Hierarchical Task Network Planning with Task Insertion and State Constraints

Zhanhao Xiao1,2, Andreas Herzig1,3, Laurent Perrussel1, Hai Wan4,5, and Xiaoheng Su4
1IRIT, University of Toulouse, Toulouse, France
2AIRG, Western Sydney University, Penrith, Australia
3IRIT, CNRS, Toulouse, France
4School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
5Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, China
zhanhao.xiao@ut-capitole.fr

Abstract

We extend hierarchical task network planning with task insertion (TIHTN) by introducing state constraints, called TIHTNS. We show that just as for TIHTN planning, all solutions of the TIHTNS planning problem can be obtained by acyclic decomposition and task insertion, entailing that its plan-existence problem is decidable without any restriction on decomposition methods. We also prove that the extension by state constraints does not increase the complexity of the plan-existence problem, which stays $\text{2-NEXPTIME}$-complete, based on an acyclic progression operator. In addition, we show that TIHTNS planning covers not only the original TIHTN planning but also hierarchy-relaxed hierarchical goal network planning.

1 Introduction

Hierarchical task network (HTN) planning [Erol et al., 1994] is an approach for building plans via step-wise refinement of high-level tasks into lower-level tasks in a top-down manner. A task network may contain both compound (high-level) and primitive tasks. Primitive tasks correspond to STRIPS-like actions that can be applied in states where their preconditions are met, while compound tasks are abstractions: for every compound task, the domain features a set of decomposition methods, each mapping the task to a task network. The complexity of the plan-existence problem for HTN planning ranges up to undecidability even for propositional HTN planning [Erol et al., 1994]. Even so, hierarchical planning approaches are often chosen for real world application scenarios [Lin et al., 2008; Biundo et al., 2011]. On the other hand, it is usually a challenge to provide a complete domain which includes all possible methods for all compound tasks, while defining only a partially hierarchical domain is not sufficient to produce all desired solutions. Several HTN researchers have investigated how partially hierarchical domain knowledge can be exploited during planning without relying on the standard HTN formalism [Kambhampati et al., 1998; Biundo and Schattenberg, 2001; Geier and Bercher, 2011; Shivashankar et al., 2013]. Among them, hierarchical task network planning with task insertion (TIHTN) planning [Geier and Bercher, 2011] relaxes the restriction on solutions and allows solutions generated not only by the decomposition of compound tasks, but also by the insertion of primitive tasks from outside the decomposition hierarchy.

Unfortunately, only ordering constraints are considered in propositional TIHTN planning [Geier and Bercher, 2011] and lifted TIHTN planning [Alford et al., 2015b]. In contrast, state constraints are taken into account in the conventional HTN planning [Erol et al., 1994], but there is few work on considering them in TIHTN planning.

In this paper, we investigate the extension of TIHTN with state constraints, noted TIHTNS for short. We first prove that, just as for the TIHTN planning problem, all solutions of the TIHTNS problems can be obtained by acyclic decomposition and task insertion, entailing that it is decidable without any restrictions on decomposition methods. We then show that the extension by state constraints does not cause an increase in complexity of the plan-existence problem, which stays $\text{2-NEXPTIME}$-complete, based on an acyclic progression operator. We also show that TIHTNS planning includes lifted TIHTN planning. As under task insertion semantics, hierarchical goal network (HGN) planning [Shivashankar et al., 2013] and goal-task network (GTN) planning [Alford et al., 2016] can be translated to lifted TIHTN, our framework, TIHTNS, actually covers the two kinds of planning approaches. In addition, we give an alternative embedding of hierarchy-relaxed HGN (HR-HGN) [Shivashankar et al., 2017] in TIHTNS without introducing fresh operators.

State constraints State constraints can capture the pre- and postcondition of compound tasks, though in the standard HTN planning there is no notion of pre- and postcondition of compound tasks. A compound task is considered as accomplished if its subtasks are accomplished. With a state constraint, a formula, as a postcondition, can be required to hold after accomplishing a compound task.

Ordering constraints cannot fully represent state constraints. The ‘immediate’ state constraint, which requires a formula holds immediately before or after a compound task, can be simulated via introducing a virtual subtask to check whether the formula holds. However, the ‘maintenance’ state constraints (also called trajectory constraints in the Planning Domain Definition Language 3 (PDDL3) [Gerevini and Long, 2005]) cannot be represented easily. For instance, suppose a robot is required to always keep 10% battery for emergency. Its initial compound task is to “clean a room”, de-
composed into “clean the ground” (t₁) and “clean the table” (t₂). Suppose that “clean the ground” requires the full battery. Then there is no solution for the TIHTNS planning problem with such a state constraint.¹

Furthermore, state constraints are introduced into PDDL3 [Gerevini and Long, 2005] to support hard constraints over state properties of a trajectory and the specification of preferences. Later [Sohrabi et al., 2009] extends PDDL3 into HTN planning where state constraints are used to capture user preferences. State constraints are also necessary in real-world applications, such as in web service composition where state constraints are used to describe user preferences [Lin et al., 2008] and to additionally capture the enforcement of regulations [Sohrabi and McIlraith, 2009].

