32nd International Conference on
Automated Planning and Scheduling

June 13–24, 2022, virtually from Singapore

HPlan 2022

Proceedings of the 5th Workshop on

Hierarchical Planning

Program Committee

Ron Alford	The MITRE Corporation
Gregor Behnke	University of Amsterdam
Pascal Bercher	The Australian National University
Arthur Bit-Monnot	Laboratory for Analysis and Architecture of Systems, LAAS-CNRS, Toulouse, France.
Alberto Camacho	University of Toronto
Maurício Cecílio Magnaguagno	PUCRS
Lavindra de Silva	University of Cambridge
Juan Fernández-Olivares	University of Granada
Alban Grastien	The Australian National University
Daniel Höller	Saarland University, Saarland Informatics Campus
Jane Jean Kiam	University of the Bundeswehr Munich
Songtuan Lin	The Australian National University
James Mason	Jet Propulsion Laboratory
Simona Ondrčková	Charles University
Sunandita Patra	IIT Kharagpur
Dominik Schreiber	Karlsruhe Institute of Technology
Vikas Shivashankar	Amazon Robotics
Shirin Sohrabi	IBM
David Speck	University of Freiburg
Álvaro Torralba	Aalborg University
Julia Wichlacz	Saarland University
Ying Xian Wu	The Australian National University
Zhanhao Xiao	Sun Yat-sen University

Organizing Committee

Pascal Bercher	The Australian National University
Jane Jean Kiam	University of the Bundeswehr Munich, Germany
Arthur Bit-Monnot	Laboratory for Analysis and Architecture of Systems, LAAS-CNRS, Toulouse, France.
Ron Alford	The MITRE Corporation

Preface

The motivation for using hierarchical planning formalisms is manifold. It ranges from an explicit and predefined guidance of the plan generation process and the ability to represent complex problem solving and behavior patterns to the option of having different abstraction layers when communicating with a human user or when planning cooperatively. This led to numerous hierarchical formalisms and systems. Hierarchies induce fundamental differences from classical, non-hierarchical planning, creating distinct computational properties and requiring separate algorithms for plan generation, plan verification, plan repair, and practical applications. Many techniques required to tackle these – or further – problems in hierarchical planning are still unexplored. With this workshop, we bring together scientists working on many aspects of hierarchical planning to exchange ideas and foster cooperation.

In 2022, the 5th edition of the workshop, we, for the second time, received an astonishing 14 submissions that underwent review (a 15th was submitted but couldn't be completed before the deadline when reviewing started), 9 of which were accepted unconditionally and 3 further ones got accepted after a round of revision. As in all previous HPlan-workshops, each paper received at least three reviews, while some received four if the reviewers disagree. To ensure the high quality of reviews comparable to major top-tier conferences, each reviewer was assigned at most 2 submissions according to their preferences and expertise, which were expressed during the paper bidding phase.

Like in previous years, a range of topics is addressed in the papers of this workshop. You find a table of contents later in the proceedings, which include both scientific papers as well as challenge papers.

Furthermore, as in previous years, we also encourage presentation of work accepted or published at other conferences or journals. This year, we have a presentation on a previous work published at AAAI on Propositional Dynamic Logic and another published at FLAIRS on plan verification. This way, the authors can share their findings at the ICAPS event, and specifically with the attendees of the HPlan-workshop and therefore with the hierarchical planning community.

Due to the high number of accepted papers, we planned a 6-hour workshop. All papers are shortly announced with teaser talks of 5 to 7 minutes, and then discussed in more depth in two virtual poster sessions, each taking 75 and 90 minutes respectively and hosting 7 posters each, carried out via gather.town. We have witnessed vivid and constructive discussions during the poster sessions in the previous years, taking even hours longer than the officially allocated 6 hours. We believe that the attendees can continue this year to enjoy the convivial poster sessions to discuss.

As in previous years, we also feature an invited talk, this year by Ugur Kuter, who introduces his work on a framework for collaborative decentralized planning and the importance of hierarchical representation and reasoning in this. On the next pages you will find an abstract of the talk, as well as a bio of the speaker.

Pascal, Jane, Arthur, Ron
HPlan Workshop Organizers,
June 2022

Invited Talk

Each year so far we had one or two invited talks. This year, our invited talk is by *Ugur Kuter*.

Hierarchies in Decentralized Collaborative Planning

This talk will introduce a framework for collaborative decentralized planning for a spectrum of user-centered and fully-autonomous, and thus, diverse, elastic, and possibly virtual, collaborative teams. We stand on the great strides from previous distributed AI ideas, combining those with modern advances in AI planning, thereby reflecting and explicitly representing and working with issues such as information locality, uncertainty, communications, and others. We will discuss the essential features of this framework, argue why we found hierarchical representations and reasoning important in our applications. We will describe some of the challenges that we face in this domain, and some of the initial approaches we took towards addressing these challenges.

Bio

Ugur Kuter, Research Fellow, Smart Information-Flow Technologies (SIFT)

Dr. Kuter is studying intelligence and intelligent behavior in both individuals and groups, human and computational. He is particularly interested in understanding and developing intelligent systems that can observe, think, and (inter-)act with other such systems, including humans. Over the years, a large part of Dr. Kuter's research is involved particularly around Automated (AI) Planning, Machine Learning, Reasoning with / about Uncertainty, Automation Self-Confidence, Knowledge Representations, Graph Analysis, Semantic, Social Networks, Game Theory, Evolutionary Computation, and recently, Quantum Cognition. Dr. Kuter has co-authored more than 100 peer-viewed research publications, including several best papers and other awards.

Table of Contents

Scientific Papers

Challenge Papers

An Accurate HDDL Domain Learning Algorithm from Partial and Noisy Observations

Maxence Grand, Damien Pellier, Humbert Fiorino

Univ. Grenoble Alpes, LIG, Grenoble, France
{Maxence.Grand, Damien.Pellier, Humbert.Fiorino}@univ-grenoble-alpes.fr

Abstract

The Hierarchical Task Network (HTN) formalism is very expressive and used to express a wide variety of planning problems. In contrast to the classical STRIPS formalism in which only the action model needs to be specified, the HTN formalism requires to specify, in addition, the tasks of the problem and their decomposition into subtasks, called HTN methods. For this reason, hand-encoding HTN problems is considered more difficult and more error-prone by experts than classical planning problem. To tackle this problem, we propose a new approach (HierAMLSI) based on grammar induction to acquire HTN planning domain knowledge, by learning action models and HTN methods with their preconditions. Unlike other approaches, HierAMLSI is able to learn both actions and methods with noisy and partial inputs observation with a high level or accuracy.

1 Introduction

The Hierarchical Task Network (HTN) formalism (Erol, Hendler, and Nau 1994) is very expressive and used to express a wide variety of planning problems. This formalism allows planners to exploit domain knowledge to solve problems more efficiently (Nau et al. 2005) when planning problems can be naturally decomposed hierarchically in terms of tasks and task decompositions. The standard language used to model HTN problem is HDDL (Hierachical Domain Description Language) (Höller et al. 2020). In contrast to the classical PDDL language used to model STRIPS problems in which only the action model needs to be specified, HDDL requires to specify the task model of the problem. A task model can be primitive and compound. A primitive task model is described by PDDL operators. A compound tasks model is described using HTN methods. An HTN method describes the set of primitive and/or compound task required to decompose a specific compound task. For this reason, hand-encoding HTN problems is considered more difficult and more error-prone by experts than classical planning problem. This makes it all the more necessary to develop techniques to learn HTN domains.

