Abstract
Hierarchical Task Network (HTN) planning with Task Insertion (TIHTN planning) is a formalism that hybridizes classical planning with HTN planning by allowing the insertion of operators from outside the method hierarchy. This additional capability has some practical benefits, such as allowing more flexibility for design choices of HTN models: the task hierarchy may be specified only partially, since “missing required tasks” may be inserted during planning rather than prior planning by means of the (predefined) HTN methods.

While task insertion in a hierarchical planning setting has already been applied in practice, its theoretical properties have not been studied in detail, yet – only EXPSPACE membership is known so far. We lower that bound proving NEXPTIME-completeness and further prove tight complexity bounds along two axes: whether variables are allowed in method and action schemas, and whether methods must be totally ordered. We also introduce a new planning technique called acyclic progression, which we use to define provably efficient TIHTN planning algorithms.

1 Introduction
Hierarchical Task Network (HTN) planning [Erol et al., 1996] is a planning approach, where solutions are generated via step-wise refinement of an initial task network in a top-down manner. Task networks may contain both compound and primitive tasks. Whereas primitive tasks correspond to standard STRIPS-like actions that can be applied in states where their preconditions are met, compound tasks are abstractions thereof. That is, for every compound task, the domain features a set of decomposition methods, each mapping that task to a task network. Solutions are obtained by applying methods to compound tasks thereby replacing these tasks by the task network of the respective method. HTN planning is strictly more expressive than non-hierarchical (i.e., classical) planning. The complexity of the plan existence problem ranges up to undecidable even for propositional HTN planning [Erol et al., 1996; Geier and Bercher, 2011; Alford et al., 2015]. Even the verification of HTN solutions is harder than in classical planning [Behnke et al., 2015].

Hierarchical planning approaches are often chosen for real-world application scenarios [Nau et al., 2005; Lin et al., 2008; Biundo et al., 2011] due to the ability to specify solution strategies in terms of methods, but also because human expert knowledge is often structured in a hierarchical way and can thus be smoothly integrated into HTN planning models. On the other side, these methods also make HTN planning less flexible than non-hierarchical approaches, because only those solutions may be generated that are “reachable” via the decomposition of the methods. So, defining only a partially hierarchical domain is not sufficient to produce all desired solutions. Several HTN researchers have thus investigated how partially hierarchical domain knowledge can be exploited during planning without relying on the restricted (standard) HTN formalism [Kambhampati et al., 1998; Biundo and Schattenberg, 2001; Alford et al., 2009; Geier and Bercher, 2011; Shivashankar et al., 2013].

The most natural way to overcome that restriction is to allow both the definition and decomposition of compound tasks and the insertion of tasks from outside the decomposition hierarchy as it is done, for example, by Kambhampati et al. (1998) and Biundo and Schattenberg (2001) in Hybrid Planning – a planning approach fusing HTN planning with Partial-Order Causal-Link (POCL) planning.

This additional flexibility also pays off in terms of computational complexity: allowing arbitrary insertion of tasks into task networks lowers the complexity of the plan existence problem from undecidable (for standard HTN planning) to
EXPSPACE membership (for HTN planning with task insertion – TIHTN planning) [Geier and Bercher, 2011]. This reduction of the complexity also has its negative consequences, however. Some planning problems that can be easily expressed in the HTN planning setting may not be expressed in the TIHTN setting (others only with a blowup of the problem size) [Höller et al., 2014]. Also, the set of TIHTN solutions of a problem may not correspond to the ones the domain modeler intended: in HTN planning – as opposed to TIHTN planning – only those plans are regarded solutions that can be obtained by decomposition only. Thus, certain plans may be ruled out even if they are executable.

In this paper, we investigate the influence of method structure (partially vs. totally ordered) and of variables (propositional vs. lifted TIHTN problems), and provide tight complexity bounds of the plan existence problem for the four resulting classes of TIHTN problems. The results are summarized in Table 1 (and compared to the respective results from HTN planning in Table 2 in the last section). Notably, we show that propositional TIHTN planning is NEXPTIME-complete, easier than the previously known EXPSPACE upper bound. Besides providing tight complexity bounds for the plan existence problem, another contribution is a new algorithm, called acyclic progression, that efficiently solves TIHTN problems. The paper closes with a discussion about the new complexity findings for TIHTN planning that puts them into context of already known results for HTN planning.

