and, by default, move it one to the right in every subsequent configuration of N. We will show how to adjust for this when checking the transitions.
- A method $(\mathit{wait}, \mathit{tn})$ where tn contains four totally-ordered tasks which sequentially call the operators $\mathit{start\_check}$, $\mathit{respond\_check}$, $\mathit{start\_sync}$, and $\mathit{respond\_sync}$. A wait task will then eat up one $\mathit{call\_check}$ followed by one $\mathit{call\_sync}$.
- For each $c \in \Gamma$, we have a method $(\mathit{check}, \mathit{tn})$ that, when the head is not currently present, will ensure that character $K+1$ chars later is identical. tn contains eight totally-ordered tasks: $\mathit{start\_check}$, $\mathit{no\_head}$, check_c, $\mathit{respond\_check}$, $K \cdot \mathit{wait}$, $\mathit{start\_check}$, check_c and $\mathit{respond\_check}$.

- For each transition $(s', c', \text{Left}) \in \delta(s, c)$, we have the task (check, tn), where tn sets the new state and ensures that the character $K+1$ chars later follows a legal transition. tn contains the following totally-ordered tasks: $\mathit{start\_check}$, $\mathit{step\_s\_s'\_c\_l}$, $2 \cdot \mathit{step\_head}$, $\mathit{respond\_check}$, $K \cdot \mathit{wait}$, $\mathit{start\_check}$, $\mathit{check}_{c'}$ and $\mathit{respond\_check}$. Notice how we call $\mathit{step\_head}$ twice to make the head move left (instead of moving to the right by default). We define check methods for Right moving transitions similarly, omitting the $2 \cdot \mathit{step\_head}$ subtask.
- We define a task $\mathit{config\_checks}(v_1, \ldots, v_k)$ with methods similar to the doubly-exponential counters of Section 4. $\mathit{config\_checks}(1, 0, \ldots, 0)$ will be responsible for launching K check tasks, starting in each cell of the configuration. Instead of the two subtasks of the oe counting methods, each method $(\mathit{config\_checks}, tn)$ has three tasks $t_1, t_2 \prec t_3$ (so t_1 is not ordered with respect to the rest), where $\alpha(t_1) = \alpha(t_3) = \mathit{config\_checks}'$, and $\alpha(t_2) = \mathit{wait}'$, where $\mathit{config\_checks}'$ and wait' are the appropriate decrement of $\mathit{config\_checks}$. We will add a method $(\mathit{config\_checks}_{+1}, tn)$ that launches exactly $K+1$ checks for each of the cells of the configuration.

Let $\mathcal{P} = (\mathcal{L}, \mathcal{O}, \mathcal{M}, s_I, tn_I)$ be an HTN problem with the above $\mathcal{L}$, $\mathcal{O}$, and $\mathcal{M}$. Let s contain the proposition for N's initial state q_0, and let tn_I contain the tasks $t_1 \prec t_2 \prec t_{done}$, t_3, t_4, and $t_5 \prec t_6$, where:

- $\alpha(t_1) = \mathit{config\_producers}_{init}$
- $\alpha(t_2) = K \cdot \mathit{config\_producers}$
- $\alpha(t_3) = (K+1) \cdot \mathit{heads}$
- $\alpha(t_4) = \alpha(t_6) = K \cdot \mathit{config\_checks}_{+1}$
- $\alpha(t_5) = (K+1) \cdot \mathit{wait}$
- $\alpha(t_{done}) = \mathit{done}$

t_1 and t_2 are the driving tasks of the problem, asserting one character in W at a time, and driving the rest of the tasks with $\mathit{call\_head}$ and $\mathit{call\_check}$. t_1 lays out the initial tape cells, with the separator first followed by K 0 cells. t_2 does K sequential copies of the unconstrained producer. The producer for each odd numbered w_i is validated by the check steps started by t_4, while the even numbered ones ($i > 0$) are validated by the check tasks from t_6, which were forced to wait one full configuration before starting by t_5. Once a producer validates a cell under the head of a configuration that leads to an accepting state, the $\mathit{check\_done}$ operator can be applied, and the rest of the tasks can short circuit via the done operator. Thus, P simultaneously generates and checks a witness W that N halts in 2^{2^k} steps.

