Fully Observable Nondeterministic HTN Planning – Formalisation and Complexity Results

Dillon Chen, Pascal Bercher

The Australian National University, Canberra, Australia
{dillon.chen, pascal.bercher}@anu.edu.au

Abstract

Much progress has been made in advancing the state of the art of HTN planning theory in recent years. However, scarce studies have been made with regards to the theory and complexity of HTN problems on nondeterministic domains. In this paper we provide a novel formalisation for fully observable nondeterministic HTN planning. We propose and study different solution criteria which differ in when nondeterministic action outcomes are considered: at plan generation or at plan execution. We integrate our solution criteria with notions of weak and strong plans canonical in nondeterministic planning and identify similarities and differences with plans in other fields of AI planning.

We also provide completeness results for a majority of HTN problem subclasses and show the significant result that problems are not made any harder under nondeterminism for certain solution criteria by using compilation techniques to deterministic HTN planning. This supports and justifies the practicality and scalability of extending HTN problems over nondeterministic domains to deal with real world scenarios.

1 Introduction

Hierarchical Task Network (HTN) planning is a powerful planning formalisation focused on problem decomposition. Tasks in HTN planning can be either primitive or compound. The former corresponds to classical planning actions, while the latter are abstract notions of actions which can be decomposed into a set of subtasks by methods defined in an HTN problem. The hierarchy induced by such compound tasks gives HTN planning and its variants such as HTN planning with task insertion (TIHTN) much expressive power by the ability to model problems of various complexities, from standard non-hierarchical planning to undecidable problems (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Alford et al. 2014; Alford, Bercher, and Aha 2015a,b).

Background and Related Work

The applications and extensions of hierarchical planning to deal with uncertainties in the real world are manifold: integrating HTNs for solving and learning standard nondeterministic planning problems with planners such as YoYo (Kuter et al. 2005, 2009) and ND-SHOP2 (Kuter and Nau

2004) and the learning algorithm HTN-MAKER[ND] (Hogg, Kuter, and Muñoz-Avila 2009), integrating plan repair (Goldman, Kuter, and Freedman 2020; Höller et al. 2020), integration with Belief-Desire-Intention agents, which consist of goals analogous to decomposition in HTN planning, for planning ahead in dynamic systems (Sardiña, de Silva, and Padgham 2006; Meneguzzi and de Silva 2015), integrating planning and learning via hierarchical methods in dynamically changing environments (Patra et al. 2020), and planning over partially observable domains (Kuter et al. 2007; Zhuo, Muñoz-Avila, and Yang 2014; Richter and Biundo 2017).

To the best of our knowledge there do not exist any formalisations for an extension of standard deterministic HTN problems that feature uncertainty in action outcomes. The formalisation that is closest to a Fully Observable Nondeterministic (FOND) HTN approach is that used in the planners YoYo and ND-SHOP2 (Kuter et al. 2005, 2009) which employs HTN techniques for solving standard nondeterministic planning problems. In other words, their model uses HTNs to serve as advice on how to solve the underlying classical FOND problem, not to find a decomposition of the initial plan as in HTN planning. The case is similar for HTN-MAKER[ND] which learns HTNs in order to speed up search for the ND-SHOP2 planner over standard nondeterministic planning domains.

Contributions

Our proposed formalisation extends standard HTN planning by introducing nondeterminism to primitive tasks and introducing additional solution criteria for solving nondeterministic HTN problems: linearisation- and outcome-dependent solutions. The former deals with nondeterministic action effects at plan generation and the latter at plan execution. We also define weak and strong variants of such solution criteria canonical to nondeterministic planning (Cimatti et al. 2003) which describe the likeliness of plan success.

Abundant studies have also been made regarding the theory of planners with uncertainty (Geffner and Bonet 2013) but none for FOND HTN planners outside the theoretical and empirical complexity analysis for the ND-SHOP2 and YoYo planners over certain example domains (Kuter and Nau 2004; Kuter et al. 2005, 2009). We will study the complexity of the plan existence problem for the various solution

Hierarchy	Order	FOD	FOND		
			Weak	**Strong**	
				linearisation-dependent	*outcome-dependent*
primitive	total	P*	NP [4.1]	P*	[4.8]
	partial	NP$^\alpha$	NP [4.2]	NP [4.7]	PSPACE [5.1]
no recursion (acyclic)	total	PSPACE$^\beta$	PSPACE [4.4]	PSPACE	[4.8]
	partial	NEXPTIME$^\beta$	NEXPTIME [4.4]	NEXPTIME [4.7]	EXPSPACE* [5.2]
regular	total	PSPACE$^\alpha$	PSPACE [4.5]	PSPACE	[4.8]
	partial	PSPACE$^\alpha$	PSPACE [4.5]	PSPACE [4.7]	EXPSPACE* [5.3]
tail-recursion	total	PSPACE$^\beta$	PSPACE [4.4]	PSPACE	[4.8]
	partial	EXPSPACE$^{\alpha,\beta}$	EXPSPACE [4.4]	EXPSPACE [4.7]	semidecidable* [3.1]
arbitrary recursion	total	EXPTIME$^\beta$	EXPTIME [4.4]	EXPTIME	[4.8]
	partial	semi- & undecidable$^{\alpha,\gamma}$	semi- & undecidable [3.1]	semi- & undecidable [3.1]	semi- & undecidable [3.1]

Table 1: Comparison of complexity results for FOD and FOND HTN planning. Results marked $^\alpha$ are shown by Erol, Hendler, and Nau (1996), and $^\beta$ by Alford, Bercher, and Aha (2015a). The undecidability result $^\gamma$ was reproduced by Geier and Bercher (2011) for the HTN formalisation used in this study. Classes are complete unless marked * where only membership is known. Prop. 2.12 collapses weak problems and Prop. 2.15 collapses totally ordered strong problems.

criteria and defined HTN problem subclasses where recursion on compound task decomposition is restricted: acyclic, regular and tail-recursive problems (Erol, Hendler, and Nau 1996; Alford et al. 2012; Alford, Bercher, and Aha 2015a). We show that for several of the solution criteria, problems can be compiled into their deterministic counterparts without much overhead and have the same complexity. We also provide completeness results for a majority of other problems and loose bounds for everything else. Complexity results are summarised in Table 1 alongside results from Fully Observable Deterministic (FOD) HTN planning and relations and compilations between problems are illustrated in Fig. 1.
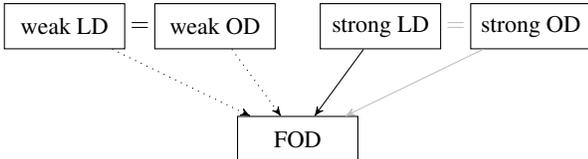

Figure 1: Relations and reductions between problems. Solid arrows between problems indicate that we have specified explicit polynomial reductions for all problem subclasses and dotted arrows for all but primitive and regular problems. Gray symbols are true only if total order on tasks is enforced.

2 Formalism

Here we will provide a complete formalisation of FOND HTN planning by introducing nondeterminism to primitive tasks in the formalisation for deterministic HTN planning by Bercher, Alford, and Höller (2019). We begin with the fundamental building blocks of HTN planning: task networks, the concept of partially ordered action sequences. We then use task networks to construct an HTN planning domain, problem and finally all of our proposed solution criteria.

Definition 2.1 (Task Network). A *task network* tn is a tuple $\langle T, \prec, \alpha \rangle$ where

- T is a finite set of *task id symbols*,
- $\prec \subseteq T \times T$ is a strict partial order on T,
- $\alpha : T \to N$ maps a task id to some task name in N.

The requirement of task id symbols and names arises from the fact that some tasks, such as 'move', may occur several times in a task network. Thus, task id symbols are unique but the mapping α is not necessarily injective. Usually when we refer to a 'task' of a task network, we mean task id symbol.

We also introduce an equivalence between two task networks. Two networks $tn = \langle T, \prec, \alpha \rangle$ and $tn' = \langle T', \prec', \alpha' \rangle$ are *isomorphic* if there exists a bijection $\sigma : T \to T'$ where for all $t, t' \in T$, $(t, t') \in \prec$ iff $(\sigma(t), \sigma(t')) \in \prec'$ and $\alpha(t) = \alpha'(\sigma(t))$. In other words, they have the same underlying structure but with different task id symbols.

