Complexity results for HTN planning*

Kutluhan Erol, James Hendler and Dana S. Nau

Computer Science Department, Institute for Systems Research, and Institute for Advanced Computer Studies University of Maryland, College Park, MD 20742, USA
E-mail: {kutluhan,hendler,nau}@cs.umd.edu

Most practical work on AI planning systems during the last fifteen years has been based on Hierarchical Task Network (HTN) decomposition, but until now, there has been very little analytical work on the properties of HTN planners. This paper describes how the complexity of HTN planning varies with various conditions on the task networks, and how it compares to STRIPS-style planning.

1. Introduction

In AI planning research, planning practice (as embodied in implemented planning systems) tends to run far ahead of the theories that explain the behavior of those systems. There is much recent analysis of the properties of total- and partial-order planning systems using STRIPS-style planning operators. STRIPS-style planning systems, however, were developed more than twenty years ago, and most of the practical work on AI planning systems during the last fifteen years has been based on Hierarchical Task Network (HTN) decomposition (e.g., NOAH [11], NONLIN [12], DEVISER [13], and SIPE [14]).

Until now, there has been very little analytical work on the properties of HTN planners. One of the primary obstacles impeding such work has been the lack of a clear theoretical framework explaining what a HTN planning system is, although two recent papers [9, 15] have provided important first steps in that direction. A primary goal of our current work is to define, analyze, and explicate features of the design of HTN planning systems.

Our work has progressed far enough to do complexity analyses of HTN planning similar to analyses which Bylander [2], Erol et al. [5] performed for planning with STRIPS-style operators. In particular, we have examined how the complexity of determining whether a plan exists depends on the following factors: (1) restrictions on the existence and/or ordering of non-primitive tasks in task networks, (2) whether

* This work was supported in part by NSF Grant NSFD CDR-88003012 to the Institute for Systems Research, and NSF grant IRI9306580 and ONR grant N00014-91-J-1451 to the Computer Science Department.

the tasks in task networks are required to be totally ordered, and (3) whether variables are allowed.

The paper is organized as follows. Section 2 contains an overview of HTN planning and our formalization of HTN planning. Section 3 contains the complexity results, and section 3.3 investigates the relation between HTN planning and STRIPS-style planning.

2. Basics of HTN planning

2.1. *Overview*

This section contains an informal description of HTN planning, intended to provide an intuitive feel for HTN planning. The precise description is presented in the sections 2.2 through 2.4.

HTN planning can be described best by contrasting it with its predecessor, STRIPS-style planning[1]. The representations of the world and the actions in HTN planning are very similar to those of STRIPS-style planning. Each state of the world is represented by the set of atoms true in that state. Actions correspond to state transitions, that is, each action is a partial mapping from the set of states to set of states. However, actions in HTN planning are usually called *primitive tasks*.

The difference between HTN planners and STRIPS-style planners is in what they plan for, and how they plan for it. STRIPS-style planners search for a sequence of actions that would bring the world to a state that satisfies certain conditions, i.e., attainment goals. Planning proceeds by finding operators that has the desired effects, and by asserting the preconditions of those operators as subgoals. On the other hand, one of the motivations for HTN planning was to close the gap between AI planning techniques and operations-research techniques for project management and scheduling [12]. HTN planners search for plans that accomplish *task networks*, and they plan via task decomposition and conflict resolution, which we shall explain shortly.

A task network is a collection of tasks that need to be carried out, together with constraints on the order in which the tasks are carried out, the way variables are instantiated, and what literals must be true before or after each task is performed. For example, Fig. 1 contains a task network for a trip to Las Vegas. Constraints allow the user to specify how he desires the tasks to be performed. For instance, in the Las Vegas example, the user might desire to get rich before seeing a show, which can be represented as an ordering constraint. Unlike STRIPS-style planning, the constraints may or may not contain conditions on what must be true in the final state.

[1] We use the term "STRIPS-style" planning to refer to any planner (either total- or partial-order) in which the planning operators are STRIPS-style operators (i.e., operators consisting of three lists of atoms: a precondition list, an add list, and a delete list). These atoms are normally assumed to contain no function symbols.

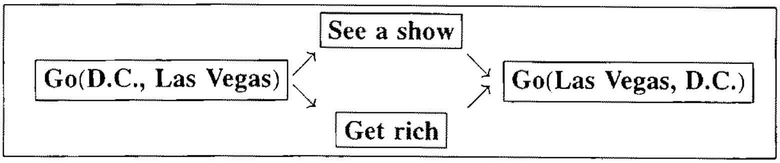

Fig. 1. A task network.

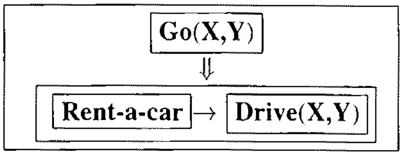

Fig. 2. A (simplified) method for going from X to Y.

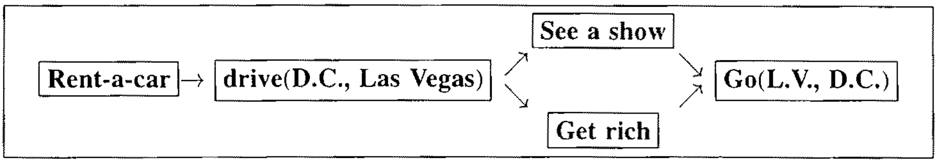

Fig. 3. A decomposition of the task network in Fig. 1.

A task network that contains only primitive tasks is called a *primitive task network*. Such a network might occur, for example, in a scheduling problem. In this case, an HTN planner would try to find a schedule (task ordering and variable bindings) that satisfies all the constraints.

In the more general case, a task network can contain *non-primitive tasks*. Non-primitive tasks are those that cannot be executed directly, and the planner needs to figure out how to accomplish them. They represent activities that involve performing multiple tasks. For example, consider the task of travelling to New York. There are several ways to accomplish it, such as flying, driving or taking the train. Flying would involve tasks like making reservations, going to the airport, buying ticket, boarding the plane, and it would only work under certain conditions such as availability of tickets, being at the airport on time, having enough money for the ticket, etc.

Ways of accomplishing non-primitive tasks are represented using constructs called *methods*. A method is of the form (α, d) where α is a non-primitive task, and d is a task network. It states that one way to accomplish the task α is to achieve all the tasks in the task network d without violating the constraints in d. Figure 2 presents a (simplified) method for accomplishing **Go(X,Y)**.

Planning proceeds by starting with the the initial task network d, and doing the following steps repeatedly, until no non-primitive tasks are left: pick a non-primitive task α in d and a method (α, d'). Then modify d by "decomposing" α (i.e., replace α with the tasks in d', and incorporate the constraints of d' into d). Figure 3 demonstrates how to do a decomposition on the task network presented in Fig. 1 using the method displayed in Fig. 2. Once no non-primitive tasks are left in d, the next problem is to find a totally-ordered ground instantiation σ of d that satisfies all of the constraints. If this can be done, then σ is a successful plan for the original problem.

In practice, HTN planning also has several other aspects. In particular, functions are often provided which can "debug" partially reduced task networks to eliminate potential problems. These "critic" functions are used to handle constraints, resource limitations, and to provide domain-specific guidance. The formalization described in [6] explains critics and the relationship between these and the constraints described above. For the purposes of this paper, the critics do not affect worst-case behavior, and thus we will omit this detail.

Here are some examples to further clarify the distinctions between different types of tasks and STRIPS-style goals. Building a house requires many other tasks to be performed (laying the foundation, building the walls, etc.), thus it is a compound task. It is different from the goal task of "having a house", since buying a house would achieve this goal task, but not the compound task of building a house (the agent must build it himself). As another example, the compound task of making a round trip to New York cannot easily be expressed as a single goal task, because the initial and final states would be the same. Goal tasks are very similar to STRIPS-style goals. However, in STRIPS-style planning, *any* sequence of actions that make the goal expression true is a valid plan, where as in HTN planning, only those plans that can be derived via decompositions are considered as valid. This allows the user to rule out certain undesirable sequences of actions that nonetheless make the goal expression true. For example, consider the goal task of "being in New York", and suppose the planner is investigating the possibility of driving to accomplish this goal, and suppose that the agent does not have a driver's licence. Even though learning how to drive and getting a driver's licence might remedy the situation, the user can consider this solution unacceptable, and while writing down the methods for **be-in(New York)**, she can put the constraint that the method of driving succeeds only when the agent already has a driver's licence.