The rest of the paper is structured as follows. Section 2 presents the definition of TIHTNS. Section 3 presents how to embed lifted TIHTN and HR-HGN into TIHTNS. Section 4 presents the complexity results of TIHTNS. Section 5 concludes and discusses the future work.

2 Extended TIHTN Planning

In this section we adapt the original TIHTN planning formalism of [Geier and Bercher, 2011; Alford et al., 2015b]. First, we define a function-free first order language L from a set of variables and a finite set L₀ of predicates and constants. Next we take parts of variables in L as task symbols to identify tasks. Every task is associated with an action (task name), which is syntactically a first-order atom in L. That is, every action typically is associated with an arity and contains variables that can be eliminated via grounding. For example, by grounding the action “openDoor(X)” where X is quantified as a door, we can obtain a set of ground actions “openDoor(a)”, “openDoor(b)”, etc. Those actions which can be executed directly are called operators, noted O, while others are called compound actions.² Noted C. Each operator o ∈ O is represented as a triple of formulas: precondition pre(o), positive effect add(o), and negative effect del(o) where add(o), del(o) are finite sets of atomic formulas. The variables in add(o) and del(o) should be associated with the variables in the operator so that they are instantiated simultaneously. For example, the positive effect of “openDoor(X)” being “Open(X)”, after grounding “openDoor(a)” has positive effect “Open(a)” but not “Open(b)”. The positive and negative effect of an operator should not conflict, i.e., add(o) ∩ del(o) = Ø for every o ∈ O. We suppose that O contains the ‘empty’ primitive operator skip where pre(skip) = T and add(skip) = del(skip) = Ø.

For a function f : R → S, its restriction to a subset X of its domain is f|X = {(r, s) ∈ f | r ∈ X}. Given a binary relation Q ⊆ R × R, we define its restriction to X ⊆ R by Q|X = Q ∩ (X × X); similarly for ternary relations. We extend the set union operator ∪ to tuples: (Q₁, Q₂) ∪ (Q₁', Q₂') = (Q₁ ∪ Q₁', Q₂ ∪ Q₂') and extend functions to sequences: f(t₁,…,tₙ) = (f(t₁),…,f(tₙ)).

2.1 HTN Problems

Task networks A task network tn = (T, Δ, α) is a tuple, where

- T is a finite and non-empty set of task symbols;
- Δ ⊆ (T∪{nil}) × L × (T∪{nil}) is a set of constraints over T
- α : T → C ∪ O labels every task with an action.

With Function α, we allow multiple instances of an action in a task network. Compared to the ordering constraints in [Geier and Bercher, 2011] which are in form of task-task pairs, we use a triple (tᵢ, φ, tⱼ) to denote a state constraint which intuitively means that formula φ must be true in all states between tᵢ and tⱼ. Specially, we introduce an idle task symbol nil which designates a task that is accomplished immediately: (nil, φ, tⱼ) and (tᵢ, φ, nil) mean formula φ holds immediately before tᵢ and after tᵢ, respectively. We suppose nil only occurs in constraints. When φ is the truth constant T then the state constraint (tᵢ, φ, tⱼ) becomes an ordering constraint that just requires that tᵢ is before tⱼ. A task is primitive if it is associated to an operator, otherwise compound. A task network is primitive if it only contains primitive task.

We say that two task networks tn = (T, Δ, α) and tpn = (T’, Δ’, α’) are isomorphic, noted tn ≃ tpn, if there exists a bijection δ : T → T’ where for all t, t’ ∈ T it holds that α(t) = α’(δ(t)) and (t, φ, t’) ∈ Δ iff (δ(t), φ, δ(t’)) ∈ Δ’. Methods Non-primitive task networks contain compound tasks which cannot be executed directly by the agent, and decomposition methods tell us how to decompose these hierarchically. Each decomposition method m is a tuple (c, tnₘ), where c is a compound action, called the method’s head, and tnₘ is a task network, whose inner tasks are called the method’s subtasks. The intuition is that compound action c can be reduced by the subtask network tnₘ.

HTN problems TIHTNS problems only differ HTN problems in the solution criterion and share the syntactical problem description. An HTN domain is a tuple D = (L, C, O, M) where M is a set of methods and C ∩ O = Ø. An HTN problem is a tuple P = (D, s₁, tnᵢ₁) where s₁ is the initial ground state and tnᵢ₁ is the initial task network.

The semantics of HTN planning is given through grounding. According to [Alford et al., 2015], it is easy to translate a lifted HTN problem into a ground (or propositional) HTN problem as the set of relations and constants L₀ is finite. For an HTN problem (L, C, O, M, s₁, tnᵢ₁), we use P = (L, C, O, M, sᵢ, tnᵢ) to denote the ground (or propositional) problem obtain from it.

A ground state is a subset of the ground atoms in L₀. A set of ground operators O determines a state-transition function γ : 2L₀ × O → 2L₀, where:

- γ(s, o) is defined iff s |= pre(o);
- γ(s, o) = (s \ del(o)) ∪ add(o) if γ(s, o) is defined.

¹For the original TIHTN planning, there is an intuitive attempt to simulate that constraint: introducing a virtual task primitive task pt whose precondition is “more than 10% battery” and introducing two ordering constraints t₁ ∨ pt and pt ∨ t₂. Assume “charge” means to charge the battery, then the plan (t₁; “charge”; pt; t₂) is a solution of the original TIHTN planning problem. But it is counter-intuitive and the state constraint is still violated.

