ABSTRACT

Title of dissertation: SEARCH COMPLEXITIES FOR HTN PLANNING

Ronald Alford, Doctor of Philosophy, 2013

Dissertation directed by: Professor Dana Nau
Department of Computer Science

Hierarchical Task Network (HTN) planning is the problem of decomposing an initial task into a sequence of executable steps. Often viewed as just a way to encode human knowledge to solve classical planning problems faster, HTN planning is more expressive than classical planning, even to the point of being undecidable in the general case. However, HTN planning is not just a way to solve planning problems faster, but is itself a search problem that can benefit from its own distinct search algorithms and heuristics.

The dissertation examines the complexities of various HTN planning problem classes in order to motivate the development of heuristic search algorithms for HTN planning which are guaranteed to terminate on a large class of syntactically identifiable problems, as well as domain independent heuristics for those algorithms to use. This will allow HTN planning to be used in a number of areas where problems may be unsolvable, including during the initial development of a domain and for use in policy generation in non-deterministic planning environments.

In particular, this dissertation analyzes two commonly used algorithms for

HTN planning and describes the subsets of HTN problems that these algorithms terminate on. This allows us to discuss the run-times of these algorithms and compare the expressivity of the classes of problems they decide. We provide two new HTN algorithms which terminate on a strictly broader and more expressive set of HTN problems.

We also analyze the complexity of delete-free HTN planning, an analogue to delete-free classical planning which is the base of many classical planning heuristics. We show that delete-free HTN planning is NP-complete, putting the existence of strict-semantics delete-relaxation-based HTN heuristics out of reach for practical purposes.

Finally, we provide a translation of a large subset of HTN planning to classical planning, which allows us to use a classical planner as a surrogate for a heuristic HTN planner. Our experiments show that even small amounts and incomplete amounts of HTN knowledge, when translated into PDDL using our algorithm, can greatly improve a classical planner's performance.

SEARCH COMPLEXITIES FOR HTN PLANNING

by

Ronald Alford

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Advisory Committee:
Professor Dana S. Nau, Chair/Advisor
Dr. Ugur Kuter, Co-Advisor
Professor Michael C. Laskowski, Deans Representative
Professor William Gasarch
Professor Don Perlis

Acknowledgments

This work would not have been possible without Professor Dana Nau and his generous support and mentorship throughout my research. Together with Dr. Ugur Kuter, they acted as crucible and forge for my ideas; refining the essence of what was interesting, and hammering it into respectable shape. Special acknowledgements also go to Vikas Shivashankar, who suffered through many hours of my ramblings and whiteboard scribbling long before my ideas became coherent.

I thank the rest of the committee, Professor William Gasarch, Professor Michael C. Laskowski, and Professor Don Perlis, both for their willingness to a part of this process and their past impact on my education and research.

Lastly, I am greatly indebted to my family, who have been a relentless source of support, encouragement and motivation throughout my studies.

This work was supported in part by ARO grant W911NF1210471 and ONR grant N000141210430. The information in this dissertation does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

Table of Contents

List of Figures v

List of Abbreviations vi

1 Introduction 1
- 1.1 Related Work 3
- 1.2 HTN Planning 5
- 1.3 Classical Planning 8

2 HTN Problem Spaces 10
- 2.1 Decomposition Problem Spaces 12
  - 2.1.1 Decidability under decomposition 14
  - 2.1.2 Complexity of $\leq_1$-stratifiable problems 17
- 2.2 Progression Problem Spaces 20
  - 2.2.1 Decidability under progression 22
  - 2.2.2 Identifiability of finite progression spaces 25
  - 2.2.3 Complexity of $\leq_r$-stratifiable problems 27
- 2.3 Total Order Partition Problem Spaces 28
  - 2.3.1 Total Order Partitions 28
  - 2.3.2 Search in TOD and TOP spaces 31
  - 2.3.3 Decidability under problem partitioning 33
  - 2.3.4 Complexity of $\leq_1$- and $\leq_r$-ordered problems 36
- 2.4 Practical Considerations 41
- 2.5 Discussion 42

3 Complexities for Delete-Free HTN Planning 45
- 3.1 Delete-Free Task Insertion HTN Problems 47
- 3.2 Solving $\text{HTN}^{1+\text{pre}}_{1+\text{eff}}$ Problems is NP-hard 50
- 3.3 Showing $\text{HTN}_{+\text{eff}}$ Problems are in NP 53
  - 3.3.1 Decomposition Trees 54
  - 3.3.2 Forming a witness to the solvability of an HTN problem 58
  - 3.3.3 Bounding the breadth of the witness tree 61

3.3.4 Bounding the height of the witness tree 63
3.4 Discussion 65

4 HTN Planning via translation to PDDL 68
4.1 Basic Definitions and Notation 69
4.1.1 Classical Planning 69
4.1.2 TSTN Planning 70
4.1.3 Using TSTN Planners for Classical Planning 71
4.2 Translating TSTN to Classical 72
4.2.1 Preliminaries 72
4.2.2 Translating Operators 73
4.2.3 Translating Methods 74
4.2.4 Translating Planning Problems 75
4.3 Properties 75
4.4 Implementation and Experiments 77
4.4.1 Towers of Hanoi 78
4.4.2 Blocks World 81
4.4.3 Office Delivery 83
4.5 Discussion 84

5 Conclusions 86

List of Figures

2.1 DHTN(D, s_0, tn_0) A simple decomposition based HTN planner. . . . 13

2.2 PHTN(D, s_0, tn_0) A simple progression based HTN planner. 21

2.3 TODHTN(D, s, tn) A procedure to explore the TOD problem space. 32

2.4 Syntactic containment: Every $\leq_1$-stratifiable problem is both $\leq_r$-stratifiable and $\leq_1$-ordered; every $\leq_r$-stratifiable problem is $\leq_r$-ordered; and every $\leq_1$-ordered problem problem is $\leq_r$-ordered. 43

2.5 Complexities of plan-existence for propositional STRIPS and HTN planning with various restrictions on method structures. Edges represent known polynomial encodings. Completeness results for regular HTN problems and STRIPS are provide by Erol et al. [1996] and Bylander [1994] 43

3.1 Complexity of plan-existence for propositional delete-free STRIPS and HTN planning with various restrictions (k-length-plan-existence is NP-complete in all cases). Arrows represent subclass relationships. The STRIPS results are from Bylander [1994]; the other results are new. 46

4.1 FF's CPU time in the Towers of Hanoi domain, with and without the translated domain knowledge. Each data point is FF's average CPU time on 100 runs. The asterisk is explained in the text. 80

4.2 FF's CPU time in the Blocks World, with and without the translated domain knowledge. Each data point is FF's average CPU time on 100 randomly generated planning problems. The asterisks are explained in the text. 80

4.3 FF's CPU time in the Office domain, with and without the translated domain knowledge. In the graph at left, the number of packages is fixed at 40 and the number of rooms varies. In the graph at right, the number of rooms is fixed at 40 and the number of packages varies. Each data point is FF's average CPU time on 100 randomly generated planning problems. 82

List of Abbreviations

DHTN	Decomposition HTN planning algorithm
FF	Fast-Forward planner
HTN	Hierarchical Task Network planning
LAMA	Landmark planner
PDDL	Planning Domain Definition Language
PHTN	Progression HTN planning algorithm
SHOP2	Simple Hierarchical Ordered Planner (version 2)
STRIPS	Standford Research Institute Problem Solver language
TIHTN	Task-Insertion HTN planning
TODHTN	Totally-Ordered Decomposition HTN planning algorithm
TOPHTN	Totally-Ordered Progression HTN planning algorithm

Chapter 1: Introduction

Despite being more expressive than classical planning, Hierarchical Task Network (HTN) planning is often viewed as just a way to encode human knowledge to solve classical planning problems faster. The last decade has seen a number of advancements in classical planning that reduce planning time, increase plan quality, and generally broaden classical planning's applicability. However, many of these techniques to not directly transfer, since unlike classical planning, HTN planning is in general undecidable, so no sound and complete algorithm will terminate on all problems. The objective of this thesis is to study the underlying computational complexities of using search techniques to solve HTN planning problems. The work includes:

- Developing heuristic search algorithms that terminate on large subsets of HTN problems, including the syntactically-identifiable sets of HTN problems which are decidable [Erol et al., 1996]

- Classifying what subsets of HTN problems these algorithms terminate on, describing the expressivity of those sets and providing run-time bounds of the algorithms on those sets.

- Establishing the complexity of delete-free HTN planning, which will have significant impact on the development of domain independent heuristics for HTN planning.

- Providing a proof of concept, showing that domain independent heuristics and an understanding of the search space can lead to more effective search techniques.

This work is organized into three chapters, described below.

Chapter 2 presents four algorithms for HTN planning. Two of these algorithms can be directly adapted for efficient heuristic search. The other two rely on an AND/OR search which is less amenable to many heuristic search techniques. Each algorithm is associated with a class of problems, identifiable syntactically in polynomial time, on which the algorithm is guaranteed to terminate. The chapter provides upper and lower complexity bounds on these classes, including new results for previously known decidable classes.

Chapter 3 examines the complexities of delete-free HTN planning. In classical planning, the ability to find a plan in domains with positive preconditions and no negative effects in polynomial time is fundamental to most classical planning heuristics. I prove that almost all propositional delete-free HTN planning is NP-complete. This means that, unless $\mathsf{P} = \mathsf{NP}$, that any polynomial time heuristic based on solving delete-free HTN problems must also relax the semantics of HTN planning.

Chapter 4 presents a technique for translating a subset of HTN problems into classical planning problems (in specific, PDDL [Fox and Long, 2003]). This means that, even without a dedicated HTN heuristic, any PDDL planner's heuristic becomes an HTN planning heuristic when used on the translated problem. Moreover, since most classical planners are guaranteed to terminate, they can be used to prove whether any translatable problem is solvable or not. My experiments show that for the subset of problems that are translatable, planning-via-translation can be an effective way to find HTN plans.

1.1 Related Work

Kutluhan Erol's seminal work on HTN planning theory [Erol et al., 1996] formalized HTN planning, as well as provided several important complexity and decidability results. The most significant result in Erol et al.'s work is that HTN planning is in general undecidable by showing how to encode the intersection of any two context-free grammars as an HTN problem. Erol shows a number of syntactic restrictions of HTN planning are decidable. Chapter 2 discusses these decidable fragments further.

Geier and Bercher [2011] show that you can regain decidability by relaxing what it means to be a solution to an HTN problem. If one allows arbitrary operators to be inserted into the final plan, then any solution to a problem can be mimicked by an acyclic decomposition of the initial task network plus operator insertion. This means that a Task-Insertion HTN (TIHTN) planner would only need to explore acyclic decompositions of the initial task (which are finite) and if those decompositions can be made executable through task insertion (which is decidable). Chapter 3 discusses task-insertion HTN planning further, showing that finding a plan in TIHTN problems with positive preconditions and effects can be done in polynomial time.

SHOP2 [Nau et al., 2003] is the most well known HTN planning implementation, known for both efficiency and flexibility. SHOP2 plans for tasks in the order they are to be executed, and can be configured to use either depth- or breadth-first search. Normally run in depth-first search mode, SHOP2 requires that the author order the methods such that every search path terminates or ends in a solution. HTNPBP [Sohrabi et al., 2009] is a preference-based HTN planner based on SHOP2, which performs a best-first search over state-based preferences, and it has the same method-ordering requirements that SHOP2 has.

Elkawkagy et al. [2012] developed the Landmark-Aware heuristic HTN planner. The landmark-aware heuristic analyzes what tasks must occur in any solution to that plan, then orders the plans by which one has the fewest required tasks, and thus requires the least effort to turn into a solution. The planner itself is a refinement-based HTN planner which maintains a set of partially refined plans ordered by its heuristic. At each iteration the planner removes the most promising plan it has found, produces a set of possible refinements of the plan.

Bercher and Biundo [2012] developed a heuristic for preference based planning in a hybrid HTN and partially-ordered causal link framework. Given a task network, they take the subset consisting of just the primitive operators, and translate that into a classical planning problem. They then run a Relaxed Graphplan variant to estimate the best quality solution to this problem.

1.2 HTN Planning

In this section, we present a propositional HTN planning formalism, using the notation presented in [Geier and Bercher, 2011].

It will be important for us to have a notation for the *restriction* of a function or relation to some subset of its domain. For this, we will use a *bar notation* that is defined as follows. For a binary relation $R \subseteq A \times A$, the restriction of R to any $X \subseteq A$ is

$$R|_X = \{(p_1, p_2) \in R \mid p_1, p_2 \in X\}.$$

Similarly, for a function $f : P \to Q$, the restriction of f to any $X \subseteq P$ is

$$f|_X = \{f(p) = q \ \mid \ p \in X\}.$$

Hierarchical Task Network (HTN) planning is the problem of decomposing an initial task into a sequence of executable steps. A task is an activity to be accomplished labeled with a task name, which is a proposition symbol, and will be either primitive, corresponding to a concrete action, or compound, representing an abstract activity.

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 partial order over T.
- $\alpha : T \rightarrow X$ is a mapping from the task symbols to a finite set of 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 [Erol et al., 1996]. We say a task network $(T, \prec, \alpha)$ is *equivalent* to another task network $(T', \prec', \alpha')$ if there is an isomorphism $\phi : T \rightarrow T'$ such that $\forall t_0, t_1 \in T \mid (t_0 \prec t_1) \Leftrightarrow (\phi(t_0) \prec' \phi(t_1))$ and $\forall t \in T \mid \alpha(t_0) = \alpha'(\phi(t_0))$. We refer to the set of all task networks over a set of task names X as TN_X.

An *HTN domain* is a tuple (L, C, O, M), where L is a function-free first order language, C is a finite set of compound task names, O is a finite set of primitive task names, and $M \subseteq C \times TN_{C \cup O}$ is a set of methods over C and O. The proposition symbols in C, O and L are pairwise disjoint.

Each primitive task name $o \in O$ is associated with a *planning operator*, which is the triple $(prec(o), add(o), del(o))$ where $prec(o)$ is a propositional formula over L and $add(o)$ and $del(o)$ are disjoint subsets of literals from L. The combined set of free variables of $prec(o)$, $add(o)$, and $del(o)$ are the *parameters* of o.

Note that the semantic models for the operators in O forms an implicit *state transition function* for the planning domain:

$$\gamma : 2^L \times O \rightarrow 2^L,$$

where:

- A state is any subset of L. The finite set of states in a planning domain is denoted as 2^L in the above definition of γ;

- $\gamma(s, o)$ is defined iff $s \models \mathit{prec}(o)$; and
- $\gamma(s, o) = (s \setminus \mathit{del}(o)) \cup \mathit{add}(o)$.

We will refer to the particular γ defined by a domain D as γ_D, or just γ if it is clear from the context.

We call a task network *primitive* if $\alpha(t) \in O$ for every $t \in T$. Otherwise, the task network is *non-primitive.*

We can *decompose* a non-primitive task network $\mathit{tn}_1 = (T_1, \prec_1, \alpha_1)$ if there is a non-primitive task $t \in T_1$ such that $\alpha(t) \in C$ and has a corresponding method $m = (\alpha(t), (T_m, \prec_m, \alpha_m)) \in M$. More formally, we define the notion of *task decomposition* as follows. Assume without loss of generality that $T_1 \cap T_m = \emptyset$. Then the decomposition of tn_1 by m into a task network tn_2 (written $\mathit{tn}_1 \xrightarrow{t,m}_D \mathit{tn}_2$) is given by:

$$\begin{aligned}
T_1' &\coloneqq T_1 \setminus \{t\}\,; \\
T_2 &\coloneqq T_1' \cup T_m; \\
\prec_2 &\coloneqq \prec_1|_{T_1'} \\
&\quad \cup \prec_m \\
&\quad \cup \{(t_1, t_2) \in T_1' \times T_m \mid (t_1, t) \in \prec_1\} \\
&\quad \cup \{(t_2, t_1) \in T_m \times T_1' \mid (t, t_1) \in \prec_1\}\,; \\
\alpha_2 &\coloneqq \alpha_1|_{T_1'} \cup \alpha_m; \\
\mathit{tn}_2 &\coloneqq (T_2, \prec_2, \alpha_2)\,.
\end{aligned}$$

If there is a finite sequence of task decompositions from $\mathit{tn}_1 \rightarrow_D \mathit{tn}_2 \rightarrow_D \ldots \rightarrow_D \mathit{tn}_n$, then we write $tn_1 \rightarrow_D^* tn_n$.

An *HTN planning problem* is a tuple (D, s_0, tn_0), where $D = (L, C, O, M)$ is an HTN domain, $s_0 \in 2^L$ is a state in D, and $tn_0 = (\{t_0\}, \emptyset, \{(t_0, x_0)\})$ is the initial task network containing a single task x_0.