2.1.1. Alternative views of non-primitive tasks

There appears to be some general confusion about the nature and role of tasks in HTN planning. This appears largely due to the fact that HTN planning emerged, without a formal description, in implemented planning systems [11, 12]. Many ideas introduced in HTN planning (such as nonlinearity, partial order planning, etc.) were formalized only as they were adapted to STRIPS-style planning, and only within that context. Those ideas not adapted to STRIPS-style planning (such as compound tasks and task decomposition) have been dismissed as mere efficiency hacks. In our formalism, we have tried to fill the gaps, and replace informal descriptions with precise definitions, without omitting HTN planning constructs.

Our formalism is mostly shaped after NONLIN [12] and the works of Yang and Kambhampati [9, 15] on hierarchical planning. However, our terminology for referring to non-primitive tasks is slightly different from theirs, which instead uses the term "high level actions" [11, 15]. Although this term has some intuitive appeal, we prefer not to use it, in order to avoid any possible confusion with STRIPS-style actions. STRIPS-style actions are atomic, and they always have the same effect on the world; non-primitive tasks can be decomposed into a number of primitive tasks, and the effect of accomplishing a non-primitive task depends not only on the methods chosen for doing decompositions, but also on the interleavings with other tasks.

Compound tasks are also different from STRIPS-style goals. As we have discussed earlier, compound tasks represent activities for which the final state might be totally irrelevant, or in the case of round-trip example, the final state might be the same as the initial state.

Yet another view of HTN planning totally discards compound tasks, and views methods for goal tasks as heuristic information on how to go about achieving the goals (i.e., which operator to use, in which order achieve the preconditions of that operator, etc.). Although this is a perfectly coherent view, we find it restrictive, and we believe there is more to HTN planning, as we try to demonstrate in our formalism and in the section on expressive power.

2.2. *Syntax for HTN planning*

Our language $\mathcal{L}$ for HTN planning is a first-order language with some extensions, and it is fairly similar to the syntax of NONLIN [12]. The vocabulary of $\mathcal{L}$ is a tuple $\langle V, C, P, F, T, N \rangle$, where V is an infinite set of variable symbols, C is a finite set of constant symbols, P is a finite set of predicate symbols, F is a finite set of *primitive*-task symbols (denoting actions), T is a finite set of *compound*-task symbols, and N is an infinite set of symbols used for labeling tasks. All these sets of symbols are mutually disjoint.

A *state* is a list of ground atoms. The atoms appearing in that list are said to be true in that state and those that do not appear are false in that state.

A *primitive task* is a syntactic construct of the form $do[f(x_1, \ldots, x_k)]$, where $f \in F$ and $x_1, \ldots, x_k$ are terms. A *goal task* is a syntactic construct of the form $achieve[l]$, where l is a literal. A *compound task* is a syntactic construct of the form $perform[t(x_1, \ldots, x_k)]$, where $t \in T$ and $x_1, \ldots, x_k$ are terms. We sometimes refer to goal tasks and compound tasks as non-primitive tasks.

A *task network* is a syntactic construct of the form $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$, where

- each α_i is a task;
- $n_i \in N$ is a label for α_i (to distinguish it from any other occurrences of α_i in the network);
- ϕ is a boolean formula constructed from variable binding constraints such as $(v = v')$ and $(v = c)$, ordering constraints such as $(n \prec n')$, and state constraints such as (n, l), (l, n), and (n, l, n'), where $v, v' \in V$, l is a literal, $c \in C$,

$$[(n_1 : achieve[clear(v_1)])(n_2 : achieve[clear(v_2)])(n_3 : do[move(v_1, v_3, v_2)]) \\ (n_1 \prec n_3) \wedge (n_2 \prec n_3) \wedge (n_1, clear(v_1), n_3) \wedge (n_2, clear(v_2), n_3) \wedge (on(v_1, v_3), n_3) \\ \wedge \neg(v_1 = v_2) \wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]$$

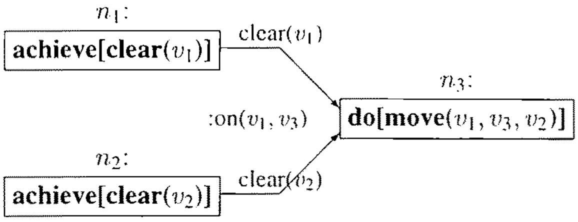

Fig. 4. A task network, and its graphical representation.

and $n, n' \in N$[2]. Intuitively (this will be formalized in the "Operational Semantics" section), $(n \prec n')$ means that the task labeled with n must precede the one labeled with n'; (n, l), (l, n) and (n, l, n') mean that l must be true in the state immediately after n, immediately before n, and in all states between n and n', respectively. Both negation and disjunction are allowed in the constraint formula.

A task network containing only primitive tasks is called a *primitive task network*.

As an example, Fig. 4 shows a blocks-world task network and its graphical representation. In this task network there are three tasks: clearing v_1, clearing v_2, and moving v_1 to v_2. The task network also includes the constraints that moving v_1 must be done last, that v_1 and v_2 must remain clear until we move v_1, that v_1, v_2, v_3 are different blocks, and that $on(v_1, v_3)$ be true immediately before v_1 is moved. Note that $on(v_1, v_3)$ appears as a constraint, not as a goal task. The purpose of the constraint $(on(v_1, v_3), n_3)$ is to ensure that v_3 is bound to the block under v_1 immediately before the move. Representing $on(v_1, v_3)$ as a goal task would mean moving v_1 onto some block v_3 before we move it onto v_2, which is not what is intended.

A *plan* is a sequence σ of ground primitive tasks.

An *operator* is of the form $[operator\ f(v_1, \ldots, v_k)(pre : l_1, \ldots, l_m)(post : l'_1, \ldots, l'_n)]$, where f is a primitive task symbol, and $l_1, \ldots, l_m$ are literals describing when f is executable, $l'_1, \ldots, l'_n$ are literals describing the effects of f, and $v_1, \ldots, v_k$ are the variable symbols appearing in the literals.

A *method* is a construct of the form (α, d) where α is a non-primitive task, and d is a task network. As we will define formally in the "Operational Semantics" section, this construct means that one way of accomplishing the task α is to accomplish the task network d, i.e., to accomplish all the subtasks in the task network without

[2] We also allow n, n' to be of the form $first[n_i, n_j, \ldots]$ or $last[n_i, n_j, \ldots]$ so that we can refer to the task that starts first and to the task that ends last among a set of tasks, respectively.

violating the constraint formula of the task network. For example, a blocks-world method for achieving $on(v_1, v_2)$ would look like $(achieve(on(v_1, v_2)), d)$, where d is the task network in Fig. 4. To accomplish a goal task $(achieve[l])$, l needs to be true in the end, and this is an implicit constraint in all methods for goal tasks. If a goal is already true, then an empty plan can be used to achieve it. Thus, for each goal task, we (implicitly) have a method $(achieve[l], [(n : do[f]), (l, n)])$ which contains only one dummy primitive task f with no effects, and the constraint that the goal l is true immediately before $do[f]$.

Each primitive task has exactly one operator for it, where as a non-primitive task can have an arbitrary number of methods.

2.3. *Planning domains and problems*

A planning domain is a pair $\mathcal{D} = \langle Op, Me \rangle$, where Op is a set of operators, and Me is a set of methods.