Definition 2.2 (Planning domain). An *HTN domain* $\mathcal{D}$ is a tuple $\langle \mathcal{F}, N_P, N_C, \delta, \mathcal{M} \rangle$ where we have

- a finite set of *facts* $\mathcal{F}$,
- a finite set of *primitive task names* N_P,
- a finite set of *compound task names* N_C,
- an action mapping $\delta : N_P \to \mathcal{A}$,
- a finite set of *decomposition methods* $\mathcal{M}$,

where $N_P \cup N_C = N$ are disjoint, and $\mathcal{A} \subseteq 2^{\mathcal{F}} \times 2^{2^{\mathcal{F}} \times 2^{\mathcal{F}}}$ denotes the set of nondeterministic primitive tasks. A *primitive task* or *action* is a tuple of preconditions and effects $a = (\text{pre}(a), \text{eff}(a))$ with $\text{pre}(a) \in 2^{\mathcal{F}}$ and $\text{eff}(a) = \{(\text{add}_i(a), \text{del}_i(a)) \mid 1 \leq i \leq n\}$ for n dependent on a. For ease of readability, we will use deterministic $(\text{add}, \text{pre}, \text{del})$ notation actions which only have one effect. This is reminiscent of classical planning actions and so we formalise the idea of applying primitive tasks to states.

Define $\mathcal{S} = 2^{\mathcal{F}}$ where states are associated with a set of facts. Let $\tau : \mathcal{A} \times \mathcal{S} \to \{\top, \bot\}$ denote executability with

$$\tau(a, s) = \begin{cases} \top & \text{if } \text{pre}(a) \subseteq s \\ \bot & \text{otherwise.} \end{cases}$$

For ease of notation later, we also define τ on primitive task names and primitive tasks by $\tau(n, s) = \tau(\delta(n), s)$ and $\tau(t, s) = \tau(\delta(\alpha(t)), s)$ for $n \in N_P$ and $t \in T$ respectively.

Next we define an application function $\gamma : \mathcal{A} \times \mathcal{S} \to 2^{\mathcal{S}}$. For $a \in \mathcal{A}, s \in \mathcal{S}$, if $\tau(a, s) = \bot$, $\gamma(a, s)$ is undefined. Otherwise, we have

$$\gamma(a, s) = \{(s \setminus \text{del}_i(a)) \cup \text{add}_i(a) \mid 1 \le i \le |\text{eff}(a)|\}.$$

Similarly define the application function for primitive task names and primitive tasks by $\gamma(n, s) = \gamma(\delta(n), s)$ and $\gamma(t, s) = \gamma(\delta(\alpha(t)), s)$.

Definition 2.3 (Planning problem). An *HTN problem* $\mathcal{P}$ is a tuple $\langle \mathcal{D}, s_I, tn_I \rangle$ with $\mathcal{D}$ an HTN domain, $s_I \in 2^{\mathcal{F}}$ an initial state and tn_I an initial task network.

Next, we introduce the machinery to refine our task networks by decomposing compound tasks. Compound tasks and methods contribute to the abstract concept of hierarchy between tasks in an HTN problem as decomposition induces an implicit transitive relation between compound tasks.

Definition 2.4 (Decomposition). Define $m = (c, tn_m)$ with $c \in N_C$ and $tn_m = \langle T_m, \prec_m, \alpha_m \rangle$ to be a *(decomposition) method*. We can apply m to $tn_1 = \langle T_1, \prec_1, \alpha_1 \rangle$ if there exists some $t \in T_1$ where $\alpha_1(t) = c$, and in this case we say m decomposes t in tn_1 to generate a task network $tn_2 = \langle T_2, \prec_2, \alpha_2 \rangle$ with

$$\begin{aligned} T_2 &:= T_1' \cup T_m', \\ \prec_2 &:= (\prec_1 \cup \prec_m') |_{T_1'} \\ &\quad \cup \{(t_1, t_2) \in T_1' \times T_m' \mid (t_1, t) \in \prec_1\} \\ &\quad \cup \{(t_1, t_2) \in T_m' \times T_1' \mid (t, t_2) \in \prec_1\}, \\ \alpha_2 &:= (\alpha_1 \cup \alpha_m') |_{T_1'}, \end{aligned}$$

where $T_1' = T_1 \setminus \{t\}$ and $tn_m' = \langle T_m', \prec_m', \alpha_m' \rangle$ is a task network isomorphic to tn_m such that $T_1' \cap T_m' = \emptyset$. The requirement for disjoint T_1' and T_m' is such that the function $(\alpha_1 \cup \alpha_m') |_{T_1'}$ is well defined and $(\prec_1 \cup \prec_m') |_{T_1'}$ is still partial. The $|_{T_1'}$ symbol denotes restriction of the α map and $\prec$ ordering in the canonical way to only tasks in T_1'.

Note that in the context of deterministic HTN planning, there are two possible ways to define method *preconditions*: one which appends preconditions to methods which is seen in the HTN domains of YoYo (Kuter et al. 2009) and one which enforces that the appended precondition be checked exactly before a first task inside the corresponding method as seen in deterministic HTN planner SHOP2 (Nau et al. 2003) and in HDDL[1], an extension of PDDL for hierarchical planning problems (Höller et al. 2020). The first version can be compiled away easily by adding a first task with the

---

[1]Note that this definition was implemented only as an optional feature in HDDL and not necessarily enforced.

same precondition and no effect to the task network, whereas the second version has no obvious compilation even in the deterministic setting. However, this second definition could also be integrated into the provided formalism if desired.

We now begin to describe solution criteria. We adapt the ideas of *weak* and *strong* solutions canonical to planners in nondeterministic domains as described by Cimatti et al. (2003). Readers familiar with these concepts will notice that *strong cyclic* solutions have been omitted in this study due to the lack of space and thus will be left for future work.

Definition 2.5 (Primitive task network). Let $\mathcal{P}$ be an HTN problem and tn be a task network. Then tn is a *primitive task network* if $\alpha(T) \subseteq N_P$, meaning that it only consists of primitive tasks. We say that tn is an *achievable primitive task network* if it can be decomposed from the initial task network tn_I of $\mathcal{P}$ by a finite number of methods from $\mathcal{M}$.

We propose two ways to define solutions for FOND HTN problems: via linearisation and policies. Both have their own trade-offs between complexity and plan flexibility.

Definition 2.6 (Linearisation). Let $tn = \langle T, \prec, \alpha \rangle$ be a task network and $T' \subseteq T$ with $|T'| = n$. A *linearisation of T'* is an ordering of all its elements compatible with the partial ordering $\prec$. A *linearisation of tn* is a linearisation of T.

Definition 2.7 (Linearisation-dependent solution). Let $\mathcal{P}$ be an HTN problem and $tn = \langle T, \prec, \alpha \rangle$ a task network. Then we say that a linearisation $t_1, \ldots, t_n$ of tn is a *strong linearisation-dependent (LD) solution* for $\mathcal{P}$ if tn is an achievable primitive task network, and for all $1 \le i \le n$ and for all $s \in S_i$ it holds that $\tau(t_i, s) = \top$, where S_i is defined recursively by

$$S_i := \begin{cases} \bigcup_{s \in S_{i-1}} \gamma(t_{i-1}, s), & \text{if } 1 < i \le n \\ \{s_I\}, & \text{if } i = 1. \end{cases}$$

On the other hand, a linearisation $t_1, \ldots, t_n$ of tn is defined to be a *weak LD solution* if tn is an achievable primitive task network and there exists states $s_1, \ldots, s_n, s_{n+1}$ with $s_1 = s_I$ and for all $1 \le i \le n$, we have $\tau(t_i, s_i) = \top$ and $s_{i+1} \in \gamma(t_i, s_i)$.

For strong solutions, this is equivalent to saying that all tasks are guaranteed to be executed in order of the linearisation regardless of nondeterministic task effects. This concept is similar to conformant plans (Goldman and Boddy 1996) in standard non-hierarchical planning under uncertainty. On the other hand, weak LD solutions only require linearisations to be executable for favourable task effects.