2 Lifted HTN Planning with Task Insertion

Geier and Bercher (2011) defined a propositional (set-theoretic) formalization of HTN and TIHTN planning problems. Both problem classes are syntactically identical – they differ only in the solution criteria. Recently, we extended their formalization of HTN planning to a lifted representation based upon a function-free first order language [Alford et al., 2015], where the semantics are given via grounding. For the purpose of this paper, we replicate the definitions for lifted HTN planning and extend them by task insertion to allow specifying lifted TIHTN planning problems.

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 $T$.
- $\alpha : T \rightarrow X$ maps from task symbols to task names.

Since task networks are only partially ordered and any task name might be required to occur several times, we need a way to “identify” them uniquely. For that purpose, we use the task symbols $T$ as unique place holders. A task network is called ground if all task names occurring in it are variable-free.

A lifted TIHTN problem is a tuple $(L, O, M, s_1, tn_I)$, where:

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

We define the semantics of lifted TIHTN planning through grounding. For the details of the grounding process, we refer to [Alford et al., 2015]. The ground (or propositional) TIHTN planning problem obtained from $(L, O, M, s_1, tn_I)$ is given by $P = (L, O, M, s_1, tn_I)$.

The operators $O$ form an implicit state-transition function $\gamma : 2^L \times O \rightarrow 2^{s\oplus O}$ for the problem, where:

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

### Executability

A sequence of ground operators $(o_1, \ldots, o_n)$ is executable in a state $s_0$ if there exists a sequence of states $s_1, \ldots, s_n$ such that $\forall 1 \leq i \leq n \gamma(s_i, o_i) = s_i$. A ground task network $tn = (T, \prec, \alpha)$ is primitive iff it contains only task names from $O$. $tn$ 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 $(\alpha(t_1), \ldots, \alpha(t_n))$ is executable in $s_0$.

### Task Decomposition

Primitive task networks can only be obtained by refining the initial task network via decomposition of compound tasks. 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, let $tn = (T, \prec, \alpha)$ be a task network and let $m = (\alpha(t), (T_m, \prec_m, \alpha_m)) \in M$ be a method. Without loss of generality, assume $T \cap T_m = \emptyset$. Then the decomposition of $t$ in $tn$ by $m$ into a task network $tn'$, written $tn \xrightarrow{t,m} \gamma_d \rightarrow tn'$, is given by:

$$T' := (T \setminus \{t\}) \cup T_m$$

$$\prec' := \{ (t, t') \in \prec \mid t, t' \in T' \} \cup \prec_m$$

$$\cup \{(t_1, t_2) \in T_m \times T \mid (t_1, t_2) \in \prec \}$$

$$\cup \{(t_1, t_2) \in T \times T_m \mid (t_1, t) \in \prec \}$$

$$\alpha' := \{ (t, n) \in \alpha \mid t \in T' \} \cup \alpha_m$$

$$tn' := (T', \prec', \alpha')$$

If $tn'$ is reachable by any finite sequence of decompositions of $tn$, we write $tn \xrightarrow{\gamma_d} \rightarrow tn'$.

### Task Insertion

TIHTN planning, in addition to decomposition, allows tasks to be inserted directly into task networks. Let $tn = (T, \prec, \alpha)$ be a task network, $t$ be a fresh task symbol not in $T$, and $o$ be a primitive task name. Then task
insertion, written \(tn \xrightarrow{\alpha} tn'\), results in the task network \(tn' = (T \cup \{t\}, \prec, \alpha \cup \{(t, o)\})\). If \(tn'\) is reachable by any sequence of insertions to \(tn\), we write \(tn \rightarrow^*_I tn'\).

Task insertion commutes with decomposition, i.e., if \(tn_i \xrightarrow{m \gamma} tn_2 \xrightarrow{\alpha} tn_3\), then there exists a \(tn'_2\) such that \(tn_i \xrightarrow{m \gamma}\ x \rightarrow^*_D tn_2 \xrightarrow{\alpha} tn_3\). If \(tn'\) is reachable by any sequence of decompositions and insertions to \(tn\), we write \(tn \rightarrow^*_D tn'\).

