Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problems

Ron Alford[1], Gregor Behnke[2], Daniel Höller[2], Pascal Bercher[2], Susanne Biundo[2], David W. Aha[3]
[1]ASEE Postdoctoral Fellow; Naval Research Laboratory; Washington, DC; USA
[2]Institute of Artificial Intelligence; Ulm University; Ulm; Germany
[3]Navy Center for Applied Research in AI; Naval Research Laboratory; Washington, DC; USA
ronwalf@volus.net | {*first.last*}@uni-ulm.de | david.aha@nrl.navy.mil

Abstract

Hierarchical Task Network (HTN) planning is a formalism that can express constraints which cannot easily be expressed by classical (non-hierarchical) planning approaches. It enables reasoning about procedural structures and domain-specific search control knowledge. Yet the cornucopia of modern heuristic search techniques remains largely unincorporated in current HTN planners, in part because it is not clear how to estimate the goal distance for a partially-ordered task network. When using SHOP2-style progression, a task network of yet unprocessed tasks is maintained during search. In the general case it can grow arbitrarily large. However, many – if not most – existing HTN domains have a certain structure (called tail-recursive) where the network's size is bounded. We show how this bound can be calculated and exploited to automatically translate tail-recursive HTN problems into non-hierarchical STRIPS representations, which allows using both hierarchical structures and classical planning heuristics. In principle, the approach can also be applied to non-tail-recursive HTNs by incrementally increasing the bound. We give three translations with different advantages and present the results of an empirical evaluation with several HTN domains that are translated to PDDL and solved by two current classical planning systems. Our results show that we can automatically find practical bounds for solving partially-ordered HTN problems. We also show that classical planners perform similarly with our automatic translations versus a previous hand-bounded HTN translation which is restricted to totally-ordered problems.

Introduction

Hierarchical task network (HTN) planning has found consistent success in many real-world applications (Nau et al. 2005), which include games (Jacopin 2014; Ontañón and Buro 2015), robotics (Weser, Off, and Zhang 2010), and web service composition (Sirin et al. 2004). The success of this approach can be attributed to two primary advantages it has compared with non-hierarchical planning approaches. First, it is strictly more expressive than classical (non-hierarchical) planning (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Höller et al. 2014; 2016; Alford, Bercher, and Aha 2015a), which means that it allows specifying problems that cannot be expressed using the classical planning formalism. Second, the task hierarchy in HTN planning can encode domain-specific search control, which is essential because there are only a few approaches for domain-independent heuristic search in hierarchical planning (Elkawkagy et al. 2012; Bercher, Keen, and Biundo 2014). In contrast, a large number of classical planning heuristics and search strategies have been studied during the past 15 years. In this paper we address this disparity by presenting translations that encode tail-recursive HTN problems into classical planning problems.

Alford, Kuter, and Nau (2009) highlighted the importance of bridging this gap and introduced a translation from HTN problems to PDDL. However, their work was limited to a severely restricted case of HTN planning problems: their translation is only applicable to totally-ordered HTN problems and required a bound on recursion that had to be specified by hand. In this paper, we extend the class of translatable HTN problems to include partially-ordered problems that are tail-recursive (Alford, Bercher, and Aha 2015a). Furthermore, we automate the translation so that it no longer depends on a user-specified bound.

This fully automatic translation has several benefits. First, when the planning process itself is not of importance, then there is no compelling need to design and implement specialized HTN planning systems if we have only tail-recursive problems, as they can be solved using non-hierarchical planning systems using our translation. Second, the task hierarchy can be limited to encode structural constraints on solutions (Höller et al. 2014; 2016) – i.e., search control must not be included into the task hierarchy due to the use of domain-independent classical planning heuristics. Third, even with modeled search control, search in the translated classical setting might be faster than in the original HTN setting as the control knowledge as well as classical planning heuristics may be exploited.

The paper is structured as follows. We first explain preliminaries for classical and HTN planning. Then, we describe our three translations for tail-recursive HTN problems to PDDL. They all require a certain bound as parameter. Our first translation is quadratic in the bound and only requires the standard STRIPS language feature of PDDL. Our second translation is for the special case of totally ordered HTNs, and is linear in the bound. The last translation is both linear in the bound and applies to partially-ordered problems, but

requires the ADL language feature (derived predicates and quantified effects). The next two sections then provide polytime algorithms to calculate approximations for the lower and upper bound required for the translation. Finally, we present an empirical evaluation with two modern planners, BFS(f) (Lipovetzky et al. 2014) and Jasper (Xie, Müller, and Holte 2014), and show that they can efficiently reason with the translated knowledge.

Background

PDDL Planning Problems The most common description language for classical planning is PDDL (Fox and Long 2003). PDDL is an expressive and feature-rich language. We use the notation of Helmert (2009) to describe the subset of PDDL used in this paper.

A *PDDL operator* o is a tuple (χ, e), where χ is a first-order logic formula called the *precondition* of o, and e, called the *effect* of o, is a conjunction of literals that is optionally universally quantified over some of its variables. The free variables in χ and e form the *parameters* of o.

A *PDDL derived predicate* d is a tuple (ϕ, ψ), where ϕ is a first-order atom called the *head* of d and ψ is a first-order formula called the *body* of the axiom. Intuitively, the head atom of d must hold in any state where ψ holds. For the semantics of derived predicates to be well-defined, a set of derived predicates must be *stratifiable* and the head of any derived predicate may not appear in the effect of any operator (Thiébaux, Hoffmann, and Nebel 2005).

A *PDDL problem* $\mathcal{P}$ is a tuple $(\mathcal{L}, \mathcal{D}, \mathcal{O}, s, \mathcal{G})$, where:

- $\mathcal{L}$ is a function-free first-order language,
- $\mathcal{D}$ is a set of stratifiable derived predicates,
- $\mathcal{O}$ is a set of PDDL operators,
- s in $\mathcal{L}$ is a set of grounds atoms called $\mathcal{P}$'s *initial state*,
- $\mathcal{G}$ is a first-order formula called the *goal* of P.

The semantics of PDDL problems is defined through grounding. Given that $\mathcal{L}$ is function-free, we can create a *ground* planning problem $P = (L, D, O, s, G)$, where:

- L is a propositional language created from $\mathcal{L}$,
- D is the grounding of $\mathcal{D}$, where the head of each derived predicate is a ground atom and the body of each derived predicate is a variable-free formula.
- O is the grounding of $\mathcal{O}$, where each operator $o = (\chi, e)$ is a pair of a propositional formula χ and a conjunction e of a set of propositional literals. We use $add(o)$ and $del(o)$ to refer to the positive and negative literals of e, respectively.
- G is the variable-free grounding of $\mathcal{G}$.