A task network tn is *executable* in a state s_0 for domain D if tn is primitive and there exists some total ordering (consistent with $\prec$ in tn) over the tasks $t_1, \ldots, t_n$ and the sequence of states $s_1, \ldots, s_n$ that arise from applying the primitive tasks (i.e., actions) $t_1, \ldots, t_n$ in that order in the initial state s_0: i.e.,

$$\forall_{i=0 \ldots n-1} \gamma(s_i, \alpha(t_{i+1})) = s_{i+1}$$

We say that tn^* is an *HTN solution* to a planning problem $P = (D, s_0, tn_0)$ if tn^* is executable in s_0 and $tn \rightarrow_D^* tn^*$. In this paper, we are concerned with two HTN decision problems: *plan-existence*, for whether a problem has any solution, and *k-length-plan-existence*, for whether a problem has a solution of k or fewer operators.

1.3 Classical Planning

Classical planning is the problem of finding a sequence of operators that takes one from the starting state to a goal state. Unlike HTN planning, there are no tasks, and operators can be applied to the state whenever the state supports their preconditions.

Formally, a classical planning problem is the tuple (L, O, s, g), where L is a set of propositions, O is a set of planning operators defined as above, $s \subset L$ is the initial state, and g is the goal, specified as a propositional formula. L and O implicitly form a state transition function γ defined identically to the γ in the previous section.

A planning problem is solvable if either $s \models g$ or if there is some operator o such that $\gamma(s, o) = s'$ and s' is solvable. We refer to this specific formulation of classical planning as STRIPS [Fikes and Nilsson, 1972]. This is an implicit graph search problem. Bylander [1994] and Erol et al. [1995] show that STRIPS is PSPACE-complete.

Both plan-existence and k-length-plan-existence are the same for both HTN planning and STRIPS.

Chapter 2: HTN Problem Spaces

Unlike HTN planning, classical planning is decidable [Erol et al., 1995]. Thus it is possible to guarantee the termination of many classical planners, through the use of loop-checking tests to prevent the planner from generating infinite cyclic paths through a finite search space.

In contrast, HTN planning is only semi-decidable [Erol et al., 1996], and every sound and complete HTN planner has an infinite set of problems on which it will never return. Although some syntactic restrictions of HTN planning are fully decidable, efficient loop-detection tests have not yet been developed, and there is a gulf between the classes of HTN problems that are known to be decidable, and the classes of problems on which current HTN planners can guarantee termination.

Part of the reason for this gulf is that much less is understood about the search spaces of HTN planners than those of classical planners. Different kinds of HTN planners have different ways of combining plan generation with task decomposition—and the structure of the problem space can vary greatly depending on how those things are done.

The following sections show how to search for solutions to HTN problems while retaining broad decidability guarantees:

1. I characterize HTN planning as a search of a *problem space* in which each node is an HTN planning problem. I present four different classifications of HTN problem spaces:

    - *Decomposition Space* (DS), which UMCP [Erol et al., 1994] and the Landmark-Aware HTN Planner [Elkawkagy et al., 2012] search;
    - *Progression Space* (PS), which SHOP and SHOP2 [Nau et al., 1999, 2003] and HTNPBP [Sohrabi et al., 2009] search;
    - *Total Order Partition* decomposition and progression spaces (TODS and TOPS), two new kinds of problem spaces for which there are not yet any existing planners.

2. Each section provides sufficient (and in one case, necessary) conditions to guarantee that each kind of problem space will be finite. These conditions can be evaluated up-front to see if an HTN planning problem has a finite problem space. For each kind of problem space, I show that there are no broader conditions for finiteness that look only at possible decomposition of tasks.

3. For each of the finiteness conditions, I provide upper and lower complexity bounds, showing both the expressivity of problems in those classes, and refining current known complexities.

4. I describe simple loop-detection tests that can be added to HTN planners that will guarantee termination when the problem space is finite.[1] These tests

[1]State-space classical planners often use a loop-detection test of the form "have we seen this

appear to be applicable to a wide variety of HTN planning problems, and their use will allow HTN planners to terminate in cases where they would not otherwise do so.

5. The HTN-to-PDDL translation algorithm described in Chapter 4 requires a user-specified upper bound on the HTN recursion depth, and the translation is correct only when this bound is sufficiently high. By characterizing the translation as a mapping from HTN progression spaces into classical state spaces, one can compute a correct bound automatically whenever the finiteness conditions in Item 2 are satisfied.

6. I show that TODS and TOPS are finite for strictly broader classes of problems than DS and PS, and I provide new sound-and-complete HTN planning algorithms for TODS and TOPS. The algorithms are guaranteed to terminate whenever the problem space is finite.

2.1 Decomposition Problem Spaces

The definition of HTN solvability leads to a natural definition of a problem space as a directed graph, where nodes are task networks, and edges are decompositions from one task network to another. The initial task network of a problem forms the root of the graph, and it is solvable if and only if there is a path in the graph from it to a primitive executable task network.

state before?" In contrast, the loop-detection tests here are basically "have we seen this problem before?"

Formally, for an HTN domain $D = (L, C, O, M)$, the directed graph (V, E) is the *decomposition problem space* of an HTN problem (D, s_0, tn_0) if and only if (V, E) is the minimal graph containing tn_0 such that $tn \in V$ and $tn \rightarrow_D tn'$ implies that $tn' \in V$ and $(tn, tn') \in E$.

```
Input: $D = (L, C, O, M)$ - an HTN domain
Input: $(s_0, tn_0)$ - an initial state and task network
$V \leftarrow Fringe \leftarrow \{tn_0\}$;
while $Fringe \neq \emptyset$ do
    Choose and remove some $tn \in Fringe$;
    if $tn$ is primitive and $(D, s_0, tn)$ is executable then
        return $tn$;
    $children \leftarrow \{tn' | tn \rightarrow_D tn'\}$;
    $Fringe \leftarrow Fringe \cup (children \setminus V)$;
    $V \leftarrow V \cup children$;
return fail;
```

Figure 2.1: DHTN(D, s_0, tn_0) A simple decomposition based HTN planner.

Algorithm 2.1 (DHTN) shows a simple decomposition space HTN planner. DHTN starts off with the initial task network tn, and maintains a set of known decompositions V and a fringe of unexpanded decomposition.[2] At every iteration, it chooses some task network in its fringe. If tn is primitive and executable, DHTN returns tn. Otherwise, it removes tn from the fringe and adds tn's immediate decompositions. If at any time the fringe is exhausted, DHTN returns failure.

The computation $Fringe \leftarrow Fringe \cup (children \setminus V)$ guarantees that DHTN will never add a previously visited task network to the fringe. This is a simple loop-detection test that can be added to other HTN planning algorithms.

Given an HTN problem P, DHTN is sound, complete, and terminating for

[2] *Fringe* and V functionally correspond to the open and closed sets of A*, respectively.

any problem for which the decomposition problem space is finite. However, if the problem space is infinite, DHTN's completeness depends on how it picks elements out of the fringe (with some acceptable choices being first-in-first-out or taking the network with the fewest tasks). If the problem has no solution and the problem space is infinite, then DHTN will not terminate no matter how elements are chosen from the fringe.

Relation to other work.

The HTN decomposition problem space formalizes the spaces searched by existing planners such as UMCP [Erol et al., 1994] and Elkawkagy's landmark-aware HTN planner. Unlike DHTN, neither UMCP nor Elkawkagy's planner check to see if they have already expanded a problem before, so a finite decomposition space is not enough to ensure their termination.

2.1.1 Decidability under decomposition

For an HTN domain $D = (L, C, O, M)$, a *non-recursive* task name is one which has a finite *k-level-mapping.* k is a partial function from $C \cup O \to \mathbb{Z}^+$ defined via induction:

- For all $o \in O$, $k(o) = 0$.
- For $c \in C$, if there is a finite number $n \in \mathbb{Z}^+$ such that n is the greatest k-level of any subtask of c in M, then $k(c) = n + 1$.

If a task in a task network has a k-level, any decomposition of that task replaces it with a set of tasks which have a lower k-level. Since we can only repeat this a finite number of times, this leads to one of the first results of [Erol et al., 1996]:

Theorem 2.1 *Let $P = (D, s_0, tn_0)$ be an HTN planning problem with every task in tn_0 has a finite k-level mapping. Then the decomposition problem space for P is finite.*

So the decomposition problem space is finite for non-recursive problems, but limited recursion is fine as long as it does not increase the size of the task network. It turns out we can syntactically identify every problem for which the problem space is finite. To this end, we say that a task network tn is $\leq_1$-*stratifiable* if there exists a total preorder $\leq_1$ on the reachable task names of tn such that if c is a reachable task name of tn and $(c, (T, \prec, \alpha))$ is a method, then either:

- If the task network contains only one task ($T = \{t\}$), then the task name must not be on a higher stratum ($\alpha(t) \leq_1 c$)
- Otherwise, all task names must be on a lower stratum ($\forall_{t \in T} \alpha(t) <_1 c$).

Example 2.2 *Let $P = (D, s_0, tn_0)$ where tn_0 contains the single task r, $D = (L, C, O, M)$, $C = \{r, s\}$, $O = \{a, b\}$, and:*

$$M = \begin{cases} (r, (\{x_1\}, \emptyset, \alpha(x_1) = s)) \\ (r, (\{x_1\}, \emptyset, \alpha(x_1) = a)) \\ (s, (\{x_1\}, \emptyset, \alpha(x_1) = r)) \\ (s, (\{x_1, x_2\}, \emptyset, \{\alpha(x_1) = b, \alpha(x_2) = b\})) \end{cases}$$

Then there is a two level $\leq_1$-stratification, where since r and s decompose to one another they must be on the same level, and a and b can be on the lower level together, since they are primitive.

Theorem 2.3 *Let $P(D, s_0, tn_0)$ be an HTN planning problem. Then the decomposition problem space for P is finite if and only if tn_0 is $\leq_1$-stratifiable.*

Proof. ($\Leftarrow$) If there exists a $\leq_1$-stratification, then every decomposition either produces a task network of the same size, or replaces a task with a set of tasks whose task names are in a lower level. Since the stratification is finite, the number of times repeated decomposition can increase the size of the task network is also finite.

($\Rightarrow$) If there is no $\leq_1$-stratification, then the transitive $\leq_1$ constraints are inconsistent. This means there exists a task name c and method $m = (c, tn_m)$ with more than one subtask such that $c \leq_1 \ldots <_1 c$, and so c is a reachable subtask of tn_m. Since c is a reachable subtask of tn_0, we can decompose tn_0 into a task network containing c, then repeatedly use the decomposition chain going through m to create a task network of arbitrary size. Since any finite problem space has a bound on the size of task networks appearing in it, the problem space for P is infinite. $\square$

Given an HTN domain D, we can find a $\leq_1$-stratification of a task network $tn = (T, \prec, \alpha)$ in time polynomial in D by performing a topological sort of the task names given the constraints. If this procedure fails, the decomposition space is infinite. If it succeeds, it returns a stratification. If the height of the stratification is

h and the maximum number of tasks in a method is b, then the largest task network in the decomposition space of an HTN problem (D, s, tn) is bounded by $|T| \cdot b^h$.

2.1.2 Complexity of $\leq_1$-stratifiable problems

It is not immediately clear how expressive $\leq_1$-stratifiable HTN planning is, nor how long we can expect a decomposition based planner to take to solve those problems. In this section, we provide upper and lower complexity bounds for $\leq_1$-stratifiable problems. In particular, $\leq_1$-stratifiable HTN planning is at least as expressive as STRIPS-style planning.

Theorem 2.4 *Deciding plan-existence and k-length-plan-existence $\leq_1$-stratifiable problems is in* NEXPTIME *(non-deterministic exponential time).*

Proof. First, show that we can compile any $\leq_1$-stratifiable domain to a domain with a k-level mapping in polynomial time. Let t be a task such that $t \leq_1 t_1 \leq_1 t_2 \leq_1 \ldots \leq_1 t_n$. Then by adding a method (t, tn) to the domain for every method (t_i, tn) and then removing every method (t, tn') where tn' contains one of the t_i, we transform the domain to one with the same set of solutions, but now t can exist on a separate strata then $t_1, \ldots, t_n$. If we repeat this process, we end up with a problem that has the same set of solutions as the original, but is now has a k-level mapping with no recursion.

With no recursion, we can only decompose the task network at most m^k times, where m is the size of the largest method and k is the minimum k-level mapping. So searching for a solution to $\leq_1$-stratifiable problems is in NEXPTIME. □

We can view the NEXPTIME result as the gap between a blind search of the progression space and using a perfect heuristic to guide the search. If DHTN is given oracle access to a perfect heuristic (one that estimates the exact size of the minimum solution to any problem), then always selecting the problem from its fringe with the minimum heuristic value will lead DHTN to terminate in exponential time on $\leq_1$-stratifiable problems. On the other hand, if it selects poorly and expands every node in the decomposition space before terminating, it may take double-exponential time.

Erol et al. [1996] shows that HTN planning with k-level problems is decidable, but does not give a lower complexity bound. By mapping STRIPS into a $\leq_1$-stratifiable problem, we can give a PSPACE lower bound:

Theorem 2.5 *Deciding plan-existence and k-length-plan-existence for $\leq_1$-stratifiable problems is* PSPACE*-hard.*

Proof. Let $P = (L, O, s, g)$ be a STRIPS problem. Here we encode P as a $\leq_1$-stratifiable HTN problem with the same set of solutions. We will take advantage of the fact the minimum length solution to any classical plan does not visit the same, and so must be of length $2^{|L|} - 1$ or less.

Let O' be a set of primitive tasks containing tasks for every operator in O, plus:

- A no-op task, whose operator has no preconditions or effects
- A task o_g, whose operator has a precondition of g and no effects.

Let C contain the set of task names $a_1, \ldots, a_{|L|}$, and M contain:

- $(a_1, tn_{\texttt{no-op}})$, where $tn_{\texttt{no-op}}$ contains only the no-op task.

- A method (a_1, tn_i) for each primitive task $o_i \in O' \setminus \{o_g\}$, where tn_i contains the task o_i.

- A method (a_j, tn_j) for $2 \ldots |L|$, where tn_j contains a_{j-1} followed by itself.

Let the initial task network tn contain the tasks $a_{|L|}$ followed by g'. Then the $\leq_1$-stratifiable problem $P' = ((L, C, O', M), s, tn)$ can be decomposed into any sequence of $2^{|L|} - 1$ no-ops and operators from O, followed the operator o_g which tests for the goal condition g. Thus any to P implies a solution to P', and vice-versa.

Since STRIPS is PSPACE-complete [Bylander, 1994], $\leq_1$-stratifiable HTN planning is PSPACE-hard. $\square$

This gives a loose bound on the complexity of planning for le_1-stratifiable problems. The NEXPTIME upper bound is based on the exponential relationship between the height of a $\leq_1$-stratifiable problem's minimum stratification and the maximum size of a task network reachable under decomposition. However, many HTN planning problems have a fixed small stratification height, and problem instances only vary by the size of the state space and the number of tasks in the initial task network. Tighter bounds can be found by parameterizing the class based on the height of a problem's smallest stratification. For a constant c, we say a problem is $\leq_1^c$-stratifiable if there exists a $\leq_1$-stratification with a height of c or less. This matches with common practice in HTN planning, since most domains have a fixed (and small) stratification (see Section 2.4).

Then the largest task network reachable under decomposition is bounded by a polynomial of degree $c+1$. Later on, Section 3.3 shows that finding an executable decomposition of a problem of a fixed size is in NP. Meanwhile, Erol et al. [1996] shows that merely ordering the leaves of this tree is NP-hard, leading to a tight and much reduced bound for $\leq_1^c$-problems:

Theorem 2.6 *For fixed c, then deciding plan-existence or k-length-plan-existence for $\leq_1^c$-stratifiable HTN problems is* NP*-complete.*

2.2 Progression Problem Spaces

Decomposition spaces are not the only way to solve HTN planning problems. Let $P = (D, s, tn)$ be an HTN planning problem, where $D = (L, C, O, M)$ and $tn = (T, \prec, \alpha)$. If there exists a task $t \in T$ with no predecessors (one such that $\forall_{t' \in T} t' \not\prec t$) and its operator $\alpha(t) \in O$ is applicable in s, then we can *progress* the problem from (D, s, tn) to the problem $P' = (D, \gamma_D(s, \alpha(t)), tn \setminus \{t\})$ (where the notation $tn \setminus \{t\}$ simply means removing any occurrence of t from T, $\prec$, and α). If there exists an unconstrained task $t \in T$ which is non-primitive ($\alpha(t) \in C$), then any decomposition $tn \xrightarrow{t,m}_D tn'$ is a valid progression of (D, s, tn) to (D, s, tn'). We write progression as $P \xrightarrow{t}_P P'$.