**Solutions** Under HTN semantics, a problem \(P\) is solvable iff \(tn_1 \rightarrow^*_I tn'\) and \(tn'\) is executable in \(s_1\). The task network \(tn'\) is then called an HTN solution of \(P\). Under TIHTN semantics, \(P\) is solvable iff there exists a \(tn'\) such that \(tn_1 \rightarrow^*_I tn'\) and \(tn'\) is executable in \(s_1\).

The following definitions go beyond those in [Alford et al., 2015; Geier and Bercher, 2011].

**Acyclic Decompositions** Let \(\overline{tn}\) be a task network sequence starting in \(tn_1\) and ending in a task network \(tn'\), s.t. \(tn_1 \rightarrow^* D tn'\). For each network \(tn_i \in \overline{tn}\), every task symbol was either present in \(tn_i\), inserted directly via task insertion, or is the result of a sequence of decompositions of a task symbol in \(tn_i\). For the tasks arrived at by decomposition, we can define their ancestors in the usual way: For \(tn_i, tn_{i+1} \in \overline{tn}\) with \(tn_i \xrightarrow{m \gamma} tn_{i+1}\), \(t_i\) is an ancestor of each task \(t' \in \overline{tn}_{i+1}\) that comes from \(tn_i\); and ancestry is transitive, i.e., if \(t_i\) is an ancestor of \(t_j\) and \(t_j\) is an ancestor of \(t_k\), then \(t_i\) is an ancestor of \(t_k\). \(\overline{tn}\) is acyclic if for every task \(t\) its final task network, the ancestors of \(t\) all have unique task names. Thus, an acyclic series of decompositions and insertions \(tn_t \rightarrow^*_D tn'\) makes no use of recursive methods, regardless of their presence in the set of methods.

Geier and Bercher (2011) represent ancestry using decomposition trees and show that, given a TIHTN solution \(tn\) that is obtained via cyclic method application, one can repeatedly remove cycles (replacing orphaned sequences of decomposition with task insertion) to arrive at an acyclic solution \(tn'\).

The next corollary follows as a special case of the application of Lemma 1 and Lemma 2 by Geier and Bercher (2011).

**Corollary 1.** Let \(P = (\mathcal{L}, O, M, s_f, tn_1)\) be a ground planning problem and let \(tn = (\langle T, \prec, \alpha \rangle, \emptyset)\) be a task network such that \(tn_1 \rightarrow^*_D tn\) and \((t_1, \ldots, t_k)\) is an executable task sequence of \(tn\) that does not violate the ordering constraints \(\prec\). Then there exists a task network \(tn' = (\langle T', \prec', \alpha' \rangle, \emptyset)\) with an acyclic decomposition \(tn \rightarrow^*_D tn'\) and an executable task sequence \((t'_1, \ldots, t'_k)\) of \(tn'\) not violating \(\alpha'\), such that \(\forall_i \alpha(t_i) = \alpha'(t'_i)\).

**3 Acyclic Progression for TIHTN Planning**

HTN planners generally solve problems either using decomposition directly [Erol et al., 1994; Bercher et al., 2014], or by using progression [Nau et al., 2003], which interleaves decomposition and finding a total executable order over the primitive tasks [Alford et al., 2012]. Since progression-based HTN algorithms can be efficient across a number of syntactically identifiable classes of HTN problems [Alford et al., 2015], it makes a useful starting point for designing efficient TIHTN algorithms.

We define acyclic progression for TIHTN planning as a procedure that performs progression on a current state. To that end, it maintains a task network of primitive and compound tasks that still need to be applied to the current state or to be decomposed, respectively. To avoid recursive definitions, it maintains a set of ancestral task names. More precisely, search nodes are tuples \((s, tn, h)\), where \(s\) is a state, \(tn = (\langle T, \prec, \alpha \rangle, \emptyset)\) is a task network, and \(h\) is a mapping of the tasks in \(T\) to the set of ancestral task names, represented as a set of task-task-name pairs. For each node, there are three possible operations:

- **Task insertion:** If \(o\) is an operator such that \(s \models prec(o)\), then \((\gamma(s, o), tn, h)\) is an acyclic progression of \((s, tn, h)\)
- **Task application:** If \(t \in T\) is an unconstrained primitive task \((\forall t' \neq t'') t \not\prec t''\) and \(s \models prec(o(t))\), then we can apply \(o(t)\) to \(s\) and remove it from \(tn\) and \(h\). So, given:

\[
\begin{align*}
T' &:= T \setminus \{t\} \\
\prec' &:= \{(t_1, t_2) \in \prec \mid t_1 \neq t \land t_2 \neq t\} \\
\alpha' &:= \{(t', n) \in \alpha \mid t' \neq t\} \\
\gamma' &:= (T', \prec', \alpha') \\
h' &:= \{(t', n) \in h \mid t' \neq t\}
\end{align*}
\]

then \((\gamma(s, o(t)), tn', h')\) is an acyclic progression of \((s, tn, h)\).

- **Task decomposition:** If \(t \in T\) is an unconstrained non-primitive task, \((\alpha(t), tn_0)\in M\) is method with \(tn_m = (T_m, \prec_m, \alpha_m)\), and none of the task names in \(\alpha\) occur as an ancestral task name of \(t\) in \(h\), then we can decompose \(t\) and update the history. So if \(tn \xrightarrow{\alpha} tn'\) and \(h' := \{(t', n) \in h \mid t' \neq t\}
\]
\[
\cup \{(t_m, n) \mid t_m \in T_m, (t, n) \in h\}
\]
\[
\cup \{(t_m, o(t)) \mid t_m \in T_m\}
\]

then \((s, tn', h')\) is an acyclic progression of \((s, tn, h)\).

If there is a sequence of acyclic progressions from the triple \((s, tn, h)\) to \((s', tn', h')\), we write \((s, tn, h) \rightarrow^*_P (s', tn', h')\). Notably, acyclic progression is only acyclic over decompositions, not states reached. If there is any sequence of acyclic decompositions to an empty task network, then the problem has a TIHTN solution.

**Lemma 2.** Given a ground planning problem \(P = (\mathcal{L}, O, M, s_f, tn_1)\), there is a series of acyclic progressions \((s_1, tn_1, \emptyset) \rightarrow^*_P (s, tn_0, \emptyset)\) (where \(tn_0\) is the empty task network) if and only if \(P\) is solvable under TIHTN semantics.

**Proof.** \((\Rightarrow)\) Let \(TD\) be the subsequence of the acyclic progression \((s_1, tn_1, \emptyset) \rightarrow^*_P (s, tn_0, \emptyset)\) containing all the decompositions performed, \(TA\) be the subsequence of task applications, \(TI\) be the subsequence of task insertions, and \(TIA\) be the subsequence of both task applications and insertions.

Since task application only removes primitive tasks, \(TD\) must be a decomposition of \(tn_1\) to a primitive task network \(tn'\), specifically one with a partial order \(\prec'\) which is consistent with the order and content of task applications, \(TA\). Then, given that task insertion commutes with decomposition, \(TI\) gives us a set of insertions \(tn' \rightarrow^*_I tn''\), and \(TIA\) is a witness that \(tn''\) has an executable ordering.
If \( P \) is solvable, by Corollary 1 there is an acyclic decomposition sequence \( t_n \) such that \( t_n \rightarrow t_{n+1} \), and a sequence of insertions \( t_n \rightarrow t'_{n+1} \) such that \( t'_{n+1} = (T', \prec', \alpha') \) has an executable ordering \( TIA = (t_1, \ldots, t_k) \). Insertions commute with both themselves and with decomposition, and decompositions commute with each other so long as one decomposed task is not an ancestor of the other. So the following procedure gives an acyclic progression of \( P \) to the empty network. Given that \( t_i \) is the last task from \( TIA \) to be applied to the state (by insertion or application) and the current triple under progression is \((s_j, t_{n_j}, h_j)\):

- If there is a non-primitive task \( t_j \) in \( t_{n_j} \) such that \( t_j \) is an ancestor of \( t_{i+1} \) in the sequence \( t_{n_j} \), use the decomposition from \( t_{n_j} \) to decompose \( t_j \).
- If \( t_{i+1} \) exists in \( t_{n_j} \), progress it out of the task network with task application and move on to \( t_{i+2} \).
- Else, \( t_{i+1} \) was obtained by insertion, and so use insertion to apply \( \alpha' (t_{i+1}) \) to \( s_j \) and move on to \( t_{i+1} \).