²In the original HTN, “non-primitive or compound task name” is used to distinguish from STRIPS-like actions, i.e., operators.
A sequence of operators \(\langle o_1, \ldots, o_n \rangle\) is executable in a state \(s_0\) iff there exists a sequence of states \(s_1, \ldots, s_n\) such that \\
\(\forall 1 \leq i \leq n; \gamma(s_{i-1}, o_i) = s_i\).

**Example 1.** Suppose we have an initial compound action \(goMC\) for “go to Melbourne center”, operator \(o_1\) for “fly to Melbourne”, and operator \(o_2\) for “take a taxi to the center”. We use \(At(Mc)\) and \(At(Ma)\) to denote “being at Melbourne center” and “being at Melbourne airport”. The primitive operators are:

- \(o_1 = \langle T, \{At(Ma)\}, \emptyset \rangle\)
- \(o_2 = \langle At(Ma), \{At(Mc)\}, \{At(Ma)\} \rangle\)

and the method is:

- \(m = \langle goMC, tn_m \rangle\), where \(tn_m = (t_2, \emptyset, (t_2, o_1))\)

Then the HTN problem \(P\) is \((L, goMC, \{o_1, o_2\}, m, s_I, tn_I)\) with \(s_I = \emptyset\) and \(tn_I = (t_1, (t_1, At(Mc), nil), (t_1, goMC))\).  

### 2.2 Task Decomposition

Next we borrow the notion of decomposition in [Geier and Bercher, 2011] and define how to decompose a compound task into a task network. In order to indicate the starting point and the end point of a compound task \(t\), we introduce a pair of ‘dummy’ primitive tasks, noted \(\ast t\) and \(t\ast\). As nil only occurs in constraints, we suppose the restriction of constraint set \(\Delta\) to a set of tasks \(T\) is \(\Delta|_T = \Delta \cap ((\cup \{nil\}) \times (\cup \{nil\}))\).

**Definition 1 (Decomposition).** Given an HTN domain \(\mathcal{D} = (L, C, O, M, \alpha)\), let \(tn = \langle T, \Delta, \alpha \rangle\) be a task network and \(t \in T\) be a compound task. Let \(m = \langle \alpha(t), (T_m, \Delta_m, \alpha_m) \rangle\) and \(T_m \cap T = \emptyset\). The decomposition of task \(t\) by method \(m\) is \(tn' = \langle T', \Delta', \alpha' \rangle\) where:

- \(T' = (T \setminus \{t\}) \cup T_m \cup \{\ast t, t\ast\}\)
- \(\Delta' = \Delta|_{T \setminus \{t\}} \cup \Delta_m \cup \{(\ast t, T, t_j), (t_j, T, t\ast) \mid t_j \in T_m\}\)
- \(\{t_1, \varphi, t_1\} \mid (t_1, \varphi, t) \in \Delta\)
- \(\{t_2, \varphi, t_2\} \mid (t, \varphi, t_2) \in \Delta\)
- \(\alpha' = \alpha|_{T \setminus \{t\}} \cup \alpha_m \cup \{(\ast t, \text{skip}), (t\ast, \text{skip})\}\)

We write \(tn \longrightarrow_{tn'} tn'\) when \(tn'\) is the decomposition of task \(t\) by \(m\).

In the resulting task network, the decomposed compound task is replaced with subtasks defined by the method applied and its corresponding starting and terminating tasks. The latter two are dummy tasks and are mapped to the action skip. All state constraints about the decomposed task \(t\) are propagated by \(\ast t\) and \(t\ast\) in \(\Delta'\). More precisely, if \(\varphi\) holds before \(t\) then it also holds before \(\ast t\) and if \(\varphi^\ast\) holds after \(t\) then it also holds after \(t\ast\). Subtasks should satisfy the inner constraints introduced by the decomposition method and should be performed between \(\ast t\) and \(t\ast\).

**Example 2** (Example 1 continued). We apply the method \(m\) in \(tn_t\) to decompose \(t_1\), i.e., \(tn_t \longrightarrow_{t_1, m} tn'\), where \(tn'\) is:

- \(T' = \{t_2, \ast t_1, t_1\ast\}\)
- \(\Delta' = \{(t_2, T, t_1\ast), (\ast t_1, T, t_2), (t_1\ast, At(Mc), nil)\}\)
- \(\alpha' = \{(t_2, o_1), (\ast t_1, \text{skip}), (t_1\ast, \text{skip})\}\)

\(\text{We will sometimes omit the braces of singleton sets.}\)

### Decomposition tree

The hierarchical procedure of task decomposition can be viewed as a tree. Given an HTN domain \(\mathcal{D} = (L, C, O, M)\), a decomposition tree is a five-tuple \(Tr = (T, E, L, \alpha, \beta)\) where \((T, E)\) is a tree, rooted in \(t_0\) which is a fresh task symbol, with nodes \(T\) and with directed edges \(E\) pointing towards the leaves; \(L\) is a set of constraints over \(T\); function \(\alpha : T \rightarrow C \cup O\) links tasks and actions where \(\alpha(t_0) = c_0\) such that \(c_0\) is a fresh compound action with an intuition “to achieve the initial task network”. \(\beta : T \rightarrow M'\) labels inner nodes with methods where \(M' = M \cup \{(c_0, tn_0)\}\) and \(tn_0\) is some task network.