The operators and derived predicates form an implicit *state transition function* $\gamma : 2^{\mathcal{L}} \times O \to 2^{\mathcal{L}}$, where:

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

P is solvable iff $(s, D) \models G$ or there is an operator $o \in O$, such that $\gamma(s, o) = s'$ and (L, D, O, s', G) is solvable.

HTN Planning Problems We use the lifted HTN planning formalism from Alford, Bercher, and Aha (2015a), augmented with method preconditions. Note that method preconditions can be compiled away into effect-free primitive operators that are constrained to come before the rest of the method's tasks.

In our formalization, every task has a *task name*, which is syntactically a first-order atom. Given a set of task names X, a *task network* is a tuple $tn = (T, \prec, \alpha)$, such that:

- T is a finite (possibly empty) set of *task instance symbols*.
- $\prec$ is a partial order over T.
- $\alpha : T \to X$ is a mapping from the task instance symbols to a finite set of task names.

The task instance symbols function as place holders for task names, allowing multiple instances of a task name to exist in a task network. A task instance $t \in T$ is *unconstrained* if no other instance is required to come before it, i.e., $\forall_{t' \in T}\, t' \not\prec t$. Conversely, we say that t is the *last task* if $\forall_{t' \in T}\, t' \prec t$.

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

- $\mathcal{L}$ is a function-free first order language.
- $\mathcal{D}$ is set of stratifiable derived predicates, as given before.
- $\mathcal{O}$ is a set of *HTN operators*, where each $o \in \mathcal{O}$ is a triple (n, χ, e), where n is a (*primitive*) task name not occurring in $\mathcal{L}$, and (χ, e) is a PDDL operator.
- $\mathcal{M}$ is a set of *HTN methods*, where each $m \in \mathcal{M}$ is a triple (c, χ, tn), where c is a (*non-primitive* or *compound*) task name not occurring in $\mathcal{O}$ or $\mathcal{L}$, χ is the precondition of m, and tn is a task network. We say a task name n is *reachable* from a task name c if there is a method $(c, \chi, tn) \in \mathcal{M}$ where n is a task name occurring in tn, or if there is a third task name c' such that n is reachable from c' and c' is reachable from c.
- s is the initial ground state and tn is the (initial) task network with task names from $\mathcal{O}$ and $\mathcal{M}$.

As with PDDL planning problems, we define the semantics of HTN planning through grounding. Given $\mathcal{L}$, we can create a ground HTN planning problem $P = (L, D, O, M, s, tn')$ where L is a propositional language, and D, O, M, and tn' are all variable-free (see the paper by Alford, Bercher, and Aha (2015b) for a more detailed description of the grounding process). The ground operators O form a state-transition function γ, defined as before.

In the grounded HTN problem P above with initial task network $tn = (T, \prec, \alpha)$, we can *apply* a ground operator $o = (n, \chi, e)$ if there is an unconstrained task instance $t \in T$ such that $\alpha(t) = n$ and $(s, D) \models \chi$. Then the result is a new problem $P' = (L, D, O, M, s', tn')$, where $s' = \gamma(s, o)$ and $tn' = (T \setminus \{t\}, \{(t_1, t_2) \in \prec \mid t_1 \neq t\}, \{(t_1, n) \in \alpha \mid t_1 \neq t\})$. We denote this operator *a*pplication with $P \to_a^o P'$.

We can *decompose* P's task network $tn = (T, \prec, \alpha)$ in a state s if there is an unconstrained task instance $t \in T$ such that $\alpha(t)$ is a non-primitive task name and there is a corresponding method $m = (\alpha(t), \chi, (T_m, \prec_m, \alpha_m)) \in M$, such that $(s, D) \models \chi$. Assume WLOG that $T \cap T_m = \emptyset$.

Then the decomposition of tn by m on t is given by the task network $tn' = (T', \prec', \alpha')$ with:

$$T' \coloneqq (T \setminus \{t\}) \cup T_m$$
$$\prec' \coloneqq \{(t', t'') \in \prec \mid t', t'' \neq t\} \cup \prec_m \cup \{(t_1, t_2) \in T_m \times T \mid (t, t_2) \in \prec\}$$
$$\alpha' \coloneqq \{(t', n) \in \alpha \mid t' \in T \setminus \{t\}\} \cup \alpha_m$$

This results in a new problem P' with the same initial state as P, but with the new task network tn'. We denote this $\underline{d}$ecomposition by $P \rightarrow_d^m P'$.

Given a planning problem P, both an operator application $P \rightarrow_a^o P'$ and a decomposition $P \rightarrow_d^m P'$ is called a *progression*, denoted by $P \rightarrow_p^x P'$, where $x \in \{o, m\}$ denotes the applied operator or method, respectively. Any finite sequence of progressions $P_0 \rightarrow_p^{x_1} P_1 \rightarrow_p^{x_2} \ldots \rightarrow_p^{x_n} P_n$ is called a progression as well and denoted by $P_0 \rightarrow_p^* P_n$. The corresponding sequence $\bar{x} = x_1, \ldots, x_n$ is called a *(progression) solution* for P if P_n has an empty task network. Finding a solution using progression is then a process of searching through possible sequences of progressions. Note that in our definition of progression, the applied sequence of methods is also specified, whereas HTN solutions typically only consists of the task network that is produced by the respective method applications. From the plan-existence point of view, progression is strictly equivalent to purely decomposition-based definitions of HTN solvability (Alford et al. 2012).

We can translate HTN problems into PDDL planning problems (also cf. Thm. 2) whenever we can bound the size of the task network needed to find a solution:

Definition 1 (Progression Bound)**.** *Given a solution $\bar{x}$ to an HTN problem P and the corresponding progression $P \rightarrow_p^* P'$, we define $\bar{x}$'s* progression bound *as the largest number of tasks in any task network visited by $P \rightarrow_p^* P'$. The smallest (largest) progression bound of any solution of P is called P's* minimum (maximum) progression bound*.*

The progression bounds of a task network tn are the bounds of P where the initial task network is replaced by tn. The progression bounds of a task name n are the bounds of P where the initial task network is replaced by a task network containing only n.

Not all problems have a finite maximum progression bound, but all solvable problems have a finite minimum progression bound. Note that a problem's progression bounds are not directly related to the length of a problem's solutions, other than that the minimum progression bound is smaller than the optimal plan length.