Intuitively, a progression interleaves a decomposition with imposing an executable total order over the primitive tasks. The following theorem establishes an equivalence between these two paradigms:

Theorem 2.7 *An HTN problem P is solvable if and only if P is executable or there* *exists a P' such that $P \rightarrow_P P'$ and P' is solvable.*

Progression, then, also leads to a natural definition of the problem space as a directed graph. The *progression problem space* of an HTN problem (D, s_0, tn_0) is the minimal directed graph (V, E) containing (s_0, tn_0) such that $(s, tn) \in V$ and $(s, tn) \rightarrow_P (s', tn')$ implies that $(s', tn') \in V$ and $((s, tn), (s', tn')) \in E$.

```
**Input**: $D = (L, C, O, M)$ - an HTN domain
**Input**: $(s_0, tn_0)$ - an initial state and task network
$V \leftarrow Fringe \leftarrow \{(s_0, tn_0)\}$; $E \leftarrow \emptyset$;
**while** $Fringe \neq \emptyset$ **do**
    Choose some $(s, tn) \in Fringe$;
    **if** *tn is primitive and* $(D, s, tn)$ *is executable* **then**
        **return** path in $(V, E)$ from $(s_0, tn_0)$ to $(s, tn)$;
    $children \leftarrow \{(s', tn') | (s, tn) \rightarrow_P (s', tn')\}$;
    $Fringe \leftarrow (Fringe \setminus \{(s, tn)\}) \cup (children \setminus V)$;
    $V \leftarrow V \cup children$;
    $E \leftarrow E \cup \{((s, tn), (s', tn')) | (s', tn') \in children\}$;
**return** fail;
```

Figure 2.2: PHTN(D, s_0, tn_0) A simple progression based HTN planner.

PHTN (Algorithm 2.2) is a simple progression-based HTN planner. PHTN maintains a directed graph of HTN problems reachable from the initial problem and expands problems from the leaves of this graph. When it encounters a primitive executable problem, it returns the entire sequence of progressions from initial problem to its solution.

PHTN is sound, complete, and terminating for any HTN problem which has a finite progression problem space. The computation $Fringe \leftarrow (Fringe \setminus \{(s, tn)\}) \cup (children \setminus V)$ guarantees that PHTN will not add previously visited task networks to the fringe. This loop-detection test can be added to other HTN planning algorithms. But as with DHTN, if the problem space is infinite and there is no solution, PHTN will never return.

Relation to other work. Progression problem spaces provide an implicit formalization of the problem space behind several existing HTN planning works such as SHOP2 [Nau et al., 2003], HTNPBP [Sohrabi et al., 2009] and the HTN-PDDL translation in the previous section. As with decomposition-based planners, finiteness does not guarantee termination; e.g., neither SHOP2 nor HTNPBP check if they have already expanded a problem.

2.2.1 Decidability under progression

Given an HTN domain D, suppose that the task network for every method in a domain consisted of a set of primitive tasks and at most one non-primitive task which is constrained to come after them. Erol et al. [1996] call this a *regular* domain and prove a decidability result for regular HTN domains. We adapt their result to our progression problem spaces as follows:

Theorem 2.8 *Given a regular HTN domain D, any HTN problem $P = (D, s_0, tn_0)$ has a finite progression problem space.*

The proof follows Erol et al.'s results. Intuitively, given any problem $P = (D, s_0, tn_0)$ with a regular domain, no progression can increase the number of non-primitive tasks in a task network. Furthermore, any primitive task introduced along with a non-primitive task must be progressed out of a task network before the non-primitive task can be decomposed. Thus, this bounds the size of the task networks in the progression problem space of P.

We can extend this class of decidable problems using the same stratification technique we used for decomposition. A task network tn is *$\leq_r$-stratifiable* if there exists a total preorder $\leq_r$ on the reachable task names of tn such that for every method $(c, (T, \prec, \alpha))$ with a reachable task name c:

- 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 $t \in T$, $\alpha(t) <_r c$.

If an HTN planning problem P's task network is $\leq_r$-stratifiable, then any progression P replaces a task with at most one task of the same level, with the rest occurring at a lower level. Since the lower level tasks are constrained to come before this task, they must be progressed out of the task network before this task can be decomposed. Since the stratification is finite, this gives us a bound on the maximum size of the network, and produces our next finiteness result:

Theorem 2.9 *Given an HTN problem $P = (D, s_0, tn_0)$, if tn_0 is $\leq_r$-stratifiable, then the progression problem space of P is finite.*

What happens to the problem space if there is no stratification? Unlike with decomposition, now both the structure of the transition function and the set of methods can affect which problems are in the problem space. This limits our result on when the problem space must be infinite:

Theorem 2.10 *Given an HTN problem $P = (D, s_0, tn_0)$ such that every problem in the domain is solvable and there is no $\leq_r$-stratification of the reachable subtasks of tn_0, then the progression problem space of P is infinite.*

Proof. Given that there is no $\leq_r$-stratification, the $\leq_r$-constraints must be inconsistent, meaning there must be two reachable task names (not necessarily distinct) such that $b <_r c$ and $c \leq_r b$. So there is a method $m = (c, (T, \prec, \alpha))$, and two tasks $t_1, t_2 \in T$ such that $t_1 \not\prec t_2$ and $\alpha(t_2) = b$. Since c is a reachable task from tn_0 and every problem in P is solvable, there is a series of progressions such that c progresses to a task network containing itself. Since we did not need to progress t_1 out of tn_0 in order to expand t_2, we can use this loop to create a task network of arbitrary size. Thus the progression problem space of P is infinite. □

Since $\leq_r$-stratifiable is a strict broadening of the $\leq_1$-stratifiability definition, if a task network is $\leq_1$-stratifiable, it is also $\leq_r$-stratifiable. Like $\leq_1$-stratifications, we can find $\leq_r$-stratifications with a topological sort of the reachable task names of a task network. For an HTN problem P with the task network $tn = (T, \prec, \alpha)$ and a stratification of tn of height h and a largest method size of b, a task in a stratum can contribute at most 1 plus b times the bound of the strata below it to the maximum size of a task network reachable under progression. This gives a total bound of $|T| \cdot \sum_{i=0}^{h-1} b^i = |T| \cdot \frac{1-b^h}{1-b}$ for the maximum size of a task network in the progression problem space of P.[3]

Relation to other work.

The HTN to PDDL translation algorithm in Chapter 4 essentially maps the progression problem space into a classical domain. The algorithm adds a fixed num-

[3]One can do much better than that by directly inspecting the stratification, but that is beyond the scope of this thesis.

ber of identifiers, specified by the user, to represent the structure of the task network. In order for the translation to be correct, there must be more identifiers available than there are tasks in the largest task network encountered under progression. If the initial task network is $\leq_r$-stratifiable, we can use the bound in the previous paragraph instead of asking the user.

2.2.2 Identifiability of finite progression spaces

Since we could identify the exact set of problems which have a finite decomposition space, there is a natural question about whether we can identify more problems which have a finite progression space.

Theorem 2.10 says that if the initial task network is not $\leq_r$-stratifiable, and every problem in the domain is be solvable, then the progression space is infinite. To say that every problem in the domain is solvable is equivalent to the following two conditions:

- Every operator is applicable in every state (i.e., $\forall_{s \in 2^L, o \in O}$, $s \models pre(o)$).
- Every task has a primitive decomposition (transitively, not necessarily an immediate decomposition).

We can check the second condition recursively:

- Every primitive task has a primitive decomposition.
- Every task which has a method where all the tasks have a primitive decomposition also has a primitive decomposition.

Once all tasks with a primitive decomposition are marked, the rest have no primitive decomposition, and so are *trivially unsolvable.* Since any task network with a task that can't be decomposed into primitive network is unsolvable, we can preprocess the domain, removing any trivially unsolvable tasks and methods that refer to them.

This lets us identify every problem with a finite progression space that can be identified without looking at the preconditions and effects of the operators:

Theorem 2.11 *Let P be an HTN planning problem and let P' be the HTN planning problem where all of the trivially unsolvable tasks of P are removed. If P' is not $\leq_r$-stratifiable, then the following holds:*

There does not exist a function $Q(P)$ which, without examining P's operators, returns true if and only if P has a finite progression space.

Proof. Let $P = (D, s, tn)$ and $P' = (D', s, tn)$ be HTN problems, where D' is D without trivially unsolvable tasks, such that P has a finite progression space and P' has no $\leq_r$-stratification.

Let P_U and P'_U be P and P' where every operator's precondition is set to *true* and every effect is removed. Then by Theorem 2.10, P'_U has an infinite progression space. Since adding methods and tasks cannot make a problem's infinite progression space finite, P_U also has an infinite progression space.

So if $Q(P)$ returns true and $Q(P_U)$ returns false, then Q must inspect the operators of P and P_U, since that is their only difference. $\square$

2.2.3 Complexity of $\leq_r$-stratifiable problems

We can solve $\leq_r$-stratifiable problems with a non-deterministic search of the progression space that keeps track of only the current task network and state. Given that the height of the stratification gives an exponential bound on the size of task networks encountered under progression, this provides an upper bound on the complexity of planning for $\leq_r$-stratifiable problems.

Theorem 2.12 *Deciding plan-existence and k-length-plan-existence $\leq_r$-stratifiable problems is in* EXPSPACE.

As with $\leq_1$-stratifiable problems, we can parameterize the class of $\leq_r$-stratifiable problems based on the stratification height. For a constant c, we say a problem is $\leq_r^c$-stratifiable if there exists a $\leq_r$-stratification with a height of c or less. The class of *regular* HTN problems from Erol et al. [1996] is identical to the class of $\leq_r^1$-stratifiable problems.

Theorem 2.13 *For fixed c, deciding plan-existence and k-length-plan-existence $\leq_r^c$-stratifiable problems is* PSPACE*-complete.*

Proof. *Containment.* For fixed c, task networks in the progression space of $\leq_r^c$-stratifiable problems are bound in size by a polynomial of degree c. Non-deterministic search then only takes polynomial space.

Hardness. Given that for $c > 0$, the class of $\leq_r^c$-stratifiable problems includes all regular HTN problems, for which plan-existence is PSPACE-complete [Erol et al., 1996]. □

2.3 Total Order Partition Problem Spaces

This section describes two new problem spaces, based on our formulation of DS and PS. The new problem spaces will allow us to define new HTN planning algorithms which terminate for a broader class of HTN problems, as described in the subsequent section.

2.3.1 Total Order Partitions

Let $P = (D, s_0, tn)$ be an HTN planning problem where $tn = (T, \prec, \alpha)$ is its task network. Then the sequence of task networks $\langle (T_0, \prec_0, \alpha_0), \ldots, (T_k, \prec_k, \alpha_k) \rangle$ is a *total order partition* of tn if:

- Their union equals tn:
  - $T = \bigcup_{i=0}^{k} T_i$
  - $\prec = \bigcup_{i=0}^{k} \prec_i$
  - $\alpha = \bigcup_{i=0}^{k} \alpha_i$
- They are disjoint and tasks between networks are ordered, i.e. for any T_i and T_j such that $i < j$:
  - $T_i \cap T_j = \emptyset$
  - $\forall_{t_i \in T_i, t_j \in T_j} t_i \prec t_j$

A total order partition is a serialization of the task network into smaller problems, which we can attempt to solve the smaller problems sequentially without interactions from other tasks:

Theorem 2.14 *Let $P = (D, s_0, tn)$ and let $\langle tn_0, \ldots, tn_k\rangle$ be a total order partition of tn. Then P is solvable iff there exists a sequence of states $s_1, \ldots, s_{k+1}$ such that for all $i \leq k$ (D, s_i, tn_i) has a solution with an ending state of s_{i+1}.*

Proof. ($\Leftarrow$) If the sequence of partitions is solvable, then each tn_i decompose into some primitive task network tn'_i. Applying these decompositions to the corresponding tasks in tn gives you a primitive task network tn' which will have an ending state of s_{i+1}.

($\Rightarrow$) Suppose P has a primitive executable decomposition tn'. Since it is executable, there is a total order over the tasks of tn. Since decomposition preserves ordering, we can split that sequence into a solution for the partition. □

Given that $\prec$ is a consistent partial order, there will be a unique *longest total order partition* from which none of the reduced problems can be further reduced. If the longest total order partition contains only a single task newtork, we call that a *trivial partition.*

TOD and TOP problem spaces.

Total order partitions give us two AND/OR problem spaces for HTN planning, one defined over decomposition and one over progression.

Given an HTN domain $D = (L, C, O, M)$, the *total-order decomposition (TOD) problem space* for an HTN problem $P = (D, s_0, tn_0)$ is the minimal directed labeled graph (V, E) containing (s_0, tn_0) such that for every $(s, tn) \in V$:

- If tn has only a trivial total order partition and $tn \rightarrow_D tn'$, then (s, tn') is also in V with the edge $((s, tn), 0, (s, tn')) \in E$.
- Otherwise, let $\{tn_1, \ldots, tn_k\}$ be the longest total order partition of tn. Edges will point to reduced problems, labeled with where in the sequence the reduce problem lies. Then:
  - $(s, tn_1) \in V$ with the edge $((s, tn), 1, (s, tn_1))$.
  - For $i < k$, if there exists an edge $((s, tn), i, (s', tn_i)) \in E$ and (s', tn_i) has a solution with an ending state of s'', then there exists $(s'', tn_{i+1}) \in V$ with the edge $((s, tn), i + 1, (s'', tn_{i+1}))$.

For each problem $P = (s, tn)$ in the TOD problem space with a non-trivial total order partition $tn_1, \ldots, tn_k$, we label P's outgoing edges with the integer corresponding to its reduced task networked (edges corresponding to decomposition are given an arbitrary label of 0). If there is an edge $((s, tn), i, (s_i, tn_i)) \in E$ for $i > 1$, then by the definition there must also be a state s_{i-1} and an edge $((s, tn), i - 1, (s_{i-1}, tn_{i-1})) \in E$ such that (s_{i-1}, tn_{i-1}) has a solution with an ending state of s_i. So if there is an edge $((s, tn), k, (s_k), tn_k)) \in E$ such that (D, s_k, tn_k) has a solution with an ending state of s_{k+1}, then there must be a chain of states that solves the partition, and so P has a solution with an ending state of s_{k+1}.

The *total-order progression (TOP) problem space* is defined similarly to the TOD problem space, replacing decomposition with progression.

2.3.2 Search in TOD and TOP spaces

We now describe two new HTN-planning algorithms, called TODHTN and TOPHTN, which perform an AND/OR search over the TOD and TOP problem spaces, respectively.

Algorithm 2.3 shows a high-level description of the TODHTN procedure. TODHTN maintains a set of variables as PHTN - a directed but now edge-labeled graph (V, E) of HTN problems, a set of HTN problems $(Fringe)$, and a new map X, which maps HTN problems to a set of known possible end states. TODHTN then begins a two phase iterative process of *selecting* a node from the fringe to expand, then *propagating* the consequences through the graph.

Every iteration of TODHTN selects a problem (s, tn) from the fringe and examines its task network. Nothing is added to the graph in this phase, but instead TODHTN marks new edges and ending states to add later during the propagation phase. If the task network has a non-trivial total order partition $\langle tn_1, \ldots \rangle$, TODHTN marks the edge from (s, tn) to its first reduced child (s, tn_1). If the network is non-primitive, it marks the edges to all the immediate decompositions of tn. Otherwise the network is primitive, TODHTN marks all the possible ending states (if any).

The propagation phase itself is split into two parts: adding edges, and propagating ending states. When TODHTN adds an edge from one problem to its child, it checks to see if the child problem is already in the graph. If not, it adds the problem to the fringe. If the child is already in the current graph, TODHTN marks all of the child's known endings states for propagation to its parent (noting the edge

```
Input: $D = (L, C, O, M)$ - an HTN domain
Input: $(s, tn)$ - an initial state and task network
$V \leftarrow Fringe \leftarrow \{(s, tn)\}$;
$X(s, tn) \leftarrow \emptyset$;
while $Fringe \neq \emptyset$ & $X(s, tn) = \emptyset$ do
    // Pick and expand a fringe node
    Choose and remove some $(s', tn') \in Fringe$;
    if there is a total order $tn_1 \prec \ldots \prec tn_n$ over $tn'$ then
        Insert $((s', tn'), 1, (s', tn_1))$ into $NewE$;
    else if $tn'$ is nonprimitive then
        Insert $((s', tn'), 0, (s', tn''))$ into $NewE$ for every decomposition $tn' \rightarrow_D tn''$;
    else
        $(s', tn')$ is primitive, so add $((s', tn'), 0, s_e)$ to $NewX$ for every ending state
        $s_e$ of $(s', tn')$;
    while $NewE \neq \emptyset$ & $NewX \neq \emptyset$ do
        // Add edges, collect end states
        foreach $(v_1, k, v_2) \in NewE$ do
            if $v_2 \notin V$ then
                Insert $v_2$ into $Fringe$ and $V$;
            else
                For each $s_e \in X(v_2)$, add $(v_1, k, s_e)$ to $NewX$;
            Insert $(v_1, k, v_2)$ into $E$;
        $NewE \leftarrow \emptyset$;
        // Propagate end states
        while $NewX \neq \emptyset$ do
            Choose and remove some $((s_p, tn_p), k, s_e)$ from $NewX$;
            Let $tn_1, \ldots, tn_n$ be the longest total order partition over $tn_p$;
            if $0 < k < n$ then
                Insert $((s_p, tn_p), k + 1, (s_e, tn_{k+1}))$ into $NewE$;
            else if $s_e \notin X(s_p, tn_p)$ then
                Insert $s_e$ into $X(s_p, tn_p)$;
                foreach $(v, j, (s_p, tn_p)) \in E$ do
                    Insert $(v, j, s_e)$ into $NewX$;
if $X(s, tn) \neq \emptyset$ then
    return the preorder traversal of a subgraph of $(V, E)$ showing a solution;
else
    return $FAILURE$;
```

Figure 2.3: TODHTN(D, s, tn) A procedure to explore the TOD problem space.

label).