We write \(\tau(Tr)\) for the nodes (tasks) of the decomposition tree \(Tr\) and \(ch(Tr, t)\) for the direct children of \(\tau(Tr)\) in \(Tr\). Hereafter \(sub(t)\) denotes the set of subtasks of \(t\). The leaf nodes of \(Tr\) together with the constraints about these nodes define a task network, denoted by \(\vartheta(Tr)\).

**Definition 2** (valid decomposition tree). A decomposition tree \(Tr\) is valid w.r.t. an HTN problem \(P\) if the root node of \(Tr\) is \(t_0\) where \(\beta(t_0) = (c_0, tn_0)\) and for any inner node \(t\) where \(\beta(t) = (c, tn_m)\), the followings hold:

1. \(\alpha(t) = c\)
2. if \(t = t_0\) then \(ch(Tr, t) = sub(t)\), otherwise \(ch(Tr, t) = sub(t) \cup \{\ast t, t\ast\}\) such that:
   - \((sub(t), \Delta_{\ast t}, \alpha_{\ast t}) \simeq tn_m\)
   - for every \(t' \in sub(t)\), \(t' \in T, \ast t, T, t\ast \in \Delta\)
3. for every \(t' \in \tau(Tr) \cup \{nil\}\):
   - if \((t, \varphi, t') \in \Delta\) then \((\ast t, \varphi, t') \in \Delta\)
   - if \((t', \varphi, t) \in \Delta\) then \((t', \varphi, \ast t) \in \Delta\)
4. there are no constraints in \(\Delta\) except for those demanded by criterion 2. and 3.

When \(tn'\) is reachable from \(tn\) by a finite sequence of decompositions then we write \(tn \longrightarrow_{tn'} tn'\).

The decomposition tree focuses on the whole procedure of decomposition and records all intermediate tasks and constraints during the procedure. In contrast, in a task network the application of a decomposition method abandons the decomposed task. However, they are compatible as the following proposition states.

**Proposition 1.** Given an HTN problem \(P\), \(tn_1 \longrightarrow_{tn_1} tn_2\) if there exists a valid decomposition tree \(Tr\) with respect to \(P\) where \(\vartheta(Tr) = tn_2\).

### 2.3 Solutions

A solution of an HTN problem is a sequence of primitive tasks which is also called a plan of the problem.

**Consistency with constraints**

Given a primitive task network \(tn = \langle T, \Delta, \alpha \rangle\) where \(|T| = n\), let \(\sigma : T \rightarrow \{1, \ldots, n\}\) be a bijection. We use \(\sigma\) to form a total ordering, noted \(\sigma(tn)\), of tasks in \(T\) as: \((\sigma^{-1}(1), \ldots, \sigma^{-1}(n))\) where \(\sigma^{-1}\) is the inverse function of \(\sigma\), i.e., \(\sigma^{-1}(\sigma(t)) = t\). Suppose \(\sigma(\sigma(tn))\) is executable in \(s_0\), i.e., there exists a sequence \(s_1, \ldots, s_n\) such that \(\gamma(s_{i-1}, \sigma(t_i)) = s_i\) for every \(i\) such that \(1 \leq i \leq n\). We say that \(\sigma(tn)\) is consistent with \(\Delta\) in \(s_0\) if for every \(\sigma^{-1}(i), \sigma^{-1}(j) \in T\) the following hold:

- for every \((nil, \varphi, \sigma^{-1}(j)) \in \Delta, s_{j-1} = \varphi;\)
for every \((\sigma^{-1}(i), \varphi, \text{nil}) \in \Delta, s_i \models \varphi\);
for every \((\sigma^{-1}(i), \varphi, \sigma^{-1}(j)) \in \Delta, i < j \) and \(s_k \models \varphi\) for every \(i \leq k < j\).

Intuitively, a ‘maintenance’ state constraint \((t, \varphi, t')\) is satisfied if all states between \(t\) and \(t'\) satisfy \(\varphi\). An ‘immediate’ state constraint \((\text{nil}, \varphi, t)\) is satisfied if \(\varphi\) holds in the state right before \(t\) occurs (or \(t\) if \(t\) is a primitive task), in other words right before all subtasks of \(t\). Finally, an ‘immediate’ state constraint \((t, \varphi, \text{nil})\) is satisfied if \(\varphi\) holds right after the state where \(t\) (or \(t\) if \(t\) primitive task) occurs, in other words right after all subtasks of \(t\). Note that it is impossible to satisfy \((t, \bot, t')\) because there is no state \(s\) such that \(s \models \bot\).

**Executable** A task network \(tn\) is primitive iff it contains only primitive tasks. A primitive task network \(tn\) is executable in a state \(s\) iff there exists a total ordering \(\sigma(\text{tn})\) of the tasks in \(tn\) that is consistent with \(\Delta\) in \(s\). We called such a \(\sigma(\text{tn})\) a plan in \(s\), noted \(\sigma_{\text{tn}, s}\).