Three translations

The set-theoretic definition of HTN progression provides a convenient base for any translation of HTN problems into PDDL. However, support for universally-quantified preconditions and conditional effects is spotty, and there are planners with otherwise quite useful capabilities that lack support for even disjunctions and negations in preconditions (Benton, Coles, and Coles 2012). Therefore, we start with a translation that uses only positive, conjunctive preconditions with standard STRIPS-style effects. We make three assumptions about HTN problems, all of which can be enforced without loss of generality:

- Every non-empty method must have a last task instance (if none exists, a no-op is inserted as an artificial last task). Empty methods $((\texttt{c}\ \ldots), \chi, (\emptyset, \emptyset, \emptyset))$ will instead be treated as an operator $((\texttt{c}\ \ldots), \chi, \emptyset)$.
- Every variable appearing in a method or operator must be *typed*. Equivalently, every such variable must appear in at least one positive atom in the negation-normal form of its respective precondition.
- There is a single ground initial task.

An *HTN to STRIPS* translation

Given an HTN problem $\mathcal{P} = (\mathcal{L}, \mathcal{D}, \mathcal{O}, \mathcal{M}, s, tn)$ and a progression bound $b \in \mathbb{N}$, we define the *HTN2STRIPS translation* as the PDDL problem $\mathcal{P}'_b = (\mathcal{L}', \mathcal{D}, \mathcal{O}', s', \mathcal{G}')$ as follows. $\mathcal{L}'$ is the union of $\mathcal{L}$ along with:

- Constants $t_0, \ldots, t_{b+1}$ representing instance symbols.
- The predicate $(\texttt{task}_{name}\ ?x_1 \ldots ?x_n\ ?t)$ for each primitive and abstract task name $(\texttt{name}\ ?x_1 \ldots ?x_n)$, where $?t$ associates the task name with a task instance symbol.
- A predicate $(<\ ?i\ ?j)$ for specifying a total order over $t_0, \ldots, t_{b+1}$, and $(\texttt{next}\ ?i\ ?j)$, representing an ordered list of instance symbols not currently used for a task, where t_0 and t_{b+1} are placeholders for the beginning and end of the list.
- A predicate $(\texttt{consents}\ ?i\ ?j)$, which is true for every task instance t_i, t_j $(1 \leq \{i, j\} \leq b$ and not including $t_0, t_{b+1})$, such that $t_i \prec t_j$ is *not* true in the current task network.

Let $(\texttt{init}\ c_1 \ldots c_n)$ be the initial task name. Then, the new initial state s' and goal $\mathcal{G}'$ (stating that all instance symbols are free) are:

$$s' = s \cup \{(\texttt{task}_{init}\ c_1 \ldots c_n\ t_1), (\texttt{next}\ t_0\ t_2)\} \cup \{(\texttt{next}\ t_i\ t_{i+1}) \mid 2 \leq i \leq b\} \cup \{(<\ t_i\ t_j) \mid 0 \leq i \leq b, i < j \leq b+1\} \cup \{(\texttt{consents}\ t_i\ t_j) \mid i, j \in \{1..b\}\}$$
$$\mathcal{G} = \textstyle\bigwedge_{0 \leq i \leq b} (\texttt{next}\ t_i\ t_{i+1})$$

For each HTN operator $o = ((\texttt{o}\ \ldots), \chi, \psi) \in \mathcal{O}$, $\mathcal{O}'$ contains the PDDL operator $o' = (\chi \wedge \chi', \psi \wedge \psi')$, where:

$$\chi' = (\texttt{task}_o\ \ldots\ ?t) \wedge (\texttt{next}\ ?t_p\ ?t_n) \wedge (<\ ?t_p\ ?t) \wedge (<\ ?t\ ?t_n) \wedge \textstyle\bigwedge_{1 \leq i \leq b} (\texttt{consents}\ t_i\ ?t)$$
$$\psi' = \neg(\texttt{task}_o\ \ldots\ ?t) \wedge \neg(\texttt{next}\ ?t_p\ ?t_n) \wedge (\texttt{next}\ ?t_p\ ?t) \wedge (\texttt{next}\ ?t\ ?t_n) \wedge \textstyle\bigwedge_{1 \leq i \leq b} (\texttt{consents}\ ?t\ t_i)$$

The precondition χ' contains χ along with the $\texttt{task}_o$ predicate to bind the task parameters to an associated instance symbol in the state, as well as $\texttt{consents}$ predicates to ensure that there are no preceding tasks, referencing each instance symbol by its constant. The remainder of the precondition with $<$ and $\texttt{next}$ symbols are for locating where

in the list of free instance symbols $?t$ should be reinserted in order to preserve the list ordering with respect to $<$ to limit the increase in reachable states associated with the free list. The effect ψ' removes the task predicate, reinserts the instance symbol in the free list, and asserts that the instance symbol no longer precedes any other task.

For each method $m = ((\texttt{c}\ \ldots), \chi, tn) \in \mathcal{M}$, where $tn = (\{t_1, \ldots, t_n\}, \prec, \alpha)$, t_1 is tn's last task, and $\alpha(t_i) = (\texttt{m}_\texttt{i}\ v_1, \ldots, v_{k_i})$ for $1 \leq i \leq n$ and terms $v_j, 1 \leq j \leq k_i$, $\mathcal{O}'$ contains the PDDL operator $m' = (\chi', \psi')$, where:

$$\begin{aligned}\chi' = {} & \chi \wedge (\texttt{task}_c\ \ldots\ ?t_1) \wedge \textstyle\bigwedge_{1 \leq i \leq b} (\texttt{consents}\ t_i\ ?t_1) \\ & \wedge (\texttt{next}\ t_0\ ?t_2) \wedge \textstyle\bigwedge_{2 \leq i \leq n} (\texttt{next}\ ?t_i\ ?t_{i+1}) \\ \psi' = {} & \neg (\texttt{task}_c\ \ldots\ ?t_1) \wedge (\texttt{next}\ t_0\ ?t_{n+1}) \\ & \wedge \neg (\texttt{next}\ t_0\ ?t_2) \wedge \textstyle\bigwedge_{2 \leq i \leq n} \neg (\texttt{next}\ ?t_i\ ?t_{i+1}) \\ & \wedge \textstyle\bigwedge_{1 \leq i \leq n} (\texttt{task}_{m_i}\ v_1 \ldots v_{k_i}\ ?t_i) \\ & \wedge \textstyle\bigwedge_{t_i \prec t_j} \neg (\texttt{consents}\ ?t_i\ ?t_j)\end{aligned}$$

Here, again, χ' contains χ, the method's task symbol that binds the instance variable $?t_1$, and the appropriate `consents` predicates. The effects remove $n-1$ instance symbols from the 'next' list, which are used to assert the method's task network into the state, and remove `consents` atoms whenever there is a precedence constraint between two tasks. Critically, ψ' reuses $?t_1$ for the last task, keeping its `consents` constraints, ensuring the last task constrains any task previously constrained by $?t_1$.