A problem’s **acyclic progression bound** is the size of the largest task network reachable via any sequence of acyclic progressions. Only decomposition can increase the task network size. Since decomposition is required to be acyclic, every problem has a finite acyclic progression bound. Given the tree-like structure of decomposition, if \( m \) is the max number of subtasks in any method and \( n \) is the number of task names, and \( T \) is the set of task symbols in the initial task network, then \(|T| \cdot m^n\) is an acyclic progression bound of the problem.

If methods are totally ordered (i.e., the \( \prec \) relation in each method is a total order), then we reach a much tighter bound:

**Theorem 3.** Propositional TIHTN planning for problems with totally-ordered methods is PSPACE-complete.

**Proof.** Classical planning provides a PSPACE lower bound [Geier and Bercher, 2011]. Here, we provide an upper bound.

Let \( P = (L, O, M, s_j, t_{n_j}) \) be a ground TIHTN problem where each method is totally ordered. The initial task network \( t_{n_j} \) may be partially ordered. Letting \( t_{n_j} = (T, \prec, \alpha) \), there are \(|T| \) initial tasks. Since the methods are totally ordered, any sequence of progressions \((s_j, t_{n_j}, 0) \rightarrow t_{n+1} \) preserves that \( t_n \) can be described by the relationship of \(|T|\) or fewer totally ordered chains of tasks.

Progression can only affect the unconstrained tasks in the chains. While the acyclic decomposition phase of progression can lengthen a chain by \( m - 1 \) tasks (\( m \) being the size of the largest method), each of those tasks has a strictly larger set of ancestral task names, and the size of that set is capped. If \( n \) is the number of ground task names, each chain can only grow to a length of \( n \cdot (m - 1) \). So the acyclic progression bound of problems with totally ordered methods is \(|T| \cdot n \cdot (m - 1)\).

Since the size of any state is also polynomial, totally ordered proposition TIHTN plan existence is PSPACE-complete.

**Theorem 4.** TIHTN planning is EXPSPACE-complete for lifted problems with totally-ordered methods.

**Proof.** Grounding a lifted domain produces a worst-case exponential increase in the number of task names providing EXPSPACE membership. Lifted classical planning provides the EXPSPACE lower bound [Erol et al., 1995].

**4 2^k Bottles of Beer on the Wall**

At most \( 2^k \) tasks need to be inserted between each pair of two consecutive primitive tasks in any acyclic decomposition [Geier and Bercher, 2011]. This provides an upper bound on the number of task insertions needed to show a TIHTN problem is solvable. The song “m Bottles of Beer on the Wall” uses a decimal counter to encode a bound on the number of refrains in space logarithmic to \( m \) [Knuth, 1977]. Much like this, we will give transformations of TIHTN operators so that a binary counter ensures strict upper bounds on the number of primitive tasks in any solution. This will limit the length of any sequence of progressions, giving us upper complexity bounds for TIHTN planning.

Clearly, we could just limit acyclic progression to a bounded depth instead. However we will also use the bounding transformation below in the following section to provide a polynomial transformation of acyclic HTN problems which preserves solvability under TIHTN semantics.

Let \( P \) be a propositional problem with a set of operators \( O \) and language \( L \). Given a bound of the form \( 2^k \), we create a language \( L' \) which contains \( L \) and the following propositions: \( counting, count_init, \) and \( counter_i \) and \( decrement_i \) for \( i \in \{1 \ldots k + 1\} \). We define the operator set \( O' \) to be each operator in \( O \) with the added the precondition \( \neg counting \) and the additional effect \( counting \land decrement_i \). We define another set of operators \( O_{count} \) with the following operators:

- \( (count_{init}_{op}, \neg count_{init}, count_{init} \land counter_{k}) \), initializing the counter to the value \( 2^k \).
- \( (decrement_i \_0\_op, pre, eff) \) for \( i \in \{1 \ldots k\} \), where:
  \( pre := count_{init} \land decrement_i \land \neg counter_i \land\)
  \( eff := \neg decrement_i \land decrement_{i+1} \land counter_{i+1} \)
setting the \( i \)th bit of the counter to 1 if it was zero and moves on to the next bit.
- \( (decrement_i \_1\_op, pre, eff) \) for \( i \in \{1 \ldots k\} \), where:
  \( pre := count_{init} \land decrement_i \land counter_{i} \land\)
  \( eff := \neg decrement_i \land \neg counting \land \neg counter_i \land\)
setting the \( i \)th bit of the counter to 0 and stops counting.