Many machine learning approaches have been proposed to facilitate the acquisition of PDDL domain acquisition and to learn the underlying action model, e.g, ARMS (Yang, Wu, and Jiang 2007), FAMA (Aineto, Celorrio, and Onaindia 2019), LOCM (Cresswell, McCluskey, and West 2013), LSONIO (Mourão et al. 2012), AMLSI (Grand, Fiorino, and Pellier 2020a,b). In these approaches, training data are either (possibly noisy and partial) intermediate states and plans previously generated by a planner, or randomly generated action sequences (i.e. random walks). On the other hand, few approaches have been proposed to learn HTN domains. However, it is possible to mention CAMEL (Ilghami et al. 2002), HTN-Maker (Hogg, Munoz-Avila, and Kuter 2008; Hogg, Kuter, and Munoz-Avila 2009), LHTNDT (Nargesian and Ghassem-Sani 2008) or HTN-Learner (Zhuo et al. 2009). The major drawbacks of these approaches are: (1) they consider to have complete and noiseless observations as input; (2) they only learn HTN methods except HTN-Learner, i.e., they consider that the action model is known a priori and (3) the learned domains are not *accurate* enough to be used "as is" in a planner. A step of expert proofreading is still necessary to correct them. Even small syntactical errors can make sometime the learned domains useless for planning

To deal with these drawbacks, we propose in this paper, a new learning algorithm for HDDL domains, called HierAMLSI. HierAMLSI is based on AMLSI (Grand, Fiorino, and Pellier 2020a,b), a PDDL domain learner based on grammar induction. HierAMLSI takes as input a set of partial and noisy observations and learns a full HDDL planning domain with action model and HTN methods. We show experimentally that HierAMLSI is highly accurate even with highly partial and noisy learning datasets minimising HDDL domain proofreading by experts. In many HDDL ICP benchmarks HierAMLSI does not require any correction of the learned domains at all.

The rest of the paper is organized as follows. In section 2 we present the problem statement. In section 3 we give some backgrounds on the AMLSI approach. In section 4, we detail the HierAMLSI steps. Then, section 5 evaluates the performance of HierAMLSI on IPC benchmarks. Finally, Section 6 concludes with the related works.

2 Formal Framework

Section 2.1 introduces a formalization of STRIPS planning domain learning consisting in learning a transition function of a grounded planning domain and in expressing it as PDDL operators and Section 2.2 extends this formalization to HTN domains.

2.1 STRIPS Planning

In this section we use definitions and notations proposed by (Höller et al. 2016) and adapt them to the learning problem.

A STRIPS planning problem is a tuple $P = (L, A, S, s_0, s, \delta, \lambda)$, where L is a set of logical propositions describing the world states, S is a set of state labels, $s_0 \in S$ is the label of the initial state, and $g \subseteq S$ is the set of goal labels. λ is an observation function $\lambda : S \rightarrow 2^L$ that assigns to each state label the set of logical propositions true in that state. A is a set of action labels.Action preconditions, positive and negative effects are given by the functions $prec$, add and del that are included in $\delta = (prec, add, del)$. $prec$ is defined as $prec : A \rightarrow 2^L$. The functions add and del are defined in the same way. Without loss of generality, we chose this unusual formal framework inspired by (Höller et al. 2016) in order to define the STRIPS learning problem as the lifting of a state transition system into a propositional language.

The function $\tau : A \times S \rightarrow \{true, false\}$ returns whether an action is applicable to a state, i.e. $\tau(a, s) \Leftrightarrow prec(a) \subseteq \lambda(s)$. Whenever action a is applicable in state s_i, the state transition function $\gamma : A \times S \rightarrow S$ returns the resulting state $s_{i+1} = \gamma(s_i, a)$ such that $\lambda(s_{i+1}) = [\lambda(s_i) \setminus del(a)] \cup add(a)$.

A sequence $(a_0 a_1 \dots a_n)$ of actions is applicable to a state s_0 when each action a_i with $0 \leq i \leq n$ is applicable to the state s_i. Given an applicable sequence $(a_0 a_1 \dots a_n)$ in state s_0, $\gamma(s_0, (a_0 a_1 \dots a_n)) = \gamma(\gamma(s_0, a_0), (a_1 \dots a_n)) = s_{n+1}$. It is important to note that this recursive definition of γ entails the generation of a sequence of states $(s_0 s_1 \dots s_{n+1})$. A goal state is a state s such that $g \in G$ and $\lambda(g) \subseteq \lambda(s)$. s satisfies g, i.e. $s \models g$ if and only if s is a goal state. An action sequence is a solution plan to a planning problem P if and only if it is applicable to s_0 and entails a goal state.

In formal languages, a set of rules is given that describe the structure of valid words and the language is the set of these words. For STRIPS planning problem $P = (L, A, S, s_0, G, \delta, \lambda)$, this language is defined as ($0 \leq i \leq n$):

$$\mathcal{L}(P) = \{\omega = (a_0 a_1 \dots a_n) | a_i \in A, \gamma(s_0, \omega) \models g\}$$

We know that the set of languages generated by STRIPS planning problems are regular languages (Höller et al. 2016). In other words, a STRIPS planning problem $P = (L, A, S, s_0, G, \delta, \lambda)$ generates a language $\mathcal{L}(P)$ that is equivalent to a Deterministic Finite Automaton (DFA) $\Sigma = (S, A, \gamma)$. S and A are respectively the nodes and the arcs of the DFA, and γ is the transition function.

For any arc $a \in A$, we call *pre-set* of a the set $preset(a) = \{s \in S \mid \gamma(s, a) = s'\}$ and *post-set* of a the set $postset(a) = \{s' \in S \mid \gamma(s, a) = s'\}$ (see Figure 1).

A STRIPS learning problem is as follow: given a set of observations $\Omega \subseteq \mathcal{L}(P)$, is it possible to learn the DFA Σ, and then infer P?

For instance, suppose $\Omega = \{a, ab, ba, bab, abb, \dots\}$ such that $s_0 \xrightarrow{a} s_2$, $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_2$, $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_2 \xrightarrow{b} s_2 \dots$ (Grand, Fiorino, and Pellier 2020b) show that it is possible to learn Σ (see Figure 1)

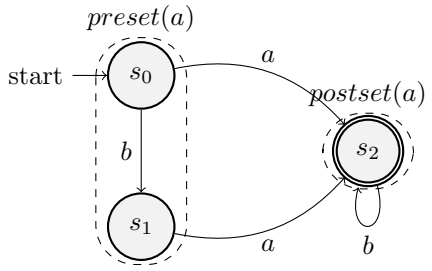


Figure 1: An example of DFA with pre-states and post-states

and infer P with actions $\{a, b\}$, the initial state s_0 and some states marked as goal $G = \{s_2\}$.

2.2 HTN Planning

By extension based on the notation of (Höller 2021), an HTN planning problem is a tuple $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$. As for STRIPS problems, L is a set of logical propositions describing the world states, S is a set of state labels, $s_0 \in S$ is the label of the initial state, $g \subseteq S$ is the set of goal label, λ is the observation function and preconditions, positive and negative effects are given by the functions $prec$, add and del included in δ.

A is the set action (or primitive task) labels and C is a set of compound (or non primitive) task labels, with $C \cap A = \emptyset$. Tasks are maintained in *task networks*. A task network is a sequence of tasks (for simplicity, we consider only Totally Ordered domain). Let $T = C \cup A$. A task network is an element out of T^* (* is the Kleene operator). Compound tasks are decomposed using methods. The set M contains all method labels. Methods are defined by the function $\sigma : M \rightarrow C \times T^*$. Then, a coumpound task c is decomposable in a state s if and only if there exists a revelant method $m \in M$ such that: $\sigma(m) = (c, \phi)$ and $prec(m) \in s$. The function $\zeta : T^* \times S \rightarrow T^*$ gives the decomposition function. For a totally orderer task network $\omega = \omega_1 t \omega_2$, ζ is defined as follows:

$$\zeta(\omega_1 t \omega_2, s) = \begin{cases} \omega_1 t \omega_2 & \text{if } t \text{ is a primitive task} \\ \omega_1 \phi \omega_2 & \text{if } t \text{ is a coumpound task} \\ & \text{and } t \text{ is decomposable in } \gamma(\omega_1, s) \\ \emptyset & \text{Otherwise} \end{cases}$$

As $\omega_1 t \omega_2$ is a totally ordered task network, ω_1 contains only primitive tasks. We denote $\omega \rightarrow^* \omega'$ that ω can be decomposed into ω' by 0 or more method applications. Finally, ω_I is the initial task network.