When processing a new ending state s_e to propagate, if s_e is a solution to the interior part of a partition ($0 < k < n$), then that state is the start state for a next child in the partition, (s_e, tn_{k+1}), and TODHTN marks the edge to that problem for later addition. Otherwise, s_p is an end state for (s_p, tn_p), and if it is an ending state that TODHTN didn't already know about, it propagates it to the parents problems of (s_p, tn_p).

Since TODHTN follows the definitions of the TOD problem space in expanding nodes from the fringe, it is a sound HTN planner. If the TOD problem space is finite, then TODHTN is complete and will eventually terminate when it runs out of nodes from the fringe to expand and ending states to propagate. If the TOD problem space is infinite, then TODHTN's completeness depends upon how it chooses nodes out of the fringe (such as FIFO). If the problem is unsolvable and the problem space infinite, no matter how TODHTN chooses it will never return.

TOPHTN is defined nearly identically to TODHTN, substituting progression for decomposition.

2.3.3 Decidability under problem partitioning

We note that, since total order partitions split task networks into smaller task networks without introducing new tasks, the TOD and TOP problem spaces of a problem are finite if the decomposition or progression problem spaces are finite, respectively.

TOD and TOP are also finite for a strictly broader class of problems. Let $tn_1, \ldots, tn_k$ be the longest total-order partition of a task network tn for some number k. We say tn is $\leq_1$-*ordered*, if each tn_i, for $i = 1, \ldots, k$, in the longest total-order partition of tn is either a singleton or $\leq_1$-stratifiable. An HTN method (c, tn) is $\leq_1$-*ordered* if the task network tn is $\leq_1$-ordered. If every method in a domain is $\leq_1$-ordered, we call that domain $\leq_1$-ordered, and if an HTN planning problem's domain and initial task network is $\leq_1$-ordered, then so is the problem.

Theorem 2.15 *If P is $\leq_1$-ordered, it has a finite TOD problem space.*

Proof. Let $P = (D, s_0, tn_0)$ be a $\leq_1$-ordered HTN planning problem. Since the initial task network tn_0 is $\leq_1$-ordered by the condition of the theorem, it is either a singleton, $\leq_1$-stratifiable, or tn_0 has a non-trivial total order partition.

The proof proceeds by showing that every problem in TOD problem space of P has a task network tn, produced by decomposition over tn_0 in s_0, that satisfies at least one of the following conditions:

- *tn is a singleton.* Consider a node with a singleton task network. Its task is either primitive, i.e., the node has no children, or the node is non-primitive, i.e., by decomposition, its children each correspond to some method in D.

- *tn matches to the initial network (tn_0) or some HTN method's task network.* we have already shown the first case. For the second case, consider a non-singleton node (s, tn) with a task network tn that corresponds to some method in D, which means that tn can be produced by applying an HTN method to a nonprimitive task in a state. Then since that method is $\leq_1$-ordered, tn either has a non-trivial total order partition, or tn is $\leq_1$-stratifiable. In the first case, any children of (s, tn) have singleton task networks or are $\leq_1$-stratifiable.

- *tn is $\leq_1$-stratifiable:* Note that in this case, we already know there are only a finite number of problems reachable from (s, tn) in the TOD problem space.

Thus, since the number of states in D is finite, the TOD problem space of P is finite. □

We define $\leq_r$-ordered problems similarly, replacing $\leq_1$-stratification with $\leq_r$-stratification. The finiteness of the TOP problem space of $\leq_r$-ordered problems can be proved similarly.

We can prove an infiniteness theorem for TOD and TOP problem spaces which is similar to the progression space theorem. Here we pick the TOP space, since it provides a way to show the same for TOD as well:

Theorem 2.16 *Let $P = (D, s_0, tn_0)$ be an HTN problem where D has a method (c, tn) where c is a reachable task name of tn_0 and tn is not $\leq_r$-ordered. If every problem in D is solvable, then the TOP space of P is infinite.*

Proof. Since c is a reachable task name from tn_0 and every problem in D is solvable, we can reach some problem p using the method (c, tn). tn is not $\leq_r$-ordered, so its longest total order partition $\langle tn_1, \ldots, tn_k \rangle$ has some non-singleton task network tn_i (possibly equal to tn) which is not $\leq_r$-stratifiable.

From Theorem 2.10 we know that there is a chain of progressions that can produce a task network of arbitrary size in the progression space of any problem

(D, s, tn_i). Let t_1 be the first progressed task in this chain. Since the partition of tn was maximal, there is some other task t_2 that is not constrained to come before or after t_1. Since the chain of progressions did not need to progress t_2 out of the task network, at no point in the sequence of progressions is there a non-trivial total order partition of a problem.

This means that the chain of progressions behaves identically in the TOP space as it does in the progression space, and so the TOP problem space of P is infinite.

□

The proof of infiniteness for TOD spaces proceeds similarly, since once a task network has no non-trivial total order partition, no sequence of decompositions can restore it. As with progression spaces, $\leq_1$-ordered and $\leq_r$-ordered are the broadest class of finite problems identifiable without inspecting the state transition function.

2.3.4 Complexity of $\leq_1$- and $\leq_r$-ordered problems

Notice that by the proof of Theorem 2.15, every node in the TOD space is either a singleton, matches a task network, or is $\leq_1$-stratifiable. This lets us bound complexity of planning on $\leq_1$-stratifiable problems:

Theorem 2.17 *plan-existence and k-length-plan-existence are in* NEXPTIME *for $\leq_1$-ordered problems.*

Proof. Let $P = (D, s, tn)$ be a $\leq_1$-ordered problem. Let TODHTN' be TODHTN modified so that in the main loop, if it picks a $\leq_1$-stratifiable problem, it fully solves the problem for each possible end state. An exponential number of NEXPTIME (Theorem 2.4) operations is still in NEXPTIME.

So TODHTN' only adds non-$\leq_1$-stratifiable problems to the fringe, and by the proof of Theorem 2.15, there are at most $O\left(2^{|D|}\right)$ of them. Given that a problem is only added to the fringe once and we only propagate end states from a given problem $2^{|L|}$ times, there are at most an exponential number of iterations of the main loop, each finishing in NEXPTIME. So $\leq_1$-ordered planning is in NEXPTIME.

□

By similar reasoning, $\leq_r$-ordered planning is in EXPSPACE.

The $\leq_1$-ordered and $\leq_r$-ordered problems both include what Erol et al. [1996] calls *totally ordered problems.* A problem is *totally ordered* if there is a total order over the initial task network and over every method's task network. Where Erol et al. prove that planning for totally ordered problem is decidable via a dynamic programming argument, we can repurpose the proof of Theorem 2.15 again to provide a bound on the size of TOD and TOP spaces for totally ordered problems:

Theorem 2.18 *If P, where $D = (S, C, O, M, \alpha)$, is totally ordered, then there are at most $1+2^{|L|} \cdot (|M| + |C| + |O|)$ vertices in both the TOD and TOP problem spaces of P.*

Given a bound B on the number of vertices, TOD- and TOPHTN maintain $O\left(B \cdot \left(B + 2^{|L|}\right)\right)$ space for the graph and map of vertices to ending states. Given that a vertex is only added to the fringe once and we only propagate end states from a given vertex $2^{|L|}$ times, TOD- and TOPHTN run in $O\left(B^2 \cdot 2^{|L|}\right)$ time, which matches

Erol et al. [1996]'s EXPTIME upper bound for totally-ordered HTN problems.

Now we move on to establishing an EXPTIME lower bound on the complexity of totally-ordered problems. In games such as chess, checkers, and certain versions of Go, deciding whether the first player can force a win is EXPTIME complete [Fraenkel and Lichtenstein, 1981, Robson, 1983, 1984]. In the remainder of this section, we show how to encode these games as totally-ordered HTN planning problems.

We restrict ourselves to two-player, perfect information, zero sum games who have a polynomial number of different moves and whose mechanics are no harder than PSPACE. Specifically, checking a move's applicability, applying a move, and checking win and lose conditions must all be in PSPACE. We additionally require that, when these checks are translated into STRIPS, which is PSPACE-complete, every action has an inverse, so that $\forall_{s \in 2^L} \gamma\left(\gamma\left(s, m\right), m^{-1}\right) = s$.

With invertible moves, we can create HTN tasks which function as *checks* which ensure some condition holds, and then returns to the state the task started in. We do this via palindromes. Suppose $P = (L, O, s, g)$ is a STRIPS problem with invertible actions. To represent this as a check, we create an HTN domain with the same operators as P plus an operator o_g which checks the goal g. We add one compound task t to the domain with one method that contains only o_g, and a method for each of the operators $o \in O$ whose task networks contain o_i followed by t followed by o_i^{-1}. Thus any expansion of t drives down to a state that satisfies g, and then reverses all the changes it made along the way.

We will assume two players, labeled A and B, each with:

- Invertible move operators $a_1, \ldots, a_j$ and $b_1, \ldots, b_k$ which can only be applied if they are valid moves.
- Two checks for whose turn it is ($turn_A$ and $turn_B$).
- Checks for winning, losing, and their negations ($lost_A$, won_A, *not* $lost_A$, etc.).
- A check for each of player B's actions on whether they are inapplicable (b_i *inapplicable*, etc).

We can now move on to encoding game tree search as a totally-ordered HTN planning problem:

Theorem 2.19 *For totally-ordered problems, plan-existence is* EXPTIME*-complete.*

Proof. Let $D = (L, C, O, M)$ be an HTN domain with the tasks described above of an EXPTIME-complete game, and let s be the game state to evaluate. We will show how to encode game tree search as a totally-ordered HTN problem, so that the problem only has a solution if player A can force a win.

First, we add a new compound task to the domain, *play*, which will implement the game tree evaluation. The methods for *play* will be recursive, with one termination condition: a method for *play* that contains the check '*won A* ' as its only task. A key property to note is that all the methods we introduce for '*play*' will return to the same state they start in, letting us evaluate multiple potential moves from the same state.

For each of player A's potential move a_i, we add the method $(play, tn_{ai})$, where tn_{ai} contains the sequence of tasks $\langle turn_A, not\ lost_A, a_i, play, a_i^{-}1 \rangle$. If it is player A's turn and A has not lost, the HTN planner can pick any of the methods corresponding to one of A's moves, apply the move (if it is applicable), evaluate '*play*' in the new state, and revert the move. Successfully evaluating this move means that player A can force a win in this state by picking the move a_i.

We add one method $(play, tn_B)$ for evaluating all of player B's moves, where tn_B contains the sequence of tasks $\langle turn_B, not\ won_B, try\ b_1, \ldots, try\ b_k \rangle$. For each compound task '$try\ b_i$,' we add two methods: one with the check 'b_i *inapplicable*,' and one with the sequence of tasks $\left\langle b_i, play, b_i^- 1 \right\rangle$.

So if it is player B's turn and B has not won, the above method for '*play*' is the only method which can be applied past its first two tasks. The method then cycles through each of B's valid moves using '$try\ b_i$ to skip the applicable ones. For each applicable move, it applies the move, evaluates '*play*' in the resulting state, and reverts the move. This ensures that all of B's potential moves have been evaluated, and that player A can for a win for each of them.

Let tn_{play} be the task network containing just the task '*play*'. Then the existence of a play for the HTN problem $P = (D, s, tn_{play})$ implies that player A can force a win from state s. Since the game was EXPTIME-complete, P was a totally-ordered HTN problem, totally-ordered HTN planning is in EXPTIME, totally-ordered HTN planning is EXPTIME-complete. □

So totally-ordered HTN planning is EXPTIME complete, meaning that TOD-HTN and TOPHTN are both asymptotically optimal for totally-ordered HTN planning.

This also lends evidence that TODHTN and TOPHTN decide a larger set of problems than DHTN and PHTN, even though we have not shown a strict separation. That is for a fixed constant c, any $\leq_1^c$-stratifiable or $\leq_r^c$-stratifiable problem can be encoded as a totally-ordered planning problem (a strict subset of $\leq_1^c$- and $\leq_r^c$-ordered problems), but, unless $\mathsf{PSPACE} = \mathsf{EXPTIME}$, not vice versa.

2.4 Practical Considerations

As a result of the theoretical analyses presented in this chapter, a practical question arises:

> *Do existing HTN planning domains satisfy the finiteness criteria of HTN problem spaces?*

The SHOP2 distribution is distributed with five different HTN domain models, namely Logistics, Blocks-World, Depots, Towers of Hanoi and Robot-Navigation.[4] All of these are both $\leq_r$-stratifiable and $\leq_1$-ordered, with Logistics also being $\leq_1$-stratifiable. This suggests that typical HTN domains models (even complicated ones such as Blocks-World and Towers of Hanoi, which encode optimal problem-solving strategies) will most likely satisfy our finiteness criteria.

Thus, our theoretical and empirical analyses over HTN problem spaces suggest the polynomial-time computable conditions for the finiteness of the HTN problem spaces, and the loop-detection tests based on those finiteness conditions, will be practically useful in at least two ways:

[4] `http://www.cs.umd.edu/projects/shop/`

- Authors of HTN domain descriptions will be able to use our theoretical finiteness conditions as guidelines so as to obtain guarantees on termination.

- HTN planning systems incorporating the search algorithms provided in this chapter can determine whether conditions for finiteness are satisfied during planning. If the conditions are satisfied, the planner can freely choose any search procedure without worrying about termination, and therefore, completeness. Otherwise, the planner can choose to fall back onto a search strategy like breadth-first search that guarantees completeness. This is useful for systems such as SHOP2 where depth-first search is empirically much faster than breadth-first search.

2.5 Discussion

This chapter provides a classification of HTN problem spaces, that provides a better understanding of the conditions under which HTN planning algorithms can safely terminate (see Figure 2.4 for a summary), as well as a number of complexity results and relationships for these conditions (see Figure 2.5). Although this work is primarily theoretical, it may potentially lead to several practical benefits.

First, there is reason to believe that loop-checking tests based on the finiteness criteria will be widely applicable (see Section 2.4), and it should be straightforward to incorporate them into several existing HTN planners, although the implementation is left future work. This will enable those planners to backtrack in cases where they otherwise might never return, thereby enabling the planners to solve a larger

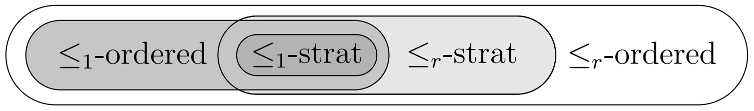

Figure 2.4: Syntactic containment: Every $\leq_1$-stratifiable problem is both $\leq_r$-stratifiable and $\leq_1$-ordered; every $\leq_r$-stratifiable problem is $\leq_r$-ordered; and every $\leq_1$-ordered problem problem is $\leq_r$-ordered.

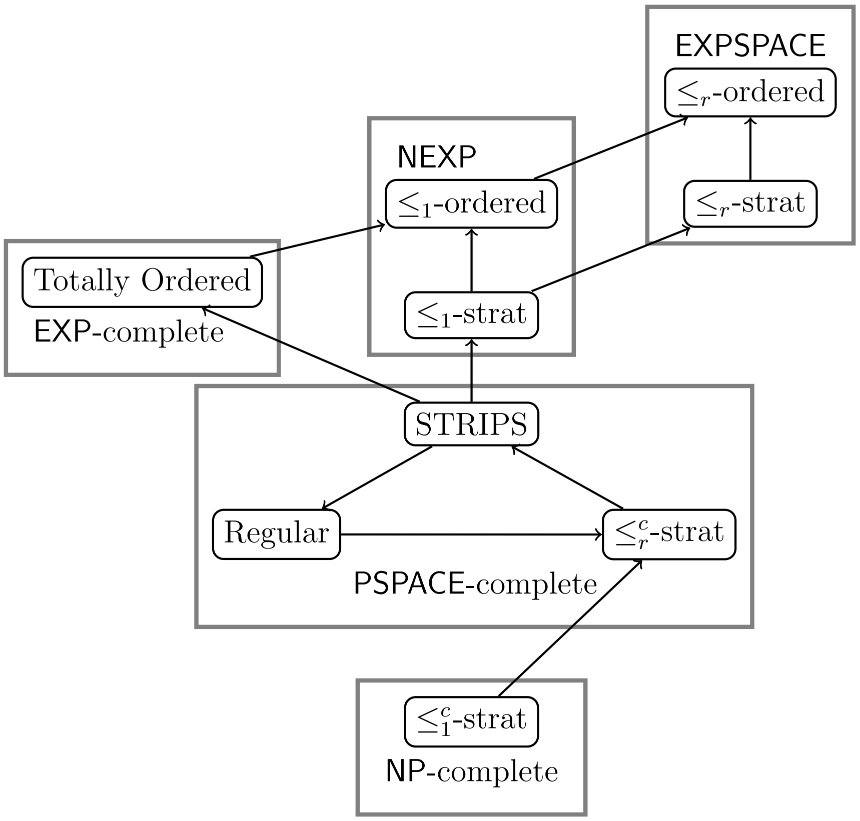

Figure 2.5: Complexities of plan-existence for propositional STRIPS and HTN planning with various restrictions on method structures. Edges represent known polynomial encodings. Completeness results for regular HTN problems and STRIPS are provide by Erol et al. [1996] and Bylander [1994]

class of problems. It might also make some planners less sensitive to the order in which the HTN methods appear in the planner's input, making it easier to write HTN domain descriptions.