A *planning problem instance* is a triple $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$, where $\mathcal{D}$ is a planning domain, I is the initial state, and d is the task network we need to plan for. The language of **P** is the HTN language $\mathcal{L}$ generated by the constant, predicate, and task symbols appearing in **P**, along with an infinite set of variables and an infinite set of node labels. Thus, the set of constants, predicates and tasks are all part of the input.

Next, we define some restrictions on HTN-planning problems. **P** is *primitive* if the task network d contains only primitive tasks. This corresponds to the case where the planner is used only for scheduling. **P** is *regular* if all the task networks in the methods and d contain at most one non-primitive task, and that non-primitive task is ordered with respect to all the other tasks in the network. Surprisingly, this class of HTN-planning problems is closely related to STRIPS-style planning, as we shall see in section 3.3. **P** is *propositional* if no variables are allowed. **P** is *totally ordered* if all the tasks in any task network are totally ordered.

PLAN EXISTENCE is the following problem: given $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$, is there a plan that solves **P**?

2.4. *Operational semantics*

In this section, we give a fixed point definition for the set of solutions for a given HTN-planning problem. Description of an equivalent model-theoretic semantics appear in [6].

First, we define how primitive tasks change the world when executed. Similar to the way it is done in STRIPS-style planning, we verify that the primitive task is executable and then update the input state based on the effects of the primitive task. More precisely, Let s be a state, and $f \in F$ be a primitive task symbol with the corresponding operator $[f(v_1, \ldots, v_k)(pre : l_1, \ldots, l_m)(post : l'_1, \ldots, l'_n)]$. We define the resulting state from executing f with ground parameters $c_1, \ldots, c_k$ as

$$apply(s, f, c_1, \ldots, c_k) = \begin{cases} \text{Undefined} & \text{if } l_i\theta \text{ is false in } s \text{ for some } i \text{ in } 1 \ldots m, \\ (s - E_n\theta) \cup E_p\theta & \text{otherwise,} \end{cases}$$

where θ is the substitution $\{c_i/v_i \mid i = 1 \ldots k\}$, and E_n, E_p are the sets of negative and positive literals in $l'_1, \ldots, l'_n$, respectively.

Next, we define the set of plans for a ground primitive task network. Let d be a primitive task network (one containing only primitive tasks), and let I be the initial state. A plan σ is a *completion* of d at I, denoted by $\sigma \in comp(d, I, \mathcal{D})$, if σ is a total ordering of the primitive tasks in a ground instance of d that satisfies the constraint formula of d. More precisely, Let $\sigma = (f_1(c_{11}, \ldots, c_{1k_1}), \ldots, f_m(c_{m1}, \ldots, c_{mk_m}))$ be a plan, s_0 be the initial state, and $s_i = apply(s_{i-1}, f_i, c_{i1}, \ldots, c_{ik_i})$ for $i = 1 \ldots m$ be the intermediate states, which are all defined (i.e., the preconditions of each f_i are satisfied in s_{i-1} and thus actions in the plan are executable). Let $d = [(n_1 : \alpha_1) \cdots (n_m : \alpha_m), \phi]$ be a ground primitive task network, and π be a permutation such that whenever $\pi(i) = j$, $\alpha_i = do[f_j(c_{j1}, \ldots, c_{jk_j})]$. Then $\sigma \in comp(d, s_0, \mathcal{D})$, if the constraint formula ϕ of d is satisfied. The constraint formula is evaluated as follows:

- $(c_i = c_j)$ is true, if c_i, c_j are the same constant symbols;
- $first[n_i, n_j, \ldots]$ evaluates to $\min\{\pi(i), \pi(j), \ldots\}$;
- $last[n_i, n_j, \ldots]$ evaluates to $\max\{\pi(i), \pi(j), \ldots\}$;
- $(n_i \prec n_j)$ is true if $\pi(i) < \pi(j)$;
- (l, n_i) is true if l holds in $s_{\pi(i)-1}$;
- (n_i, l) is true if l holds in $s_{\pi(i)}$;
- (n_i, l, n_j) is true if l holds for all s_e, $\pi(i) \leqslant e < \pi(j)$;
- logical connectives $\neg, \wedge, \vee$ are evaluated as in propositional logic.

If d is a primitive task network containing variables, then

$$comp(d, s_0, \mathcal{D}) = \{\sigma \mid \sigma \in comp(d', s_0, \mathcal{D}),\ d' \text{ is a ground instance of } d\}.$$

If d contains non-primitive tasks, then the set of completions for d is the empty set.

Now, we define how to do task decompositions. Let $d = [(n : \alpha)(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$ be a task network containing a non-primitive task α. Let $me = (\alpha', [(n'_1 : \alpha'_1) \ldots (n'_k : \alpha'_k), \phi']$ be a method[3], and θ be the most general unifier of α and α'. Then we define *reduce*(d, n, me) to be the task network obtained from $d\theta$ by replacing $(n : \alpha)\theta$ with the task nodes of the method, and incorporating the constraint formula of the method into the constraint formula of $d\theta$. More precisely,

$$reduce(d, n, me) = [(n'_1 : \alpha'_1)\theta \ldots (n'_k : \alpha'_k)\theta\ (n_1 : \alpha_1)\theta \ldots (n_m : \alpha_m)\theta, \phi'\theta \wedge \psi],$$

where ψ is obtained from $\phi\theta$ with the following modifications:

[3] All variables and node labels in the method must be renamed with variables and node labels that do not appear anywhere else.

- replace $(n < n_j)$ with $(\mathit{last}[n'_1, \ldots, n'_k] < n_j)$, as n_j must come after every task in the decomposition of n;
- replace $(n_j < n)$ with $(n_j < \mathit{first}[n'_1, \ldots, n'_k])$;
- replace (l, n) with $(l, \mathit{first}[n'_1, \ldots, n'_k])$, as l must be true immediately before the first task in the decomposition of n;
- replace (n, l) with $(\mathit{last}[n'_1, \ldots, n'_k], l)$, as l must be true immediately after the last task in the decomposition of n;
- replace (n, l, n_j) with $(\mathit{last}[n'_1, \ldots, n'_k], l, n_j])$;
- replace (n_j, l, n) with $(n_j, l, \mathit{first}[n'_1, \ldots, n'_k])$;
- everywhere that n appears in ϕ in a *first*[ ] or a *last*[ ] expression, replace it with $n'_1, \ldots, n'_k$.

We define $\mathit{red}(d, I, \mathcal{D})$, the set of reductions of d as

$$\mathit{red}(d, I, \mathcal{D}) = \{d' \mid d' \in \mathit{reduce}(d, n, \mathit{me}),\ n \text{ is the label for a non-primitive task in } d, \\ \text{and } \mathit{me} \text{ is a method in } \mathcal{D} \text{ for that task}\}.$$

Thus, a plan σ solves a primitive task network d at initial state I, iff $\sigma \in \mathit{comp}(d, I, \mathcal{D})$; a plan σ solves a non-primitive task network d at initial state I, iff σ solves some reduction $d' \in \mathit{red}(d, I, \mathcal{D})$ at initial state I .

Now, we can define the set of plans $\mathit{sol}(d, I, \mathcal{D})$ that solves a planning problem instance $\mathbf{P} = \langle d, I, \mathcal{D}\rangle$:

$$\begin{aligned}
\mathit{sol}_1(d, I, \mathcal{D}) &= \mathit{comp}(d, I, \mathcal{D}), \\
\mathit{sol}_{n+1}(d, I, \mathcal{D}) &= \mathit{sol}_n(d, I, \mathcal{D}) \cup \bigcup_{d' \in \mathit{red}(d, I, \mathcal{D})} \mathit{sol}_n(d', I, \mathcal{D}), \\
\mathit{sol}(d, I, \mathcal{D}) &= \bigcup_{n < \omega} \mathit{sol}_n(d, I, \mathcal{D}).
\end{aligned}$$

Intuitively, $\mathit{sol}_n(d, I, \mathcal{D})$ is the set of plans that can be derived in n steps, and $\mathit{sol}(d, I, \mathcal{D})$ is the set of plans that can be derived in any finite number of steps. In [6], we prove that the set of solutions according to the model-theoretic semantics coincide with $\mathit{sol}(d, I, \mathcal{D})$.