Since $\mathcal{P}$ is solvable iff there is a run of N that terminates 2^{2^k}, partially-ordered lifted acyclic HTN planning is 2-NEXPTIME-complete. Replacing the top level tasks with tail-recursive tasks would encode a strictly space-bound NTM. The only variables used in the encoding were for counting K repetitions of tasks. Replacing K with a merely-exponential bound would let us encode N in a fully propositional problem. Using combinations of either of these modifications (tail-recursive top level tasks or using $K = 2^k$) gives the remaining lower bounds of the theorem. □

7 Conclusions

We proposed a straight-forward extension for propositional HTN planning to a lifted representation that is based upon a function-free first-order logic. We studied how the variables/constants, the (partial) order of tasks, and various variants of recursion interact w.r.t. the complexity of the plan existence problem. Our results have straight-forward implications for other hierarchical planning formalisms, such as *hierarchical goal networks* (HGNs), that have a direct correspondence with HTN planning (Shivashankar et al. 2012), and for *hybrid planning*, a framework that fuses HTN planning with partial-order causal-link (POCL) planning (Biundo and Schattenberg 2001).

Apart from giving deeper theoretical insights of the complexity and expressiveness of HTN planning, our work also has implications on the design of future HTN algorithms. For example, the TOPHTN algorithm from Alford et al. (2012) uses a mixture of progression and problem-decomposition, and is able to decide every totally-ordered propositional problem in EXPTIME. However, it also takes exponential space. The progression-based algorithm from the same paper (PHTN) can be made space efficient on the tail-recursive subset of these problems, taking only polynomial space. This leaves an obvious gap in the literature that could be filled with an algorithm that can decide both the classes of problems efficiently.

Some HTN problems can also be solved via compiling them into non-hierarchical planning problems. Alford, Kuter, and Nau (2009) describe such a translation for totally-ordered tail-recursive problems. It should be straight-forward to extend this technique to tail-recursive problems with arbitrary ordering. However, our results show that any such translation for partially-ordered problems must yield an exponential blow-up in the general case.

In future work, we want to extend our results to provide tight complexity bounds for hierarchical planning with task insertion (TIHTN planning) – a variant of hierarchical planning that allows to insert tasks into task networks without the need to decompose abstract tasks (Geier and Bercher 2011; Shivashankar et al. 2013). Other interesting problems include extending the NP-completeness results for both propositional acyclic regular HTN problems (Alford et al. 2014) and for HTN plan verification (Behnke, Höller, and Biundo 2015) into our lifted HTN formalism.

Acknowledgment This work is sponsored in part by OSD ASD (R&E) and by the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems" funded by the German Research Foundation (DFG). The information in this paper does not necessarily reflect the position or policy of the sponsors, and no official endorsement should be inferred.

References

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN problem spaces: Structure, algorithms, termination. In *Proceedings of the 5th Annual Symposium on Combinatorial Search (SoCS)*, 2–9. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2014. On the feasibility of planning graph style heuristics for HTN planning. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, 2–10. AAAI Press.

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 1629–1634. AAAI Press.

Alur, R.; Kannan, S.; and Yannakakis, M. 1999. Communicating hierarchical state machines. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, 169–178.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling*. AAAI Press.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief (a preliminary report on combining state abstraction and HTN planning). In *Proceedings of the 6th European Conference on Planning (ECP)*, 157–168. AAAI Press.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 94(1-2):165–204.

Chandra, A. K.; Kozen, D. C.; and Stockmeyer, L. J. 1981. Alternation. *Journal of the ACM* 28(1):114–133.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, volume 94, 1123–1128. AAAI Press.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1991. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. *Artificial Intelligence* 76:75–88.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* 76(1):75–88.

Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity of UCPOP with the efficiency of graphplan. In *Proceedings of the 4th European Conference on Planning: Recent Advances in AI Planning (ECP)*, 221–233. Springer-Verlag.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 1955–1961. AAAI Press.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning: theory & practice*. Morgan Kaufmann.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, 447–452. IOS Press.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 2, 981–988. International Foundation for Autonomous Agents and Multiagent Systems.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL planning system: a more perfect union of domain-independent and hierarchical planning. In *Proceedings of the Twenty-Third international Joint Conference on Artificial Intelligence (IJCAI)*, 2380–2386. AAAI Press.