It is possible that the task network is not executable in a state. For instance, the primitive task network \(tn'\) in Example 2 is not executable in \(s_2\) because apart from skip it only involves the operator \(o_1\) and it is impossible to satisfy \(At(\text{Mc})\). However, if we extend the task network by inserting some tasks, then we can make it executable.

**Insertion** Let \(tn = (T, \Delta, \alpha)\) and \(tn' = (T', \Delta', \alpha')\) be two task networks where \(tn'\) is primitive. Inserting \(tn'\) into \(tn\) results in the task network \(tn \cup tn'\). Note that it is not required that \(T' \cap T = \emptyset\) because the insertion can involve some constraints about tasks in \(tn\).

**Solutions** With respect to some initial \(tn\), if \(tn'\) is reachable by a finite sequence of decompositions and an insertion we write \(tn \rightarrow_D tn'\). The plan obtained only by decomposition is called an HTN solution while the plan obtained additionally by insertion is called a TIHTNS solution.

Let \(tn\) be a primitive task network such that there exists a plan of \(tn\) in \(s_1\). Given an HTN problem \(P\), we call \(\sigma_{\text{tn}, s_1}\) an HTN solution of \(P\) if \(tn \rightarrow_D tn'\); we call \(\sigma_{\text{tn}, s_1}\) a TIHTNS solution of \(P\) if \(tn \rightarrow_D tn'\).

**Example 2 continued.** The plan \((s_1, t_2, t_3, t_1^*)\) where \(\alpha(t_3) = \alpha_{25}\) is a TIHTNS solution of \(P\).

When considering whether an HTN problem \(P\) has a TIHTNS solution, we call \(P\) a TIHTNS problem.

The next proposition states that the state constraints about compound tasks are satisfied in the solutions of the problem.

**Proposition 2.** Given a TIHTNS problem \(P\), suppose \(Tr\) is a valid decomposition tree with respect to \(P\) and \(tn'\) is the final primitive task network obtained from \(\vartheta(Tr)\) by insertion and \(\sigma_{tn', s_1}\) is a solution of \(P\). Then all constraints in \(Tr\) are satisfied by \(\sigma_{tn', s_1}\).

**Sketch of proof.** Suppose there are two compound tasks \(t_i\) and \(t_j\) in \(Tr\) and \(tn' = (T', \Delta', \alpha')\). For every \((t_i, \varphi, t_j) \in \Delta\), we have \((t_i^*, \varphi, t_j^*) \in \Delta'\) and then all states \(s_k\) such that \(\sigma(t_i^*) \leq k < \sigma(t_j^*)\), \(s_k \models \varphi\). For every \((t_i, \varphi, \text{nil}) \in \Delta\), we have \((t_i^*, \varphi, \text{nil}) \in \Delta'\) for all subtasks \(st \in \text{sub}(t_i)\) and then \(s_{\sigma(t_i^*)} \models \varphi\) and \(\sigma(st) < \sigma(t_i^*)\) for all \(st \in \text{sub}(t_i)\). The case of \((\text{nil}, \varphi, t_j)\) is similar. If one or both of \(t_i, t_j\) are primitive then just consider \(s_{t_i^*}\) and \(t_j^*\) as \(t_i\) and \(t_j\), respectively. Then all constraints in \(Tr\) are satisfied.

**Remark 1.** The state constraint here we define is weaker than that in the original HTN planning [Erol et al., 1995]. Their semantics considers every compound task starts by its first ‘real’ subtask and terminates by its last ‘real’ subtask instead of the virtual starting and terminating tasks. The distinction between two semantics stands on whether it is allowed to insert tasks between \(st\) and the first real subtask of \(t\) (between the last real subtask and \(t^*\)). Our weaker semantics can capture the pre- and postcondition of compound actions better. In Example 1, after the subtask \(t_2\) of goMC is accomplished, the agent’s desirable goal “being in the center” does not hold while by allowing the insertion of \(t_3\), the goal is achieved.

### 3 Embedding TIHTN and HGN

Recently HTN researchers work on enhancing the semantics of HTN planning and have proposed variants of HTN planning. Propositional TIHTN was first proposed in [Geier and Bercher, 2011] and later was extended into lifted TIHTN in [Alford et al., 2015b].

**Sketch of proof.** Under the lifted TIHTN semantics, consider the decomposition of \(t\) by \(m\), the new ordering constraint set is obtained as: (1) keep the constraints not involving \(t\); (2) introduce the constraints about the subtasks; (3) if \(t\) is before \(t'\) then all subtasks are before \(t'\); (4) if \(t\) is after \(t'\) then all subtasks are after \(t'\). Under the extended semantics, according to Definition 1, the change of state constraints includes (1) and (2) and is analogous to (3) and (4). If \(t\) is before \(t'\), i.e., \((t, t', t')\), there will be \((t^*, t, t')\) introduced. As \((t_m, t, t^*)\) for all subtasks \(t_m\) are introduced, it entails that \((t_m, t, t')\) which means \(t_m\) is before \(t'\). Then (3) is simulated and the case of (4) is similar. The proposition follows.

Alford et al. [2016] combine HTN and HGN planning into goal-task network (GTN) planning where an element of the network consists of a goal and a task. They also show that allowing task insertion, HGN and GTN planning problems can be translated polynomially into lifted TIHTN problems. Therefore, our extension of TIHTNS actually also covers them.