However, the definition of strong LD solutions appears to be very strict as nondeterministic task effects are only considered at plan generation and not execution. Thus, we will introduce the more flexible solution using policies. A policy tells a planner what action to execute given a current state and task history. The input for remembering previously applied tasks is necessary for preserving task ordering.

Definition 2.8 (Policy). A *policy* π *for a primitive task network* tn is a function $\pi : 2^T \times \mathcal{S} \to T$ where $\langle (T', s), t \rangle \in \pi$ only if $t \notin T'$, and $\forall \tilde{t} \in T \setminus T', \tilde{t} \not\prec t$.

We now define an execution structure and sequence induced by a policy introduced by Cimatti et al. (2003) for planners over nondeterministic domains which is conceptually analogous to a state transition system. We also introduce an additional requirement to an execution sequence in the existing literature where for any path in the system, the tasks which trace out the path preserve partial ordering.

Definition 2.9 (Execution structure). Let $\mathcal{D}$ be an HTN domain and tn be a primitive task network. Let $S \subseteq \mathcal{S}$ and $T' \subseteq T$. Then S is *executable* by T' if there exists a sequence of states $s_1, \ldots, s_{n+1}$ in S (possibly with repetition) and a linearisation $t_1, \ldots, t_n$ of T' with $s_{i+1} \in \gamma(t_i, s_i)$ for all $1 \le i \le n$.

The *execution structure* induced by a policy π for tn is a tuple $K = \langle \mathcal{Q}, \mathcal{R} \rangle$ where $\mathcal{Q} \subseteq \mathcal{S}$ and $\mathcal{R} \subseteq \mathcal{S} \times T \times \mathcal{S}$ are minimal sets satisfying the conditions $s_I \in \mathcal{Q}$, and if $s \in \mathcal{Q}$ and there exists $S \subseteq \mathcal{Q}, T' \subseteq T$ with

- $s \in S$,
- S is executable by T',
- $\pi(T', s) = t$, and
- $\tau(t, s) = \top$,

then for all $s' \in \gamma(t, s)$ we have $s' \in \mathcal{Q}$ and $(s, t, s') \in \mathcal{R}$.

Definition 2.10 (Execution sequence). Let $K = \langle \mathcal{Q}, \mathcal{R} \rangle$ be the execution structure induced by a policy. An *execution sequence* of K is a finite sequence of task-state tuples $\langle t_1, s_1 \rangle, \ldots, \langle t_k, s_k \rangle$ with

- $s_1 = s_I$,
- $\tau(t_k, s_k) = \top$,
- $(s_i, t_i, s_{i+1}) \in \mathcal{R}$ for $1 \le i < k$, and
- $t_1, \ldots, t_k$ is a linearisation of $\{t_i \mid 1 \le i \le k\}$.

An execution sequence is *complete* if $t_1, \ldots, t_k$ is a linearisation of tn.

Definition 2.11 (Outcome-dependent solution). Let $\mathcal{P}$ be an HTN problem and tn a task network. We say that a policy π for tn is a *strong outcome-dependent (OD) solution* if tn is an achievable primitive task network and every execution sequence of K induced by π is complete.

We say that π for tn is a *weak OD solution* if tn is an achievable primitive task network and there exists a complete execution sequence of K induced by π.

In other words for strong solutions, if we were to fix effects for each task, then there exists an executable linearisation of such tasks with fixed effects. Thus, the main difference between strong OD and LD solutions is that task linearisation is dynamic and determined at plan execution for the former and fixed at plan generation for the latter.

The following two propositions compare our proposed solution definitions. The first states that the two definitions for weak solutions are equivalent such that we can collapse linearisation and outcome dependency. However, this is false for strong solutions where implication only goes one way.

Proposition 2.12. *Let $\mathcal{P}$ be a FOND HTN problem. There exists a linearisation of tn that is a weak LD solution iff there exists a policy for tn that is a weak OD solution.*

Proof. ($\Longrightarrow$) Suppose the linearisation $t_1, \ldots, t_n$ of tn is a weak LD solution for $\mathcal{P}$. Then by definition there exists states $s_1, \ldots, s_n$ with $\tau(t_i, s_i) = \top$ for $1 \le i \le n$ with $s_1 = s_I$. Then define a policy π for tn by

$$\pi = \{\langle (T_i, s_i), t_{i+1} \rangle \mid 1 \le i < n\}$$

where $T_i = \{t_j \mid 1 \le j < i\}$. Its induced execution structure is $K = \langle \mathcal{Q}, \mathcal{R} \rangle$ with $\mathcal{Q} = \{s_1\} \cup \bigcup_{1 \le i < n} \gamma(t_i, s_i)$ and $\mathcal{R} = \bigcup_{1 \le i < n} \{(s_i, t_i, \tilde{s}) \mid \tilde{s} \in \gamma(t_i, s_i)\}$. A complete execution sequence of K is given by $\langle t_1, s_1 \rangle, \ldots, \langle t_n, s_n \rangle$. Thus, π for tn is a weak OD solution.

($\Longleftarrow$) Suppose the policy π for tn is a weak OD solution with induced execution structure $K = \langle \mathcal{Q}, \mathcal{R} \rangle$. Then it has a complete execution sequence $\langle t_1, s_1 \rangle, \ldots, \langle t_n, s_n \rangle$ and so the linearisation $t_1, \ldots, t_n$ of tn is a weak LD solution. $\square$

Proposition 2.13. *Let $\mathcal{P}$ be a FOND HTN problem. If there exists a linearisation of tn that is a strong LD solution, then there exists a policy for tn which is a strong OD solution. However, the converse statement does not hold.*

Proof. ($\Longrightarrow$) Suppose $t_1, \ldots, t_n$ of tn is a strong LD solution. Define the policy π for tn by

$$\pi = \bigcup_{1 \le i < n} \{\langle (T_i, s_{i,j}), t_{i+1} \rangle \mid s_{i,j} \in S_i\},$$

again with T_i defined in the proof of the previous proposition. Its induced execution structure is $K = \langle \mathcal{Q}, \mathcal{R} \rangle$ with $\mathcal{Q} = \{s_1\} \cup \bigcup_{1 \le i < n} \bigcup_{s_{i,j} \in S_i} \gamma(t_i, s_{i,j})$ and $\mathcal{R} = \bigcup_{1 \le i < n} \bigcup_{s_{i,j} \in S_i} \{(s_{i,j}, t_i, \tilde{s}) \mid \tilde{s} \in \gamma(t_i, s_{i,j})\}$.

The execution sequences of K are all of the form $\langle t_1, s_{1,j_1} \rangle, \ldots, \langle t_n, s_{n,j_n} \rangle$ where $s_{1,j_1} = s_I$ and $s_{i,j_i} \in \gamma(t_{i-1}, s_{i-1,j_{i-1}})$ and are complete since $t_1, \ldots, t_n$ is a linearisation of tn by assumption. Thus, π for tn is a strong OD solution.

($\Longleftarrow\!\!\!/$) Consider the following HTN problem where a strong OD solution exists but not a strong LD solution. Let $\mathcal{D} = \langle \mathcal{F}, N_P, \emptyset, \delta, \emptyset \rangle$ with $\mathcal{F} = \{1, 2\}$, $N_P = \{a, b, c\}$, and mapping δ defined by

$$a \mapsto (\emptyset, \{(\{1\}, \emptyset), (\{2\}, \emptyset)\}),$$
$$b \mapsto (\{1\}, \{2\}, \emptyset),$$
$$c \mapsto (\{2\}, \{1\}, \emptyset).$$

Define the problem $\mathcal{P} = \langle \mathcal{D}, s_I, tn_I \rangle$ with $s_I = \emptyset$ and $tn_I = \langle T, \prec, \alpha \rangle$, $T = \{t_1, t_2, t_3\}$, $\prec = \{(t_1, t_2), (t_1, t_3)\}$ and α defined by $t_1 \mapsto a, t_2 \mapsto b, t_3 \mapsto c$.

Then a strong OD solution is π for tn_I defined by

$$(\emptyset, s_I) \mapsto t_1, \quad \begin{array}{ll} (\{t_1\}, s_1) \mapsto t_2, & (\{t_1, t_2\}, s_3) \mapsto t_3, \\ (\{t_1\}, s_2) \mapsto t_3, & (\{t_1, t_3\}, s_3) \mapsto t_2, \end{array}$$

with $s_1 = \{1\}, s_2 = \{2\}, s_3 = \{1, 2\}$.