Second, this work provides a useful bound for the HTN-to-PDDL translation algorithm in Chapter 4. That algorithm requires an upper bound on the size of the largest task network in the HTN progression space, and if the user supplies too low a bound, then the translation algorithm will produce a classical planning domain that is not a correct translation of the original HTN planning domain. By computing the correct bound automatically, it is easier to guarantee a correct translation.

Third, the chapter presents new HTN planning algorithms that will terminate in cases where previous HTN planning algorithms would not terminate (not even with the incorporation of the loop-checking tests described above). In our future work, I hope to implement this algorithm and test its performance against existing HTN planners such as SHOP2 and Elkawkagy et al.'s Landmark-Aware HTN planner.

This chapter examines the algorithms and search complexities of HTN planning. However, effective search requires an informed and efficient heuristic. The next chapter sets a hard theoretical boundary on how informed a polynomial time HTN heuristic can be.

Chapter 3: Complexities for Delete-Free HTN Planning

Planning has been shown to be theoretically intractable in general. Bylander [1994] showed that even the simplest interesting variant of classical planning is PSPACE-complete. Hierarchical Task Network (HTN) planning is even harder: depending on the particular variant, the complexity can be anywhere from EXPTIME to undecidable [Erol et al., 1996].

To combat the complexity of classical planning, modern classical planners use efficiently computable state-based heuristics that often work very well in practice [Helmert, 2006, Hoffmann and Nebel, 2001, Bonet and Geffner, 2001, Nguyen and Kambhampati, 2001]. The most influential among these is arguably the *Relaxed Planning Graph* heuristic used in the FF planner [Hoffmann and Nebel, 2001], which solves the propositional delete-free version of the given problem in polynomial time, and computes a heuristic value based on that solution. Relaxed planning-graph heuristics have since been developed for a variety of purposes, e.g., probabilistic planning [Yoon et al., 2007, Teichteil-Königsbuch et al., 2010], propositional landmark generation [Richter and Westphal, 2012], metric planning [Hoffmann, 2003].

In this paper, for propositional delete-free HTN planning, we prove results about the complexity of two well-known decision problems, plan-existence and k-

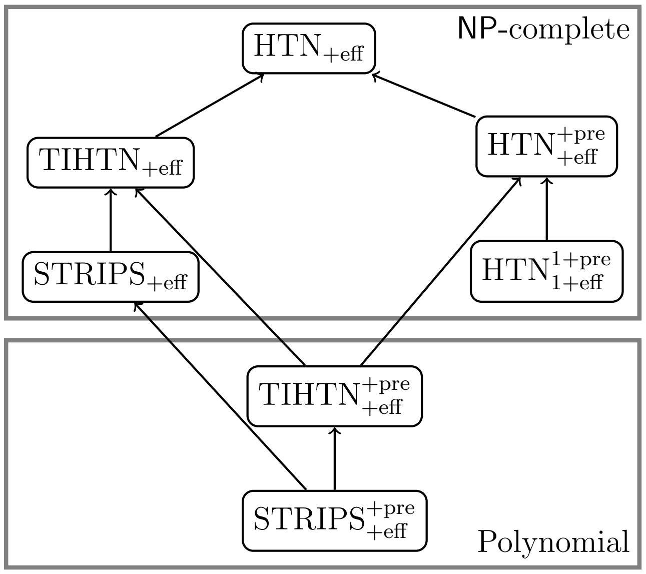
NP-complete
$\text{HTN}_{+\text{eff}}$
$\text{TIHTN}_{+\text{eff}}$
$\text{HTN}^{+\text{pre}}_{+\text{eff}}$
$\text{STRIPS}_{+\text{eff}}$
$\text{HTN}^{1+\text{pre}}_{1+\text{eff}}$
$\text{TIHTN}^{+\text{pre}}_{+\text{eff}}$
$\text{STRIPS}^{+\text{pre}}_{+\text{eff}}$
Polynomial

Figure 3.1: Complexity of plan-existence for propositional delete-free STRIPS and HTN planning with various restrictions (k-length-plan-existence is NP-complete in all cases). Arrows represent subclass relationships. The STRIPS results are from Bylander [1994]; the other results are new.

length-plan-existence, under various conditions.

Fig. 3.1 summarizes the results, using the following notation. TIHTN is propositional HTN planning with *task insertion* (see Section 3.1 and [Geier and Bercher, 2011]); "+pre" (resp. "+eff") means all preconditions (resp. effects) are positive; "1+pre" (resp. "1+eff") means at most one positive and no negative preconditions (resp. effects). Here is how the results bear on the feasibility of relaxation-based search heuristics for HTN planning:

- Even for very restricted cases, delete-free propositional HTN planning is NP-complete. Thus unless P=NP, there is no direct analogy of Relaxed GraphPlan for HTN problems.

- If the HTN planning semantics is modified to allow task insertion and all of the preconditions and effects are positive, then plan-existence is polynomial-time computable. Thus, it may be possible to use this or other relaxations to develop search heuristics for HTN planning.

In this chapter, we consider only propositional *delete-free* planning problems, where operators contain only positive effects (i.e., $\forall_{o \in O} del(o) = \emptyset$). Deferring to Bylander [1994], we refer to this restricted class of problems as members of $\text{HTN}_{+\text{eff}}$. When problems are further restricted to contain only operators with positive preconditions, we say these problems belong to $\text{HTN}^{+\text{pre}}_{+\text{eff}}$. In the highly restricted case where both the preconditions and effects of operators contain at most a single positive literal, we say these problems belong to $\text{HTN}^{1+\text{pre}}_{1+\text{eff}}$.

3.1 Delete-Free Task Insertion HTN Problems

Before we get to delete-free HTN problems, we shall first consider delete-free versions of a variant of HTN planning: HTN Planning with Task Insertion (TIHTN) [Geier and Bercher, 2011]. In TIHTNs, a problem is still modeled in terms of an initial state and a task network that needs to be decomposed, but insertion of tasks is now allowed without requiring them to be inserted by the decomposition of a compound task that is present in the task network. As Geier and Bercher show, this feature of TIHTNs relaxes HTN planning enough to regain decidability of plan existence even in cases when the original HTN problem remains undecidable.

The following theorem shows that in $\text{TIHTN}^{+\text{pre}}_{+\text{eff}}$, plan existence is polynomial-time computable:

Theorem 3.1 *If $P = (D, s_0, tn_0)$ (where $D = (L, C, O, M)$) is a Task Insertion*

HTN planning problem with positive preconditions and effects (TIHTN$^{+pre}_{+eff}$), then plan-existence for P is decidable in time $O\left(|O|^2 + |M|^2\right)$.

Proof. We iteratively apply operators from O to s_0 until we reach a fixed point state s where no more operators are applicable, much like Relaxed GraphPlan (taking $O\left(|O|^2\right)$ time).

Then the following algorithm iterates through the list of methods at most $|M|$ times finding a solution for at least one non-primitive task in all but its last iteration, starting from the non-primitive tasks.

1. For every primitive task $o \in O$ where $\gamma(s, o) = s$, mark o as solvable.
2. Iterate through the methods in M. If $m = (c, tn)$ is a method such that all the tasks names in tn are marked as solvable, mark c as solvable.
3. Repeat line 2 if it marked any new task names as solvable.
4. Return $TRUE$ if all task names in tn_0 are solvable, return $FALSE$ otherwise.

Since at least one method is marked in every pass, this takes $O\left(|O| + |M|^2\right)$ time, resulting in an overall time complexity of $O\left(|O|^2 + |M|^2\right)$. □

We shall now establish lower bounds on complexities of both plan-existence and k-length-plan-existence for the remaining delete-free TIHTN planning classes.

Firstly, we note that (delete-free) TIHTN problems can be encoded as (delete-free) HTN problems as follows: given a TIHTN domain $D = (L, C, O, M)$, we add for every $t \in C$ and $o \in O$ a method to M that decomposes t into a pair of subtasks $\langle o, t \rangle$. Similarly, we can also show that (delete-free) STRIPS problems can be encoded as (delete-free) TIHTN problems by simply adding a dummy operator o with the goal as its precondition and no effects and letting the initial task network consist of o.

Since we know that plan-existence for $\text{STRIPS}_{+\text{eff}}$ and k-length-plan-existence for both $\text{STRIPS}^{+\text{pre}}_{+\text{eff}}$ and $\text{STRIPS}_{+\text{eff}}$ is NP-hard [Bylander, 1994], it follows immediately from the encoding from STRIPS to TIHTN problems that plan-existence for $\text{TIHTN}_{+\text{eff}}$ and k-length-plan-existence for both $\text{TIHTN}^{+\text{pre}}_{+\text{eff}}$ and $\text{TIHTN}_{+\text{eff}}$ are also NP-hard.

Now using the encoding from TIHTN to HTN problems, we can similarly lower bound the complexities for some HTN planning problem classes. In particular, we can show that plan-existence for $\text{HTN}_{+\text{eff}}$ is NP-hard and that k-length-plan-existence for both $\text{HTN}_{+\text{eff}}$ and $\text{HTN}^{+\text{pre}}_{+\text{eff}}$ are NP-hard.

Table 3.1: Summary of results from Section 3.

Problem	plan-existence	k-length-plan-existence
$\text{TIHTN}^{+\text{pre}}_{+\text{eff}}$	P	NP-hard
$\text{TIHTN}_{+\text{eff}}$	NP-hard	NP-hard
$\text{HTN}^{+\text{pre}}_{+\text{eff}}$	-	NP-hard
$\text{HTN}_{+\text{eff}}$	NP-hard	NP-hard

Table 3.1 summarizes the complexity results from this section. One thing yet to be done is to estimate the complexity of solving $\text{HTN}^{+\text{pre}}_{+\text{eff}}$problems. As we shall see in the following section, while the task insertion variant of this problem ($\text{TIHTN}^{+\text{pre}}_{+\text{eff}}$) is solvable in polynomial time, $\text{HTN}^{+\text{pre}}_{+\text{eff}}$problems are much harder to

solve.

3.2 Solving $\text{HTN}^{1+\text{pre}}_{1+\text{eff}}$ Problems is NP-hard

We begin our analysis on delete-free HTN planning problems by focusing on a restricted case, where

- Every method is totally ordered
- Every method is *regular*, such that non-primitive tasks only occur as the last task in the method.
- The methods are *acyclic*, meaning there are only a finite number of solutions to the initial problem.
- Every operator has at most one (positive) literal in its precondition and at most one (positive) proposition in its effect.

We call the class of such HTN planning problems as $\text{HTN}^{1+\text{pre}}_{1+\text{eff}}$ problems for the rest of the paper. In the following theorem, we establish the NP-hardness of plan-existence for $\text{HTN}^{1+\text{pre}}_{1+\text{eff}}$ (and thus, for $\text{HTN}^{+\text{pre}}_{+\text{eff}}$) by showing a reduction from CNF-SAT:

Theorem 3.2 *Plan existence for HTN^{1+pre}_{1+eff} planning is* NP*-hard.*

Proof. Let $E = e_1 \wedge e_2 \wedge \ldots \wedge e_n$ be a CNF-SAT formula, where each conjunct is a disjunction over a set of variables $v_1, \ldots, v_m$ and their negations.

To give an encoding, we need to present a delete-free HTN planning problem where any solution implies a satisfying assignment for E, and no solution implies

E is unsatisfiable. The encoding of E is the HTN domain $D = (L, C, O, M)$ and problem $(D, \emptyset, tn_0)$, all given below.

Let the set of propositions L consist of two symbols for each variable, $v_i\text{-}true$ and $v_i\text{-}false$ representing a true and false assignment to v_i, respectively.

Let the set of operators O consist of four operators for each variable v_i, two for setting the value of the variable and two for checking its truth or negation:

- An operator $set\text{-}v_i\text{-}true$, with

$$prec(set\text{-}v_i\text{-}true) = true,$$
$$add(set\text{-}v_i\text{-}true) = \{v_i\text{-}true\},$$
$$del(set\text{-}v_i\text{-}true) = \emptyset.$$

- An operator $set\text{-}v_i\text{-}false$, with

$$prec(set\text{-}v_i\text{-}false) = true,$$
$$add(set\text{-}v_i\text{-}false) = \{v_i\text{-}false\},$$
$$del(set\text{-}v_i\text{-}false) = \emptyset.$$

- An operator $check\text{-}v_i\text{-}true$, with

$$prec(check\text{-}v_i\text{-}true) = v_i\text{-}true,$$
$$add(check\text{-}v_i\text{-}true) = del(check\text{-}v_i\text{-}true) = \emptyset.$$

- An operator $check\text{-}v_i\text{-}false$, with

$$prec(check\text{-}v_i\text{-}false) = v_i\text{-}false,$$
$$add(check\text{-}v_i\text{-}false) = del(check\text{-}v_i\text{-}false) = \emptyset.$$

C and M consist of a set of non-primitive tasks with methods that first choose a variable assignment for each variable, and then checks that each conjunct of the expression is satisfied.

For each variable v_i, C contains the non-primitive task *set*-v_i. For $i < m$, we introduce two methods for *set*-v_i: one which calls *set*-v_i-*true* and then *set*-v_{i+1}, and another which calls *set*-v_i-$false$ and then *set*-v_{i+1}. For *set*-v_m, we introduce two methods as above but which call *check*-e_1 after setting the variable true or false (instead of calling *set*-v_{i+1}).

For each conjunct e_i of E, C contains the task *check*-e_i. Since e_i is a disjunction of literals, let $l_1, \ldots, l_k$ be the disjuncts of e_i. For each literal l_j, we encode a method for *check*-e_i: if l_j is of the form $\neg v_l$ for some variable v_l, then the method calls *check*-v_l-$false$ followed by *check*-e_{i+1}. Otherwise, l_j is of the form v_l, and the method calls *check*-v_l-*true* followed by *check*-e_{i+1}. The methods for *check*-e_m omit the call to check the next expression.

The initial task network tn_0 contains a single task, *set*-v_1. Any primitive decomposition of the tn_0 must first call *set*-v_i-*true* or *set*-v_i-$false$ (but not both) for each variable, and then check that one literal is true for each conjunct in E. Thus there exists a solution to the HTN problem iff there is a satisfying assignment for the variables in E.

Since the encoding is linear with respect to the length of E and CNF-SAT is NP-hard, delete-free HTN planning is NP-hard. □

Any of the first three restrictions on method structure is enough to place a regular HTN planning problem in a decidable fragment of the language [Erol et al., 1996]. This leaves only two obvious syntactic restrictions that would make a delete-free HTN problem solvable in polynomial time: either restrict the initial task network to be primitive, or restrict all operators to have zero effects.

3.3 Showing $\text{HTN}_{+\text{eff}}$ Problems are in NP

Here we show that if there is a solution of length k to a delete-free HTN planning problem, then there exists a polynomial size witness, verifiable in polynomial time, proving that there is exists a solution of size k or smaller. This places both plan-existence and k-length-plan-existence in NP for delete-free HTN planning.

The outline of the proof is as follows: We present *decomposition trees* [Geier and Bercher, 2011], which can be used as a witness that a task network is derivable from the initial network, and these trees can be verified in time polynomial in the size of the tree. We then digress to show that deciding whether a problem has a solution when the primitive tasks that change the state are fixed in advance is in NP. Since solutions in delete-free domains can only change the state a polynomial number of times, this lets us use a decomposition tree of polynomial width as part of the witness to the solvability of $\text{HTN}_{+\text{eff}}$ problems. Finally, we also provide a polynomial bound on the height of a decomposition tree necessary to show that problem is solvable.

3.3.1 Decomposition Trees

Geier and Bercher [2011] introduced the idea of *decomposition trees*, which is a representation of how the initial compound task c_I can be transformed to a task network tn via a sequence of decompositions.[1] This section presents their definitions below, modified slightly to suit our purposes.