3. Results

Our complexity results are summarized in Table 1. In the following sections, we state the theorems and discuss their implications.

Table 1
Complexity of HTN planning.

Restrictions on non-primitive tasks	Must every HTN be totally ordered?	Are variables allowed?	
		no	yes
none	no	undecidable[α]	undecidable[α][β]
	yes	in EXPTIME; PSPACE-hard	in DEXPTIME; EXPSPACE-hard
"regularity" ($\leqslant 1$ non-primitive task, which must follow all primitive tasks)	doesn't matter	PSPACE-complete	EXPSPACE-complete[γ]
no non-primitive tasks	no	NP-complete	NP-complete
	yes	polynomial time	NP-complete

[α]Decidable with acyclicity restrictions.

[β]Undecidable even when the planning domain is fixed in advance.

[γ]In PSPACE when the planning domain is fixed in advance, and PSPACE-complete for some fixed planning domains.

3.1. *Undecidability results*

It is easy to show that we can simulate context-free grammars within HTN planning by using primitive tasks to emulate terminal symbols, compound tasks to emulate non-terminal symbols, and methods to encode grammar rules. More interesting is the fact that we can simulate any two context-free grammars, and with the help of task interleavings and constraints, we can check whether these two grammars have a common string in the languages they generate. Whether the intersection of the languages of two context-free grammars is non-empty is a semi-decidable problem [8]. Thus:

Theorem 1. PLAN EXISTENCE is strictly semi-decidable, even if **P** is restricted to be propositional, to have at most two tasks in any task network, and to have only totally ordered methods.

This result might seem surprising at first, since the state space (i.e., the number and size of states) is finite. If the planning problem were that of finding a path from the initial state to a goal state (as in STRIPS-style planning), indeed it would be decidable, because, for that problem, whenever there is a plan, there is also a plan that does not go through any state twice, and thus we need to examine only a finite number of plans. On the other hand, HTN planning can represent compound tasks accomplishing which might require going through the same state many times, and thus we have the undecidability result.

Instead of encoding each context-free grammar rule as a separate method, it is possible to encode these rules with predicates in the initial state, and to have a method containing variables and constraints such that only those decompositions corresponding to the grammar rules encoded in the initial state are allowed. Hence, even when the domain description (i.e., the set of operators and methods) is fixed in advance, it is possible to find planning domains for which planning is undecidable, as stated in the following theorem:

Theorem 2. There are HTN planning domains that contain only totally ordered methods each with at most two tasks, for which PLAN EXISTENCE is strictly semi-decidable.

3.2. Decidability and complexity results

One way to make PLAN EXISTENCE decidable is to restrict the methods to be acyclic. In that case, any task can be expanded up to only a finite depth, and thus the problem becomes decidable. To this end, we define a *k-level-mapping* to be a function *level*() from ground instances of tasks to the set $\{0, \ldots, k\}$, such that whenever we have a method that can expand a ground task α to a task network containing a ground task α', $level(\alpha) > level(\alpha')$. Furthermore, $level(\alpha)$ must be 0 for every primitive task α.

Intuitively, *level*() assigns levels to each ground task, and makes sure that tasks can be expanded into only lower level tasks, establishing an acyclic hierarchy. In this case, any task can be expanded to a depth of at most k. Therefore,

Theorem 3. PLAN EXISTENCE is decidable if **P** has a k-level-mapping for some integer k.

Examples of such planning domains can be found in manufacturing, where the product is constructed by first constructing the components and then combining them together.

Another way to make PLAN EXISTENCE decidable is to restrict the interactions among the tasks. Restricting the task networks to be totally ordered limits the interactions that can occur between tasks. Tasks need to be achieved serially, one after the other; interleaving subtasks for different tasks is not possible. Thus interactions between the tasks are limited to the input and output state of the tasks, and the "protection intervals", i.e., the literals that need to be preserved, which are represented by state constraints of the form (n, l, n').

Under the above conditions, we can create a table with an entry for each task, input/output state pair, and set of protected literals, that tells whether it is possible to achieve that task under those conditions. Using dynamic programming techniques we can compute the entries in the table in DOUBLE-EXPTIME, or in EXPTIME if the problem is further restricted to be propositional. As shown in the next section, STRIPS-style planning can be modeled using HTNs that satisfy these conditions, so we can use the complexity results on STRIPS-style planning in [2, 5] to establish a lower bound on the complexity of HTN planning. Thus:

Theorem 4. PLAN EXISTENCE is EXPSPACE-hard and in DOUBLE-EXPTIME if **P** is restricted to be totally ordered. PLAN EXISTENCE is PSPACE-hard and in EXPTIME if **P** is further restricted to be propositional.

If we restrict our planning problem to be regular, then there will be at most one non-primitive task in any task network (both the initial input task network, and those we obtain by expansions). Thus, subtasks in the expansions of different tasks cannot be interleaved, which is similar to what happens in Theorem 4. But in Theorem 4, there could be several non-primitive tasks in a task network, and we needed to keep track of all of them (which is why we used the table). If the planning problem is regular, we only need to keep track of a single non-primitive task, its input/final states, and the protected literals. Since the size of a state is at most exponential, the problem can be solved in exponential space. But even with regularity and several other restrictions, it is still possible to reduce an EXPSPACE-complete STRIPS-style planning problem (described in [5]) to the HTN framework. Thus:

Theorem 5. PLAN EXISTENCE is EXPSPACE-complete if **P** is restricted to be regular. It is still EXPSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

When we further restrict our problem to be propositional, it is still possible to define a reduction from propositional STRIPS-style planning, which is proven to be PSPACE-complete [2]. Thus the complexity goes down one level:

Theorem 6. PLAN EXISTENCE is PSPACE-complete if **P** is restricted to be regular and propositional. It is still PSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

If we allow variables but instead fix the planning domain $\mathcal{D}$ (i.e., the set of methods and operators) in advance, then the number of ground atoms and ground tasks is polynomial in the length of the input to the planner. Hence the complexity of regular HTN planning with a fixed planning domain with variables is no harder than the complexity of regular propositional HTN planning, which is shown to be PSPACE-complete in Theorem 6[4].

In the proof of Theorem 5.17 in [5] a set of three STRIPS operators with variables for which planning is PSPACE-complete is presented. The reduction described in the proof of Theorem 5 transforms this set of operators into a regular HTN planning domain (with variables) for which planning is PSPACE-complete. Hence:

[4] In a related result, [2] shows that propositional STRIPS-style planning is also PSPACE-complete. We investigate the relation between regular HTN planning and STRIPS-style planning in section 3.3.

Theorem 7. If **P** is restricted to be regular and $\mathcal{D}$ is fixed in advance, then PLAN EXISTENCE is in PSPACE. Furthermore, there exists a fixed regular HTN planning domain $\mathcal{D}$ for which PLAN EXISTENCE is PSPACE-complete.

Suppose a planning problem is primitive, and either propositional or totally ordered. Then the problem's membership in NP is easy to see: once we nondeterministically guess a total ordering and variable binding, we can check whether the constraint formula on the task network is satisfied in polynomial time. Furthermore, unless we require the planning problem to be both totally ordered and propositional, our constraint language enables us to represent the satisfiability problem, and thus we get NP-hardness. Hence:

Theorem 8. PLAN EXISTENCE is NP-complete if **P** is restricted to be primitive, or primitive and totally ordered, or primitive and propositional. However, PLAN EXISTENCE can be solved in polynomial time if **P** is restricted to be primitive, totally ordered, and propositional.

3.3. *Expressivity: HTNs versus STRIPS representation*

There has not been a clear consensus on what HTNs can and cannot represent. It was generally believed that although HTNs are more flexible compared to STRIPS-style planning, anything that can be done in HTN planning can be also done in STRIPS-style planning. Due to the lack of a formalism for HTN planning, such claims could not be proved or disproved. We address this question using the formalism in this paper.