Taken together, each operator in $\mathcal{O}'$ represents either an HTN operator or method, and implements the definition of progression accordingly. Most classical planners should be able to handle the output since the transformation only adds a conjunction of positive literals to the preconditions and simple adds and deletes to the effects. The cost of this, though, is that operator schemata grow linearly with the chosen progression bound, while the initial state grows quadratically. We also had to ensure each method had a last task. This can increase the progression bound for some problems. Next we show that our transformation retains the same set of solutions. We proof our claim only for the first translation and omit the proof for the other two for the sake of brevity.

Theorem 2 (Solution Equivalence)**.** *Let P be an HTN planning problem. For every solution $\bar{x}$ to P there is some $b \in \mathbb{N}$, such that P's HTN2STRIPS translation P'_b has a solution $\bar{x}'$ that is equivalent to $\bar{x}$. Further, for all $b \in \mathbb{N}$ and all solutions $\bar{x}$ to the respective HTN2STRIPS translation P'_b, P has a solution $\bar{x}'$ that is equivalent to $\bar{x}$.*

Proof sektch. Let $\bar{x}$ be a solution to P and $P = P_1 \rightarrow_p^{x_1} P_2 \rightarrow_p^{x_2} \ldots \rightarrow_p^{x_{n-1}} P_n$ be a corresponding progression. Let $b = \max_{i=1}^{n} |tn(P_i)|$ be the maximal size of the visited task networks. Then, $\bar{x}'$ ($\bar{x}'$ being equivalent to $\bar{x}$) is a valid solution to the HTN2STRIPS translation P'_b.

Let $b \in \mathbb{N}$ and $\bar{x}' = x'_1, \ldots, x'_n$ be a solution of the HTN2STRIPS translation P'_b. As P'_b simulates progression, the sequence $P \rightarrow_p^{x_1} P_1 \rightarrow_p^{x_2} \ldots \rightarrow_p^{x_n} P_n$ (with $\bar{x} = x_1, \ldots, x_n$ being equivalent to $\bar{x}'$) is a valid progression sequence for P. Since $\bar{x}'$ is a solution to P'_b, the task network of P_n must be empty and $\bar{x}$ forms a solution to P. □

For the remaining two translations, we don't give the respective theorems, as their proofs work analogously.

A *TOHTN to STRIPS* translation

HTN problems where the initial task network and all the methods' networks are totally ordered are called *totally-ordered HTNs*, or TOHTNs. Any progression of a TOHTN problem is still totally-ordered, and a totally-ordered task network can be represented as a simple list with only one unconstrained task at a time, simplifying our translation.

Given a TOHTN problem $\mathcal{P} = (\mathcal{L}, \mathcal{D}, \mathcal{O}, \mathcal{M}, s, tn)$ and a progression bound $b \in \mathbb{N}$, we define the *TOHTN2STRIPS translation* of $\mathcal{P}$ as the PDDL problem $(\mathcal{L}', \mathcal{D}, \mathcal{O}', s', \mathcal{G}')$ as follows. $\mathcal{L}'$ is defined as the union of $\mathcal{L}$ with the instance symbols $t_0, \ldots, t_b$, the `task` and $<$ predicates, but not the `permits` or `next` predicates. Instead, we repurpose $<$ as a stack with a new unary predicate $(\texttt{head}\ ?t)$ as the stack pointer. The initial state s' contains s, the $<$ ordering of $t_0, \ldots, t_b$, the initial $(\texttt{task}_n\ \ldots\ t_1)$ atom, and the atom $(\texttt{head}\ t_1)$. The goal $\mathcal{G}'$ is simply $(\texttt{head}\ t_0)$.

For each operator $((\texttt{o}\ \ldots), \chi, \psi) \in \mathcal{O}$, $\mathcal{O}'$ contains the PDDL operator (χ', ψ'), where:

$$\begin{aligned}\chi' = {} & \chi \wedge (\texttt{task}_o\ \ldots\ ?t) \wedge (\texttt{head}\ ?t) \wedge (<\ ?t_p\ ?t) \\ \psi' = {} & \psi \wedge \neg (\texttt{task}_o\ \ldots\ ?t) \wedge \neg (\texttt{head}\ ?t) \wedge (\texttt{head}\ ?t_p)\end{aligned}$$

Then $\mathcal{O}'$ is applicable when o's task is at the head of the stack. The effects retract the task name from the state and decrement the stack pointer, and otherwise behave as ψ. Again, empty methods are translated as operators.

For each non-empty method $((\texttt{c}\ \ldots), \chi, tn)$ where $t_n \prec \ldots \prec t_1$ is the total ordering of instance symbols in tn, $(\texttt{m}_\texttt{i}\ v_1, \ldots, v_{k_i})$ is the task name associated with each t_i, $\mathcal{O}'$ contains the PDDL operator (χ', ψ'), where:

$$\begin{aligned}\chi' = {} & \chi \wedge (\texttt{task}_c\ \ldots\ ?t_1) \wedge (\texttt{head}\ ?t_1) \\ & \wedge \textstyle\bigwedge_{1 \leq i \leq n-1} (<\ ?t_i\ ?t_{i+1}) \\ \psi' = {} & \neg (\texttt{task}_c\ \ldots\ ?t_1) \wedge (\texttt{head}\ ?t_n) \wedge \neg (\texttt{head}\ ?t_1) \\ & \textstyle\bigwedge_{1 \leq i \leq n} (\texttt{task}_{m_i}\ v_1 \ldots v_{i_k}\ ?t_i)\end{aligned}$$

This method only needs to retract its current task name, assert the list of subtasks, and move the pointer accordingly.

This translation is closely related to the translation of totally-ordered problems found in (Alford, Kuter, and Nau 2009), which we refer to here as *TOHTN09 translation*. That translation relies on a user-specified bound of non-tail recursion, which is closely related to our progression bounds. The TOHTN2STRIPS translation adds fewer predicates and operators than the TOHTN09 translation, though this does not necessarily mean fewer operators and preconditions in the grounded domain, since any bound on progression will be larger (polynomially) than a corresponding bound on non-tail recursion.

An *HTN to ADL* translation

The set-theoretic definition of progression is directly and trivially representable with the quantified preconditions and quantified conditional effects provided by ADL. We omit the full definition of our *HTN2ADL translation* due to lack of

space, but it roughly follows the outline provided by the HTN2STRIPS translation, with the exception that it uses `consents`'s converse predicate (`constrains` t_1 t_2), which is true whenever $t_1 \prec t_2$ holds in the current task network. Since the translation directly uses the definition of decomposition, which has a quantified condition in asserting decomposed task constraints, we no longer need the restriction that every method has a last task. Enforcing that condition entails adding place-holder subtasks to methods, which can increase the minimum progression bound, meaning the HTN2ADL translation can solve some partially-ordered problems with a smaller progression bound than the HTN2STRIPS translation.