Then in the problem \( P' \) with operators \( O' \cup O_{count} \) and the language \( L' \), any executable sequence consists of an alternating pattern of an operator from \( O' \) followed by a sequence of counting operators from \( O_{count} \). Since there is no operator to decrement \( counter_{k+1} \), we can only apply \( 2^k \) operators from \( O' \) to the state. Notice that by setting the appropriate \( counter_i \) propositions in the \( count_{init}_op \) operator, we could have expressed any bound between 0 and \( 2^{k+1} - 1 \).

We can extend this to doubly-exponential bounds for lifted problems. Let \( P \) be a lifted problem with language \( L \) and operators \( O \), and let \( 2^{2^k} \) be our bound on operators from \( O \) to encode. Let \( L' \) contain \( L \) and the following predicates: \( counting(), count_{init}(), \) and \( counter(v_1, \ldots, v_k) \)
and decrement \((v_1, \ldots, v_k)\), and let 0, 1 be arbitrary distinct constants in \(L'\).

As with the counter; predicates, the ground counter\((\ldots)\) predicates will express a binary counter in the state, with a binary index (the variables) into the exponential number of bits. Let \(cr_1, \ldots, cr_k\) be predicates such that each \(cr_i\) has the form counter \((v_k, \ldots, v_{i+1}, 0, 1, \ldots, 1)\) and let dec \(1, \ldots, dec_k\) be predicates such that each \(dec_i\) has the form decrement \((v_k, 0, 1, \ldots, 1)\) where each \(v_m\) is a variable. So:

- \(dec_1 = \text{decrement} \left(v_k, \ldots, v_2, 0\right)\)
- \(dec_2 = \text{decrement} \left(v_k, \ldots, v_3, 0, 1\right)\)
- \(dec_{k-1} = \text{decrement} \left(v_k, 0, 1, \ldots, 1\right)\)
- \(dec_k = \text{decrement} \left(0, 1, \ldots, 1\right)\)

Similarly, let \(dec'_1, \ldots, dec'_k\) be predicates of the form decrement \((v_k, \ldots, v_{i+1}, 1, 0, \ldots, 0)\), so:

- \(dec'_1 = \text{decrement} \left(v_k, \ldots, v_2, 1\right)\)
- \(dec'_2 = \text{decrement} \left(v_k, \ldots, v_3, 1, 0\right)\)
- \(dec'_{k-1} = \text{decrement} \left(v_k, 1, 0, \ldots, 0\right)\)
- \(dec'_k = \text{decrement} \left(1, 0, \ldots, 0\right)\)

So if \(v\) is an assignment of \(v_1, \ldots, v_k\) to \(\{0, 1\}\) and we view the proposition \(\text{dec}_i \left[v\right]\) as an instruction to decrement the \(i\)th bit of the counter, then \(\text{dec}'_i \left[v\right]\) is for decrementing the bit with index \(j + 1\).

Let \(O'\) consist of each operator in \(O\) with the added the precondition \(\neg \text{counting}()\) and the additional effect \(\text{counting}'() \land \text{decrement}(0, \ldots, 0)\). We define \(O_{\text{count}}\) to be the following operators:

- \((\text{count}_i\_\text{init}\_\text{op}(), \text{pre}, \text{eff}),\) where:
  \[
  \text{pre} := \neg \text{count}_i() \\
  \text{eff} := \text{count}_i() \land \text{counter}(1, 0, \ldots, 0)
  \]
  which initializes the counter to the value \(2^k\).

- \((\text{decrement}_i\_\text{start}\_\text{op}(), \text{pre}, \text{eff})\) for \(i \in \{1..k\}\), where:
  \[
  \text{pre} := \text{count}_i() \land \text{dec}_i \land \neg \text{cr}_i \\
  \text{eff} := \neg \text{dec}_i \land \text{dec}_{i+1} \land \text{cr}_i
  \]
  which, if \(v\) is an assignment of \(v_1, \ldots, v_k\) to \(\{0, 1\}\), sets \(\text{cr}_i \left[v\right]\) to 1 if it was zero and moves on to the next bit.