A solution to an HTN planning problem is a task network ω with:

1. $\omega_I \rightarrow^* \omega$, i.e. it can be reached by decomposing ω_I.
2. $\omega \in A^*$, i.e. all tasks are primitive.
3. $\gamma(s_0, \omega) \models g$, i.e. ω is applicable in s_o and results in a goal state.

Finally, we can define an HTN planning problem $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$ as a formal language:

$$\mathcal{L}(P) = \{\omega = (t_0 t_1 \dots t_n) | t_i \in A, \gamma(s_0, \omega) \models g, \omega_I \rightarrow^* \omega\}$$

An HTN learning problem is as follow: given a set of observations $\Omega \subseteq \mathcal{L}(P)$, is it possible to learn P?

Unlike STRIPS planning problems, the language $\mathcal{L}(P)$ is not necessary regular (Höller et al. 2014) and cannot be represented as a DFA. As mentioned by (Höller et al. 2014; Höller 2021), $\mathcal{L}(P)$ is the intersection of two languages:

1. $\mathcal{L}_{\mathcal{H}}(P) = \{\omega \in A^* | w_I \rightarrow^* \omega\}$ which is defined by the decomposition hierarchy, i.e. by the compound tasks and methods.
2. $\mathcal{L}_{\mathcal{C}}(P) = \{\omega \in A^* | \gamma(s_0, \omega) \in g\}$ which is defined by the state transition system defined by the preconditions and effects of the primitive tasks. This language is regular.

The key idea of our approach is to learn the DFA $\Sigma = (S, A, \gamma)$ corresponding to the regular language $\mathcal{L}_{\mathcal{C}}(P)$, and modify the DFA to approximate the language $\mathcal{L}(P)$ with the DFA $\Sigma = (S, T, \gamma)$ and then infer P.

3 The AMLSI Approach

In this section we will summarized the AMLSI approach on which HierAMLSI is based. For more detail see (Grand, Fiorino, and Pellier 2020b,a).

AMLSI generates the set of observations Ω by using random walks to learn $\Sigma = (S, A, \gamma)$ and deduce $P = (L, A, S, s_0, G, \delta, \lambda)$. AMLSI assumes L, A, S, s_0 known and the observation function λ possibly partial and noisy (a partial observation is a state where some propositions are missing and a noisy observation is a state where the truth value of a proposition is erroneous). No knowledge of the goal states G is required. Once Σ is learned, AMLSI has to deduce δ from the transition function γ. Concretely, δ can be represented as a STRIPS planning domain containing all the actions of the problem P and by induction the classical PDDL operators.

The AMLSI algorithm consists of 4 steps: (1) generation of the observations, (2) learning the DFA corresponding to the observations, (3) induction of the PDDL operators from the learned DFA; (4) finally, refinement of these operators to deal with noisy and partial state observations:

Step 1: AMLSI generates a random walk by applying an action from the initial state of the problem. If the action is applicable in the current state the sequence of actions from the initial state is valid and is added to I^+, the set of positive samples. Otherwise the random walk is stopped and the sequence is added to I_-, the set negative samples.

Step 2: To learn the DFA $\Sigma = (S, A, \gamma)$ AMLSI uses a variant (Grand, Fiorino, and Pellier 2020b) of a classical regular grammar learning algorithm called RPNI (Oncina and García 1992). The learning is based on both I^+ and I^-.

Step 3: AMLSI begins by inducing the preconditions and effects of the actions. For the preconditions $prec(a)$ of action a, AMLSI computes the logical propositions that are in all the states preceding a in Σ:

$$prec(a) = \cap_{s \in preset(a)} \lambda(s)$$

For the positive effects $add(a)$ of action a, AMLSI computes the logical propositions that are never in states before the execution of a, and always present after a execution:

$$add(a) = \cap_{s \in preset(a)} \lambda(s) \setminus prec(a)$$

Dually,

$$del(a) = prec(a) \setminus \cap_{s \in postset(a)} \lambda(s)$$

Once preconditions and effects are induced, actions are lifted to PDDL operators based on OI-subsumption (subsumption under Object Identity) (Esposito et al. 2000): first of all, constant symbols in preconditions and effects are substituted by variable symbols. Then, the less general preconditions and effects, i.e. preconditions and effects encoding as many propositions as possible, are computed as intersection sets. This generalization method allows to ensure that all the necessary preconditions, i.e. the preconditions allowing to differentiate the states where actions are applicable from states where they are not, to be rightfully coded in the corresponding operators.

Step 4: To deal with noisy and partial state observations, AMLSI starts by refining the operator effects to ensure that the generated operators allow to regenerate the induced DFA. To that end, AMLSI adds all the effects ensuring that each transition in the DFA are feasible. Then, AMLSI refines the preconditions of the operators. As in (Yang, Wu, and Jiang 2007), it makes the following assumptions: the negative effects of an operator must be used in its preconditions. Thus, for each negative effect of an operator, AMLSI adds the corresponding proposition in the preconditions. Since effect refinements depend on preconditions and precondition refinements depend on effects, AMLSI repeats these two refinements steps until convergence, i.e., no more precondition or effect is added. Finally, AMLSI performs a Tabu Search to improve the PDDL operators independently of the induced DFA, on which operator generation is based. Once the Tabu Search reaches a local optimum, AMLSI repeats all the three refinement steps until convergence.

4 The HierAMLSI approach

HierAMLSI generates the set of observations Ω by using random walks to learn $\Sigma = (S, T, \gamma)$ and deduce $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$. HierAMLSI makes the same assumptions than AMLSI: L, A, S, s_0 are known but also C, the decomposition function ζ and the observation function λ possibly partial and noisy (a partial observation is a state where some propositions are missing and a noisy observation is a state where the truth value of a proposition is erroneous). No knowledge of the goal states g is required. Once Σ is learned, HierAMLSI has to deduce δ from the transition function γ and σ from the decomposition function ζ.

Like AMLSI, HierAMLSI uses random walks to generate Ω and makes the same assumptions.

HierAMLSI has been devised to solve HTN learning problems. In practice, it computes P as HDDL domain and problem files (Höller et al. 2020; Höller et al. 2019).

Regarding the training dataset, HierAMLSI uses random walks to generate Ω. HierAMLSI makes the same assumptions than AMLSI. Once the DFA is learned, HierAMLSI generates the set of methods σ and infers the action precondition, positive and negative effect functions in δ from the state transition function γ. Finally, the methods and operators of the HDDL domain file are induced from σ and δ.

The HierAMLSI approach consists of 4 steps:

1. *Generation of the observations.* HierAMLSI produces a set of observations Ω by using a random walk. In section 4.1, we will present how HierAMLSI is able to efficiently exploit these observations by taking into account not only the fact that some task are decomposable in certain states and their decomposition but also that others are not.
2. *DFA Learning.* HierAMLSI learns a DFA approximating the language $\mathcal{L}(P)$ (see Section 4.2).
3. HTN *Methods generation.* HierAMLSI generates from the DFA learned previously a set of HTN Methods allowing to decompose all tasks observed in Ω (see Section 4.3).
4. *Action model and* HTN *Methods precondition learning.* Once HTN Methods have been learned, HierAMLSI have to learn the action model, i.e. primitive tasks preconditions and effects and the HTN Methods preconditions. To do this, HierAMLSI treat HTN Methods as primitive tasks and learn an action model containing both methods and primitive tasks using the learning and refinement techniques proposed by the AMLSI approach (see Section 3).

The rest of this Section will be illustrated using the IPC[1] Gripper domain. In this domain, a robot moves balls in different rooms using its two grippers r and l. This domain contains three compound tasks: $goto(r_a)$ the robot goes into the room r_a, $move1ball(b_1\ r_a)$ the robots move the ball b_1 in the room r_a and $move2balls(b_1\ b_2\ r_a)$ the robot moves balls b_1 and b_2 in the room r_a.

4.1 Observation Generation

The data generation process is similar to the generation method of the AMLSI algorithm (Grand, Fiorino, and Pellier 2020b). To generate the observations in Ω, HierAMLSI uses random walks by querying a blackbox. HierAMLSI chooses randomly a (primitive or compound) task t. If the task t is decomposable, the blackbox returns the final decomposition ϕ containing only primitive task and this decomposition is added to the current primitive task sequence. This procedure is repeated until the selected task is not applicable to the current state. The applicable prefix of the primitive task sequence is then added to I_+, the set of positive samples, and the complete sequence, whose last task is not applicable, is added to I_-, the set of negative samples. Random walks are repeated until I_+ and I_- achieve an arbitrary size.