Given a planning problem P, a *decomposition tree* $g = (T, E, \prec, \alpha, \beta)$ is a five-tuple satisfying the following properties:

- (T, E) is a tree with nodes T and directed edges E pointing towards the leaves;
- $\prec$ is a partial order defined over T;
- $\alpha : T \rightarrow C \cup O$ is a labeling function that labels the nodes in T with task names;
- β is a labeling function that labels each inner node with a method $m = (c, tn_m)$ and an isomorphism from tn_m to the children of that node.

Moreover, $T(g)$ is defined to refer to the tasks of g and $\mathsf{ch}(g, t)$ to refer to the direct children of $t \in T(g)$ in g.

The following definition states the conditions under which a decomposition tree encodes a decomposition of the initial task network. A decomposition tree

[1]Note that the restriction for having a single task for the initial task network of an HTN planning problem is only for the sake of simplifying the exposure of our theoretical results; the definitions and theorems in this section can be adapted to work without this restriction by generalizing the notion of decomposition trees, described below, to decomposition forests.

$g = (T, E, \prec, \alpha, \beta)$ is *valid* with respect to a planning problem $P = (D, s_0, c_I)$ if and only if the root node of g is labeled with the initial task name c_I and for any inner node t, where $\beta(t) = ((c, tn_m), f)$, the following conditions hold:

1. $\alpha(t) = c$,

2. f is a valid isomorphism of the task network induced in g by $\mathsf{ch}(g, t)$ and tn_m; i.e.

$$(\mathsf{ch}(g,t), \prec|_{\mathsf{ch}(g,t)}, \alpha|_{\mathsf{ch}(g,t)}) \underset{f}{\cong} tn_m,$$

3. $\forall t' \in T, c' \in \mathsf{ch}(g, t)$, it holds that

   (a) if $t \prec t'$ then $c' \prec t'$;

   (b) if $t' \prec t$ then $t' \prec c'$.

4. there are no other ordering constraints in $\prec$ other than those demanded by conditions 2 and 3.

Informally, the above conditions capture the following checks for each inner node t: condition 1 verifies the applicability of the method $m = \beta(t)$ that t is labeled with; condition 2 verifies that m's task network is correctly represented in the tree; condition 3 ensures that the ordering constraints are inherited correctly after the application of m; and condition 4 ensures the minimality of $\prec$.

The definition of a decomposition tree and its validity to an HTN planning problem is identical to Geier and Bercher's definition, save for the addition of the explicit isomorphism at each inner node t, mapping $\mathsf{ch}(g, t)$ to the subtask network of the method applied at t. This modification is made so that the validity of a decomposition tree can be checked in time polynomial in the size of the tree[2]. Note that the theoretical results in Geier and Bercher [2011] still hold unchanged even with these modifications. This is an important point as we shall be using their theorems (which they proved under their definition) in our proofs.

Note that the leaves of a decomposition tree g form a task network, which is called the yield of g. Formally, the *yield of a decomposition tree* $g = (T, E, \prec, \alpha, \beta)$ is a task network defined as follows. Let $T' \subseteq T$ be the set of all leaf nodes in g. Then, $\mathsf{yield}(g) = (T', \prec|_{T'}, \alpha|_{T'})$.

Geier and Bercher [2011] use the above definitions to prove the following useful property of valid decomposition trees:

Theorem 3.3 *Given a planning problem $P = (D, s_0, c_I)$, the following holds for any task network $tn \in TN_{C \cup O}$. There exists a valid decomposition tree g with $\mathsf{yield}(g) = tn$ if and only if $c_I \rightarrow_D^* tn$.*

In other words, the reachability of tn from c_I via a sequence of method decompositions can be proved by providing a valid decomposition tree for the problem P whose yield is tn. This property, as we shall see later, will be instrumental in proving that delete-free HTN planning is in NP.

Given a decomposition tree $g = (T, E, \prec, \alpha, \beta)$ and a node $t \in T$, the subtree of g induced by t, written as $g[t]$, is

$$g[t] = (T', E', \prec|_{T'}, \alpha|_{T'}, \beta|_{T'}),$$

[2]Since graph isomorphism is not known to be in P, this would not be possible without our modification.

where (T', E') is the subtree of (T, E) rooted at t.

Definition 3.4 *Let $g = (T, E, \prec, \alpha, \beta)$ be a decomposition tree and $t_i, t_j \in T$ be two nodes of g. The result of the subtree substitution of t_i with t_j on g, written as $g[t_i \leftarrow t_j]$, is given as follows:*

- *If t_i is the root node of g, then $g[t_i \leftarrow t_j] = g[t_j]$.*
- *Otherwise, $g[t_i \leftarrow t_j] = (T', E', \prec|_{T'}, \alpha|_{T'}, \beta|_{T'})$, with*
  - *$T' = (T \setminus T(g[t_i])) \cup T(g[t_j])$,*
  - *$E' = E|_{T'} \cup \{(p, t_j)\}$, where p is the parent node of t_i in g.*

Note that this operation in general will *not* lead to valid decomposition trees. However, if applied under the right conditions, the result of the subtree substitution can still describe valid decompositions as described by the following result [Geier and Bercher, 2011]:

Theorem 3.5 *Let $g = (T, E, \prec, \alpha, \beta)$ be a valid decomposition tree for an HTN planning problem P. If we are given two nodes $t_i \in T, t_j \in T(g[t_i])$ such that $\alpha(t_i) = \alpha(t_j)$, then $g[t_i \leftarrow t_j]$ is also a valid decomposition tree for P.*

In other words, if t_i and t_j map to the same task names and t_j is a descendant of t_i in g, then replacing t_i (and its subtree) with t_j (and its subtree) still results in a valid decomposition tree. This technique can therefore be used to eliminate cyclic decompositions from a tree while still retaining validity.

3.3.2 Forming a witness to the solvability of an HTN problem

We are going to use decomposition trees to show that delete-free HTN planning is in NP. Note that if a valid decomposition tree's yield is primitive and executable, then we can use the tree as a checkable proof that its problem is solvable. However, even in the restricted case where none of the operators have an effect, the minimal solution size (measured in the number of tasks) may still be exponential. So we need to be able to present a witness that includes both a tree with a non-primitive yield, and a polynomial size proof that some expansion of that yield is executable.

Definition 3.6 *Let $P = (D, s_0, tn_0)$ be an HTN planning problem, where $D = (L, C, O, M)$ and $tn_0 = (T_0, \prec_0, \alpha_0)$. A* state-transition preserving *solution for P is one in which the only state-changing actions are the ones that were already in tn_0, i.e., it is a primitive task network tn such that:*

- *$tn_0 \rightarrow_D^* tn$, where $tn = (T, \prec, \alpha)$*
- *tn has an executable ordering over its tasks ($t_1, \ldots, t_n$, executing over the states s_0 to s_n)*
- *If $t_i \notin T_0$ then $s_{i-1} = \gamma(s_{i-1}, \alpha(t_i)) = s_i$*

Given a sequence of states, a *solution table* for finding a state-transition preserving solution consists of a row for each combination of start state, end state, and task name. Each row in a solution table has a *value*, defined as follows:

- For each row with a primitive task name, the value of that row is 1 if the ground instance of the operator for that primitive task name is both applicable in any state between the start state and end state, inclusively, and the operator does not change said state. Otherwise, the value of the row is ∞.

- For each row with a non-primitive task name, we associate a method used to decompose the task, and a set of pointers back into the table supporting that the method is executable (without changing the state) between the start and end state for the row. The value for the row is then the sum of the values of its supporting rows.

We can check the table by first checking the primitive entries of the table, and then repeatedly scanning the table to find rows whose supports have already been checked. This leads into the following lemma:

Lemma 3.7 *Both the plan-existence and the k-length-plan-existence problems for finding a state-transition preserving solution are in* NP.

Proof. Let $P = (D, s_0, tn_0)$ be an HTN planning problem where $D = (L, C, O, M)$, $tn_0 = (T_0, \prec_0, \alpha_0)$ such that P has a state-transition preserving solution.

By definition, in any state-transition preserving solution, only the primitive tasks already in tn_0 may change the state. So given a fixed, executable ordering over the primitive tasks of tn_0 and the states associated with that ordering $(s_0, \ldots, s_n)$, the decompositions of non-primitive tasks in tn_0 interact with each other only in what states they start and end on (constraining the end and start states, respectively, of tasks required to come before or after). Start and end states for a task determine what decompositions (if any) are executable over that sequence of states. This lets us construct a solution table as described above.

Once the solution table is constructed, a witness to the solvability of P (i.e., a witness that there exists a state-preserving solution for P) consists of a total order over the primitive tasks of tn_0, a solution table described above for the sequence of states traversed by those primitive tasks, and a set of pointers into the table for each non-primitive task in tn_0. The value of the solution is the sum of the primitive tasks in the row of the solution table that holds tn_0, plus the sum of the values sizes of the supporting table entries. Since the validity of the ordering and table are verifiable in polynomial time, both plan-existence and k-length-plan-existence for finding a state-transition preserving solution are both in NP. $\square$

We can now use a decomposition tree as a proof that an HTN problem is solvable, even if the yield of that tree is non-primitive:

Definition 3.8 *Let tn be a task network, g be valid decomposition tree of tn, and stp be a witness that the yield of g has a state-transition preserving solution. Then, a* witness *to the solvability of an HTN problem $P = (D, s, tn)$ is the pair (g, stp) of a valid decomposition tree g of tn with stp.*

Since checking the validity of a tree is polynomial in the size of the tree, and checking the witness that the yield of the tree has a state-transition preserving solution is polynomial in the size of the yield and the number of task names, it follows that the combined witness is also in P. Furthermore, note that every solvable HTN planning problem has a witness, even non-delete-free problems. However, the existence of a polynomial-sized witness is only likely in delete-free planning, where a fix-point state is reachable in a polynomial number of actions. In the remaining sections, we show that delete-free HTN planning problems always have a witness of polynomial size.

3.3.3 Bounding the breadth of the witness tree

Given a delete-free HTN problem and its witness, (g, stp), we know there are at most $|O|$ primitive tasks which change the state in any execution of the yield of g, where O is the set of operators. We now show how to restrict a decomposition tree to its minimal valid subtree that contains those operators.

Definition 3.9 (Saplings) *Given a tree $g = (T, E, \prec, \alpha, \beta)$ and a set of tree nodes $S \subset T$, let T' be the set of nodes along any path from a node in S to the root of g (inclusively) and the siblings of each and every node along the path. Formally, T' is the smallest subset of T such that:*

- $S \subseteq T'$
- $\forall_{t,t' \in T} t' \in T' \wedge (t, t') \in E \implies t \in T'$
- $\forall_{t,t_1,t_2 \in T'} (t_1 \in T') \wedge \{(t, t_1), (t, t_2)\} \subseteq E \implies t_2 \in T'$

Let T'' contain the inner nodes of T'. The the sibling-augmented path tree *or S-sapling of T is the decomposition tree given by the tuple:*

$$\left(T', E_{|T'}, \prec_{|T'}, \alpha_{|T'}, \beta_{|T''}\right)$$

Proposition 3.10 *Given a tree $g = (T, \prec, \alpha, \beta)$ and a set $S \subseteq T$, then the S-sapling of g is a valid decomposition tree.*

Proof. Any subtree of g containing the root satisfies all but condition 3 of definition a valid decomposition tree. Since the construction of a sapling either preserves all children of node or none of them, condition 3 also holds. $\square$

Given a witness (g, stp) for a delete-free problem, we can create a sapling using just the primitive tasks that change the state.

Lemma 3.11 *Let (g, stp) be a witness that a delete-free HTN problem $P = (D, s_0, tn_0)$ with domain $D = (L, C, O, M)$ is solvable. Let $(T, \prec, \alpha) = yield(g)$, and let $S \subseteq T$ be the set of tasks that change the state in the order specified by stp. Then if g' is the S-sapling of g, there exists a witness stp' such that the yield of g' has state-transition preserving solution of the same size or smaller than the yield of g.*

Proof. Given that *stp* is the witness that g has a state-transition preserving solution, let $(<, B, R) = stp$, where:

- $<$ is $\langle t_1, \ldots, t_n \rangle$ which is the total ordering over the primitive tasks in the yield of g. Let $s_1, \ldots, s_m$ be the distinct states that sequence produces (omitting repeated states).
- B is the solution table for the sequence $s_1, \ldots, s_m$. Assume WLOG that B is optimal, giving the best possible cost for each entry in B.
- R is the set of pointers into B for non-primitive tasks in the yield of g.

Let S be the set of primitive tasks that change the state, or
$S = \{t_i \in yield(g) \mid s_{i-1} \neq s_i\}$, and let g' be the S-sapling of g.

Now we provide a witness that g' has a state-transition preserving solution. Let $<'$ be the same ordering as $<$ restricted to tasks in S. Since tasks in $<$ but not in $<'$ did not change the state, an execution of $<'$ produces the same sequence $s_1, \ldots, s_m$ of distinct states that $<$ did, and so we can reuse the same solution table B.

For the set of supports, any task t in the yield of g' which was not in the yield of g must have children in the yield of g which, under the given ordering $<$, were all either primitive tasks which did not change the state or were non-primitive with state-transition preserving expansions with entries in R. So B, the solution table, must have an entry for s_i, s_j, t with finite value, where s_i and s_j are the first state and last in the sequence $s_0, \ldots, s_m$ where either primitive descendant was executed or the first state used in R for a non-primitive descendant. So we can construct a new set of supports R' using the above method for any task in the yield of g' but not in R, and directly using the entry from R otherwise.

So $stp' = (<', B, R')$ is a witness that g' has a state-transition preserving solution. Moreover, since B remains the same and R' was calculated from B and R, stp' must indicate that g' has a solution in B with the same or lower value as g. $\square$

3.3.4 Bounding the height of the witness tree

The above lemma lets us take any witness (g, stp) to a problem's solvability and construct a new witness which is composed of a polynomial number of paths to the root g (plus siblings). This is not quite enough to show that delete-free planning is in NP, since those paths may not be polynomial in length. However, in those cases, we can use a variant of the pumping lemma [Comon et al., 2007] to produce a new witness with polynomially-bounded length paths:

Theorem 3.12 *Let $P = (D, s_0, tn_0)$ (where $D = (L, C, O, M)$) be a solvable delete-free HTN planning problem, with P having a minimal solution size of k . Then there exists a witness (g, stp) that P has a solution size of $\leq k$, with $|T(g)| \leq m \cdot |C| \cdot |O|^2$, where m is the size of the largest task network in M.*

Proof. If P is solvable, there exists a tree g_p with an executable, primitive yield of optimal cost k. Let (g, stp) be the S-sapling witness as constructed above in lemma 3.11, where S is the set of tasks in the witness that change the state. Then (g, stp) is a witness that P has a solution of size $\leq k$.

Suppose g has a height that is greater than $|C| \cdot |O|$. Since g is constructed from a series of paths from nodes to the root, this means that there is some path from a node in S to the root of that length.

Let $t_1, \ldots, t_n$ be the tasks along that path. Since that path is joined at most $|S| - 1$ times by other paths from S to the root ($|S| \leq |O|$) and since there are only $|C|$ task names to assign, there must be some segment $t_i, \ldots, t_j$ between joins such that $\alpha(t_i) = \alpha(t_j)$, and no descendants of t_i not on the path to t_j has a descendants that is in S.

Since no descendants of t_i that are not also a descendants of t_j are in S, then all of those descendants must have a state-transition preserving solution under stp'. Let $g' = g\,[t_i \leftarrow t_j]$ be the tree obtained by substituting t_j for t_i. Since we only removed tasks which did not change the state, the yield of g' is a strict subset of the yield of g. So we can create a witness stp' that g' has a state-transition preserving solution by restricting the set of supports in stp to the tasks remaining in the yield of g'. That solution must have a cost strictly less than k. This would violate our assumption k was the minimal solution size.

So g must have a height that is less than or equal to $|C| \cdot |O|$. $\square$

Since we can always find a polynomial sized witness to the minimal-sized solution, this means that finding k-size solution (or any solution) to a delete-free HTN problem ($\text{HTN}_{+\text{eff}}$) is in NP. Given that both plan-existence and k-length-plan-existence are NP-hard for $\text{HTN}_{+\text{eff}}$, the last of our results is trivial:

Theorem 3.13 *For $HTN_{+\mathit{eff}}$, both plan-existence and k-length-plan-existence are* NP*-complete.*

From this theorem and the subclass relationships shown in Figure 3.1, the other classes considered in this paper fall in NP as well and are thus NP-complete (with the exception of plan-existence for $\text{TIHTN}^{+\text{pre}}_{+\text{eff}}$, of course). Table 3.2 summarizes our final set of results for delete-free HTN planning.

3.4 Discussion

In classical planning, relaxing the planning problem by removing negative preconditions and effects has been quite useful in the development of efficiently computable search heuristics. This chapter shows that this relaxation will not—by itself—produce efficiently computable HTN planning heuristics, because the relaxed

Table 3.2: Summary of results after Section 3.3.