Automatic bounds for tail-recursive HTNs

Since HTN planning is in general undecidable (Erol, Hendler, and Nau 1996; Geier and Bercher 2011), not every HTN problem will have a finite progression bound. Even for problems that do have a finite progression bound, finding either the minimum or maximum progression bounds may be as difficult as solving the problem itself. In the rest of the section, we show that, for a large class of HTN planning problems (those that are *tail-recursive*), we can approximate the upper and lower progression bounds in poly-time. Our approach is also applicable to non tail-recursive planning problems (see examples in the evaluation). Here the bound b used in the translation has to be increased until a solution is found. Due to the undecidability of HTN planning, we cannot stop at any point if the translated problem is unsolvable and hence cannot prove unsolvability with this technique.

Tail-Recursive HTN Planning

Many HTN problems are *tail-recursive*, in that any task can only recurse through the last task instance of any of its associated methods. Tail-recursive problems are guaranteed to have a finite progression bound, and they can be identified through a syntactic test called $\leq_r$-stratifiability (Alford et al. 2012). An HTN problem $\mathcal{P}$ is $\leq_r$*-stratifiable* if there exists a total preorder $\leq_r$ on the task names appearing in the grounded problem P of $\mathcal{P}$ such that for every method $(c, \chi, (T, \prec, \alpha))$ in P:

- If there is a last task $t_r \in T$, then $\alpha(t_r) \leq_r c$.
- 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. Totally-ordered and partially-ordered tail-recursive problems have worst-case progression bounds that are polynomial and exponential in the height of shortest stratification, respectively.

We note that a $\leq_r$-stratification of the predicate symbols appearing in $\mathcal{P}$ implies the existence of a polynomial-height $\leq_r$-stratification of P without having to generate the grounded model.

Determining the upper bound

The worst-case upper bounds for tail-recursive problems are only very rough estimates of the actual maximum progression bound, but calculating the exact maximum progression is as hard as the planning problem itself. We present an algorithm to compute the exact maximum progression bound of the *state relaxation* of a problem P in time polynomial to its stratification. The state relaxation is achieved by deleting all preconditions occurring in P, effectively making states irrelevant. The maximum progression bound of the state relaxation overestimates the actual maximum progression bound.

For the rest of this section and the next, we will assume that the problem we are calculating the bounds for has been state-relaxed. First, we show how to bound the progression of a task network assuming we already know the maximum progression bounds of all tasks contained in it.

Let $tn = (T, \prec, \alpha)$ be a task network, and let $w(\cdot)$ be a function assigning a maximum progression to each task name appearing in T.

Lemma 3. *A task $t \in T$ contributes exactly 0, 1, or $w(\alpha(t))$ to the maximum progression bound of tn*

Proof. The maximum progression bound of tn is the size of the largest task network on the way to progressing all tasks out of tn. Assume at this point that t is not already progressed out (0) nor has predecessors (1). That t has a maximum progression bound and that the problem is state-relaxed implies that there is a sequence of progressions that can decompose it into a total of $w(\alpha(t))$ tasks. So tn's weight is not less than $w(\alpha(t))$ plus the remaining tasks. □

Definition 4 (Independent set)**.** *For a graph $G = (V, E)$, an* independent set *$I \subseteq V$ is a set of vertices of G, such that $\forall_{x,y \in I}$ there is no edge $(x, y) \in E$.*

For a task network $tn = (T, \prec, \alpha)$, the partial order $\prec$ forms a transitive DAG, and so a set $I \subseteq T$ is independent if and only if $\forall_{i,j \in I}\, i \not\prec j$. An independent set of tn represents a set of tasks that be unconstrained simultaneously during progression. Let $\mathcal{I}$ be the set of all independent sets of T.

Corollary 5. *Let $I \in \mathcal{I}$ and define $PB_w : \mathcal{I} \to \mathcal{N}$ as*

$$PB_w(I) = |\{j \in T \mid \exists_{i \in I}\, i \prec j\}| + \sum_{i \in I} w(\alpha(i))$$

The progression bound of tn is $\max_{I \in \mathcal{I}} PB_w(I)$.

Proof. Since each I is independent, we can always progress out all the proceeding tasks, so the progression bound is at least the sum of Lemma 3 for each remaining task.

By Lemma 3 again, the maximum bound can only be the sum over the tasks of 0, 1, or $w(\alpha(t))$, and all of the latter term must come from independent tasks. □

We can find the bound by transforming it into the problem of finding maximum weighted independent sets in transitively closed DAGs. The following proposition provides a formal definition of the problem and states that it is poly-time computable (Kagaris and Tragoudas 1999; Golumbic 2004) by applying a maximum flow algorithm.

Proposition 6. *Let $G = (V, E)$ be a transitively closed DAG with vertex weights $w : V \to \mathbb{N}$. G's maximum weighted independent set can be determined in poly-time.*

Our reduction of maximum progression bounds for task networks is given in the proof of the following theorem:

Theorem 7. *The maximum progression bound of a task network $tn = (T, \prec, \alpha)$ can be computed in poly-time.*

Proof. Let $G = (V, E)$ be a DAG representation of T and $\prec$ weighted with $w(\cdot)$. As $\prec$ is transitively closed, so is G.

We define a new graph $G^* = (V^*, E^*)$ which will be the input for the independent set algorithm by

$$V^* = \{v^+, v^- \mid v \in V\}$$
$$E^* = \{(a^+, b^+), (a^-, b^+) \mid (a, b) \in E\} \cup \{(v^-, v^+) \mid v \in V\}$$

All vertices v^+ retain the weight of v – representing that a vertex is in the progression-front, while v^- has the weight 1 – representing a vertex that is a successor of a progression-front vertex. The construction of the graph ensures that if a vertex v is in the progression-front none of its predecessors can be counted towards the weight of the progression-front.

We compute the maximum weight of an independent set of G^* and claim that it is equivalent to the maximum progression bound of tn. The constructed graph is transitively closed. Also for any vertex v, the vertex v^+ does not have an edge to any w^- where w is a successor of v, i.e., both can be part of an independent set at the same time. Similarly including v^- in an independent set excludes any vertex w^+ where w is a successor of v.