**Embedding HR-HGN** Hierarchical goal network planning is a formalism which extends classical planning to include hierarchical decomposition using methods.

HGN planning talks about goal network \(gn = (G, \prec)\) where \(G\) is a set of formulas in disjunctive normal form over ground literals, called goal formulas, and \(\prec \subseteq G \times G\) is a
strict partial order on $G$. Similar with HTN problems, an HGN problem is a tuple $P = (L, O, M, s_1, gn_1)$ where $M$ is a set of HGN methods and $gn_1$ is an initial goal network. For an HR-HGN problem, $M$ is an empty set and omitted. The solutions of HR-HGN are defined as the set of all operators sequences that are executable in the initial state $s_1$ and that achieve all initial goals according to the order. In [Shivashankar et al., 2017], by introducing fresh operators with the number of $|gn_1|$, an HR-HGN problem can be translated into a classical planning problem. Now, we translate HR-HGN planning into our framework without introducing any fresh operators. Allowing task insertion, state constraints simulate a goal formula by requiring that the formula holds immediately before an empty task. Formally, given an HR-HGN problem $P = (L, O, s_1, gn_1)$, we define a TIHTNS problem $\Gamma_G(P) = (L, \emptyset, O, \emptyset, s_1, tn)$ as:

- for every $g \in G_1$, $\lambda_g \in tn$ and $(nil, g, \lambda_g) \in \Delta$ and $\alpha(\lambda_g) = \text{skip}$
- for every $g_i < g_j$, $(\lambda_g_i, \top, \lambda_g_j)$

Next we show that the translation $\Gamma_G$ is correct.

**Proposition 4.** For an HGN problem $P$, $P$ has a solution iff $\Gamma_G(P)$ has a solution.

**Sketch of proof.** Suppose $\sigma_{tn_1,s_1}$ is a solution of $\Gamma_G(P)$. Then $tn_1$ is obtained from $tn_1$ by an insertion. It forms a sequence $s_0, \ldots, s_n$ where $s_0 = s_1$. For every $g \in G_1$, we have $\lambda_g \in T'$ and $(nil, g, \lambda_g) \in \Delta'$ then $s_i = g$ where $\sigma(\lambda_g_i) = i$; for every $g_i < g_j$, we have $(\lambda_g_i, \top, \lambda_g_j) \Delta'$ then $i' < j'$ and $s_{i'} = gi, s_{j'} = gj$, where $\sigma(\lambda_g_i) = \sigma(\lambda_g_j)$.

**4 Complexity**

In this section, we show that the solutions of TIHTNS can be obtained by acyclic decomposition and it does not increase the complexity, staying $2\text{-NEXPTIME}$-complete.

**Substitution** Now we adapt the notion of subtree substitution initially proposed in [Geier and Bercher, 2011] which replaces a subtree with another subtree. Given a decomposition tree $(T, E, \Delta, \alpha, \beta)$ and a node $t \in T$, we define the subtree of $Tr$ induced by $t$, as $Tr[t] = (T', E', \Delta', \alpha_{T'}, \beta_{T'})$ where $(T', E')$ is the subtree in $(T, E)$ which is rooted in $t$.

Let $Tr = (T, E, \Delta, \alpha, \beta)$ be a decomposition tree and $t_i, t_j \in T$ be two nodes of $Tr$ where $t_i$ is an ancestor of $t_j$. We define the result of the subtree substitution on $Tr$ that substitutes $t_i$ by $t_j$, written $Tr[t_i \leftarrow t_j] = (T', E', \Delta', \alpha_{T'}, \beta_{T'})$, where

\[ T' = (T \setminus \tau(Tr[t_i])) \cup \tau(Tr[t_j]) \]
\[ E' = E \setminus \{(p, t_j) \mid (p, t_i) \in E\} \]
\[ \Delta' = (\Delta \setminus \{\text{nil}, \varphi_1, \ast t_j, \ast t_j, \varphi_2, \text{nil} \in \Delta\}) \]
\[ \cup \{\langle t_j, \varphi_1, t_i\rangle, \langle t_j, \ast t_j, \varphi_1, t_i\rangle, \langle t_i, \varphi_1, t_i\rangle \in \Delta\} \]
\[ \cup \{\langle t_j, \varphi_2, t_i\rangle, \langle t_j, \ast t_j, \varphi_2, t_i\rangle, \langle t_i, \varphi_2, t_i\rangle \in \Delta\} \]

Compared to [Geier and Bercher, 2011], the substitution operator here requires to remove $t_j$ corresponding starting and terminating tasks before replacement because they are generated from $t_j$’s parent which will be dropped. The following proposition states that the resulting tree is still valid.

**Proposition 5.** Let $Tr = (T, E, \Delta, \alpha, \beta)$ be a valid decomposition tree with respect to TIHTNS problem $P$ and two nodes $t_i \in T, t_j \in \tau(Tr[t_i])$ with $\alpha(t_i) = \alpha(t_j)$. Then $Tr[t_i \leftarrow t_j]$ is also a valid decomposition tree w.r.t. $P$.

**Acyclic decomposition** The sequence of decompositions is acyclic if for every node $t$ in its corresponding tree $Tr$, the ancestors of $t$ have different actions.