- \((\text{decrement}_i\_\text{end}\_\text{op}(), \text{pre}, \text{eff})\) for \(i \in \{1..k\}\), where:
  \[
  \text{pre} := \text{count}_i() \land \text{dec}_i \land \text{cr}_i \\
  \text{eff} := \neg \text{dec}_i \land \neg \text{counting}() \land \neg \text{cr}_i
  \]
  which, if \(v\) is an assignment of \(v_1, \ldots, v_k\) to \(\{0, 1\}\), sets \(\text{cr}_i \left[v\right]\) to 0 and stops counting.

So after decrement \((0, 0, \ldots, 0)\) is set by an operator (and count\(\_i\_\text{init}\_\text{op}\) has been applied in the past), the only legal sequence of operators involves stepping sequentially through the \(2^i\) possible counter \((\ldots)\) predicates until the decrement operation is finished.

Similar to the propositional transformation, we can start the counter at some number which is a polynomial sum of \(2^i\) for \(i \in \{1..2^k\}\). These two transformations are the dual of the counting tasks in Theorems 4.1 and 4.2 from Alford et al. (2015). Where the counting tasks gave methods that enforced exactly \(2^c\) and \(2^k\) repetitions of a given task be in any solution, this transformation ensures that there are no more than the specified number of primitive tasks.

**Theorem 5.** Propositional TIHTN planning is in NEXPTIME; lifted TIHTN planning is in 2-NEXPTIME.

**Proof.** Use the appropriate transformation from above to limit the number of primitive operators in any solution to \([T] \cdot m^n \cdot 2^c\), where \([T]\) is the number of tasks in the initial network, \(m\) is the max method size, \(n\) is the number of non-primitive task names in the grounded problem (exponential for lifted problems), and \(2^c\) is the total number of unique states expressible by \(L\). This ensures every sequence of acyclic progressions ends after an exponential number of steps for propositional problems and a doubly-exponential number of steps for lifted problems. Thus, a depth-first non-deterministic application of acyclic progression until it reaches a solution or can progress no more is enough to prove the existence of a TIHTN solution.

5 Acyclic HTN Planning with TIHTN Planners

Theorem 5 provides upper bounds for TIHTN planning. Section 2 describes acyclic decomposition. An acyclic problem is one in which every sequence of decompositions is acyclic [Erol et al., 1996; Alford et al., 2012]. HTN planning for propositional partially ordered acyclic problems is NEXPTIME-complete and 2-NEXPTIME-complete for lifted partially ordered acyclic problems.

Theorem 6.1 of [Alford et al., 2015] encodes NEXPTIME- and 2-NEXPTIME-bounded Turing machines almost entirely within the task network of propositional and lifted acyclic HTN planning, respectively. Of particular interest here, though, is that, for a given time bound and Turing machine, every primitive decomposition of the initial task network in these encodings has exactly the same number of primitive tasks. This lets us use the bounding transformation from Section 4 to prevent rogue task insertion under TIHTN semantics:

**Theorem 6.** Propositional TIHTN planning is NEXPTIME-hard; lifted TIHTN planning is 2-NEXPTIME-hard.

**Proof.** Let \(N\) be a nondeterministic Turing machine (NTM), let \(K = 2^2^c\) be the time bound for \(N\), and let \(P\) with operators \(O\) be the encoding of \(N\) as a lifted acyclic HTN problem.

One can calculate exactly how many primitive tasks are in any decomposition of \(P\), but it is roughly of the form \(B = c \cdot K^2 + d \cdot K + 1\) for constants \(c\) and \(d\), which we can express as a polynomial sum of \(2^c\). Let \(P'\) be the \(B\)-task-bounded transformation of \(P\).

From the hardness proof of Theorem 6.1 of [Alford et al., 2015], we know: if \(N\) can be in an accepting state after \(K\) steps, there is an executable primitive decomposition of \(P\) with \(B\) tasks from \(O\), and so \(P'\) has a TIHTN solution.