All this information can be represented compactly with a diagram (Fig. 2) of the induced execution structure where circles denote states and directed arrows determine relations.

There are only two execution sequences in the execution structure induced by π given by $\langle t_1, s_I \rangle, \langle t_2, s_1 \rangle, \langle t_3, s_3 \rangle$ and $\langle t_1, s_I \rangle, \langle t_3, s_2 \rangle, \langle t_2, s_3 \rangle$ which are both complete such that π is a strong OD solution.
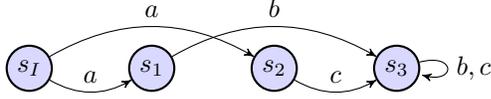
Figure 2: Execution structure induced by π.

However, there are no linearisations of tn which are strong LD solutions. The only two possible order preserving linearisations t_1, t_2, t_3 and t_1, t_3, t_2 do not satisfy the LD solution criterion as they are not executable in order for all possible nondeterministic action effects. $\square$

However, there exists a powerful condition on the ordering of tasks in an HTN domain which when met allows us to collapse strong LD and OD solutions.

Definition 2.14 (Totally ordered planning problem). An HTN problem $\mathcal{P}$ is a *totally ordered planning problem* if the ordering of its initial task network tn_I and the task networks of every method is total.

Proposition 2.15. *Let $\mathcal{P}$ be a totally ordered FOND HTN problem. There exists a linearisation of tn that is a strong LD solution iff there exists a policy π for tn which is a strong OD solution.*

Proof. For totally ordered primitive tasks, there exists only one possible candidate policy which corresponds with the linearisation of tasks induced by the total ordering. $\square$

A policy for a task network can become exponential in size which can be realised by noticing that the execution structure representation of a policy has a tree structure with branching resulting from nondeterministic task effects. However, we can verify that a policy is a solution for a primitive task network using a depth-first search on the execution structure to check whether there exists/all execution sequences are complete for weak/strong solutions respectively.

Proposition 2.16. *Let $\mathcal{P}$ be a FOND HTN problem. Let tn be a primitive task network and π a policy. It is in P in the size of π to verify that π for tn is an OD solution.* $\square$

We will show in Section 5 that the problem for determining the existence of a policy which contributes towards a solution for primitive task networks is PSPACE-complete despite the fact the a policy is exponential in size.

3 Problem Class Definitions

Erol, Hendler, and Nau (1996) showed that general FOD HTN planning is undecidable by reduction from the problem of whether the languages produced by two context-free grammars have a non-empty intersection. The proof was reproduced by Geier and Bercher (2011) to show that this fact is still true for their simplified formalisation we have built upon. Since FOD versions of FOND problems provide lower complexity bounds as the former are special cases of the latter with $|\text{eff}(a)| = 1$ for all actions, general FOND HTN planning is also undecidable. However, breadth-first search can be used to find a solution so we have semidecidability.

Theorem 3.1. *Let $\mathcal{P}$ be a FOND HTN problem. Deciding whether $\mathcal{P}$ has a solution is semi- & undecidable. This holds for $\{weak, strong\} \times \{LD, OD\}$ problems.* $\square$

This shows that it is not feasible to consider general HTN planning for practical applications. However, there exist restrictions on recursion of compound task decompositions problems easier. The first restriction which we will introduce is elimination of hierarchies in a FOND HTN problem.

Definition 3.2 (Primitive planning problem). An HTN problem $\mathcal{P}$ is *primitive* if tn_I is primitive and hence an achievable primitive task network. Note that sets N_C and $\mathcal{M}$ are now irrelevant.

The hierarchical problems we will study are regular, acyclic (Erol, Hendler, and Nau 1996) and tail-recursive problems (Alford et al. 2012; Alford, Bercher, and Aha 2015a).

Definition 3.3 (Regular problem). An HTN problem $\mathcal{P}$ is *regular* if for its initial task network $tn_I = \langle T, \prec, \alpha \rangle$ and for all its methods $(c, \langle T, \prec, \alpha \rangle) \in \mathcal{M}$ it holds that

- there is at most one compound task in T, and
- if $t \in T$ is compound, it is the *last* task, meaning that for all $t' \in T$ with $t' \neq t$ we have $t' \prec t$.

We rely on the concept of stratifications proposed by Alford et al. (2012) to help define acyclic and tail-recursive problems.

Definition 3.4 (Stratification). A *stratification* on a set S is a total order $\leq$ on S. An inclusion-maximal subset $C \subseteq S$ is a *stratum* if for all $x, y \in C$ both $x \leq y$ and $y \leq x$ holds.

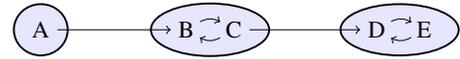

Figure 3: Stratification on S.

A stratification can be represented diagrammatically (Bercher and Höller 2018) by a directed graph (Fig. 3). For example, let $S = S_1 \cup S_2 \cup S_3$ with $S_1 = \{A\}, S_2 = \{B, C\}, S_3 = \{D, E\}$ be stratas. Add the relations $B \leq A$ and $D \leq C$ and the number of strata is still 3.

Definition 3.5 (Acyclic problem). An HTN problem $\mathcal{P}$ is *acyclic* if no compound task can reach itself via decomposition. More formally, we can define a stratification on N_C in $\mathcal{P}$ with

- $c \leq c'$ if there exists a method $(c, \langle T, \prec, \alpha \rangle) \in \mathcal{M}$ and $\alpha(c') \in T$, and
- for all $c, c' \in N_C$, if $c \leq c'$, then $c' \not\leq c$.

Definition 3.6 (Tail-recursive problem). An HTN problem $\mathcal{P}$ is *tail-recursive* if we can define a stratification on N_C of $\mathcal{P}$ where for all methods $(c, \langle T, \prec, \alpha \rangle)$ it holds that

- if there exists a last compound task $t \in T$, then we have $\alpha(t) \leq c$, and
- for any non-last compound task $t \in T$, we have $\alpha(t) \leq c$ and $c \not\leq \alpha(t)$.

In other words, the last task of a tail-recursive task network in a method (if it exists) is at most as hard as the decomposed task c, and any other non-last task is on a lower stratum and hence easier than c. Note further that regular and acyclic problems are special cases of tail-recursiveness. Although these definitions are not talked about much in Section 4, they will be studied more closely in Section 5.

4 Complexity Results via Compilation

In this section we will derive a majority of complexity results by compiling both strong LD and weak problems into deterministic problems to get membership for some problem subclasses and provide algorithms for the remaining problems. In fact, all results here will be complete with the exception of a problem shown to be in P. Recall that deterministic problems are a special case of FOND HTN planning so we also have hardness without many complex reductions.

Weak Problems

The compilation of weak problems to FOD problems will require introducing compound tasks so clearly we cannot apply this technique for primitive problems. Thus, we will give explicit complexity proofs for such problems first.

Theorem 4.1. *Let $\mathcal{P}$ be a totally ordered, primitive FOND HTN problem. Deciding whether $\mathcal{P}$ has a weak solution is* NP-*complete.*

Proof. Membership: we check that the only possible solution candidate, the linearisation $t_1, \ldots, t_n$ induced by a total ordering, is a solution by guessing the effects of the primitive tasks and verifying in linear time that the tasks are executable at each state from the effects we guessed. Specifically, guess effects $(\text{add}_i, \text{del}_i)$ for $\delta(t_i)$ and then check that for all $1 \leq i \leq n, \tau(t_i, s_i) = \top$ where s_1 is the initial state of the problem and $s_i = (s_{i-1} \setminus \text{del}_i) \cup \text{add}_i$ for $1 < i \leq n$.

Hardness: we reduce from the NP-complete SAT problem (Cook 1971). Let $X = \{x_1, \ldots, x_n\}$ be a set of boolean variables, and $\tilde{X} = \{\neg x \mid x \in X\}$ be negations. Let $\mathcal{C} = \{c_1, \ldots, c_m\}$ be a set of clauses over our variables, and say that $x_i \in c_j$ if setting x_i to true sets c_j to be true. Similarly say that $\neg x_i \in c_j$ if setting x_i to false sets c_j to be true.