Let $P \subseteq V$ be an independent set in G. Then the set $I = \{v^+ \mid v \in P\} \cup \{v^- \mid \exists w \in P \text{ and } (w, v) \in V\}$ is independent in G^*. Moreover, $PB_w(P) = \sum_{i \in I} w(i)$.

Conversely, let $I \subsetneq V^*$ be an independent set in G. If I contains a vertex a^- such that (a^-, a^+) is the only edge to a^+, then a^+ must have weight 1. So WLOG, assume I contains a^+ in such cases. Let $P = \{v \mid v^+ \in I\}$. Then P is an independent set in G, and $\sum_{i \in I} w(i) = PB_w(P)$.

So any maximal independent set in G^* has the same weight as the maximum progression bound of tn. □

Let $S = S_0 S_1 \dots S_n$ be a *maximal* $\leq_r$-stratification of a state-relaxed problem P, in the sense that two task names c_1 and c_2 are on the same stratum if and only if $c_1 \leq_r c_2 \leq_r c_1$ (i.e., they are tail-recursive with each other).

We define progression bounds over the strata in a domain in a bottom-up fashion: If a stratum contains a primitive task name (singular, by dint of maximality), the maximum progression bound for that task is 1. In the special case where no method associated with a stratum references tasks from lower strata or have an empty task network, the tasks on that network are unsolvable. We assign them a progression bound of $-\infty$, since there is no solution through them.

Otherwise, for strata S_i, let $\mathcal{M}_{S_i}$ be the set of methods associated with the non-primitive task names in S_i. We can calculate provisional maximum progression bounds for each task network using the (already computed) maximum progression bounds from lower strata, and assigning 1 as a bound for any task name occurring in S_i. Since the methods are tail-recursive and the stratification is maximal, for any given task name c, we can progress from a network containing just c to the task network from any method in $\mathcal{M}_S$, but not before progressing out any other tasks introduced along the way. So the maximum progression bound for all task names in S is the max of the provisional maximum progression bounds of the methods.

Finally, the maximum progression bound for P is just the maximum progression bound of its initial task network calculated using the bounds from S.

Determining the lower bound

In many cases, it may not be feasible to use the upper bound for the translation. Instead smaller bounds can be used for which a solution might exist, but non-solvability of the translation does not imply that the original problem does not have a solution and plans may be non-optimal. A similar technique is commonly used in SAT-planning. By computing a lower bound to the minimum progression bound we can determine the smallest bound we can reasonably use.

The minimum progression problem can be solved by a fix-point approach. We compute the minimum progression $m_p(t)$ of the planning problem that only contains t in its initial task network. For any primitive task this value is 1 and we initialize all other values with ∞. We repeatedly iterate over the set of compound tasks c and recompute their value by iterating through all their methods (c, tn) and their minimum progressions based on the current values $m_p(\cdot)$. For every tn we have to determine the minimum progression when progressing through it. Progressing through two of tns tasks simultaneously instead of one at a time can only increase the size of intermediate task networks in the progressions. The minimum progression is achieved by progression through the tasks of tn is some topological order. While progressing through a task t, the largest task network of the process has as many tasks as the minimum progression of t plus the number of tasks that are behind t in the topological ordering. This leads to the following graph problem.

Definition 8 (Min-Order problem)**.** *Given a DAG $G = (V, E)$ with weights on its vertices $w \colon V \to \mathbb{N}$. Is there a topological ordering $(v_1, \dots, v_n)$ of V such that $\forall_{1 \leq i \leq |V|} w(v_i) + |V| - i \leq k$?*

Theorem 9. MIN-ORDER *is in* ***P****.*

Proof. Ensuring the property $w(v_i) + |V| - i \leq k$ for all vertices is equivalent to $w(v_i) - i \leq k'$ for $k' = k - |V|$. We can decide whether there is a topological ordering fulfilling the criterion in a greedy fashion. Given a DAG $G = (V, E)$ we select any source v_i – a vertex without an ingoing edge – such that $v_i - 1 \leq k'$ to be the first vertex of the topological ordering. Then the vertex is removed from G and the weights of all remaining nodes are decreased by 1, resulting in a new graph G'. The topological ordering of G' constitutes the remainder of the topological ordering of G.

Suppose the algorithm fails to construct an ordering, but a valid one exists. Then at any point during the algorithm the current graph G had a solution, but the newly constructed graph G' does not. Let $(v_1, \dots, v_n)$ be a topological ordering of G that fulfills the criterion and let v_i be the vertex that

was selected by our greedy algorithm. Since v_i is a source $(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$ is a valid topological ordering of G'. For the nodes $v_1, \ldots, v_{i-1}$ we have $w_{G'}(v_j) - j = w_G(v_j) - 1 - j \leq k$, since they have the same index in the ordering for G. The index for all other nodes has decreased by 1, which is canceled out by decreasing the weight of the nodes. Thus the ordering $(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$ also fulfills the criterion. Contradiction. □

HTN planning without preconditions is a degenerate form of delete-free HTN planning with task insertion (Alford et al. 2014), which means there is always an acyclic series of progressions that can derive a witness for the lower bound (Alford, Bercher, and Aha 2015b). This means there is no necessary interdependence between rows of $m_p(t)$, and so at least one correct and final lower bound is calculated in every iteration. Thus, the $m_p(t)$ calculation iterates at most $|C|$ times, where C is the set of non-primitive task names.

Experiments

Our work presents two questions: Is our translation amenable to current classical planning techniques, and do we find reasonable automatic bounds for HTN problems. We chose two recent IPC (Vallati, Chrpa, and McCluskey 2015) planners with distinct code bases: Jasper (Xie, Müller, and Holte 2014) deriving from Fast-Downward (Helmert 2006), and BFS(f) (Lipovetzky et al. 2014) deriving from LAPKT and FF (Ramirez, Lipovetzky, and Muise 2015; Hoffmann and Nebel 2001). We ran all problems on a Xeon E5-2639 with a per problem limit of 8 GB of RAM and 45 minutes of planning time. While Jasper can handle ADL, BFS(f) is restricted to the HTN2STRIPS and TOHTN translations (as with many other IPC planners).

To evaluate the translation, we used the three domains from the evaluation of the TOHTN09 translation (Alford, Kuter, and Nau 2009). The first is an HTN adaptation of the well-known *Blocksworld* domain. In the *Towers of Hanoi* domain, the task hierarchy encodes the solution strategy that can be described easily as hierarchical procedural knowledge. In the *Office Delivery* domain, a robot has to move through rooms, open and close doors and deliver some object. For more details like the number of tasks and methods, we refer to the paper by Alford, Kuter, and Nau (2009). We translated the above problems using the automated maximum progression bound with each of our three translations[1].