When we compare HTNs and STRIPS, we observe that the HTN approach provides all the concepts (states, actions, goals) that STRIPS has. In fact, given a domain encoded as a set of STRIPS operators, we can transform it to an HTN planning domain, in low-order polynomial time. A straightforward transformation would be to declare one primitive task symbol for each STRIPS operator, and for every effect of each operator, to declare a method similar to the one in Fig. 4. Each such method contains the preconditions of the operator as goal tasks, and also the primitive task corresponding to the operator itself.

Below is a more instructive transformation, which demonstrates that the relationship between STRIPS-style planning and HTN planning is analogous to the relationship between right linear (regular) grammars and context-free grammars. We summarize the transformation below; for details see the proof of Theorem 5.

In this transformation, the HTN representation uses the same constants and predicates used in the STRIPS representation. For each STRIPS operator o, we declare a primitive task f with the same effects and preconditions as o. We also use a dummy primitive task f_d with no effects or preconditions. We declare a single compound task symbol t. For each primitive task f, we construct a method of the form

$$\boxed{perform[t]} \implies \boxed{\boxed{do[f]} \longrightarrow \boxed{perform[t]}}$$

We declare one last method $\boxed{perform[t]} \Rightarrow \boxed{do[f_d]}$. Note that t can be expanded to any sequence of actions ending with f_d, provided that the preconditions of each action are satisfied. The input task network has the form $[(n : perform[t]), (n, G_1) \wedge \cdots \wedge (n, G_m)]$ where $G_1, \ldots, G_m$ are the STRIPS-style goals we want to achieve. Note that the transformation produces regular HTN problems, which has exactly the same complexity as STRIPS-style planning. Thus, just as restricting context-free grammars to be right linear produces regular sets, restricting HTN methods to be regular produces STRIPS-style planning.

The next question is whether there exists a transformation in the other direction, that is whether it is possible to encode HTN planning problems as STRIPS-style planning problems. Intuitively, such a transformation cannot exist, because STRIPS-style planning lacks the concept of compound tasks, and its notion of goals is more restrictive than in HTN planning. For example, it does not provide means for declaring goals/constraints on the intermediate states as HTNs do. A more formal argument can be made as follows.

From Theorem 1, HTN planning with no function symbols (and thus only finitely many ground terms) is semi-decidable. Even if we require the domain description $\mathcal{D}$ to be fixed in advance (i.e., not part of the input), Theorem 2 tells us that there are HTN planning domains for which planning is semi-decidable. However, with no function symbols, STRIPS-style planning is decidable, regardless of whether or not the planning domain[5] is fixed in advance [5]. Thus:

Theorem 9. There does not exist a computable function ψ from the set of HTN planning problem instances to the set of STRIPS-style planning problem instances such that for any HTN-planning problem instance $\mathbf{P}$, and any plan σ, σ solves $\mathbf{P}$ iff $\psi(\sigma)$ solves $\psi(\mathbf{P})$[6].

Showing whether a polynomial or computable transformation exists is one way of comparing the expressivity of two languages. The lack of a computable transformation from HTN planning to STRIPS-style planning means that for some planning problems, the problem representation in STRIPS-style planning will be exponentially (or in some cases, even infinitely!) larger than in HTN planning. However, there are also some other ways in which one might want to compare expressivity. For example, for comparing the expressive power of knowledge representation languages, Baader [1] has developed an approach based on model-theoretic semantics.

According to Baader's definition, a knowledge-representation language L_2 is as expressive as L_1 iff there exists a function ψ (which does not have to be computable) that maps each set of sentences from L_1 to a set of sentences from L_2, such that the following property is satisfied whenever $\psi(\Gamma_1) = \Gamma_2$:

[5] Since STRIPS-style planning does not include methods, a STRIPS-style planning domain is simply a set of operators.

[6] In proving this theorem, we use the standard assumption that the STRIPS operators do not contain function symbols, nor do the HTN operators.

for any model of Γ_1, there is an equivalent (modulo renaming of symbols) model of Γ_2, and vice versa.

L_2 can express L_1 if and only if such a transformation exists. If L_2 can express L_1, but L_1 cannot express L_2, then L_2 is strictly more expressive than L_1.

To adapt Baader's definition to planning languages, there are two possible approaches: develop a model-theoretic semantics for planning, or use the operational semantics presented in section 2.4. As we discuss in [6], these two approaches yield different (non-equivalent) definitions of the expressivity of planning languages. However, HTN planning is more expressive than STRIPS-style planning according to both of these definitions. In [6] we prove this for the model-theoretic definition; below we present a definition of expressivity based on operational semantics, and we use this definition to prove that HTN planning is more expressive than STRIPS-style planning.

We define a planning language L_1 to be *as expressive as* a planning language L_2 iff there exists a function ψ from the set of planning problem instances in L_2 to the set of planning problem instances in L_1 such that for any planning problem instance $\mathbf{P}$ in L_2, $\mathbf{P}$ and $\psi(\mathbf{P})$ have the same set of solutions (plans) modulo symbol renaming. ψ need not be computable.

In the proofs of the undecidability theorems, we have shown that the set of solutions for an HTN-planning problem instance can be any context-free set, or even the intersection of any two context-free sets. In general, such a set cannot be expressed as the set of solutions to a STRIPS-style planning problem, because the set of solutions to a STRIPS-style planning problem correspond to a regular set (where the states in the planning domain correspond to the states of a finite automata, and the actions correspond to state transitions). Hence there does not exist a function ψ from the set of HTN-planning problem instances to the set of STRIPS-style planning problem instances that preserves the set of solutions. On the other hand, we have presented a transformation that maps each STRIPS-style planning problem instance to an HTN-planning problem instance with the same set of solutions. Thus we can conclude

Theorem 10. HTN planning is strictly more expressive than STRIPS-style planning.

The power of HTN planning comes from two things: (1) allowing multiple tasks and arbitrary constraint formulas in task networks, (2) compound tasks. Allowing multiple tasks and arbitrary formulae provides flexibility – but if all tasks were either primitive or goal (STRIPS-style) tasks, these could probably be expressed with STRIPS-style operators (albeit clumsily and using an exponential number of operators/predicates). Compound tasks provide an abstract representation for sets of primitive task networks, similar to the way non-terminal symbols provide an abstract representation for sets of strings in context-free grammars.

4. Conclusion

We have presented a formal description of HTN planning, and we have done a complexity analysis based on this formalism. From our results we can draw the following conclusions:

1. HTN planners can represent a broader set of planning domains than STRIPS-style planners. The transformations from HTN planning problems to STRIPS-style planning problems have revealed that STRIPS-style planning is a special case of HTN planning, and that the relation between them is analogous to the relation between context-free languages and regular languages. This contradicts the idea, held by some researchers, that HTNs are just an "efficiency hack".
2. Handling interactions among non-primitive tasks is the most difficult part of HTN planning. In particular, if subtasks in the expansions for different tasks can be interleaved, then planning is undecidable, under even a very severe set of restrictions. However restricting the planning problems to be totally-ordered or regular reduced the complexity significantly, because that limited the interactions among tasks.
3. In general, what restrictions we put on the non-primitive tasks has a bigger effect on complexity than whether or not we allow variables, or require tasks to be totally ordered.
4. If there are no restrictions on non-primitive tasks, then whether or not we require tasks to be totally ordered has a bigger effect (namely, decidability vs. undecidability) than whether or not we allow variables. But in the presence of restrictions on non-primitive tasks, whether or not we allow variables has a bigger effect than whether or not we require tasks to be totally ordered.

Currently, we are investigating how to use constraint satisfaction techniques such as tree search versus repair, value ordering and variable ordering heuristics to increase the efficiency of HTN planning. We are also investigating how to extend the action representation of HTNs to allow conditional, probabilistic and future effects.