4.2 DFA Learning

As mentioned in Section 2 the language $\mathcal{L}(P)$ is not necessary regular, then the purpose of this step is to learn a DFA approximating this language. More precisely, the DFA learning step is divided in 2 steps: (1) HierAMLSI learns a DFA corresponding to the language $\mathcal{L}_\mathcal{C}(P)$ which is defined by the state transition system defined by the preconditions and effects of the primitive tasks and (2) HierAMLSI adds transitions to represent compound tasks in the DFA to allow to approximate the language $\mathcal{L}(P)$.

[1] https://www.icaps-conference.org/competitions/

Algorithm 1: Heuristic HTN Methods learning

```
Initialization(M)
for i = 1 : |C| do
    for c ∈ C do
        M'[c] ← greedy(c, M, i)
    for c ∈ C do
        if |M'[c]| < |M[c]| then
            M[c] ← M'[c]
return M
```

Step 1: Primitive task DFA Learning HierAMLSI starts by using the DFA Learning algorithm proposed by (Grand, Fiorino, and Pellier 2020b) to learn the DFA containing only primitive tasks.

Step 2: Task DFA Induction Once the Primitive Task DFA has been learned, HierAMLSI induces the Task DFA by adding compound task transition in the DFA, i.e. by adding transitions whose labels are compound task labels. For instance, suppose we have the compound task $move2balls(b_1\ b_2\ r_b)$ has been decomposed by primitive tasks $(move(r_a\ r_b),\ pick(b_1\ r\ r_b),\ pick(b_2\ l\ r_b),\ move(r_b\ r_a),\ drop(b_1\ r\ r_a),\ drop(b_2\ l\ r_a))$ in node 0 and reached the node 6. Then we add the following transitions in the DFA $\gamma(move2balls(b_1\ b_2\ r_b), 0) \rightarrow 6$. Figure 2 gives an example of Task DFA.

4.3 HTN Methods Learning

Once the Task DFA is induced HierAMLSI can directly extract HTN Methods from the Task DFA. However, it is possible that a large number of HTN Methods has been generated. For instance, let's take the Task DFA in Figure 2. For the compound task $move1ball(?b\ ?r)$ HierAMLSI can generate several methods:

$$\begin{aligned}
\phi_1 &= (move(?r\ ?r_1), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r)) \\
\phi_2 &= (pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r)) \\
\phi_3 &= (goto(?r_1), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r)) \\
\phi_4 &= (goto(?r_1), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r)) \\
&\dots
\end{aligned}$$

Some of these decomposition are redundant. To facilitate proof reading we want a more compact description of the HTN Methods. More precisely, we want minimizing the set of methods allowing to decompose observed compound tasks. Then, the HTN Methods learning problem can be reduce to a variant of the set cover problem (Karp 1972) which is NP-Complete.

The Greedy algorithm (Jungnickel 1999) is a classical way to approximate the solution in a polynomial time. At each stage of the Greedy algorithm, HierAMLSI chooses

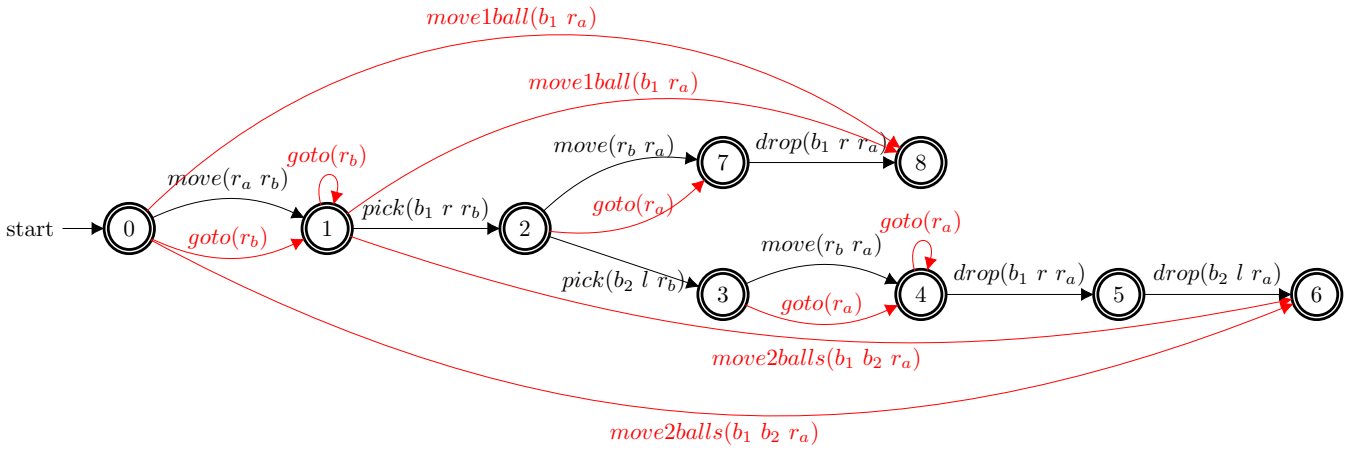


Figure 2: DFA learning steps The Primitive Task DFA is the DFA containing only Primitive Task, i.e. black transition, and the Task DFA contains Compound Tasks, i.e. red transitions, in addition

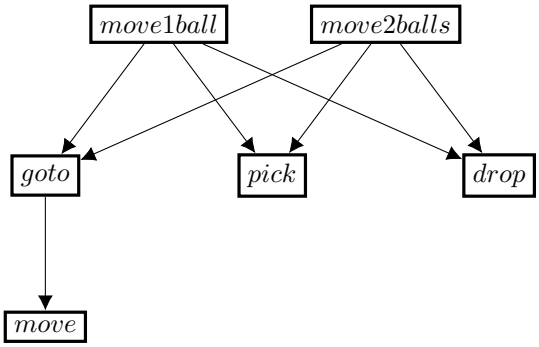


Figure 3: Tasks dependencies for the IPC Gripper domain

the method that allows to decompose the largest number of tasks.

The main drawback of this approach is that it does not take into account the fact that there are dependencies between tasks. For instance, the optimal way to decompose the compound task $move1ball(?b\ ?r)$ is $\phi_4 = (goto(?r_1), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r))$ (see Figure 3). So the compound task $move1ball(?b\ ?r)$ depends of the compound task $goto(?r)$. So, as long as all methods for the compound task $goto(?r)$ have been generated, the Greedy algorithm will always prioritize ϕ_1 and ϕ_2 to ϕ_4. Indeed, the decomposition ϕ_4 can be added to the current solution of the Greedy algorithm if and only if all methods for the compound task $goto(?r)$ have been added.

Heuristic Approach We propose a sound, complete and polynomial Heuristic approach (see Algorithm 1) taking into account dependencies between tasks. Figure 4 gives an example for the Gripper domain.

HierAMLSI starts by initializing the set of HTN methods using the decomposition ϕ observed during the observation generation step (see Section 4.1). For each Compound Task we have therefore a set of HTN Methods containing only Primitive Tasks and no Compound Task dependencies. For instance, for the Compound Task $move1ball(?b\ ?r)$ we have the two following decomposition:

$$\begin{aligned}\phi_1 = &\ (pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))\\ \phi_2 = &\ (move(?r_?\ ?r_1)), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))\end{aligned}$$

Then, at each iteration AMLSI use the greedy algorithm to compute a new set of HTN Methods with an additional Compound Task dependency. Finally, if the new HTN Methods set is smaller than the one learned in the previous iteration, then it is retained. For instance, for the Gripper domain, suppose we have the two following decomposition for the Compound Task $goto(?r)$:

$$\begin{aligned}\phi_1 = &\ ()\\ \phi_2 = &\ (move(?r_1\ ?r)))\end{aligned}$$

Then, the Greedy algorithm return only one decomposition for the Compound Task $move1ball(?b?r)$: $(goto(?r_1)), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r))$.