Problem	plan-existence	k-length-plan-existence
$\mathrm{TIHTN}^{+\mathrm{pre}}_{+\mathrm{eff}}$	P	NP-complete
$\mathrm{TIHTN}_{+\mathrm{eff}}$	NP-complete	NP-complete
$\mathrm{HTN}^{+\mathrm{pre}}_{+\mathrm{eff}}$	NP-complete	NP-complete
$\mathrm{HTN}_{+\mathrm{eff}}$	NP-complete	NP-complete

problems are NP-hard. Thus the development of search heuristics for HTN planning will require a new kind of problem relaxation.

The solution tables that we used in the proof of Lemma 3.7 are a data structure similar to planning graphs, and it might be possible to use them as a foundation to develop new heuristics and search techniques for generating compact witnesses. Such witnesses could be used to provide heuristic estimates of relaxed plan length. Furthermore, a solution table also exhibit similarities to that of a *chart* in *chart parsing* [Kay, 1986, Earley, 1986, Charniak et al., 1998, Ji, 1993], in both the way the solution table is generated and structured. This suggests that it might be possible to use the techniques to generate efficient parse-trees in chart parsing for witness inference.

Another approach may be to combine relaxed planning graphs with a relaxation of the constraints that HTN planning formalisms impose on the search process. For example, our results show that efficiently computable HTN planning problems can be produced by removing negative preconditions and effects, and also allowing task insertion (i.e., allowing the application of any executable operator, regardless of whether or not it is reachable by some decomposition). We suspect that this might relax the problem too much for the heuristic values to be useful. But we think it may be possible to develop more accurate yet efficiently computable heuristics by developing a principled compromise, e.g., by restricting the inserted tasks to those available in some decomposition of the current task. This would be an interesting topic for future research.

Given that full-semantics delete-free HTN planning is out of reach for an HTN heuristic, one might question whether less informed heuristics will provide effective search guidance. The next chapter shows totally-ordered $\leq_r$-stratifiable problems can be translated into classical planning problems. This opens up the use of classical planning heuristics for HTN planning.

Chapter 4: HTN Planning via translation to PDDL

I show that HTN planning knowledge, if it satisfies some restrictions, can automatically be translated into PDDL, and that ***even small amounts of such knowledge can greatly improve a classical planner's performance***. In particular:

- Section 4.2 describes how to translate a restricted class of HTN methods and operators into PDDL. We provide theorems showing that our translation is correct, that its time and space complexity are both linear, and that it can be used even on *partial* HTN models of a domain (which can be much easier to write than full HTN models).
- Experiments in Section 4.4 show that by translating partial HTN models into PDDL, we can substantially improve a classical planner's performance. In experiments with the well-known Fast-Forward (FF) planner [Hoffmann and Nebel, 2001] on more than 3500 planning problems, the translated knowledge improved FF's running time by several orders of magnitude, and enabled it to solve much larger planning problems than it could otherwise solve.

4.1 Basic Definitions and Notation

Chapter 1 introduced propositional HTN and classical planning formalisms for ease of analysis. Most planning implementations, however, are set to work over fragments of first-order logic. This section presents formalisms defined over function-free first-order logic for classical planning and totally-ordered HTN planning that closely match the PDDL planning language and SHOP HTN language, respectively [Fox and Long, 2003, Nau et al., 2001].

4.1.1 Classical Planning

Our definitions for classical planning are based on the ones in Ghallab et al. [2004].

Let L be the set of all literals in a function-free first-order language. A *state* is any set of ground atoms of L. A *classical planning problem* is a triple $P = (s_0, g, O)$, where s_0 is the *initial* state, g is the *goal* (a set of ground literals of L), and O is a set of operators. Each operator $o \in O$ is a triple

$$o = (\text{name}(o), \text{precond}(o), \text{effects}(o)),$$

where name(o) is o's name and argument list, and precond(o) and effects(o) are sets of literals called o's *preconditions* and *effects*. An *action* α is a ground instance of an operator. If a state s satisfies precond(α), then α is *executable* in s, producing the state $\gamma(s, \alpha) = (s - \{\text{all negated atoms in effects}(\alpha)\}) \cup \{\text{all non-negated atoms in effects}(\alpha)\}$. A *plan* is a sequence $\pi = \langle \alpha_1, \ldots, \alpha_n \rangle$ of actions. π is a *solution* for P

if, starting in s_0, the actions are executable in the order given and the final outcome is a state s_n that satisfies g.

4.1.2 TSTN Planning

Ghallab et al. [2004] describes a restricted case of HTN planning called *Total-order Simple Task Network* planning, which we'll abbreviate as TSTN Planning. The definitions are as follows.

A *task* is a symbolic representation of an activity. Syntactically, it is an expression $\tau = t(x_1, \ldots, x_q)$ where t is a symbol called τ's *name*, and each x_i is either a variable or a constant symbol. If t is also the name of an operator, then τ is *primitive*; otherwise τ is *nonprimitive*. Intuitively, primitive tasks can be instantiated into actions, and nonprimitive tasks need to be decomposed (see below) into subtasks.

A *method* is a prescription for how to decompose a task into subtasks. Syntactically, it is a four-tuple

$$m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m)),$$

where name(m) is m's name and argument list, task(m) is the task m can decompose, precond(m) is a set of preconditions, and subtasks$(m) = \langle t_1, \ldots, t_j \rangle$ is the sequence of subtasks.

A *TSTN planning problem* is a four-tuple $P = (s_0, T_0, O, M)$, where s_0 is an initial state, O is a set of operators, T_0 is a sequence of ground tasks called the *initial task list*, and M is a set of methods.

If T_0 is empty, then P's only solution is the empty plan $\pi = \langle\rangle$, and π's *derivation* (the sequence of actions and method instances used to produce π) is $\delta = \langle\rangle$. If T_0 is nonempty (i.e., $T_0 = \langle t_1, \ldots, t_k\rangle$ for some $k > 0$), then let $T' = \langle t_2, \ldots, t_k\rangle$. If t_1 is primitive and there is an executable action α with $\text{name}(\alpha) = t_1$, then let $s_1 = \gamma(s_0, \alpha)$. If $P' = (s_1, T', O, M)$ has a solution π with derivation δ, then the plan $\alpha \bullet \pi$ is a solution to P (where $\bullet$ is concatenation) whose derivation is $\alpha \bullet \delta$. If t_1 is nonprimitive and there is a method instance m such that $\text{task}(m) = t_1$, and if s_0 satisfies $\text{precond}(m)$, and if $P' = (s_1, \text{subtasks}(m) \bullet T', O, M)$ has a solution π with derivation δ, then π is a solution to P and its derivation is $m \bullet \delta$.

4.1.3 Using TSTN Planners for Classical Planning

To use a TSTN planner in a classical planning domain D (i.e., a set of classical planning problems that all have the same operator set O), the usual approach is augment D with a set M of methods and a way to translate each classical goal g into a task list T_0^g. This maps each classical planning problem $P = (s_0, g, O)$ in D into a TSTN planning problem $P' = (s_0, T_0^g, O, M)$. The mapping is *correct* if P' is solvable whenever P is, and if the solutions for P' are also solutions for P. Since the objective is for P' to have a small search space, the set of solutions for P' may be much smaller than the set of solutions for P.

In the above mapping, we will say that M is *O-complete* if every operator in $o \in O$ is *mentioned* in M, i.e., at least one method in M has a subtask that is an instance of $\text{name}(o)$.

4.2 Translating TSTN to Classical

Let $P = (s_0, T, O, M)$ be a TSTN planning problem, and suppose T_0 is a correct translation (as defined above) of a classical goal g. We now describe how to translate P (if a restriction holds) into a classical planning problem $trans(P) = (s'_0, g, O')$ that is *equivalent* to P in the following sense: as we'll show in Section 4.3, there is a one-to-one mapping from P's solution derivations to $trans(P)$'s solutions.

4.2.1 Preliminaries

We begin by introducing a restriction. For every solution π of P, let the *non-tail height* of π be the number of levels of method decomposition used to produce π, ignoring tail decomposition (i.e., decomposition of the last task in a task list). Then either we need to extend the planning language to include function symbols,[1] or else we must be given an upper bound H on the non-tail height of all solutions of P. The non-tail height is equivalent to the height of the minimum stratification in totally-ordered $\leq_r$-stratifiable problems.

We need the above restriction in order to implement a symbolic representation of a numeric counter, to keep track of the current number of levels of task decomposition. Here is how to implement the counter when H is given:[2]

[1]PDDL includes this extension, but traditional formulations of classical planning (e.g., [Ghallab et al., 2004]) do not.

[2]If H is not given but the planning language contains function symbols, we can instead use an unbounded number of ground terms $1, \mathsf{next}(1), \mathsf{next}(\mathsf{next}(1)), \ldots$.

- We'll introduce new constant symbols $\mathsf{d}_0, \mathsf{d}_1, \ldots, \mathsf{d}_H$ to denote levels of task decomposition, and a predicate symbol level so that the atom $\mathsf{level}(\mathsf{d}_i)$ can be used to mean that the current level of task decomposition is d_i. We give a special meaning to the constant symbol d_0: it marks the successful end of method decomposition process.

- To specify a total ordering of the constant symbols, we will put new atoms $\mathsf{next}(\mathsf{d}_1, \mathsf{d}_2)$, $\mathsf{next}(\mathsf{d}_2, \mathsf{d}_3)$, $\ldots$, $\mathsf{next}(\mathsf{d}_{H-1}, \mathsf{d}_H)$ into the initial state.

In addition, for each method $m(x_1, \ldots, x_k)$ and task $t(y_1, \ldots, y_j)$ we will introduce new atoms $\mathsf{do}_m(x_1, \ldots, x_k)$ and $\mathsf{do}_t(y_1, \ldots, y_j)$.

4.2.2 Translating Operators

Let o be any operator in O, and suppose $\text{name}(o) = o(x_1, \ldots, x_n)$, $\text{precond}(o) = \{p_1, \ldots, p_j\}$ and $\text{effects}(o) = \{e_1, \ldots, e_k\}$. If o is not mentioned in M (whence M is not O-complete), then $trans(o) = o$. Otherwise $trans(o)$ is the following operator o', which is like o except that it is applicable only when do_o is true, and it decrements the counter:

$$\begin{aligned}
\text{name}(o') &= o'(x_1, \ldots, x_n) \\
\text{precond}(o') &= \{\mathsf{do}_o(x_1, \ldots, x_n), p_1, \ldots, p_j, \\
&\qquad \mathsf{level}(v), \mathsf{next}(u, v)\} \\
\text{effects}(o') &= \{\neg\mathsf{do}_o(x_1, \ldots, x_n), \neg\mathsf{level}(v), \\
&\qquad \mathsf{level}(u), e_1, \ldots, e_k\}
\end{aligned}$$

We define $trans(O) = \{trans(o) \mid o \in O\}$.

4.2.3 Translating Methods

Let m be any method in M, and suppose $\text{name}(m) = m(x_1, \ldots, x_n)$, $\text{task}(m) = t(y_1, \ldots, y_{j_t})$, and $\text{precond}(m) = \{p_1, \ldots, p_{j_m}\}$. There are two cases:

Case 1: $\text{subtasks}(m) = \emptyset$ (i.e., m specifies no subtasks for t). Then $trans(m)$ is the operator m' defined as follows:

$$\text{name}(m') = m'(x_1, \ldots, x_n)$$

$$\text{precond}(m') = \{\mathsf{do}_t(y_1, \ldots, y_{j_t}), p_1, \ldots, p_j,$$
$$\mathsf{level}(v), \mathsf{next}(u, v)\}$$

$$\text{effects}(m') = \{\neg\mathsf{do}_t(y_1, \ldots, y_{j_t}), \neg\mathsf{level}(v), \mathsf{level}(u)\}$$

Case 2: $\text{subtasks}(m) = \{t_1, \ldots, t_k\}$ for $k \geq 1$. Then $trans(m)$ is the set of planning operators $\{m'_0, \ldots, m'_k\}$ defined below, where m'_0 is an operator that checks whether m is applicable, and $m'_1, \ldots, m'_k$ are operators that correspond to calling m's subtasks. The definition of m'_0 is

$$\text{name}(m'_0) = m'_0(x_1, \ldots, x_n)$$

$$\text{precond}(m'_0) = \{\mathsf{do}_t(y_1, \ldots, y_{j_t}), p_1, \ldots, p_{j_m}, \mathsf{level}(v)\}$$

$$\text{effects}(m'_0) = \{\neg\mathsf{do}_t(y_1, \ldots, y_{j_t}), \mathsf{do}_{m_1}(x_1, \ldots, x_n, v)\}$$

Intuitively, m'_0's preconditions say that t is the current task and that m's preconditions hold; and m'_0's effect do_{m_1} makes it possible to apply the planning operator m'_1 that corresponds to m's first subtask.

For $i = 1, \ldots, k-1$, if m's i^{th} subtask is $t_i(y_{i1}, \ldots, y_{ij_i})$ then m'_i is defined as follows:

$$\begin{aligned}
\text{name}(m'_i) &= m'_i(x_1, \ldots, x_n) \\
\text{precond}(m'_i) &= \{\mathsf{do}_{m_i}(x_1, \ldots, x_n, v), \mathsf{level}(v), \mathsf{next}(v, w)\} \\
\text{effects}(m'_i) &= \{\neg\mathsf{do}_{m_i}(x_1, \ldots, x_n, v), \neg\mathsf{level}(v), \mathsf{level}(w), \\
&\qquad \mathsf{do}_{t_i}(y_{i1}, \ldots, y_{ij_i}), \mathsf{do}_{m_{i+1}}(x_1, \ldots, x_n, v)\}
\end{aligned}$$

The operator m'_k, which corresponds to m's last subtask t_k, is like m'_i but omits the effects $\neg\mathsf{level}(v)$ and $\mathsf{level}(w)$.

We define $trans(M) = \bigcup_{m \in M} trans(m)$.

4.2.4 Translating Planning Problems

Finally, we define $trans(P) = (s'_0, g, O')$, where

$$\begin{aligned}
s'_0 &= s_0 \cup \{\mathsf{next}(\mathsf{d}_0, \mathsf{d}_1), \ldots, \mathsf{next}(\mathsf{d}_{k-1}, \mathsf{d}_k), \\
&\qquad \mathsf{level}(\mathsf{d}_1), \mathsf{do}_{t_0}(c_1, \ldots, c_n)\}; \\
O' &= trans(O) \cup trans(M).
\end{aligned}$$

4.3 Properties

The theorems in this section establish the correctness and computational complexity of our translation scheme.

Theorem 4.1 *Let $P = (s_0, \langle t_1, \ldots, t_k\rangle, O, M)$ (where $k \geq 0$) be any TSTN planning problem. Let $\Delta = \{$all derivations of solutions for $P\}$, and $\Pi = \{$all solutions for* *trans*$(P)\}$. *If* M *is* O*-complete, then there is a one-to-one correspondence that maps* Δ *onto* Π.

Sketch of proof. We need to define a mapping $F : \Delta \rightarrow \Pi$ and show that F is one-to-one and onto. Below we define F; the proof that it is one-to-one and onto can be done straightforwardly by induction.

Let π be a solution for P with derivation δ. Recall that δ is the sequence of the actions and method instances used to produce π, in the order that they were applied. In particular, δ is a concatenation of subsequences $\delta_1, \ldots, \delta_k$ corresponding to $t_1, \ldots, t_k$. We will let $F(\delta) = F(\delta_1) \bullet \ldots \bullet F(\delta_k)$, where $\bullet$ denotes concatenation, and where each $F(\delta_i)$ is defined recursively as follows:

If δ_i is empty, then $F(\delta_i)$ also is empty. If δ_i is nonempty (i.e., $\delta_i = \langle \alpha_{i1}, \ldots, \alpha_{ik} \rangle$), then let $\delta'_i = \langle \alpha_{i2}, \ldots, \alpha_{ik} \rangle$. There are three cases:

1. If α_{i1} is an action, then $F(\delta_i) = trans(\alpha_{i1}) \bullet F(\delta'_i)$.

2. If α_{i1} is a substitution instance $m\theta$ of a method m with substitution θ, and subtasks(m) is empty, then $F(\delta_i) = m'\theta \bullet F(\delta'_i)$, where m' is as in Case 1 of Section 3.