Then we can define a domain $\mathcal{D} = \langle \mathcal{F}, N_P, \emptyset, \delta, \emptyset \rangle$ with $\mathcal{F} = \mathcal{C}$, $N_P = \{e_1, \ldots e_n, k\}$ and δ defined by

$$e_i \mapsto \langle \emptyset, \{ (\{c_j \in \mathcal{C} \mid x_i \in c_j\}, \emptyset),$$
$$(\{c_j \in \mathcal{C} \mid \neg x_i \in c_j\}, \emptyset) \} \rangle,$$
$$k \mapsto \langle \mathcal{C}, \emptyset, \emptyset \rangle.$$

Define a planning problem $\mathcal{P} = \langle \mathcal{D}, s_I, tn_I \rangle$ with $s_I = \emptyset$, $tn_I = \langle T, \prec, \alpha \rangle$, $T = \{t_1, \ldots, t_n, t_\varepsilon\}$, the total ordering $\prec$ defined by $t_1 \prec \ldots \prec t_n \prec t_\varepsilon$ and α by $t_i \mapsto e_i$ and $t_\varepsilon \mapsto k$.

To justify how satisfiability of the set of clauses in $\mathcal{C}$ corresponds with existence of a weak solution, suppose that we have boolean assignments for each x_i such that $\mathcal{C}$ is satisfied. Then choose the first outcome of each e_i if x_i is true, and choose the second otherwise. Since the choice of x_i satisfies $\mathcal{C}$, we would have progressed to a state with all elements of $\mathcal{C}$ after action e_n so we can apply k. Conversely, if no assignment of variables satisfies $\mathcal{C}$, then no selection of outcomes

of the actions e_i will progress the initial state to $\mathcal{C}$ and hence we cannot execute k. $\square$

The above membership proof technique can also be used for partially ordered primitive problems with an additional step. Now we guess a linearisation preserving the partial order and then verify executability of such linearisation.

Corollary 4.2. *Let $\mathcal{P}$ be a primitive FOND HTN problem. Deciding whether $\mathcal{P}$ has a weak solution is in* NP. $\square$

Now we will provide the compilation technique for all classes but regular problems, with which we will deal with later individually. The idea of the compilation is that we replace each nondeterministic primitive task with a compound task with a decomposition into a deterministic task for each effect in the original one. This domain transformation concept where tasks in the original planning domains are replaced with compound tasks is similarly used for HTN plan and goal recognition (Höller et al. 2018) and HTN plan repair (Höller et al. 2020).

Lemma 4.3. *Let $\mathcal{P}$ be a FOND HTN problem. $\mathcal{P}$ can be compiled in* P *time into the deterministic version of the problem, such that $\mathcal{P}$ has a weak solution iff its compilation has a solution. Furthermore, the compilation preserves acyclic and tail-recursive problem subclasses.*

Proof. We will replace each primitive task $t_i \in N_P$ with a compound task c_i and define deterministic primitive tasks $t'_{i,j}$ and methods $m_{i,j}$ for $1 \leq j \leq n$ with n the number of effects of $a_i = \delta(t_i)$. Extend δ to include maps

$$t'_{i,j} \mapsto (\text{pre}(a_i), \text{add}_j(a_i), \text{del}_j(a_i)).$$

Then define $m_{i,j} = (c_i, tn_{i,j})$ where $tn_{i,j}$ is the task network with one task $t'_{i,j}$. The compilation is polynomial since we introduce linearly many new tasks and methods for each task in the original problem. Tail-recursive problems are preserved as the implicitly modified methods still obey the tail-recursive restriction and similarly for acyclic problems.

We now show correspondence between solutions of the original and compiled problem. Suppose a weak solution exists for the former. Then in the compiled problem, we can choose decompositions corresponding to the task effects which contribute to the weak solution for the compound tasks replacing the nondeterministic tasks to get an executable linearisation. Conversely, if a linearisation exists for the compiled deterministic problem, we can choose the task effects corresponding to methods of the compound c_i's to get a weak solution for the original FOND problem. $\square$

Given that the compilation is in polynomial time, we get the same membership results from the deterministic versions of the corresponding problem subclasses.

Theorem 4.4. *Let $\mathcal{P}$ be a general, acyclic or tail-rec. FOND HTN problem. The complexity of deciding whether $\mathcal{P}$ has a weak solution is equivalent to its FOD counterpart.* $\square$

The reason why we cannot directly adapt Lem. 4.3 for regular problems is because some decompositions are modified to be no longer regular. However, we can still use the exact same progression search proof for regular problems

(Erol, Hendler, and Nau 1996) which looks at the size of the largest task network in the model under progression in our determinised problem. This is because the space bound of the algorithm does not change for our compilation as the newly added compound tasks get replaced by a single primitive and hence do not increase the size of any task network.

Corollary 4.5. *Let $\mathcal{P}$ be a regular FOND HTN problem. The complexity of deciding whether $\mathcal{P}$ has a weak solution is equivalent to its FOD counterpart.* $\square$

Strong Linearisation-Dependent Problems

Strong LD solutions behave similarly to conformant plans for planning under uncertainty (Goldman and Boddy 1996) as both require finding a linearisation that is executable regardless of action effects. We can also compile strong LD problems to their FOD counterparts in polynomial time using a similar method for compiling nondeterministic conformant problems into classical problems (Albore, Palacios, and Geffner 2010; Palacios and Geffner 2009). However in contrast to weak solutions, the compilation also works for all FOND HTN problem subclasses.

Lemma 4.6. *Let $\mathcal{P}$ be a FOND HTN problem. $\mathcal{P}$ can be compiled in P time into the deterministic version of the problem, such that $\mathcal{P}$ has a strong LD solution iff its compilation has a solution. Furthermore, the compilation preserves all problem subclasses.*

Proof. First we modify the problem syntactically where for all $a \in \delta(N_p)$, we set $\mathrm{del}_i(a) = \mathrm{del}_i(a) \backslash \mathrm{add}_i(a)$. This is because by definition facts in the intersection $\mathrm{add}_i(a) \cap \mathrm{del}_i(a)$ are always added when a is executed so doing this changes nothing semantically. However, the below compilation will not preserve the correspondence of solutions if we do not compile this away. Then determinise the problem by collapsing task effects and replacing all $a \in \delta(N_p)$ with

$$a' = (\mathrm{pre}(a), \bigcap_{1 \le j \le n} \mathrm{add}_j(a), \bigcup_{1 \le j \le n} \mathrm{del}_j(a)),$$

where n_i is the number of effects of a_i. This takes polynomial time in the number of action effects to compile.

Solving the determinised problem is equivalent to solving the original problem as both require finding an executable linearisation. This is because we can solve a FOND problem by finding a candidate strong LD linearisation solution and verifying executability by performing the above determinisation but this commutes with the action of determinising the problem first then finding an executable linearisation.

The problem of verifying executability of a determinised linearisation is the same as of the FOND one since all the effects of a are weaker than those of a'. Specifically, the add of each a' are subsets of those of a while on the other hand the deletes of a' are supersets of a, so executability of each a' guarantees executability of a for all possible effects.

Conversely, a strong LD solution can be compiled to an equivalent deterministic one which can be seen by using an inductive argument. Assume that a length $k > 0$ prefix of a strong LD solution $\delta(t_1), \ldots, \delta(t_n)$ can be determinised. Then applying the first k actions advances the initial state to some state s. By definition of a strong LD solution, for all

$s' \in \gamma(t_{k+1}, s)$, we have $\tau(t_{k+2}, s') = \top$. This is equivalent to saying that $\tau(t_{k+2}, \bigcap_{s' \in \gamma(t_{k+1}, s)} s') = \top$ and hence $\delta(t_{k+1})$ can be determinised in the above manner. The argument for the base case $k = 0$ is the same where $s = s_I$.

Thus, we have a direct correspondence between FOD and FOND solutions since one exists iff the other does. Furthermore, problem subclasses are preserved as no changes are made to compound tasks and methods. $\square$

Again, since the above compilation is in polynomial time, we get the same membership results from the deterministic versions of the problems. Recall Prop. 2.15 to get the same result for totally ordered strong OD problems.