Lemma 1. *The Heuristic approach is sound and complete. The heuristic approach generates a set of* HTN *Methods M able to decompose all observed compound tasks in the dataset Ω.*

Proof. During the observation generation step (see Section 4.1), for each generated Compound Task t, we have its final decomposition ϕ. So at the initialization step of the Heuristic approach, there at least one method able to decompose each observed Task. The initialization is therefore sound and complete. Moreover the following steps of the Heuristic approach generates methods decomposing as many tasks as the previous steps, then the Heuristic approach is sound and complete. □

Lemma 2. *The Heuristic approach is polynomial.*

$goto(?r):$ $\phi_1 = (), \phi_2 = move(?r_1\ ?r)$
$move2balls(?b_1\ ?b_2\ ?r):$ $\phi_1 = (pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), move(?r_1\ ?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$
$\phi_2 = (move(?r\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), move(?r_1\ ?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$
$move1ball(?b\ ?r):$ $\phi_1 = (pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$
$\phi_2 = (move(?r_?\ ?r_1)), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$

(a) **Step 0:** Initialization with no compound task dependency

$goto(?r):$ $\phi_1 = (), \phi_2 = move(?r_1\ ?r)$
$move2balls(?b_1\ ?b_2\ ?r):$ $\phi_1 = (goto(?r_1), pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), goto(?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$
$move1ball(?b\ ?r):$ $\phi_1 = (goto(?r_1), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r))\}$

(b) **Step 1:** Induction with 1 compound task dependence

$goto(?r):$ $\phi_1 = (), \phi_2 = move(?r_1\ ?r)$
$move2balls(?b_1\ ?b_2\ ?r):$ $\phi_1 = (goto(?r_1), pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), goto(?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$
$move1ball(?b\ ?r):$ $\phi_1 = (goto(?r_1), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r))\}$

(c) **Step n:** Induction with n compound task dependence

Figure 4: HTN Methods learning example

Domain	# Primitive Task	# Compound Task	# Methods	# Predicates
Blocksworld	4	4	8	5
Gripper	3	3	4	4
Zenotravel	4	2	5	7
Transport	3	4	6	5
Childsnack	6	1	2	12

Table 1: Benchmark domain characteristics. From left to right, the number of Primitive Tasks, the number of Compound Tasks, the number of Methods and the number of Predicates for each IPC domain.

Proof. First of all, we have $\mathcal{O}(|I_+|)^2$ nodes in the DFA. Then, in the worst case, we have a possible HTN Method for each node pair, then we have $\mathcal{O}(|I_+|^2)$ possible HTN Methods in the DFA. Then, the complexity of the Greedy algorithm is $\mathcal{O}(|I_+|^3)$ in term of tested decomposition. Finally, according to the algorithm 1, the Greedy algorithm is repeated $|C|^2$ times. Finally, the complexity of the Heuristic approach is $\mathcal{O}(|C|^2.|I_+|^3)$. □

5 Exprimentation

The purpose of this evaluation is to evaluate the performance of HierAMLSI though two variants: (1) we evaluate the performance of HierAMLSI when only HTN Methods are learned, i.e. the action model is known and (2) we evaluate the performance of HierAMLSI when both HTN Methods are learned and the action model is unknown. We use 4 experimental scenarios[3]:

1. Complete intermediate observations (100%) and no noise (0%).
2. Complete intermediate observations (100%) and high level of noise (20%).
3. Partial intermediate observations (25%) and no noise (0%).
4. Partial intermediate observations (25%) and high level of noise (20%).

[2] $|I_+|$ denote the number of primitive tasks in the positive sample

[3] Note that these are the experimental scenarios used to test AMLSI on which HierAMLSI is built

5.1 Experimental Setup

Our experiments are based on 5 HDDL (Höller et al. 2020; Höller et al. 2019) domains (see Table 1) from the IPC 2020 competition: Blocksworld, Childsnack, Transport, Zenotravel and Gripper.

HierAMLSI learns HTN domains from one instance. To avoid performances being biased by the initial state, HierAMLSI is evaluated with different instances. Also, for each instance, to avoid performances being biased by the generated observations, experiments are repeated five times. All tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz with 16GB of memory. PDDL4J library (Pellier and Fiorino 2018) was used to generate the benchmark data.

5.2 Evaluation Metrics

HierAMLSI is evaluated using the *accuracy* (Zhuo, Nguyen, and Kambhampati 2013) that measures the learned domain performance to solve new problems.

Formally, the accuracy $Acc = \frac{N}{N^*}$ is the ratio between N, the number of correctly solved problems with the learned domain, and N^*, the total number of problems to solve. In the rest of this section the accuracy is computed over 20 problems. The problems are solved with the TFD planner (Pellier and Fiorino 2020) provided by the PDDL4J library. Plan validation is done with VAL, the IPC competition validation tool (Howey and Long 2003).

5.3 Discussion

Table 5 shows the accuracy of HierAMLSI obtained on the 5 domains of our benchmarks when varying the training data

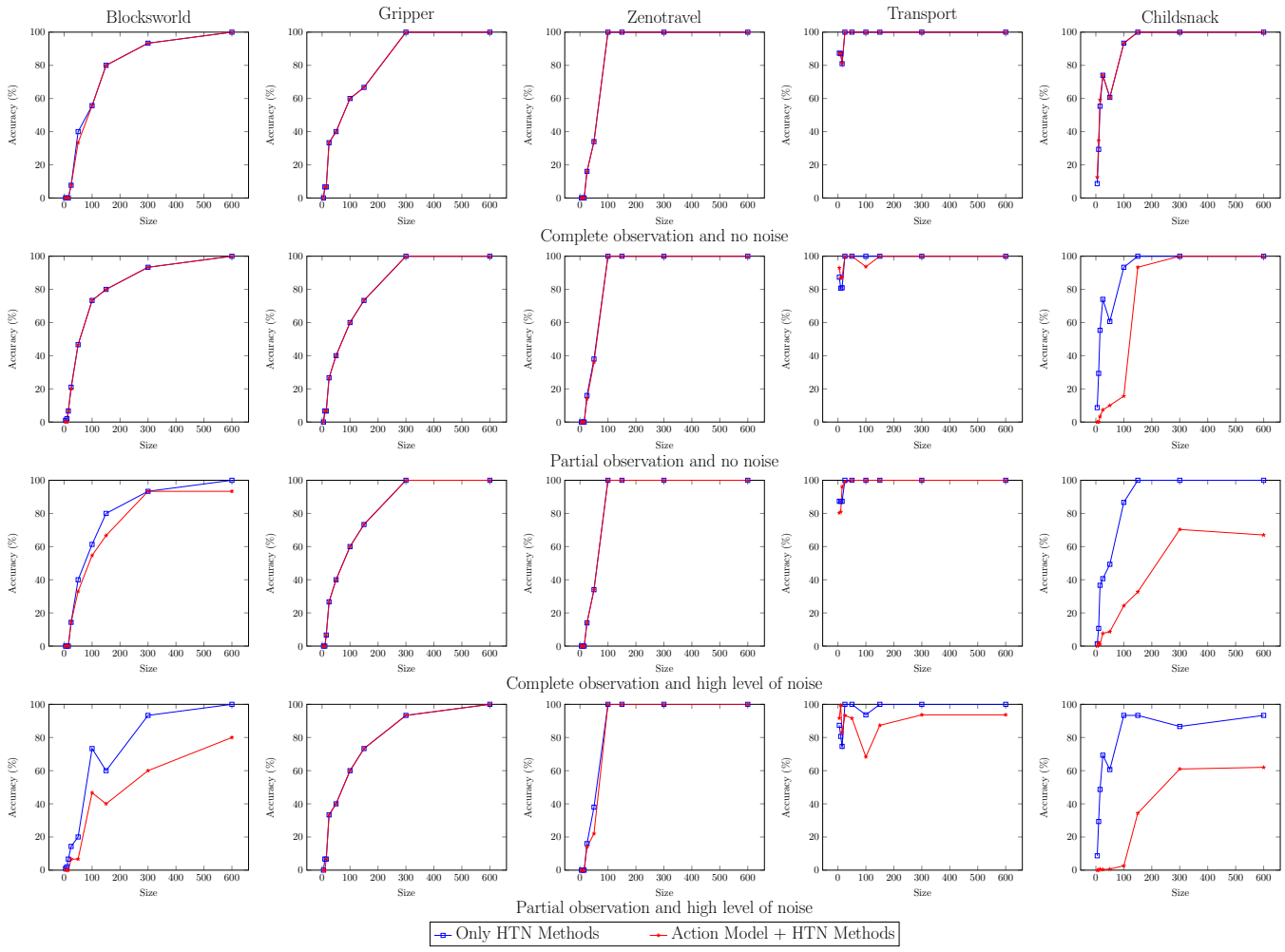


Figure 5: Performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

set size. The size of the training set is indicated in number of tasks.