Acknowledgement

We thank A. Barrett, W. Gasarch, R. Kambhampati, B. Kettler, and anonymous reviewers for their insightful comments.

Appendix

Theorem 1. PLAN EXISTENCE is strictly semi-decidable, even if **P** is restricted to be propositional, to have at most two tasks in any task network, and to have only totally ordered methods.

Proof. Membership: We can restate PLAN EXISTENCE as $\exists k$: $sol_k(d, I, \mathcal{D}) \neq \emptyset$. Thus the problem is in Σ^1.

Hardness: Given two context-free grammars G_1 and G_2, whether $L(G_1) \cap L(G_2)$ is non-empty is an undecidable problem [8]. We define a reduction from this problem to PLAN EXISTENCE as follows:

Without loss of generality, assume both G_1 and G_2 have the same binary alphabet Σ, and they are in Chomsky normal form (at most two symbols at the right hand side of production rules). Refer to [8] to see how any context-free grammar can be converted into this form. Similarly, assume that the sets of non terminals Γ_1 and Γ_2 for each grammar are disjoint; i.e., $\Gamma_1 \cap \Gamma_2 = \emptyset$. We also assume neither language contains the empty string. It is easy to check whether a CFG derives empty string. If both languages contain the empty string, then their intersection is non-empty; we can simply return a simple HTN problem that has a solution. If one of the languages does not contain the empty string, it does not affect the intersection to remove the empty string from the other language.

It is quite easy to see that using methods we can simulate context-free grammars: Primitive task symbols mimic the terminals, compound task symbols mimic the non-terminals, and methods mimic the production rules. The difficulty is in making sure there is a string produced by both G_1 and G_2. We achieve this with the help of the constraints in methods.

For each terminal $a \in \Sigma$, we introduce a proposition p_a. We also need another proposition called *turn*.

Let the initial state $I = \{turn\}$.

For each terminal $a \in \Sigma$, we introduce two primitive tasks (one for each grammar) f_{a_1} and f_{a_2} such that f_{a_1} has the preconditions $\{turn\}$ and effects $\{p_a, \neg turn\}$; f_{a_2} has the preconditions $\{p_a, \neg turn\}$ and effects $\{\neg p_a, turn\}$.

Intuitively, f_{a_1} produces p_a, and f_{a_2} consumes p_a. The proposition *turn* ensures that we use these primitive tasks alternately.

For each non-terminal B in each grammar, we introduce a compound task symbol t_B.

For each production rule $R: A \rightarrow B_1 B_2$, we introduce a method

$$(t_A, [(n_1 : \alpha_1)(n_2 : \alpha_2)\ (n_1 \prec n_2)]),$$

where

$$\alpha_i = \begin{cases} perform[t_{B_i}] & \text{if } B_i \text{ is a nonterminal,} \\ do[f_{a_1}] & \text{if } B_i \text{ is a terminal } a, \text{ and } R \text{ is a production rule of } G_1, \\ do[f_{a_2}] & \text{if } B_i \text{ is a terminal } a, \text{ and } R \text{ is a production rule of } G_2. \end{cases}$$

The input task network contains the three tasks $perform[t_{S_1}], perform[t_{S_2}]$, $do[f_{last}]$, where S_1, S_2 are the starting symbols of the grammars G_1, G_2, respectively, and f_{last} is a primitive task with no effects. The constraint formula states that $t_{S_1} \prec t_{last}$, and $t_{S_2} \prec f_{last}$, and that *turn* needs to be true immediately before f_{last}. The last condition ensures that the last primitive task f_{pa} belongs to G_2.

The task decompositions mimic the production rules of the grammars. The proposition *turn* ensures that each grammar contributes a primitive action to any plan alternatively, and the conditions with propositions p_a ensure that whenever G_1 contributes a primitive task f_{a_1}, G_2 has to contribute f_{a_2}. Thus, there is a plan iff G_1 and G_2 have a common word in their corresponding languages. $\square$

Theorem 2. There are HTN planning domains that contain only totally ordered methods each with at most two tasks, for which PLAN EXISTENCE is strictly semi-decidable.

Proof. We construct a planning domain $\mathcal{D}$ and show that planning in this domain is semi-decidable, using a reduction from the intersection of context-free grammars problem.

Domain description. We use four primitive tasks $f_{a_1}, f_{b_1}, f_{a_2}, f_{b_2}$ (they are exactly the same primitive tasks used in the previous proof), and another dummy primitive task f_{dummy} with no effects or preconditions.

We have three propositions $p_a, p_b, turn$, and a predicate $R(X, Y, Z)$, used for expressing production rules of the form $X \to YZ$.

We declare the following four operators that specify the effects of those tasks:

$$\begin{array}{lll}
(operator\ f_{a_1} & (pre : turn) & (post : p_a, \neg turn)) \\
(operator\ f_{a_2} & (pre : \neg turn, p_a) & (post : \neg p_a, turn)) \\
(operator\ f_{b_1} & (pre : turn) & (post : p_b, \neg turn)) \\
(operator\ f_{b_2} & (pre : \neg turn, p_b) & (post : \neg p_b, turn)) \\
(operator\ f_{dummy} & (pre :) & (post :))
\end{array}$$

We use five compound tasks $t(A_1), t(A_2), t(B_1), t(B_2), t(Dummy)$ corresponding to our primitive tasks.

We declare five methods describing how those compound tasks expand to their corresponding primitive tasks:

$$\begin{array}{ll}
(t(v)\ [(n : do[f_{a_1}]) & (v = A_1)]) \\
(t(v)\ [(n : do[f_{b_1}]) & (v = B_1)]) \\
(t(v)\ [(n : do[f_{a_2}]) & (v = A_2)]) \\
(t(v)\ [(n : do[f_{b_2}]) & (v = B_2)]) \\
(t(v)\ [(n : do[f_{dummy}]) & (v = Dummy)])
\end{array}$$

We use a predicate $R(v, v_1, v_2)$ to encode grammar rules. We declare a final method:

$$(t(v)[(n_1 : perform[t(v_1)])(n_2 : perform[t(v_2)]]) \ \ (n_1 \prec n_2) \wedge (n_1, R(v, v_1, v_2))])$$

Basicly, this method specifies that a task $t(X)$ can be expanded to $t(Y)t(Z)$ iff there is a production rule of the form $X \to YZ$. Thus we have a domain with 5 operators and 5 methods.

The reduction. Given two context-free grammars, here is how we create the initial state and the input task-network.

Let $G_i = \langle \Sigma, \Gamma_i, R_i \rangle$, $i = 1, 2$, be two context-free grammars. Without loss of generality, assume $\Sigma = \{a, b\}, \Gamma_1 \cap \Gamma_2 = \emptyset$, the production rules are in Chomsky normal form (at most two symbols at right hand sides), and the grammars do not use the symbols $\{A_1, A_2, B_1, B_2, Dummy\}$

Here is the initial state:

- For each production rule of the form $X \to YZ$, we assert a predicate $R(X, Y, Z)$.
- For each production rule of the form $X \to a$ from grammar i, we assert a predicate $R(X, A_i, Dummy)$. We handle rules of the form $X \to b$, similarly.
- Finally, we assert *turn*.

The input task network to the planner contains the three tasks $t(S_1), t(S_2)$, f_{dummy}, where S_1, S_2 are the starting symbols of the grammars G_1, G_2, respectively. The constraint formula states that both $t(S_1)$ and $t(S_2)$ precede f_{dummy}, and that *turn* needs to be true immediately before f_{last}. The last condition ensures that the last primitive task belongs to G_2.

How it works: The construction is very similar to that of Theorem 1. In that construction, we introduced a method for each production rule. This time, we observe that all those methods had the same structure, so instead we use a single method with variables and an extra constraint $R(X, Y, Z)$ that makes sure that we can expand $t(X)$ to $t(Y)t(Z)$ only when we have the corresponding production rule. The sequence of actions $t(S_i)$ can expand to corresponds to the strings that can be derived from S_i. The effects of the actions and the conditions on them ensure that in any final plan the actions from S_1 and S_2 alternate, and that whenever S_1 contributes an action, it has to be followed by the corresponding action in S_2. Obviously, the reduction can be done in linear time. □

Theorem 3. PLAN EXISTENCE is decidable if **P** has a k-level-mapping for some integer k.