Theorem 4.7. *Let $\mathcal{P}$ be a FOND HTN problem. The complexity of deciding whether $\mathcal{P}$ has a strong LD solution is equivalent to its FOD counterpart.* $\square$

Corollary 4.8. *Let $\mathcal{P}$ be a totally ordered FOND HTN problem. The complexity of deciding whether $\mathcal{P}$ has a strong OD solution is equivalent to its FOD counterpart.* $\square$

5 Complexity Results for Strong Outcome-Dependent Problems

Unlike for the other two solution criteria, there is no easy method for compiling a partially ordered FOND HTN problem with strong OD solutions to a deterministic HTN problem. This is reinforced by our first significant result in this section which states that partially ordered OD primitive problems are PSPACE-complete, exactly one step harder than the deterministic problem which is NP-complete. The increase of hardness should be expected given the extra flexibility and strength of strong OD solutions in comparison to strong LD solutions or conformant plans. However, this does not completely rule out the existence of a compilation given that it is yet an unknown result whether NP = PSPACE.

The result that the plan existence problem for strong primitive OD solutions is PSPACE-complete although a policy is possibly exponential in size. This mirrors the result from non-hierarchical propositional planning where it is PSPACE-complete to determine existence of a plan (Bylander 1994) but a plan can still be exponential in length. We use two tools for the proof: alternating Turing machines (ATM), an extension of Turing machines with additional *existential* and *universal* states, and reduction from a two player game with a bounded number of turns. The intuition for the former is that the alternation feature of ATMs deal with nondeterminism naturally (Rintanen 2004). The intuition for the latter is that nondeterminism is associated to an enemy in adversarial game playing and that the fixed number of actions in primitive problems fit the criteria of having a bounded number of turns in the game we reduce from.

Theorem 5.1. *Let $\mathcal{P}$ be a partially ordered, primitive FOND HTN problem. Deciding whether $\mathcal{P}$ has a strong OD solution is PSPACE-complete.*

Proof. Membership: we show that the problem is in AP, problems decidable in polynomial time by an ATM, and use that AP = PSPACE (Chandra and Stockmeyer 1976; Kozen 1976) to show PSPACE-membership. Let tn have n tasks.

The ATM begins by branching existentially to find a candidate first task t_{i_1}, then universally for all effects eff_{i_1} of said task. Then encode in classical TM language and solve the problem of whether t_{i_1} is executable in s_I and store the state s_1 progressed by the effect eff_{i_1} applied on s_I. Then branch existentially again to find a second task t_{i_2} and universally for effects eff_{i_2} of said task and encode the problem of whether t_{i_2} is executable in s_1, check that $t_{i_2} \not\prec t_{i_1}$, $t_{i_2} \neq t_{i_1}$ and store the state s_2 progressed by eff_{i_2} on s_1. Repeat until the n-th and last step where we branch existentially to find a t_{i_n} then universally for effects eff_{i_n} and encode and solve whether t_{i_n} is executable in the state s_{n-1} progressed under the corresponding search and t_{i_n} is unique from and not behind any of $t_{i_1}, \ldots, t_{i_{n-1}}$. Each of the n steps requires polynomial time to encode to an ATM and execute, so the problem is in AP = PSPACE.

The acceptance of the ATM corresponds with the existence of a policy, as this ATM algorithm asks whether for all task effects of a chosen task there exists a next task dependent on the task effect which is executable from the state progressed from such effect and so on until all tasks are executed in a sequence preserving the partial ordering.

Hardness: we reduce from an (m, n, k)-game which is PSPACE-complete[2] for $k \geq 4$ (Hsieh and Tsai 2007; Reisch 1980). In an (m, n, k)-game, two players (P1 and P2) take turns marking empty squares on a m by n grid with symbols like X and O, one for each player, and whoever marks k in a row, column or diagonal wins. The problem we reduce from is whether the first player P1 has a winning strategy against the second player P2.

We begin modelling an (m, n, k)-game by defining mn tasks mapping bijectively to mn nondeterministic $turn_i$ actions each representing a P1 move and all possible P2 countermoves in the action effects. We then define several *win* and *not_lose* actions to help determine whether an instance of a game is won by P1. Moreover, we define several *illegal* actions to deal with nondeterministic effects of $turn_i$ that might correspond to P2 playing an illegal move. Finally, we introduce a *clear* action which gets triggered when a win condition for P1 is met or an illegal is played by P2 and is used to complete a strong OD solution by making all other tasks executable. The execution of a strong OD policy can be divided into three main phases:

- *Phase 1*: playing an instance of an (m, n, k)-game,
- *Phase 2*: verifying that P1 has won a game,
- *Phase 3*: executing every remaining task as required in a strong OD solution.

Now we begin to formalise and justify our argument. First, associate each $i \in \{1, \ldots, mn\}$ with a unique square in an m by n board. Next, let L denote the set of all possible size k sets of integers between 1 and mn inclusive corresponding to length k rows, columns or diagonals (*k-line*). A loose upper bound for the cardinality of L is given

---

[2]Hsieh and Tsai proved a more general result by showing that (m, n, k, p, q)-games are PSPACE-complete for $k - p \geq \max\{3, p\}$. An (m, n, k)-game is a special case where $p = q = 1$.

by $|L| \leq 3mn$ since there are at most mn of each rows, columns and diagonals on an m by n board.

Then define an HTN domain $\mathcal{D} = \langle \mathcal{F}, N_P, \emptyset, \delta, \emptyset \rangle$ with the facts given by

$$\mathcal{F} = \{x_i, \neg x_i, o_i, \neg o_i \mid 1 \leq i \leq mn\} \cup L \cup \{R, C\}.$$

The facts x_i indicate that square i on the board is occupied by a P1 symbol and the $\neg x_i$ indicate that it is not occupied by a P1 symbol. In Phase 1 of plan execution which models a game, both x_i and $\neg x_i$ will never be set simultaneously. However, after a win condition is met, we no longer follow this rule. The same can be said similarly for o_i and $\neg o_i$ representing existence of P2 symbols on the board.

The facts in L as described earlier represent k-lines and is used with *not_lose* actions which will be defined below to check whether P2 has not won a game and P1 has not lost. Note that although elements of L are defined as sets, each element is only one fact. The fact R, denoting a running game, is modified by *fin_turn* actions which again will be defined later to model a winning move for P1. The fact C is used in conjunction with a *clear* task, also to be defined later, which allows us to enter Phase 3 of a strong OD policy.

We define task names and mapping δ of such names to primitive actions as follows, noting that the $turn_i$ are the only nondeterministic tasks

$$
\begin{aligned}
turn_i &\mapsto (\{\neg x_i\}, \\
&\quad \{(\{x_i, o_j\}, \{\neg x_i, \neg o_j\}) \mid 1 \leq j \leq mn\}) \\
fin\_turn_i &\mapsto (\{\neg x_i, \neg o_i, R\}, \{x_i\}, \{R\}) \\
win_\lambda &\mapsto (\{x_i \mid i \in \lambda\}, \{x_j, \neg x_j \mid 1 \leq j \leq mn\}, \emptyset) \\
not\_lose_i &\mapsto (\{\neg o_i\}, \{\lambda \mid i \in \lambda, \lambda \in L\}, \emptyset) \\
not\_lose_0 &\mapsto (L, \{C\}, \emptyset) \\
illegal_i &\mapsto (\{x_i, o_i\}, \{C\}, \emptyset) \\
clear &\mapsto (\{C\}, \mathcal{F}, \emptyset)
\end{aligned}
$$

for $1 \leq i \leq mn$ and one win_λ for each $\lambda \in L$. We will need to finish constructing the primitive FOND HTN problem with the task id symbol mapping and partial ordering of the initial task network before being able to explain the purpose of such tasks.