The insertion of tasks allow us to break the loop of generating same compound tasks during the decomposition procedure and find a shortcut to generate the solution. The following theorem states that we only need to consider the acyclic sequences of decompositions to compute the solution.

**Theorem 1.** A TIHTNS problem $P$ has a solution iff $P$ has a solution which is generated by an acyclic sequence of decompositions and an insertion.

**Proof.** The right-to-left direction is straightforward. Now we prove the left-to-right direction.

Suppose $\sigma_{tn_1,s_1}$ is a TIHTNS solution of $P$ where $tn_1 \rightarrow_{D}^* tn_1$ and $tn_1$ is obtained from $tn_1$ by insertion. Then $tn_1$ is a primitive task network. If the sequence of decompositions from $tn_0$ to $tn_1$ is acyclic, then it is proved. Now suppose the sequence is not acyclic. Then in its corresponding tree $Tr$, there exist two ancestors $t_i, t_j$ of some $t$ such that $\alpha(t_i) = \alpha(t_j)$. By Proposition 5, the tree $Tr[t_i \leftarrow t_j]$ is valid with respect to $P$. By Proposition 1, there exists a primitive task network $tn_2$ such that $tn_2 = \emptyset(Tr[t_i \leftarrow t_j])$ and $tn_1 \rightarrow_{D}^* tn_2$. As the substitution does not introduce any new node, the leaf nodes of $Tr[t_i \leftarrow t_j]$ are a subset of the leaf nodes of $Tr$. The only state constraints about leaf nodes which are introduced by the substitution are $(t_j, \ast t_j, \ast t_j)$ and $(\ast t_i, \ast t_i, \ast t_i)$. Now we replace all occurrences of $t_j$ such that $t_j \ast t_j, t_j$ with $t_i \ast t_i$. Then except for those constraints between subtasks of $t_j$ and either $t_j$ or $t_i$, in form of $(\ast t_j, t, \ast t_j)$ and $(t, t, \ast t_j)$, the constraints in $tn_2$ are a subset of the constraints of $tn_1$. Because $t_i$ is an ancestor of $t_j$ in $Tr$, there must be some tasks $t_i, t_i, \ldots, t_{i}^{(k)}$ such that $(\ast t_i, t_{i}^{(k)}), (t_{i}^{(k)}, t, \ast t_{i}^{(k)}), \ldots, (t_{i}^{(k)}, t, \ast t_j)$.

Thus, in the plan $\sigma_{tn_1,s_1}$, $t_i$ is before $t_j$ and then before all subtasks of $t_j$. Then those constraints, $(\ast t_i, t_r, t)$ are satisfied; the case of $t_j$ is similar. So $\sigma_{tn_1}$ includes all tasks in $tn_2$ and is consistent with all constraints in $tn_2$ and can be obtained from $tn_2$ by inserting an appropriate task network.

The next lemma allows us to only check whether there is a solution with an upper bound on length.

**Lemma 1.** If TIHTNS problem $P = (L, C, O, M, s_1, tn_1)$ has a solution then $P$ has a solution with a length of at most $|T_1| \times (k + 2)^{|C|} \times c^m \times 2^{|C|}$, where $k$ is the maximal number of subtasks in one method, $c$ and $p$ are the number of constants and predicates respectively, $n$ and $m$ are the maximal arity of compound actions and predicates respectively.

**Sketch of proof.** A valid decomposition tree generated by acyclic decomposition has at most $|T_1| \times (k + 2)^{|C|}$ leaf nodes where $|C|$ is the number of ground compound actions that is $|C| \times c^m$. As the number of states is bounded by $2^{|C|}$, where $|L_n|$ is the number of ground atoms that is $p \times c^m$, at most $2^{|C|}$ tasks can be inserted between two neighbor tasks.
As the state-transition function $\gamma$ satisfies the frame axiom, before performing an operator we can check whether a formula is true in the next state. Given a formula $\varphi$, we use $\varphi_\gamma$ to denote the formula obtained from $\varphi$ by replacing $p$ with $\top$ and $q$ with $\bot$ where $p \in \text{add}(o)$ and $q \in \text{del}(o)$. As all variables outside the effect of operator $o$ keep their truth value, we can check whether a formula $\varphi$ holds in the next state before operator $o$ is performed, as the following lemma states:

**Lemma 2.** $s \models \varphi_\gamma$ iff $\gamma(s, o) \models \varphi$.

**Acyclic progression** We adapt the acyclic progression operator proposed in [Alford et al., 2015b] to TIHTNS. To capture the ‘maintenance’ state constraints we introduce a set $\Sigma$ of pairs $(\varphi, t)$ of formula and task which means $\varphi$ should be satisfied until performing task $t$. We use $\text{Fml}(\Sigma)$ to denote the conjunction of all formulas in $\Sigma$. To forbid the recursion of tasks, when decomposing a compound task $t$ by method $m$, its subtasks cannot contain $t$’s ancestors denoted by $h(t)$, where $h : T \rightarrow 2^T$. Formally, we use a tuple $(s, \text{tn}, \Sigma, h)$ to represent that task network $tn$ still needs to be accomplished at the current state $s$. Given a tuple $(s, \text{tn}, \Sigma, h)$, we define its acyclic progression is $(s', \text{tn}', \Sigma', h')$ with $t'n' = (T', \Delta', \alpha')$, which is obtained from three possible options:

- **Task insertion:** if $s \models \text{pre}(o) \wedge \text{Fml}(\Sigma)_o$ then $s' = \gamma(s, o)$ and $\text{tn}' = \text{tn}$, $\Sigma' = \Sigma$, $h' = h$

The task inserted should satisfy the state constraints.