We present coverage results in Tab. 1 and Fig. 1 comparing planner coverage and CPU time against the base action model and the previous TOHTN09 translation. Both planners performed well against the base action model, showing that the reduction in branching factor overcame the overhead of planning with methods. All HTN translations have identical search spaces, so we have no proof that the modest performance differences with the TOHTN09 translation stem from anything other than planner implementation minutiae.

All of the above problems had a maximum progression bound of 2, in part because they were designed to work well with the TOHTN09 translation. To evaluate whether our automatic bounding algorithm finds reasonable bounds on "natural" HTN problems, we collected 60 problems from four domains in hybrid planning (Bercher, Keen, and Biundo 2014). Here, compound tasks are not symbols as in the given formalism, but they describe abstract state transitions by means of preconditions and effects. Methods map the respective compound task to an "implementing" plan (Biundo and Schattenberg 2001). In these plans, causality is represented using causal links, a concept known from POCL planning. We removed these hybrid planning features to obtain simplified models. Two of these domains, *Satellite* and *Woodworking*, were well-known IPC domains and extended for hybrid planning. The *UM-Translog* domain describes a logistics problem and was originally designed for hierarchical planning systems such as UMCP (Andrews et al. 1995) and then adapted to hybrid planning. The *SmartPhone* domain was directly modeled for hybrid planning. It describes the operation of a modern cell phone (Biundo et al. 2011).

Our automatic bounding found minimum progression bounds (PBs) between 1 and 17 for these problems (Tab. 2). Not all problems were tail recursive, so we separate out these results. For tail-recursive problems, we found maximum progression bounds of 3 to 40. The inclusive range between minimum and maximum PB form a feasible set of PBs. We ran BFS(f) and Jasper 10 times on each useable combination of translation and feasible PB, arbitrarily capping the PB of non-tail-recursive problems at twice their minimum PB.

Some combination of planner, translation, and PB solved all but one problem (the largest non-tail-recursive SmartPhone). Overall, Jasper with HTN2ADL solved 74% of the translated instances, Jasper with HTN2STRIPS solved 57%, and BFS(f) with HTN2STRIPS solved 40%. Limited to the tail-recursive instances translated with their maximum PB, solution rates were 78%, 64%, and 54%, respectively. The lower completion rate for partially-ordered problems matches with their theoretical difficulty (Alford, Bercher, and Aha 2015a).

We observed that at least 18 of the 60 original problems had solutions that matched their theoretical maximum PB and 25 problems had solutions that matched their theoretical minimum (5 had matching minimum and maximum PBs). We conclude that our bounding techniques are often sufficient to find feasible progression bounds for translation, though there is significant room for improvement.

Related & Future Work

Relatedly, Fritz, Baier, and McIlraith (2008) described theoretical translations of HTNs into ConGolog and ConGolog into a variant of PDDL, but the translation into PDDL requires unbounded state structure (e.g., function symbols in the operator parameters and the state would be sufficient). Given the equivalence between HTNs and ConGolog (Goldman 2009), our work makes it possible to provide an automatic bounded translation of tail-recursive ConGolog programs into PDDL. Whether such translations are useful is left to future work.

Some difficulties in planning with our HTN translations are undoubtedly due to the size of the method schemas.

[1] http://github.com/ronwalf/HTN-Translation

			No HTN		HTN2STRIPS		HTN2ADL		TOHTN2STRIPS		TOHTN09	
Planner	Domain	#	#	(s)	#	(s)	#	(s)	#	(s)	#	(s)
Jasper	Blocksworld	575	561	64.3	575	2.2	575	2.6	575	1.2	575	1.5
Jasper	Towers	120	110	272.6	120	77.4	120	29.7	120	9.4	120	14.4
Jasper	Office Delivery	500	441	245.0	500	6.5	500	3.9	500	8.6	500	7.2
BFS(f)	Blocksworld	1000	447	212.5	1000	1.4	–	–	1000	0.4	1000	0.5
BFS(f)	Towers	120	80	254.1	110	18.1	–	–	120	2.9	120	5.2
BFS(f)	Office Delivery	500	500	109.2	500	11.6	–	–	500	27.6	497	44.3

Table 1: Planner coverage (#) per domain variant (the column "No HTN" denotes the (non-hierarchical) base action model) and planner, along with average time in seconds (s) for problems solved by all domain variants by the planner. For Jasper on Blocksworld, 425 problems were tossed when at least one variant segfaulted or erroneously reported the problem unsolvable.

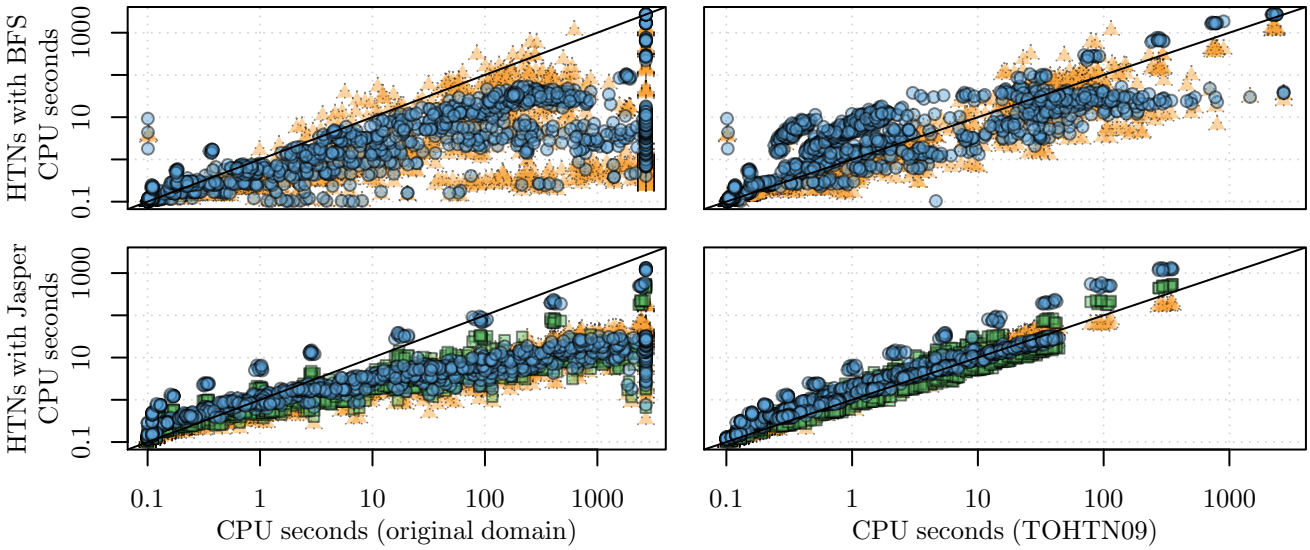

Figure 1: Log-scale comparison of planning time on Blocksworld, Robot Navigation, and Towers of Hanoi with HTN translations vs. planning time for the original domains (left) and our HTN translations vs. TOHTN09 translation (right). Circles, boxes and triangles indicate instances using the HTN2STRIPS, HTN2ADL and TOHTN2STRIPS translation, respectively.