We define the problem $\mathcal{P} = \langle \mathcal{D}, s_I, tn_I \rangle$ with $s_I = \{\neg x_i, \neg o_i \mid 1 \leq i \leq mn\} \cup \{R\}$ corresponding to an empty board. Let $tn_I = \langle T, \prec, \alpha \rangle$ be a primitive task network with α mapping task id symbols in T bijectively to each of the primitive task names defined above except *clear*, where instead we define N task id symbols mapping to *clear* by α, one copy for each of the other actions. Specifically, $N = |L| + 4mn + 1 \leq 7mn + 1$ where there are $|L|$ number of win_λ and mn of each $turn_i$, fin_turn_i, not_lose_i and $illegal_i$. Finally, define a partial ordering with $win_\lambda \prec not\_lose_i$ for all possible (λ, i) tuples with $\lambda \in L$ and $1 \leq i \leq mn$.

The reduction is polynomial as there are at most $7mn + 2$ facts and $2(7mn + 1)$ task symbols mapping surjectively into the set of task names.

Phase 1 The $turn_i$ task corresponds in the world of an (m, n, k)-game to a P1 move followed by a P2 countermove.
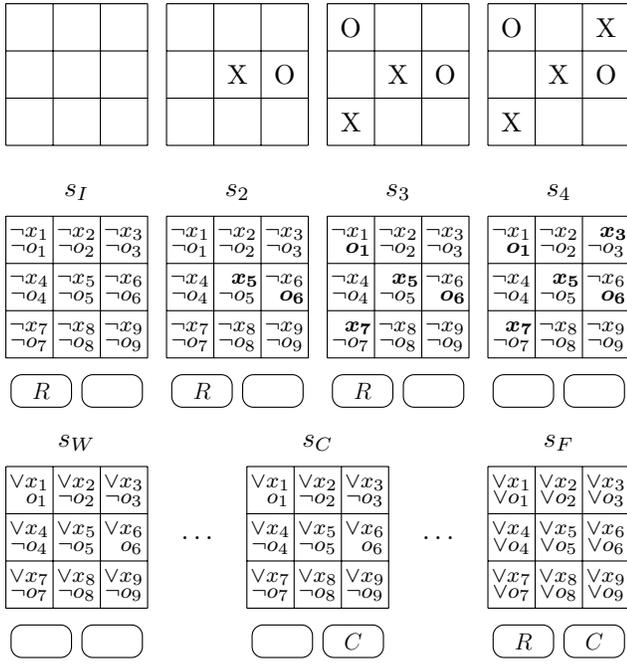
Figure 4: An instance of a Tic-Tac-Toe game. The top row illustrates the usual depiction of such a game and the bottom two rows show the state variables of corresponding states and executed tasks from the reduction to FOND HTN planning. Facts λ are not shown and $\lor x_i$ indicates both x_i and $\neg x_i$ is set and similarly for $\lor o_i$ with o_i and $\neg o_i$.

Specifically, the effects associated with adding x_i and deleting $\neg x_i$ indicate P1 placing a symbol at square i if possible, and the nondeterministic effects associated with adding o_i and deleting $\neg o_i$ indicate P2 placing a symbol at j. The *fin_turn*$_i$ also corresponds with P1 playing a move, but now with no effect for a P2 move. This is defined to be a 'winning move' for P1 which should only be played once in Phase 1 to avoid a simultaneous win for P2 from *turn*$_i$ effects. It is also defined to be only placed once in a game with the introduction of the R fact such that P1 cannot play k *fin_turn*$_i$ actions in a row to win at start of the game.

Phase 2 The *win*$_\lambda$ task sets off Phase 2 once P1 wins a game by playing a k-line. Once one *win*$_\lambda$ can be executed the remaining *win*$_\lambda$ can be as well. Then the *not_lose*$_i$ tasks can be executed and preserve the partial ordering to turn on facts in L. Setting all the facts in L allows executability of *not_lose*$_0$ which determines that P2 has not won as well.

Phase 3 Then we enter Phase 3 where we can execute all duplicates of *clear* and hence all remaining tasks yet to be executed as *clear* adds all facts in the model to a state. The partial ordering is defined in order to prevent all the *not_lose*$_i$ tasks from being executed in s_I and flag a false win.

Fig. 4 illustrates an example execution sequence corresponding to a P1 win in an $(3, 3, 3)$-game, or Tic-Tac-Toe. We begin with Phase 1 where the states s_I, s_2, s_3, s_4 correspond to the above game states of the Tic-Tac-Toe instance. The tasks and effects that progress s_I to s_4 in order are *turn*$_5$: eff$_6$, *turn*$_7$: eff$_1$ and *fin_turn*$_3$. Then we enter Phase 2 where *win*$_\lambda$ progresses s_4 to s_W representing a win state with precondition $\{x_3, x_5, x_7\}$. The state s_W is then progressed to s_C representing a clear state by first executing the remaining *win* tasks followed by *not_lose*$_2$, *not_lose*$_4$, *not_lose*$_5$, *not_lose*$_9$ and then *not_lose*$_0$. Equivalently, the planner can also execute *not_lose*$_3$, *not_lose*$_5$ and *not_lose*$_7$ before *not_lose*$_0$ instead. Finally, we enter Phase 3 where the remaining tasks are executed in an alternating fashion with a duplicate of *clear* in between each of the other tasks to progress s_C to the final state s_F.

Note that in this reduction it is possible for P2 to play an illegal move by placing a symbol on top of an already existing symbol by nondeterministic effects of *turn*$_i$. If P2 plays a symbol on top of a P1 symbol, then an *illegal*$_i$ task becomes executable which similarly to above allows executability of all duplicates of *clear* and hence every other task. This is intended since a strong OD solution corresponds to a winning strategy for P1 and we can disregard illegal moves since a winning strategy assumes no illegal moves. Otherwise if P2 plays a symbol on top of another P2 symbol, then although this move is illegal, this benefits P1 and in no way benefits P2 since this is equivalent to P2 losing a turn. Thus, this case is also not an issue in the reduction.

To see the correspondence between the solutions of the original and the reduced problem, first assume that we have a winning strategy for an (m, n, k)-game. Then a policy would execute *turn*$_i$ corresponding to P1 moves in the winning strategy in response to legal P2 moves represented by nondeterministic effects. In the case where a nondeterministic effect corresponding to an illegal move is set, execute the *illegal*$_i$ task which sets C and lets us enter Phase 3. Hence, a strong OD solution exists. Conversely, if a strong OD solution exists, we can similarly construct a winning strategy for P1 and ignore the branches where illegal P2 moves occur. By our construction, there cannot exist a policy which is executable but does not correspond to a win by definition that all tasks must be executed. □

We now study the remaining problem classes: acyclic, regular and tail-recursive. For acyclic problems where there is no recursion in task decomposition, we can adapt the membership proof of the deterministic version of the problem in Cor. 3.2 by Alford, Bercher, and Aha (2015a) which calculates an upper bound for the size of a task network under decomposition.

Theorem 5.2. *Let $\mathcal{P}$ be an acyclic FOND HTN problem. Deciding whether $\mathcal{P}$ has a strong OD solution is in* EXPSPACE.

Proof. We provide a nondeterministic decision procedure which consists of two steps as follows:

1. Nondeterministically guess and apply up to k^h decompositions in order where k is the size of the largest task network in the model and $h = |\mathcal{M}|$.

2. Verify whether the resulting primitive task network tn has a policy which is a strong OD solution (Thm. 5.1).

The bound for the number of decompositions results from performing the operation of decomposing all compound tasks in the current task network at most h times. This comes from the acyclic assumption where each method replaces the decomposed task with tasks of strictly lower stratum and the branching factor k comes from each application of this operation. Thus after h decompositions the expanded initial task network must be primitive. Step 1 takes exponential time and step 2 polynomial space with respect to k^h such that the decision procedure requires exponential space and the problem is in NEXPSPACE = EXPSPACE (Savitch 1970). $\square$

Trying to directly adapt the algorithm for regular FOD problems (Erol, Hendler, and Nau 1996) to FOND problems fails. This is because the progression search algorithm used relies on advancing an initial state to determine a solution by components corresponding to task decomposition, whereas in nondeterminism we may end up in several advanced states in a search node. To remedy this problem, we use extra space to store all possible advanced states after progression.

Theorem 5.3. *Let $\mathcal{P}$ be a regular FOND HTN problem. Deciding whether $\mathcal{P}$ has a strong OD solution is in* EXPSPACE.