- **Task performing:** for a primitive task $t \in T$, if $s \models \text{pre}(\alpha(t)) \wedge \text{Fml}(\Sigma_{\neg t})_{\alpha(t)}$ where $\Sigma_{\neg t} = \Sigma \cap (\mathcal{L} \times (T \setminus \{t\}))$ and the followings hold:
  - there is no $t'$ such that $t' \neq \text{nil}$ and $(t', \varphi, t) \in \Delta$
  - for every $(\text{nil}, \varphi, t) \in \Delta$, $s \models \varphi$
  - for every $(t, \varphi, t') \in \Delta$, $s \models \varphi_{\alpha(t)}$

then $T' = T \setminus \{t\}$, $\Delta' = \Delta|_{T'}$, $\alpha' = \alpha|_{T'}$

$\Sigma' = \Sigma - \{(\varphi, t') | (t, \varphi, t') \in \Delta \text{ and } t' \neq \text{nil}\}$

$h' = h|_{T'}$ and $s' = \gamma(s, \alpha(t))$

A primitive task is chosen only if its all predecessors have been accomplished, its precondition is satisfied and the state constraints in $\Sigma$, except for those constraints ending with the primitive task, will hold after performing it. The ‘maintenance’ state constraints starting from the primitive task will be added into $\Sigma$.

- **Task decomposition:** for a compound task $t \in T$ and a method $(\alpha(t), (T_m, \Delta_m, \alpha_m)) \in M$, if $h(t) \cap T_m = \emptyset$, then $\text{tn} \sim_{t,m} \text{tn}'$ and
  - $h' = h|_{T'} \cup \{(t_m, h(t) \cup \{\alpha(t)\}) | t_m \in T_m\}$
  - $\Sigma' = \Sigma|_{T'} \cup \{(\varphi, st) | (\varphi, t) \in \Sigma\}$

For a compound task, the method chosen to decomposed cannot contain an action which is its ancestor in order to avoid the recursion of tasks.

Note that the cardinality of $\Sigma$ may be exponential on size but $\text{Fml}(\Sigma)$ is polynomial which is restricted by the constraints in $M$. It entails that applying a step of acyclic progression is in $P$.

If all tasks can be eliminated by the acyclic progression, then the TIHTNS problem has a solution:

**Lemma 3.** Given a TIHTNS problem $\mathcal{P}$, $\mathcal{P}$ has a solution iff there is a sequence of acyclic progressions from $(s_1, \text{tn}_1, \emptyset, h_1)$ to $(s, \text{tn}_0, \emptyset, \emptyset)$ where $h_1 = \alpha_1$ and $\text{tn}_0$ is the empty task network.

As TIHTNS can cover lifted TIHTN whose plan-existence problem is 2-NEXPTIME-complete [Alford et al., 2015b], the plan-existence problem for TIHTNS is 2-NEXPTIME-hard. The next theorem states its completeness by proving the upper bound.

**Theorem 2.** Deciding whether a TIHTNS problem has a solution is 2-NEXPTIME-complete.

**Proof.** Theorem 1 reduces deciding the problem into checking all solutions generated by acyclic decomposition and Lemma 1 reduces further deciding the problem into checking solutions with an upper bound on length. By introducing a counter on operators according to [Alford et al., 2015b], the length of a solution can be restricted with a polynomial translation on the problem. As the number of decomposing compound tasks is bounded double exponentially, it can ensure that every sequence of acyclic progression terminates after a double exponential number of steps of insertion, performing and decomposition. Therefore, by applying non-deterministically acyclic decomposition, either it can reach a solution or it cannot progress any more which entails that the problem has no solution.

### 5 Conclusion

In this paper, we extended TIHTN so that state constraints can be captured. We have shown that this extension does not increase the complexity while it can represent cover planning frameworks: lifted TIHTN and HR-HGN.

State constraints are written as linear temporal logic (LTL) formulas in PDDL3. It seems not difficult to adapt the notion of consistency with constraint to LTL formulas.

The experimental results in [Shivashankar et al., 2013] show that the completeness of methods can contribute to the performance of the HGN planners. It is also interesting to extend TIHTNS again with allowing method insertion. A promising reference is [Herzig et al., 2016] which provides a dynamic view of HTN planning and models decomposition in propositional dynamic logic.

### Acknowledgments

We wants to appreciate the reviewers for their insightful comments. Zhanhao Xiao is partially supported by CSC (Chinese Scholarship Council) and Hai Wan is partially supported by National Natural Science Foundation of China under grants 61573386, Natural Science Foundation of Guangdong Province under grant 2016A030313292 and Guangdong Province Science and Technology Plan projects under grant 2016B030305007. Laurent Perrussel is partially supported by ANR-11-LABX-0040-CIMI within the program ANR-11-IDEX-0002-02.
References