First of all, we observe that when HierAMLSI learns only the set of HTN Methods, learned domains are generally optimal (Accuracy $= 100\%$) with 600 tasks whatever the experimental scenario. Also, 100 tasks are generally sufficient to learn accurate domains (Accuracy $> 50\%$). Then, when HierAMLSI learns both action model and HTN Methods performances are similar when observations are noiseless. However, when observations are noisy, performances are downgraded for some domains: Blocksworld and Childsnack when observations are complete and Blocksworld, Transport and Childsnack when observations are partial. This is due to the fact that there are learning error in the action model learned. However, learned domains remain accurate when there are at least 300 tasks in the training dataset.

To conclude, we have shown experimentally that HierAMLSI learns accurate domains. More precisely, when the action model is known, HierAMLSI generally learns optimal domains. Also the performances are downgraded when HierAMLSI has to learn the action model in addition to the set of HTN methods, but the learned domains remain accurate. Performance degradation are due to the learning errors in the action model.

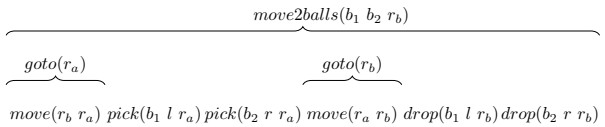
6 Related Works

Many approaches have been proposed to learn HDDL domains. These approaches can be classified according to the input data of the learning process. The input data can be plan "traces" obtained by resolving a set of planning problems, annoted plans (see Figure 6a), decomposition tree (see Figure 6b) or random walks. The input data can contain in addition to the tasks also states which can be fully observable (FO) or partially observable (PO), or noisy. Also, these approaches can be classified according to the output. The output can be the action model, the set of HTN Methods and HTN Methods preconditions. Table 2 summarises these classifications.

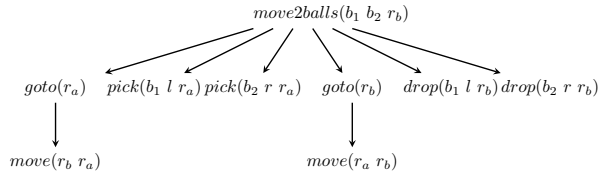
A first group of approaches only learns the set of HTN Methods. First of all, (Xiaoa et al. 2019) takes as input a set of plan traces and HTN Methods and proposes an algorithm to update incomplete HTN Methods by task insertions.

Algorithm	Input			Output		
	Input	Environment	Noise Level	Action Model	HTN Methods	HTN Methods Preconditions
(Xiaoa et al. 2019)	Plans	FO	0%		•	
HTN-Maker	Plans	FO	0%		•	
(Hogg, Kuter, and Munoz-Avila 2010)	Plans	FO	0%		•	
HTN-MakerND	Plans	FO	0%		•	
(Li et al. 2014)	Plans	FO	0%		•	
(Garland and Lesh 2003)	Annoted Plans	PO	0%		•	
LHTNDT	Annoted Plans	FO	0%		•	
CAMEL	Plans	FO	0%		•	•
HDL	Plans	FO	0%		•	•
HTN-Learner	Decomposition Trees	PO	0%	•	•	•
HierAMLSI	Random Walks	PO	20%	•	•	•

Table 2: State-of-the-art action model learning algorithms. From left to right: the kind of input data, the kind of environment: **F**ully **O**bservable or **P**artially **O**bservable, the maximum level of noise in observations, the kind of output



(a) An annoted plan



(b) A decomposition tree

Figure 6: Input examples

Then HTN-Maker (Hogg, Munoz-Avila, and Kuter 2008) and HTN-MakerND (Hogg, Kuter, and Munoz-Avila 2009) takes as input plan trace generated from STRIPS planner and annoted task provided by a domain expert. Then, (Hogg, Kuter, and Munoz-Avila 2010) proposed an algorithm based on reinforcement learning. Then, (Li et al. 2014) proposed an algorithm taking as input only plan traces. This algorithm builds, from plan traces, a context free grammar (CFG) allowing to regenerate all plans. Then, methods are generated using CFG: one method for each production rule in the CFG. Then (Garland and Lesh 2003) and (Nargesian and Ghassem-Sani 2008) proposed to learn HTN Methods from annoted plan. Annoted plan are plan segmented with the different tasks solved. Figure 6a gives an example of annoted plan. However, obtaining these annotated examples is difficult and needs a lot of human effort.

A second group of approach learns HTN Methods preconditions. First of all, the CAMEL algorithm (Ilghami et al. 2002) learns HTN Methods and their the preconditions of HTN Methods from observations of plan traces, using the version space algorithm. An annoted task is an triplet (n, Pre, Eff) where n is a task, Pre is a set of proposition known as the preconditions and Eff is a set of atoms known as the effects. These approach use annoted task to build incrementally HTN Methods with preconditions. Then, the HDL algorithm (Ilghami, Nau, and Muñoz-Avila 2006) takes as input plan traces. For each decomposition in plan traces, HDL checks if there exist a method responsible of this decomposition. If not, HDL creates a new method and initializes a new version space to capture its preconditions. Preconditions are learned in the same way as in the CAMEL algorithm.

Only HTN-Learner proposes to learn Action Model and HTN Methods from decomposition trees. A decomposition tree is a tree corresponding to the decomposition of a method. Figure 6b gives an example of decomposition tree.

7 Conclusion

In this paper we have presented HierAMLSI, a novel algorithm to learn HDDL domains. HierAMLSI is built on the AMLSI approach. HierAMLSI is composed of four steps. The first step consists, as AMLSI, in building two training sets of feasible and infeasible action sequences. In the second step, HierAMLSI induces a DFA. The third step is the generation of the HTN Methods, and the last step learns HTN Methods preconditions and the action model. Our experimental results show that HierAMLSI is able to learn accurately both action models and HTN Methods from partial and noisy datasets.

Future works will focus on extending HierAMLSI in order to learn more expressive action model.

Acknowledgments

This research is supported by the French National Research Agency under the "Investissements d'avenir" program (ANR-15-IDEX-02) on behalf of the Cross Disciplinary Program CIRCULAR.

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using *LOCM*. *Knowledge Engineering Review*, 28(2): 195–213.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In *Proc. of AAAI*, 1123–1128.

Esposito, F.; Semeraro, G.; Fanizzi, N.; and Ferilli, S. 2000. Multistrategy Theory Revision: Induction and Abduction in INTHELEX. *Machine Learning*, 38(1-2): 133–156.

Garland, A.; and Lesh, N. 2003. Learning hierarchical task models by demonstration. *MERL*.

Grand, M.; Fiorino, H.; and Pellier, D. 2020a. AMLSI: A Novel and Accurate Action Model Learning Algorithm. In *Proc of KEPS workshop*.

Grand, M.; Fiorino, H.; and Pellier, D. 2020b. Retro-engineering state machines into PDDL domains. In *Proc of ICTAI*, 1186–1193.

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *Proc of IJCAI*.

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2010. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *Proc of AAAI*, volume 24.

Hogg, C.; Munoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proc of AAAI*, 950–956.

Höller, D. 2021. Translating totally ordered HTN planning problems to classical planning problems using regular approximation of context-free languages. In *Proc of ICAPS*, volume 31, 159–167.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Prof of ECAI*, 447–452.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the Expressivity of Planning Formalisms through the Comparison to Formal Languages. In *Proc of ICAPS*, 158–165.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2019. Hierarchical Planning in the IPC. *CoRR*, abs/1909.04405.