	Problem	Translated	Theoretical PB		Observed PB		Jasper-STRIPS		Jasper-ADL		BFS(f)-STRIPS	
Domain	Count	Instances	Min	Max	Min	Max	#	(s)	#	(s)	#	(s)
Satellite	26	3340	1 – 8	5 – 40	4	14	1745	232.8	2378	229.9	1156	232.1
SmartPhone	4	190	4 – 6	8 – 11	4	8	190	2.1	190	0.5	190	0.7
SmartPhone*	3	350	6 – 17		6	16	86	145.5	161	184.3	70	214.6
UM-Translog	15	390	5 – 8	5 – 16	5	11	360	30.1	390	6.4	280	37.0
UM-Translog*	7	530	6 – 8		6	11	310	29.9	420	15.2	250	6.2
Woodworking	5	220	3 – 6	3 – 10	3	7	152	34.8	185	18.9	39	75.1
Total	60	5020					2843		3724		1985	

Table 2: Planner coverage (#) per domain variant and planner with mean time in seconds (s) of solved problems for partially-ordered domains. The "*" domains contained non-tail-recursive problems. BFS(f) and Jasper were run 10 times on each feasible combination of progression bound and translation type (neither planner is deterministic). On non-tail-recursive problems, we capped the progression bounds at twice the minimum progression bound.

Work on automatic action splitting may reduce the difficulty of grounding the translated domains (Areces et al. 2014).

Conclusion

In this paper we showed how a broad class of HTN planning problems can be automatically translated into classical planning problems. While such a translation is impossible in general (due to the undecidability of HTN planning), there is a special case that often occurs in practice for which it is possible: tail-recursive problems. In such planning problems, tasks may only recurse through the last task in a method's task network. As a consequence, the size of any task network that is produced by SHOP2-style HTN progression is bounded. We provided a poly-time algorithm that calculates an overestimation of this bound. To increase the practical

efficiency of the translation, we also provided an algorithm to determine the lower bound. We describe translations that take such a bound as input and produce a classical planning problem that simulates the progression search of the original HTN problem up to the specified bound. We extended a previous translation that worked only for totally-ordered planning problems to work with arbitrary task ordering. It can be applied to any HTN planning problem, provided a suitable progression bound is given. Our empirical evaluation with two planning systems showed that the additional domain knowledge increases the planning performance and that the new translation is competitive to the previous one.

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.; Bercher, P.; and Aha, D. W. 2015a. Tight bounds for HTN planning. In *Proc. of ICAPS*, 7–15. AAAI Press.

Alford, R.; Bercher, P.; and Aha, D. W. 2015b. Tight bounds for HTN planning with task insertion. In *Proc. of IJCAI*, 1502–1508. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN problem spaces: Structure, algorithms, termination. In *Proc. of 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 *Proc. of 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 *Proc. of IJCAI*, 1629–1634. AAAI Press.

Andrews, S.; Kettler, B.; Erol, K.; and Hendler, J. 1995. UM translog: A planning domain for the development and benchmarking of planning systems. Technical report, University of Maryland at College Park.

Areces, C.; Bustos, F.; Dominguez, M. A.; and Hoffmann, J. 2014. Optimizing planning domains by automatic action schema splitting. In *Proc. of ICAPS*, 11–19. AAAI Press.

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal planning with preferences and time-dependent continuous costs. In *Proc. of ICAPS*, 2–10. AAAI Press.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of SoCS*, 35–43. 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 *Proc. of ECP*, 157–168. AAAI Press.

Biundo, S.; Bercher, P.; Geier, T.; Müller, F.; and Schattenberg, B. 2011. Advanced user assistance based on AI planning. *Cognitive Systems Research* 12(3-4):219–236.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving hierarchical planning performance by the use of landmarks. In *Proc. of AAAI*, 1763–1769. AAAI Press.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18(1):69–93.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Fritz, C.; Baier, J. A.; and McIlraith, S. A. 2008. Congolog, sin trans: Compiling congolog into basic action theories for planning and beyond. In *Proc. of KR*, 600–610. AAAI Press.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of IJCAI*, 1955–1961. AAAI Press.

Goldman, R. P. 2009. A semantics for HTN methods. In *Proc. of ICAPS*, 146–153. AAAI Press.

Golumbic, M. C. 2004. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. Elsevier.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AI* 173(5):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system. *Journal of Artificial Intelligence Research* 14:253–302.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. of ECAI*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of ICAPS*. AAAI Press.

Jacopin, É. 2014. Game AI planning analytics: The case of three first-person shooters. In *Proc. of AIIDE*, 119–124. AAAI Press.

Kagaris, D., and Tragoudas, S. 1999. Maximum weighted independent sets on transitive graphs and applications. *Integration, the VLSI Journal* 27(1):77–86.

Lipovetzky, N.; Ramirez, M.; Muise, C.; and Geffner, H. 2014. Width and inference based planners: SIW, BFS(f), and PROBE. In *The 8th IPC*, 6–7.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Wu, D.; Yaman, F.; Muñoz-Avila, H.; and Murdock, J. W. 2005. Applications of SHOP and SHOP2. *Intelligent Systems, IEEE* 20:34–41.

Ontañón, S., and Buro, M. 2015. Adversarial hierarchical-task network planning for complex real-time games. In *Proc. of IJCAI*, 1652–1658. AAAI Press.

Ramirez, M.; Lipovetzky, N.; and Muise, C. 2015. Lightweight Automated Planning ToolKiT. http://lapkt.org/. Accessed: 2016-03-10.

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2004. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web* 1(4):377–396.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of PDDL axioms. *AI* 168(1):38–69.

Vallati, M.; Chrpa, L.; and McCluskey, T. L. 2015. The 2014 IPC: Progress and trends. *AI Magazine*. ISSN 0738-4602.

Weser, M.; Off, D.; and Zhang, J. 2010. HTN robot planning in partially observable dynamic environments. In *Proc. of ICRA*, 1505–1510. IEEE.

Xie, F.; Müller, M.; and Holte, R. 2014. Jasper: The art of exploration in greedy best first search. In *The 8th IPC*, 39–42.