Let \(tn\) be a non-executable primitive decomposition of the initial task network, \(tn_1 \rightarrow_D tn\). Since the bounding transformation does not affect the methods, this decomposition
sequence is also legal in $P'$ (with operators $O' \cup O_{\text{count}}$). Since any insertions of operators from $O'$ would put $tn$ over the limit $B$, no sequence of insertions can make this task network executable in $P'$. Thus if no run of $N$ is in an accepting state after $K$ steps, no primitive decomposition of $P$ is executable, and there is no TIHTN solution to $P'$.

Since $P'$ has a TIHTN solution iff $P$ has a solution, and $P$ encodes a $2^k$-bounded NTM, lifted TIHTN planning is $2$-NEXPTIME-hard.

The proof is the same for propositional TIHTN problems using the propositional encoding of $2^k$ time-bound NTMs into acyclic HTN problems, so propositional acyclic TIHTN planning is NEXPTIME-hard.

As a corollary, we obtain NEXPTIME and 2-NEXPTIME completeness for propositional and lifted TIHTN planning, respectively.

### 6 A comparison with HTN complexity classes

Based on the recursion structure classification for HTN problems [Alford et al., 2012], we now have an extensive classification of HTN problems by structure and complexity:

- **Acyclic** problems, discussed earlier, where every decomposition is guaranteed to be acyclic.
- **Tail-recursive** problems, where methods can only recurse through their last task. All acyclic problems are also tail-recursive.
- **Arbitrary-recursive** problems, which includes all HTN problems.

However, by Corollary 1, non-acyclic (i.e., cyclic) decompositions can be ignored, limiting the impact of recursion structure on the complexity of TIHTN planning.

This is not to say that method structure (outside of ordering) has no effect on the complexity of TIHTN planning. For instance, regular TIHTN problems (defined by Erol et al. [1996] for HTN planning) with a partially ordered initial task network and partially ordered methods are easier than non-regular (partially ordered) problems. Regular problems are a special case of tail-recursive problems, where every method is constrained to have at most one non-primitive task in the method's network, and that task must be constrained to come after all the primitive tasks. Regular problems have a linear progression bound regardless of whether the primitive tasks are totally-ordered amongst themselves. Since acyclic progression is a special case of progression, regular problems have linear acyclic progression bounds, and thus partially ordered regular problems have the same complexity under TIHTN semantics as totally-ordered regular problems:

**Corollary 7.** **TIHTN plan-existence for regular problems** is PSPACE-complete when they are propositional, and EXPSPACE-complete otherwise, regardless of ordering.

There are also times when HTN planning is simpler than TIHTN planning. Alford et al. (2014) show that HTN planning for propositional regular problems which are also acyclic is NP-complete. Since an empty set of methods and a single primitive task in the initial network is enough to encode classical planning problems under TIHTN semantics, acyclic-regular problems are still PSPACE-hard for propositional domains and EXPSPACE-hard when lifted.

So, while TIHTN planning is not always easier than HTN planning, we have shown that its complexity hierarchy is, in general, both simpler and less sensitive to method structures. In future work, we want to investigate the plan existence problem along the further axis of syntactic restrictions on the task hierarchy and methods: Alford et al. (2015) defines a new restriction on HTN problems, that of constant-free methods, that forbids mixing constants and variables in task names appearing in methods. This significantly reduces the progression bound for lifted partially ordered acyclic and tail recursive problems, and thus may also impact the complexity of those problems under TIHTN semantics.

### 7 Conclusions

We studied the plan existence problem for TIHTN planning, a hierarchical planning framework that allows more flexibility than standard HTN planning. The complexity varies from PSPACE-complete for the totally ordered propositional setting to 2-NEXPTIME-complete for TIHTN planning where variables are allowed and the methods’ task networks may be only partially ordered.

We showed that totally ordered TIHTN planning has the same plan existence complexity as classical planning (both with and without variables). Given that plan existence for both delete-relaxed propositional TIHTN and classical problems is in polynomial time [Alford et al., 2014], we hope that
many of the algorithms and heuristics developed for classical planning can be adapted for totally-ordered TIHTN problems.

We also provided a new planning technique for TIHTN planning, called acyclic progression, that let us define provably efficient TIHTN planning algorithms. We hope it inspires the creation of future planners that are both provably and empirically efficient.

Acknowledgment This work is sponsored in part by OSD ASD and 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