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 *Proc of AAAI*, 9883–9891. AAAI Press.

Howey, R.; and Long, D. 2003. VAL's progress: The automatic validation tool for PDDL2. 1 used in the international planning competition. In *Proc of the International Workshop on the International Planning Competition*, 28–37.

Ilghami, O.; Nau, D. S.; and Muñoz-Avila, H. 2006. Learning to Do HTN Planning. In *Proc of ICAPS*, 390–393.

Ilghami, O.; Nau, D. S.; Muñoz-Avila, H.; and Aha, D. W. 2002. CaMeL: Learning Method Preconditions for HTN Planning. In *Proc of ICAPS*, 131–142.

Jungnickel, D. 1999. The Greedy Algorithm. In *Graphs, Networks and Algorithms*, 129–153. Springer.

Karp, R. M. 1972. Reducibility Among Combinatorial Problems. In *Proc. of a symposium on the Complexity of Computer Computations*, 85–103.

Li, N.; Cushing, W.; Kambhampati, S.; and Yoon, S. 2014. Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences. *ACM Transactions on Intelligent Systems and Technology*, 5(2): 1–32.

Mourão, K.; Zettlemoyer, L. S.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Proc of UAI*, 614–623.

Nargesian, F.; and Ghassem-Sani, G. 2008. LHTNDT: Learn HTN Method Preconditions using Decision Tree. In *Proc of ICINCO-ICSO*, 60–65.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Muñoz-Avila, H.; Murdock, J. W.; Wu, D.; and Yaman, F. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2): 34–41.

Oncina, J.; and García, P. 1992. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis: Selected Papers from the IVth Spanish Symposium*, volume 1, 49–61. World Scientific.

Pellier, D.; and Fiorino, H. 2018. PDDL4J: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1): 143–176.

Pellier, D.; and Fiorino, H. 2020. Totally and Partially Ordered Hierarchical Planners in PDDL4J Library. *CoRR*, abs/2011.13297.

Xiaoa, Z.; Wan, H.; Zhuoa, H. H.; Herzigb, A.; Perrusselc, L.; and Chena, P. 2019. Learning HTN Methods with Preference from HTN Planning Instances. *Proc of HPlan workshop*, 31.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3): 107–143.

Zhuo, H. H.; Hu, D. H.; Hogg, C.; Yang, Q.; and Muñoz-Avila, H. 2009. Learning HTN Method Preconditions and Action Models from Partial Observations. In *Proc of IJCAI*, 1804–1810.

Zhuo, H. H.; Nguyen, T. A.; and Kambhampati, S. 2013. Refining Incomplete Planning Domain Models Through Plan Traces. In *Proc of IJCAI*, 2451–2458.

An Efficient HTN to STRIPS Encoding for Concurrent Plans

Nicolas Cavrel, Damien Pellier, Humbert Fiorino,
Univ. Grenoble Alpes - LIG
Grenoble, France
{nicolas.cavrel, damien.pellier, humber.fiorino}@univ-grenoble-alpes.fr

Abstract

The Hierarchical Task Network (HTN) formalism is used to express a wide variety of planning problems in terms of decompositions of tasks into subtaks. Many techniques have been proposed to solve such hierarchical planning problems. A particular technique is to encode hierarchical planning problems as classical STRIPS planning problems. One advantage of this technique is to benefit directly from the constant improvements made by STRIPS planners. However, there are still few effective and expressive encodings. In this paper, we present a new HTN to STRIPS encoding allowing to generate *concurrent* plans. We show experimentally that this encoding outperforms previous approaches on hierarchical IPC benchmarks.

Introduction

The Hierarchical Task Network (HTN) formalism (Erol, Hendler, and Nau 2003) is used to express a wide variety of planning problems in terms of decompositions of tasks into subtaks. HTN planning is used in many applications as, for instance, in task allocation for robot fleets (Milot et al. 2021), video games (Menif, Jacopin, and Cazenave 2014) or industrial contexts such as software deployment (Georgievski 2020). One possible explanation is that HTN formalism usually fits better for real-world applications and domain experts' mindset: a HTN planning problem is expressed as a set of tasks to achieve rather than an objective state to reach, and the "processes" achieving these tasks as *methods*, that is to say task decompositions into "simplier" subtasks. Despite the success of hierarchical planning and the recent revival of this planning technique (Bercher, Alford, and Höller 2019), there is comparatively less work on hierarchical planning than in classical STRIPS planning (Fikes and Nilsson 1971). The work of the planning community has been more focused on the development of techniques and heuristics for STRIPS planning, e.g., (Hoffmann 2000; Bryce and Kambhampati 2007; Hoffmann and Nebel 2011).

Many techniques are used to solve hierarchical planning problems. Some ad-hoc HTN solvers have been implemented (Bercher, Keen, and Biundo 2014; Erol 1996; Nau et al. 2003). Another approach consists in encoding HTN problems into SAT problems (Schreiber et al. 2019) or into constraint programming problems (Vidal and Geffner 2006).

One particular technique of encoding is to translate hierarchical planning problems as STRIPS problems. Encoding techniques benefit directly from the constant improvements of STRIPS planners. To our best knowledge, two HTN to STRIPS encodings have been published so far (Alford et al. 2016; Alford, Kuter, and Nau 2009) (very recent work have been published concurrently to this paper, which will not be studied here (Behnke et al. 2022)). However, they have some limitations on the type of problems they can address and only produce *sequential* plans. For instance, one of the best current encodings is (Alford et al. 2016). It translates any HTN problem into STRIPS, making it solvable by any STRIPS planner. However, this encoding has three downsides:

1. It depends on a *progression bound*, which is an integer parameter bounding the size of the task network, meaning that the maximum size of the task network has to be estimated before encoding the HTN problem into STRIPS.
2. The resulting solution plans are sequential, a single action being performed at each time. However, many real-world applications are intrinsically distributed, and need *concurrent* actions at each time.
3. The encoding generates an high number of actions in the translated STRIPS problem. This makes the actual grounding of the problem difficult. This is particularly true when the makespan increases, as the number of translated actions greatly expands with it.

The contribution of the paper is twofold: (1) we introduce a search procedure called CPFD (Concurrent Partial Forward Decomposition) to generate concurrent plans, and (2) propose an **encoding** of this search procedure into STRIPS actions.

The rest of this paper is as follows. Section 1 defines the problem statement. Section 2 presents the Concurrent Partial Forward Decomposition procedure (CPFD). Section 3 introduces the concrete STRIPS encoding of CPFD, called Concurrent Task Holders Decomposition encoding (CTHD). In the last section, we compare CTHD with HTN2STRIPS (Alford et al. 2016), which is the current state-of-the-art encoding from HTN to STRIPS.

Problem statement

STRIPS Planning Problems

A *STRIPS planning problem* is a tuple $P = (L, A, I, G)$ where L is a finite set of logical propositions, A is a finite set of actions, $I \subseteq L$ is the initial state, and $G \subseteq L$ is the goal.

An *action* a is a triplet $a = (pre(a), add(a), del(a))$ where $pre(a)$ is the action's *preconditions*, $add(a)$ is its positive *effects* and $del(a)$ its negative ones, each a set of propositions. Two actions (a, b) are *independent* iff $del(a) \cap (pre(b) \cup add(b)) = \emptyset$ and $del(b) \cap (pre(a) \cup add(a)) = \emptyset$. Note that action independence only depends on action definitions. In the following, for all $a \in A$, $nInd(a)$ will denote the set of actions $b \in A$ *dependent* with a.

A *state* s is a set of logical propositions. The result of applying an action a to state s is a state s' defined by the transition function $s' = \gamma(s, a) = (s - del(a)) \cup add(a)$ if $pre(a) \subseteq s$, and undefined otherwise. Let an *action layer* π be a set of pairwise independent actions, and $pre(\pi) = \bigcup_{a \in \pi} pre(a)$. $add(\pi)$ and $del(\pi)$ are defined in the same way. By extension $s' = \gamma(s, \pi) = (s - del(\pi)) \cup add(\pi)$ if $pre(\pi) \subseteq s$, and undefined otherwise. Note that the actions of π can be executed concurrently or in any sequential permutation and still yield exactly the same state s'.