3. If α_{i1} is a substitution instance $m\theta$ of a method m and subtasks(m) is nonempty (i.e., subtasks$(m) = \langle t'_1, \ldots, t'_j \rangle$ for some $j > 0$), then δ'_i is the concatenation of subsequences $\delta'_{i1}, \ldots, \delta'_{ij}$ produced by decomposing $t'_1, \ldots, t'_j$, respectively. In this case,

$$F(\delta_i) = m'_0\theta \bullet m'_1\theta \bullet F(\delta'_{i1}) \bullet \ldots \bullet m'_j\theta \bullet F(\delta'_{ij}),$$

where $m'_1, \ldots, m'_j$ are as in Case 2 of Section 3. □

Corollary 4.2 *In Theorem 4.1, if M is not O-complete, then the mapping F is one-to-one but not necessarily onto.*

Proof Sketch. If M is not O-complete, then there is at least one operator $o \in O$ that is not mentioned in M. Consequently, no instance of o will appear in any solution for P, nor in Δ, hence no instance of $trans(o)$ will appear in $\{F(\delta) \mid \delta \in \Delta\}$. But instances of $trans(o)$ can appear in solutions to $trans(P)$, in which case F is no longer onto. □

Theorem 4.3 *The time and space complexity of computing $trans(P)$ are both $O(|P| + H)$.*

Proof Sketch. For each $o \in O$, $trans(o)$ is a single operator that is computed by a linear-time scan of o, and it can be seen by inspection that the size of that operator is $O(|o|)$. Suppose there are no non-tail recursive methods in M. This means that $H = 0$ in this case. For each $m \in M$, $trans(m)$ is a set of methods that can be produced by a linear-time scan of m, and it can be seen by inspection that the set of methods has size $O(|m|)$. If there is a non-tail recursive method in M, then H is given as input and it is a fixed number. Thus, the theorem follows. □

4.4 Implementation and Experiments

We implemented an algorithm that uses our translation technique to translate TSTN domain descriptions into PDDL, and did an experimental investigation of the following question:

> *In domains that are hard for a classical planner, how much can its performance be improved by PDDL translations of partial HTN knowledge?*

For the classical planner, we used FF [Hoffmann and Nebel, 2001]. FF is perhaps one of the most influential classical planners available; many recent classical planning algorithms either directly depends on generalizations of FF or they incorporate the core ideas of FF in their systems.

For the planning domains, we chose three planning domains for which we wrote simple HTN domain descriptions with varying amounts of incompleteness: the Blocks World, the Towers of Hanoi problem, and a transportation domain called the *Office Delivery* domain.

The source code for our translation technique and the HTN method descriptions of the three planning domains described below are available at `http://www.cs.umd.edu/projects/planning/data/alford09translating/`.

4.4.1 Towers of Hanoi

The Towers of Hanoi problem causes problems for many classical planners because of its combinatorial nature. On the other hand, it is almost trivially easy to write a set of HTN methods to solve the problem without any backtracking. The methods say basically the following:

- **Method to move a disk:**

  precond: the smallest disk wasn't the last one moved subtask: move the smallest disk clockwise.

- **Method to move a disk:**

  precond: the smallest disk was the last one moved

  subtask: move the other disk.

Note that in a tower of a Towers of Hanoi problem, the largest disk is always at the bottom of the tower and no disk can be place on a smaller disk – i.e., the disks in a tower are in the increasing order by their sizes with the smallest is always at the top. Thus, whether the smallest disk was the last one moved can be checked in the above methods by examining the towers to the left or to the right of a tower.

The methods above provide an almost-complete solution to the Towers of Hanoi problem, except that the second method doesn't say where to move the disk. To use the PDDL translation, FF must figure out for itself that there is only one place the disk can be moved.

Below, "FF-Plain" refers to FF using the ordinary classical-planning definition of the Towers of Hanoi domain, and "FF-HTN" refers to FF using the PDDL translations of the HTNs described above. We varied the number n of disks from 3 to 14. For each value of n, we ran FF-HTN and FF-Plain each 100 times, averaging the running times. The reason for the multiple runs is because FF makes some random choices during each run that make its running time vary from one run to another. Fig. 4.1 shows the results.

For FF-Plain at 14 rings (the * in Fig. 4.1), two runs took longer than 2 hours (our time limit per problem) to finish. We counted these runs as 2 hours each,

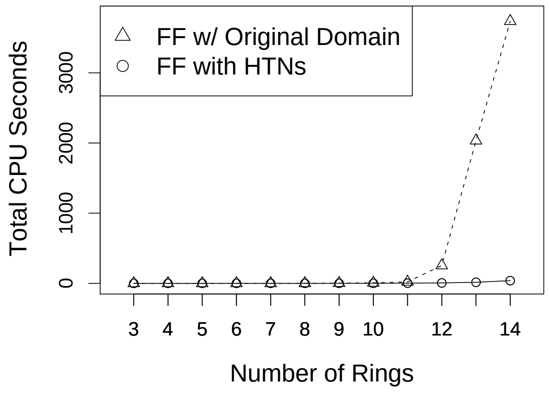

Figure 4.1: FF's CPU time in the Towers of Hanoi domain, with and without the translated domain knowledge. Each data point is FF's average CPU time on 100 runs. The asterisk is explained in the text.

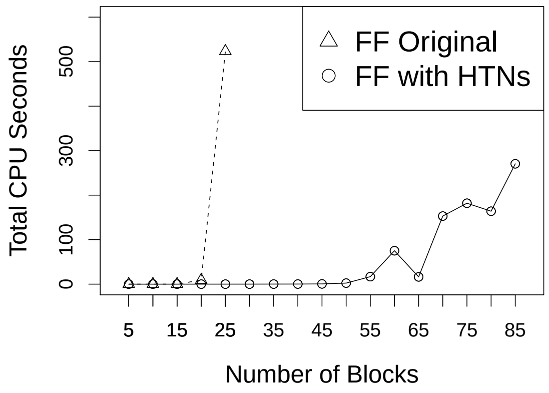

Figure 4.2: FF's CPU time in the Blocks World, with and without the translated domain knowledge. Each data point is FF's average CPU time on 100 randomly generated planning problems. The asterisks are explained in the text.

and averaged them with the other 98 runs; hence the data point for 14 rings makes FF-Plain's performance look better than it actually was.

As shown in Fig. 4.1, FF-Plain's running times grew much faster than FF-HTN's did. With 14 disks, FF-HTN was about 2 orders of magnitude faster than FF-Plain.

4.4.2 Blocks World

The Blocks World has previously been shown to pose some difficulities for FF. Complete HTN domain descriptions can work very efficiently [Ghallab et al., 2004], but are somewhat complicated. To see how well FF could do with some simple and partial HTN knowledge, we wrote HTN methods that said basically the following:

- **Method to move a block:**

  precond: the block is not in its final position

  subtasks: pick up the block; put it in its final position.

- **Method to move a block:**

  precond: the block is not in its final position

  subtasks: pick up the block; put it on the table.

At each point in the planning process, both of the methods are applicable. To use the PDDL translation of them, FF must use its heuristics to choose which of them to use.

Below, "FF-Plain" refers to FF using the ordinary classical-planning definition of the Blocks World, and "FF-HTN" refers to FF with the PDDL translations of

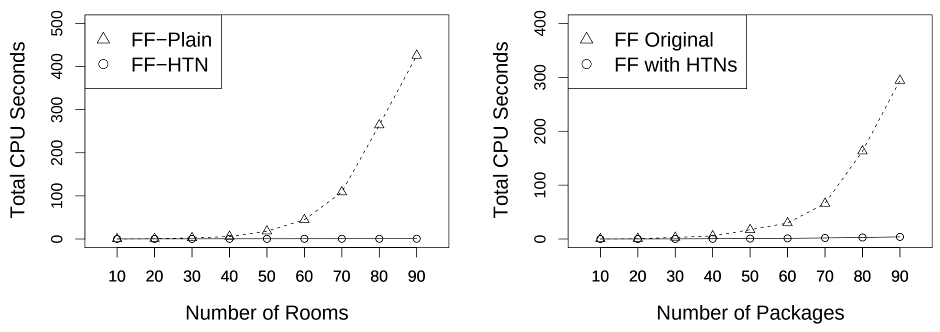

Figure 4.3: FF's CPU time in the Office domain, with and without the translated domain knowledge. In the graph at left, the number of packages is fixed at 40 and the number of rooms varies. In the graph at right, the number of rooms is fixed at 40 and the number of packages varies. Each data point is FF's average CPU time on 100 randomly generated planning problems.

the methods described above. We ran both FF-HTN and FF-Plain on 100 randomly generated n-block problems for each of $n = 5, 10, 15, \ldots, 90$, giving a total of 1800 Blocks World problems. Fig. 4.2 shows the results.

As before, we gave FF a 2-hour time limit for each run. At data points where all 100 runs took less than 2 hours each, the data point was the average time per run. At data points where 3 or fewer of the 100 runs failed to finish within 2 hours, we counted each failure as 2 hours when computing the average, and marked the data point with an asterisk. In all of the other cases, a large number of the 100 runs failed to finish with 2 hours, so we omitted those data points.

As shown in Fig. 4.2, FF-Plain could not solve problems larger than 25 blocks, but FF-HTN could solve problems up to 85 blocks. At 25 blocks, FF-HTN was about 4.2 orders of magnitude faster than FF-Plain.

4.4.3 Office Delivery

This is a transportation domain in which a robot needs to pick up and deliver packages in a building. It is similar to the well-known Robot Navigation Domain [Kabanza et al., 1997], with the following differences: (1) the problem is deterministic, (2) there is a variable number of rooms, and (3) some of the rooms can be quite far from the hallway (hence to get to a room r, the robot may need to go through many other rooms). For this domain, we wrote a very incomplete set of HTN methods:

- **Method to move all remaining packages:**

  precond: there is a package that's not at its destination

  subtasks: pick up the package; put it in its final location;

  move all remaining packages.

Above, we omitted (1) how to get to the package's location in order to pick it up, and (2) how to take the package to its destination. To use the PDDL translation of the method, the planner must figure out those things for itself.

Fig. 4.3 shows the results of our Office Delivery experiments. For the graph on the left, we fixed the number of packages at 40 and varied the number n of office rooms from 10 to 90; and for the graph on the right we fixed the number of rooms at 40 and varied the number k of packages from 10 to 90. For each combination of n and k we ran FF on 100 randomly generated problems, giving a total of 1700 problems.

As shown in Fig. 4.3, FF-Plain's running time increased much faster than FF-HTN's. On the largest problems (90 rooms and 90 packages), FF-HTN was faster than FF-Plain by about 2.8 and 1.9 orders of magnitude, respectively.

4.5 Discussion

This chapter shows that HTN planning knowledge, if it satisfies the restrictions described in Section 2.2, can easily be translated into a form usable by domain-independent PDDL planners.

In our experiments with FF, PDDL translations of small amounts of HTN planning knowledge improved FF's performance by several orders of magnitude. This occurred even though the HTN knowledge was *incomplete*, i.e., it omitted some of the knowledge that an HTN planner would need. In places where the knowledge was missing, FF simply used its ordinary planning heuristics.

FF's ability to augment the translated HTN knowledge with its own heuristics show that even if a full-semantics delete-relaxation HTN heuristic is out of reach, relaxed-semantics heuristics can effectively guide the search to quality plans. Delete relaxation heuristics with semantics that more closely match that of regular HTN planning should be more informed (and thus more effective). Since forward-search planners such as FF and LAMA are guaranteed to terminate, running translated HTN problems under FF and LAMA is a decision procedure for totally-ordered $\leq_r$-stratifiable problems. Although the experiments are not enough to show out the impact of the duplicate detection required to maintain termination, they at least show it is not an impediment to effective search.

Chapter 5: Conclusions

This research contributes to the understanding of search in hierarchical planning systems, both in how best to organize the search space and the restrictions on developing domain independent heuristics. Guaranteed termination also opens up the possibility of adapting a number of techniques from classical planning to HTN planning, such planning-as-verification [Albarghouthi et al., 2009], counter example generation [Goldman et al., 2012], and policy generation for non-deterministic problems [Kuter et al., 2008].

Specifically, this dissertation provides:

- Four algorithms for HTN planning (DHTN, PHTN, TODHTN, and TOPHTN) and four corresponding classes of syntactically-identifiable HTN problems for which those algorithms are proven to terminate ($\leq_1$-stratifiable, $\leq_r$-stratifiable, $\leq_1$-ordered, and $\leq_r$-ordered, respectively).
- Upper and lower complexity bounds for each of these identifiable sets, including new bounds on previously known decidable problem sets.
- An analysis of delete-free HTN planning, showing that almost all delete-free HTN planning is NP-complete. This illuminates one of the barriers to provid-

ing domain-independent HTN heuristics.

- A method for HTN planning via translation to PDDL for any totally-ordered $\leq_r$-stratifiable problem.

A caveat to the above work is that all the algorithms were analyzed over propositional HTN planning, while most planning implementations use function-free fragments of first-order logic (though many planners create a ground propositional domain as a pre-processing step). For the most part, this just provides an exponential bump in the complexity bounds [Erol et al., 1995, 1996], but we have not formally shown that.

Although this thesis provides no direct implementation of these algorithms nor a specialized HTN heuristic, when used with a classical planner, the HTN-to-PDDL translation does provide heuristic, terminating search for totally-ordered $\leq_r$-stratifiable problems. Not only does this provide a practical benefit for those who need heuristic HTN search, it provides a baseline from which to evaluate future HTN-specific domain-independent heuristics.

This leaves a clear future work: the implementation and evaluation of TOPHTN described in Section 2.3, and the development and evaluation of principled delete-free based HTN heuristics.

Guaranteed termination paired with heuristic search will allow for search when the solvability of the problem is in question, such as during debugging, or for policy generation for non-deterministic problems. Even when the problem is known (or assumed) to be solvable, current HTN planners such as SHOP2 Nau et al. [2003]

must default to depth-first search to effectively find plans. This requires the domain author to know what paths the planner will take, and ensure that these paths are all finite or end in a solution. Otherwise, if the path the planner chooses is infinite, the planner will fail to return a solution even when one exists. In contrast, if the problem $\leq_r$-ordered (the broadest of the syntactic classes presented here), a TOPHTN implementation is guaranteed to terminate regardless of how it orders its search space.

Bibliography

Aws Albarghouthi, Jorge Baier, and Sheila A. McIlraith. On the use of planning technology for verification. In *Proceedings of the ICAPS09 Workshop on Heuristics for Domain Independent Planning*, 2009.

Pascal Bercher and Susanne Biundo. A heuristic for hybrid planning with preferences. In *International Conference of the Florida Artificial Intelligence Research Society*. AAAI Press, 2012.

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33, 2001.

Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1):165–204, 1994.

Eugene Charniak, Sharon Goldwater, and Mark Johnson. *Edge-Based Best-First Chart Parsing*, pages 127–133. Association for Computational Linguistics, 1998.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. Released October, 12th 2007.

Jay Earley. An efficient context-free parsing algorithm. *Readings in natural language processing*, pages 25 – 33, 1986.

Mohamed Elkawkagy, Pascal Bercher, Bernd Schattenberg, and Susanne Biundo. Improving hierarchical planning performance by the use of landmarks. In *AAAI Conference on Artificial Intelligence*, pages 1763–1769, 2012.

K. Erol, J. Hendler, and D.S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *International Conference on Artificial Intelligence Planning Systems*, pages 249–254, 1994.

K. Erol, J. Hendler, and D. Nau. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence*, 18, 1996.

Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. *Artificial Intelligence*, 76:75–88, 1995.

Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1972.

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

Aviezri S Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n. *Journal of Combinatorial Theory, Series A*, 31(2):199–214, 1981.

Thomas Geier and Pascal Bercher. On the decidability of HTN planning with task insertion. In *International Joint Conference on Artificial Intelligence*, pages 1955–1961, 2011.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

Robert P. Goldman, Ugur Kuter, and Tony Schneider. Using classical planners for plan verification and counterexample generation. In *Proceedings of AAAI Workshop on Problem Solving Using Classical Planning. To appear*, 2012.

M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

J. Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

J. Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Ping Ji. A tree approach for tolerance charting. *International Journal of Production Research*, 31(5):1023–1033, 1993.

F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

Martin Kay. Algorithm schemata and data structures in syntactic processing. *Readings in natural language processing*, pages 35–70, 1986.

Ugur Kuter, Dana S. Nau, Elnatan Reisner, and Robert P. Goldman. Using classical planners to solve nondeterministic planning problems. In *International Conference on Automated Planning and Scheduling*, pages 190–197, 2008.

D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *International Joint Conference on Artificial Intelligence*, 1999.

D. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-order planning with partially ordered subtasks. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 425–430, 2001.

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

XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *International Joint Conference on Artificial Intelligence*, pages 459–466, 2001.

Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177, 2012.

John Michael Robson. The complexity of go. In *International Federation for Information Processing Congress*, pages 413–417, 1983.

John Michael Robson. N by n checkers is exptime complete. *SIAM Journal on Computing*, 13(2):252–267, 1984.

S. Sohrabi, J.A. Baier, and S.A. McIlraith. HTN planning with preferences. In *International Joint Conference on Artificial Intelligence*, 2009.

Florent Teichteil-Königsbuch, Ugur Kuter, and Guillaume Infantes. Incremental plan aggregation for generating policies in MDPs. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 1231–1238, 2010.

Sung Wook Yoon, Alan Fern, and Robert Givan. FF-Replan: A Baseline for Probabilistic Planning. In *International Conference on Automated Planning and Scheduling*, 2007.