Proof. When there exists a k-level-mapping, no task can be expanded to a depth more than k. Thus, whether a plan exists can be determined after a finite number of expansions. □

Theorem 4. PLAN EXISTENCE is EXPSPACE-hard and in DOUBLE-EXPTIME if **P** is restricted to be totally ordered. PLAN EXISTENCE is PSPACE-hard and in EXPTIME if **P** is further restricted to be propositional.

Proof. Membership: Here, we present an algorithm that runs in DOUBLE-EXPTIME, and solves the problem. In the propositional case, the number of atoms, the number of states etc. would go one level down, and thus, the same algorithm would solve the problem in EXPTIME.

The basic idea is this: for each ground task t, states s_I, s_F, and set of ground literals $L = \{l_1, \ldots, l_k\}$, we want to compute whether there exists a plan for t starting at s_I and ending at s_F while protecting the literals in L (i.e., without making them false). We store our partial results in a table with an entry for each tuple $\langle t, s_I, s_F, L\rangle$. An entry in the table has value either yes, no, or unknown.

Here is the algorithm:

1. Initialize all the entries in the table to unknown.
2. For each s_I, s_F, L and ground primitive task f_p, compute whether executing f_p at s_I results in s_F, and that all literals in L are true in both s_I and s_F. Insert the result in the table.
3. For each method $\langle t, (n_1 : \alpha_1) \dots (n_k : \alpha_k), \phi \rangle$ and the input task network do:
   - Replace each constraint of the form (n_i, l, n_j) with $(n_i, l, n_{i+1}) \wedge (n_{i+1}, l, n_{i+2}) \wedge \cdots \wedge (n_{j-1}, l, n_j)$. (For simplicity, we assume the label of a node reflects its position in the total order.)
   - Apply de Morgan's rule so that negations come before only atomic constraints.
4. Go over all the entries $\langle t, s_I, s_F, L \rangle$ in the table with value unknown, doing the following:

```
For all ground instances of methods for $t$ $\langle t, (n_1 : \alpha_1) \dots (n_k : \alpha_k), \phi \rangle$ do:
    For all $k+1$ tuples of states $(s_0, \dots, s_k)$ do:
        For all expansions $\phi'$ of $\phi$ into conjuncts do:
            (a) for each conjunct of the form $(n_i, l)$ or $(l, n_{i+1})$, check
                whether $s_i$ satisfies $l$.
            (b) For each $i \leqslant k$, let $L_i'$ be the set of literals $l$ such that
                $(n_i, l, n_{i+1})$ is a conjunct. Check whether the entry for
                $\langle t_i, s_{i-1}, s_i, L_i \rangle$ is yes.
            (c) Check whether the variable binding constraints are satisfied.
            (d) If all checks are OK, enter yes to the table for $\langle t, s_I, s_F, L \rangle$.
```

5. If step 4 modified the table, then goto step 4.
6. For all ground instances $\langle (n_1 : \alpha_1) \dots (n_k : \alpha_k), \phi \rangle$ of the input task network do

```
For all $k+1$ tuples of states $(s_0, \dots, s_k)$ do:
    For all expansions $\phi'$ of $\phi$ into conjuncts do:
        (a) for each conjunct of the form $(n_i, l)$ or $(l, n_{i+1})$, check whether
            $s_i$ satisfies $l$.
        (b) For each $i \leqslant k$, let $L_i'$ be the set of literals $l$ such that $(n_i, l, n_{i+1})$
            is a conjunct. Check whether the entry for $\langle t_i, s_{i-1}, s_i, L_i \rangle$ is yes.
        (c) Check whether the variable binding constraints are satisfied.
        (d) If all checks are OK, halt with success; if not, halt with failure.
```

The algorithm works bottom-up. For all ground tasks, state pairs and protection sets $\langle t, s_I, s_F, L\rangle$, it computes whether there exists a plan for t starting at s_I and ending at s_F that does not violate the literals in L. When step 4 terminates without any modification to the table, the table contains all the answers. Thus in step 6, we can check whether the input task network can be achieved.

The table has a doubly exponential number of entries (roughly the cube of the number of states times number of ground tasks). Step 4 is executed at most a doubly exponential number of times (when we make one modification at each step). At each execution step 4 goes over all the entries in the table, taking double exponential time. Processing each entry takes double exponential time. The resultant time is the product of these, which is still double exponential. The rest of the steps in the algorithm obviously do not take more than double exponential time. Thus the algorithm runs in double exponential time.

When we restrict the problem to be propositional, the number of states goes down from doubly exponential to exponential, and so does the size of table and the number of executions in all the steps. Thus in propositional case, the algorithm runs in exponential time.

Hardness: PLAN EXISTENCE, restricted to totally ordered regular planning domains, is a special case of our problem. But in Theorems 5 and 6, we prove that under this restricted version of PLAN EXISTENCE is EXPSPACE-hard (or PSPACE-hard in the propositional case). Thus the hardness follows. □

Theorem 5. PLAN EXISTENCE is EXPSPACE-complete if **P** is restricted to be regular. It is still EXPSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

Proof. Membership: It suffices to present a nondeterministic algorithm that uses at most exponential space and solves the problem, as EXPSPACE = N-EXPSPACE, so that is what we will do. Since all task networks will contain at most one non-primitive task, all we need to do is keep track of what atoms need to be true/false immediately before, immediately after, and along that single task. Since there are an exponential number of atoms, we can do this within exponential space. Here is the algorithm:

1. Let d be the input task network.
2. If d contains only primitive tasks, then
   non-deterministically guess a total-ordering and variable-binding.
   If it satisfies the constraint formula and the preconditions of the primitive tasks, then halt with success; if not, halt with failure.
3. Non-deterministically, guess a total-ordering and variable-binding. The task network will be of the form

$$[(n_1 : do[f_1]) \dots (n_m : do[f_m])(n : perform[t])(n'_1 : do[f'_1]) \dots (n'_u : do[f'_u])]$$

with ordering $n_1 \prec n_2 \prec \cdots \prec n_m \prec n \prec n'_1 \prec \cdots \prec n'_u$.
Note that t is the only non-primitive task in the network. Let s_i, s'_i, s_t be the states immediately after f_i, f'_i, t, respectively.

4. Eliminate all variable binding and task ordering constraints from the constraint formula using the guess in step 3.

5. Replace any constraint of the form (n_i, l, n'_j) with $(n_i, l, n_m) \wedge (n_m, l, n'_1) \wedge (n'_1, l, n'_j)$.

6. Replace any constraint of the form (n_i, l, n_j) or (n'_i, l, n'_j) with $(n_i, l) \wedge \cdots \wedge (n_{j-1}, l)$ and $(n'_i, l) \wedge \cdots \wedge (n'_{j-1}, l)$, respectively.

7. Process the constraint formula (using De Morgan's rule) so that negations apply to only atomic constraints.

8. Now the resultant constraint formula contains only conjuncts and disjuncts. For each disjunct, nondeterministically pick a component, obtaining a constraint formula containing only conjuncts.

9. Compute all the intermediate states before n and verify that all constraints of the form $(n_i, l), (l, n_i)$ are satisfied. Remove these constraints from the constraint formula.

10. For all the state constraints after n, use regression to determine what needs to be true immediately after n for those constraints to be satisfied[7].

11. Set the initial state I to $f_m(f_{m-1}(\ldots f_1(I)\ldots))$, i.e., the state that results from applying all the primitive tasks before t.

12. Now we can get rid of all the primitive tasks in the task network. The constraint formula contains only what needs to be true while we achieve t and what needs to be true immediately after we achieve t.