Proof. Define $Prim(tn)$ to return the task network tn restricted to all its primitive tasks in the obvious way. Also define $Prog(tn_P, \pi, s)$ to return the set of states corresponding to the leaf nodes of the execution structure of π for tn_P at s by a depth-first search. Then the algorithm is as follows.

1. Let $tn = tn_I$, $S_A = \{s_I\}$ and $S_B = \emptyset$.
2. Let $tn_P = Prim(tn)$ and initialise $d = \top$. For all $s \in A$:

   Nondeterministically guess a policy π for tn_P.

   Verify that π is a solution for tn_P (Prop. 2.16) and if so, let $S_B = S_B \cup Prog(tn_P, \pi, s)$. Else, set $d = \bot$.
3. If $d = \bot$ this nondeterministic branch is not a witness.
4. If tn has no compound task, terminate with success. Otherwise let $S_A = S_B$, $S_B = \emptyset$ and $tn = tn_c$, with tn_c the task network tn restricted to the only compound task c.
5. Guess a method and apply it to tn. Then go to step 2.

The task network tn is bounded by the size of the largest network in the model and guessing a policy takes exponential space relative to tn. Similarly $Prog$ takes exponential time by a polynomial time depth-first search in the exponential size policy. Moreover, the sets S_A and S_B are each bounded by the number of states which is exponential in the number of facts such that the problem is in EXPSPACE.

For correctness, observe that regularity ensures that after applying some number of methods to achieve a primitive task network, we can create policies for disjoint sets of primitive tasks corresponding to each compound regular method component as illustrated in Fig. 5. This arises from an equivalence relation on such sets induced by compound tasks being last. Hence a proof of a witness can be done in finite space by verifying that each policy component contributes to a solution. S_A stores 'initial states' for policies and S_B all possible progressed states of a policy component
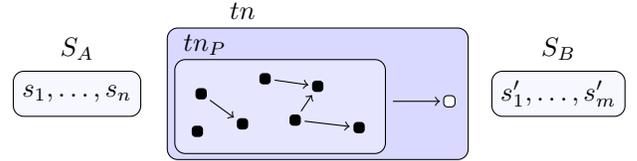

Figure 5: Visualisation of the data structures used for the decision procedure of plan existence for regular strong OD problems. Black nodes represent primitive tasks, the white node a last compound task and arrows partial order.

at an initial state. Step 4 determines for each loop of the algorithm whether there exists a policy component that is a solution for $Prim(tn)$ for all possible initial states such that it contributes to a solution for the initial problem. $\square$

However, trying progression search for strong OD tail-recursive problems fails because of the existence of possibly more than one compound task in a network. Then primitive tasks cannot be progressed and forgotten about without requiring unbounded space to deal with nondeterministic effects so it is not obvious which complexity class such a problem lies in or if the problem is even decidable at all.

6 Conclusion

We have proposed a novel formalism for extending standard HTN planning over nondeterministic domains with solution criteria spanning two dimensions. The first dimension describes the structure of a solution: it is either a conformant plan (linearisation-dependent) or policy (outcome-dependent). The second dimension describes how likely a solution executes successfully: weak and strong. The solution criteria are both canonical in the context of standard HTN planning and non-hierarchical planning under uncertainty. Furthermore, we have found methods for compiling problems with certain solution criteria into their deterministic counterparts and other relations between problems to get many tight complexity bounds with no increase in hardness.

However, hardness does increase for the strongest and most flexible solution criteria involving policies as can be seen in the base case of FOND HTN planning where hierarchies are eliminated. We have shown that nondeterministic, partially ordered primitive problems are PSPACE-complete, exactly one step harder than the NP-completeness of the deterministic problem. We also have loose upper bounds for the remaining problem subclasses whose tight bounds are left for future study alongside complexity analysis for non-primitive problems with dynamic method decomposition and TIHTN planning over nondeterministic domains.

References

Albore, A.; Palacios, H.; and Geffner, H. 2010. Compiling Uncertainty Away in Non-Deterministic Conformant Planning. In *ECAI*, 465–470. IOS Press.

Alford, R.; Bercher, P.; and Aha, D. 2015a. Tight Bounds for HTN Planning. In *ICAPS*, 7–15. AAAI Press.

Alford, R.; Bercher, P.; and Aha, D. 2015b. Tight Bounds for HTN planning with Task Insertion. In *IJCAI*, 1502–1508. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2014. On the Feasibility of Planning Graph Style Heuristics for HTN Planning. In *ICAPS*, 2–10. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *SOCS*, 2–9. AAAI Press.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *IJCAI*, 6267–6275. IJCAI.

Bercher, P.; and Höller, D. 2018. Tutorial: Introduction to Hierarchical Task Network HTN Planning. URL http://tutorial2018.hierarchical-task.net.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence* 69(1-2): 165–204.

Chandra, A. K.; and Stockmeyer, L. J. 1976. Alternation. In *17th Annual Symposium on Foundations of Computer Science (FOCS 1976)*, 98–108. IEEE.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1-2): 35–84.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, 151–158. ACM.

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

Geffner, H.; and Bonet, B. 2013. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 8(1): 1–141.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *IJCAI*, 1955–1961. AAAI Press.

Goldman, R. P.; and Boddy, M. S. 1996. Expressive Planning and Explicit Knowledge. In *AIPS*, 110–117. AAAI Press.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. Stable Plan Repair for State-Space HTN Planning. *Proceedings of the ICAPS-20 Workshop on Hierarchical Planning (HPlan 2020)* 27–35.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN Plan Repair via Model Transformation. In *KI*, 88–101. Springer.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning Hierarchical Task Networks for Nondeterministic Planning Domains. In *IJCAI*, 1708–1714. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and Goal Recognition as HTN Planning. In *ICTAI*, 466–473. IEEE.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *AAAI*, 9883–9891. AAAI Press.

Hsieh, M. Y.; and Tsai, S.-C. 2007. On the fairness and complexity of generalized k-in-a-row games. *Theoretical Computer Science* 385(1-3): 88–100.

Kozen, D. 1976. On Parallelism in Turing Machines. In *17th Annual Symposium on Foundations of Computer Science (FOCS 1976)*, 89–97. IEEE.

Kuter, U.; Nau, D.; Reisner, E.; and Goldman, R. 2007. Conditionalization: Adapting forward-chaining planners to partially observable environments. In *Proceedings of the ICAPS-07 Workshop on Planning and Execution for Real-World Systems*.

Kuter, U.; and Nau, D. S. 2004. Forward-Chaining Planning in Nondeterministic Domains. In *AAAI*, 513–518. AAAI Press.

Kuter, U.; Nau, D. S.; Pistore, M.; and Traverso, P. 2005. A Hierarchical Task-Network Planner based on Symbolic Model Checking. In *ICAPS*, 300–309. AAAI Press.

Kuter, U.; Nau, D. S.; Pistore, M.; and Traverso, P. 2009. Task decomposition on abstract states, for planning under nondeterminism. *Artificial Intelligence* 173(5-6): 669–695.

Meneguzzi, F.; and de Silva, L. 2015. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review* 30(1): 1–44.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *JAIR* 20: 379–404.

Palacios, H.; and Geffner, H. 2009. Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width. *JAIR* 35(1): 623–675.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. S. 2020. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. In *ICAPS*, 478–487. AAAI Press.

Reisch, S. 1980. Gobang ist PSPACE-vollständig. *Acta Informatica* 13: 59–66.

Richter, F.; and Biundo, S. 2017. Addressing Uncertainty in Hierarchical User-Centered Planning. In *Companion Technology*, Cognitive Technologies, 101–121. Springer.

Rintanen, J. 2004. Complexity of Planning with Partial Observability. In *ICAPS*, 345–354. AAAI Press.

Sardiña, S.; de Silva, L.; and Padgham, L. 2006. Hierarchical planning in BDI agent programming languages: a formal approach. In *AAMAS*, 1001–1008. ACM.

Savitch, W. J. 1970. Relationships Between Nondeterministic and Deterministic Tape Complexities. *Journal of Computer and System Sciences* 4(2): 177–192.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning hierarchical task network domains from partially observed plan traces. *Artificial Intelligence* 212(1): 134–157.