13. Nondeterministically, choose a method for t, expand it, and assign the resulting task network to d.

14. Go to step 2.

Hardness: In [5], we showed that plan existence problem in STRIPS representation is EXPSPACE-complete. Here we define a reduction from that problem.

The plan existence problem in STRIPS representation is defined as "Given a set of constants, a set of predicates, a set of STRIPS operators, an initial state and a goal, is there a plan that achieves the goal?"

[7] Here is how we do this. Consider the task $(n'_i : t'_i)$ and the condition (n'_i, l). When we regress this condition, we get *TRUE* if t'_i asserts l, *FALSE* if t'_i denies l, or (n'_{i-1}, l) if t'_i neither denies nor asserts l. This is what needs to be *TRUE* before t'_i, for the condition to hold after t'_i.

Given such a problem, we transform it into an HTN planning problem as follows:

We use the same set of constants and predicates. We will also have the same initial state. For each STRIPS operator o, we define a primitive task f_o that has exactly the same preconditions and postconditions as o. We also need an extra primitive task f_ε with no effects, that will be used as a dummy.

We will need a single non-primitive task t that can be expanded to any executable sequence of actions. Thus we declare the following methods for t:

- $(t, [(n_1 : f_\varepsilon), TRUE])$;
- for each STRIPS operator o

$$(t, [(n_1 : f_o)(n_2 : t), (n_1 \prec n_2)]).$$

Finally, the input task network will be of the form

$$\langle (n_1 : t), (n_1, g_1) \wedge (n_1, g_2) \cdots \wedge (n_1, g_k)) \rangle,$$

where g_i are the goals of the STRIPS problem.

The resulting HTN problem satisfies all the restrictions of the theorem. A plan σ solves the STRIPS plan existence problem instance iff σ is a solution for the HTN planning problem instance. Hence the reduction is correct. Obviously, the reduction is in polynomial time. □

Theorem 6. PLAN EXISTENCE is PSPACE-complete if **P** is restricted to be regular and propositional. It is still PSPACE-complete if **P** is further restricted to be totally ordered, with at most one non-primitive task symbol in the planning language, and all task networks containing at most two tasks.

Proof. Membership: The algorithm we presented for the membership proof in Theorem 5 works also for the propositional case. In the propositional case we have only a linear number of atoms, and as a result, the size of any state is polynomial. Thus the algorithm requires only polynomial space.

Hardness: In [2, 5], it is shown that plan existence problem in STRIPS representation is PSPACE-complete if it is restricted to be propositional. The reduction from STRIPS-style planning that we presented in the hardness proof of Theorem 5 also works for the propositional case. □

Theorem 7. If **P** is restricted to be regular and $\mathcal{D}$ is fixed in advance, then PLAN EXISTENCE is in PSPACE. Furthermore, there exists fixed regular HTN planning domains $\mathcal{D}$ for which PLAN EXISTENCE is PSPACE-complete.

Proof. When $\mathcal{D}$ is fixed in advance, the number of ground instances of predicates and tasks will be polynomial in the size of the input. Thus we can reduce it to propositional regular HTN-planning. As a direct consequence of Theorem 6, it is in PSPACE.

In the proof of Theorem 5.17 in [5], we had presented a set of STRIPS operators containing variables for which planning is PSPACE-complete. Applying the reduction defined in the hardness proof of Theorem 5 to this set of operators would give a regular HTN planning domain (containing variables) with the same set of solutions and complexity as its STRIPS counterpart. □

Theorem 8. PLAN EXISTENCE is NP-complete if **P** is restricted to be primitive, or primitive and totally ordered, or primitive and propositional. However, PLAN EXISTENCE is in polynomial time **P** is restricted to be primitive, totally ordered, and propositional.

Proof. If **P** is primitive, propositional and totally ordered, we can compute whether an atomic constraint is satisfied in linear time. In order to find a plan, all we need to do is to check whether the constraint formula is satisfied, which can be done in polynomial time.

Membership: Given a primitive task network, we can nondeterministically guess a total ordering and variable binding, and then we can verify that it satisfies the constraint formula in polynomial time. Thus the problem is always in NP.

Hardness: There are three cases:

Case 1. Primitive and propositional.

We define a reduction from satisfiability problem as follows:

Given a boolean formula, we define a planning problem such that it uses the same set of propositions as the boolean formula. we have two primitive tasks for each proposition, one that deletes the proposition, and one that adds the proposition. The initial state is empty. The input task network contains all the primitive tasks, and the constraint formula states that the boolean formula needs to be true in the final state.

If there exists a total ordering that satisfies the constraint formula, the truth values of propositions in the final state would satisfy the boolean formula; if there is a truth assignment that satisfies the boolean formula, we can order the tasks such that if a proposition p is assigned true, then the primitive task that adds it is ordered after the primitive task that deletes it (and vice versa), coming up with a plan that achieves the task network.

Obviously, the reduction can be done in linear time.

Case 2. Primitive and totally ordered.

Again, we define a reduction from satisfiability problem.

We will use two constant symbols t and f, standing for true and false, respectively.

For each proposition p in the boolean formula, we will introduce a unary predicate P, and a unary primitive task symbol $T_p(v_p)$ that has the effect $P(v_p)$.

The initial state will be empty, and the input task network will contain one task for each proposition, namely $T_p(v_p)$.

We construct a formula F from the input boolean formula by replacing each proposition p with $P(t)$. Our constraint formula will require F to be true in the final state.

If there exists a truth assignment that satisfies the boolean formula, we can construct a variable binding such that v_p is bound to t whenever p is assigned true, and v_p is bound to f otherwise. This variable binding would make sure the constraints are satisfied.

If there exists a variable binding that achieves the task network, we construct the following truth assignment that satisfies the boolean formula. We assign true to p iff v_p is bound to t.

Obviously, the reduction can be done in polynomial time.

Case 3. Primitive.

Both case 1 and case 2 are special cases of case 3, so hardness follows immediately. □

References

[1] F. Baader, A formal definition for expressive power of knowledge representation languages, in: *Proceedings of the 9th European Conference on Artificial Intelligence* (Pitman, Stockholm, Sweden, Aug. 1990).

[2] T. Bylander, Complexity results for planning, *IJCAI-91* (1991).

[3] D. Chapman, Planning for conjunctive goals, *Artificial Intelligence* 32 (1987) 333–378.

[4] M. Drummond, Refining and extending the procedural net, in: *Proc. IJCAI-85* (1985).

[5] K. Erol, D. Nau and V.S. Subrahmanian, Complexity, decidability and undecidability results for domain-independent planning, *Artificial Intelligence* (to appear). A more detailed version is available as Tech. Report CS-TR-2797, UMIACS-TR-91-154, SRC-TR-91-96, University of Maryland, College Park, MD (1992).

[6] K. Erol, J. Hendler and D. Nau, Semantics for hierarchical task network planning, Technical Report CS-TR-3239, UMIACS-TR-94-31, Computer Science Dept., University of Maryland (March 1994).

[7] R.E. Fikes and N.J. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, *Artificial Intelligence* 2(3/4) (1971).

[8] Hopcroft and Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, California, 1979).

[9] S. Kambhampati and J. Hendler, A validation structure based theory of plan modification and reuse, *Artificial Intelligence* (May 1992).

[10] A.L. Lansky, Localized event-based reasoning for multiagent domains, *Computational Intelligence Journal* (1988).

[11] E.D. Sacerdoti, The nonlinear nature of plans, in: *Proceedings of IJCAI* (1975) pp. 206–214.

[12] A. Tate, Generating project networks, in: *Proceedings of IJCAI* (1977) pp. 888–889.

[13] S.A. Vere, Planning in time: windows and durations for activities and goals, *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI* 5(3) (1983) 246–247.

[14] D. Wilkins, *Practical Planning: Extending the Classical AI Planning Paradigm* (Morgan-Kaufmann, 1988).

[15] Q. Yang, Formalizing planning knowledge for hierarchical planning, *Computational Intelligence* 6 (1990) 12–24.