A *layered plan* Π is a sequence of action layers $\langle \pi_1, \ldots, \pi_n \rangle$. Let $\gamma(s, \Pi) = \gamma(\gamma(s, \pi_1), \langle \pi_2, \ldots, \pi_n \rangle)$. π_i *precedes* π_j if $i < j$. Likewise $a_i \prec a_j$ if $a_i \in \pi_i$, $a_j \in \pi_j$, and π_i *precedes* π_j. A layered plan Π is a solution to a STRIPS planning problem $P = (L, A, I, G)$ iff $G \subseteq \gamma(s, \Pi)$ (see Fig. 1).

In the following, a conditional action will be used, we used the semantic defined by the ADL formalism (Pednault 1994). As a regular action, a conditional one is defined by a set of preconditions. Its effects however depend on a set of conditions. For each condition verified, the corresponding effect is applied. This action is used for simplicity reasons but can easily be converted into a set of non condition actions.

HTN Planning Problems

We build on STRIPS planning problem definition to define a *HTN planning problem* as a tuple $P = (L, \mathcal{T}, A, M, I, tn)$ where L is a finite set of logical propositions, $\mathcal{T}$ is a finite set of *tasks*, A is a finite set of actions, M is a finite set of methods, $I \subseteq L$ is the initial state and tn the initial task network. There are two kind of tasks: *primitive* tasks that can be resolved by a STRIPS action $a = (task(a), pre(a), add(a), del(a)) \in A$, and *compound* tasks, which can be recursively decomposed into either primitive or compound tasks by a method $m \in M$.

A *task network* is a tuple $tn = (T, \prec, \alpha)$ such that T is a finite set of tasks symbols, $\alpha : T \mapsto \mathcal{T}$ maps indexes to tasks in $\mathcal{T}$, and $\prec$ is a partial order over T representing precedence constraints: t *precedes* t' if $t \prec t'$, or equivalently $(t, t') \in \prec$. $\prec$ is transitive. A task $\alpha(t), t \in T$ is *trailing* if $\forall t' \in T$, $(t', t) \notin \prec$ (t has no predecessor in T). $trail(tn)$ will denote the set of trailing tasks in T. Symmetrically, a task $\alpha(t), t \in T$ is a *last task* if $\forall t' \in T, (t, t') \notin \prec$ (t has no successor in T).

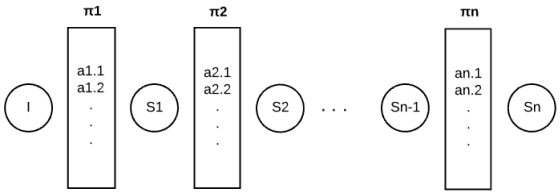


Figure 1: A layered plan with the successive states resulting from the action layer application. Circles represent states, and rectangles action layers.

A *method* is a tuple $m = (task(m), pre(m), tn(m))$ where $task(m)$ is the compound task *decomposed by the method* m, $pre(m)$ is the method's *preconditions* and $tn(m)$ is a task network. A method m is a *resolver* of a compound task τ if $task(m) = \tau$. Note that a given compound task can have various methods to resolve it: $task(m) = task(m') = \cdots = \tau$.

An action $a = (task(a), pre(a), add(a), del(a))$ can be applied to resolve a primitive task $\alpha(t)$ of the initial task network tn if t is trailing, $task(a) = \alpha(t)$ and $pre(a) \subseteq I$. The result is a new problem $P' = (L, \mathcal{T}, M, I', tn')$ where $I' = \gamma(I, a)$ and $tn' = (T \setminus \{t\}, \{(t', t'') \in \prec \mid t' \setminus = t\}, \alpha \setminus \{(t, \alpha(t))\})$. In a symmetrical manner, a method $m = (task(m), pre(m), tn(m))$ can be applied to resolve a compound task $\alpha(t)$ of the task network tn if t is trailing, $task(m) = \alpha(t)$ and $pre(m) \subseteq I$. The result of applying the method m with $tn(m) = (T_m, \prec_m, \alpha_m)$ is a new problem $P' = (L, \mathcal{T}, M, I, tn')$ where $tn' = (T', \prec', \alpha')$ and:

$$\begin{aligned} T' &= (T \setminus \{t\}) \cup T_m \\ \prec' &= \{(t', t'') \in \prec \mid t'' \neq t\} \cup \prec_m \cup \\ &\quad \{(t'', t') \in T_m \times T \mid (t, t') \in \prec\} \\ \alpha' &= \{(t', \alpha(t')), t' \in T \setminus \{t\}\} \cup \alpha_m \end{aligned}$$

In other words, in $\prec'$ we keep all the precedence constraints of $\prec$ that does not involve t, add all the precedence constraints in $\prec_m$, and propagate precedence transitivity between $\prec$ and $\prec_m$ through t.

Applying either an action a or a method m to resolve a task in a planning problem P is called a *progression*. If $t_p \in \mathcal{T}$ is a primitive task of P, resolving t_p by a is a *progression* denoted $P \mapsto^a_{t_p} P'$. Similarly, if t_c is a compound task of P, decomposing t_p using m is a *progression* denoted $P \mapsto^m_{t_c} P'$.

To conclude, a layered plan $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ is a solution for a HTN planning problem $P = (L, \mathcal{T}, A, M, I, tn)$ if (1) there exists a sequence of progressions that transforms P into $P' = (L, \mathcal{T}, M, I', (\emptyset, \prec'))$ (i.e. all the tasks of P have been resolved), and (2) $a_i \prec a_j \Leftrightarrow task(a_i) \prec' task(a_j)$ (i.e. action precedence constraints in the layered plan Π are equivalent to the primitive task precedence constraints in P'). In section 2, we show how CPFD builds a layered solution plan by applying progressions on P.

HTN to STRIPS Encoding Problems

The Hierarchical Task Network (HTN) formalism has been shown to be more expressive than STRIPS (Erol, Hendler, and Nau 2003). This means that any STRIPS problem can be formulated as a HTN problem but not the other way around. Therefore, the translation of a HTN problem into a STRIPS problem is not always possible. However, it has been proven by (Alford et al. 2016) that this translation is possible if the size of the solution task network can be bounded. Given a HTN problem and a sequential solution plan, the minimum (respectively maximum) bound is the smallest (respectively largest) number of tasks in any task network visited by the sequence of progressions carried out to find this solution plan.

In practice, not all problems have a maximum bound, but all solvable problems have a minimum bound. These bounds are not directly related to the length of a problem solutions, though the minimum progression bound is smaller than the optimal plan length[1].

Our encoding also assumes the bound existence. In addition, we make two other assumptions on the HTN problem to encode:

1. in the initial HTN problem, T is singleton. Otherwise, it is always possible to add a root dummy-task and a dummy-method to decompose it.
2. every method of the HTN problem has a task network with a *single last task*. If a method does not have it, a dummy-task with no successor is added to the task network.
3. methods have no preconditions. Otherwise, a trailing dummy action is added to the method task network with no effects and the method's preconditions.

These assumptions are made without loss of generality and will simplify the notations in the following. These assumptions were also made by the current state of the art encoding HTN2STRIPS (Alford et al. 2016).

Concurrent Partial-order Forward Decomposition

In this section, we propose a recursive procedure to solve HTN problems called CPFD (Concurrent Partial-order Forward Decomposition) and generate layered plans.

CPFD procedure is detailed in Alg. 1. CPDF is an adaptation of PFD (Partial-order Forward Decomposition) (Ghallab, Nau, and Traverso 2004) procedure to output layered plans (see Figure 1). A layered plan is solution of a HTN problem if actions resolve all the tasks of the task network, and if the ordering constraints of the actions in the layered plan satisfy the precedence constraints in this task network. CPFD tries to solve recursively the trailing tasks as in the PFD procedure. The difference lies on the resolution of the primitive tasks: while PFD adds actions to a sequential plan, CPFD adds them to layers of independent actions.

More precisely, CPFD takes as input four parameters: a HTN problem $P = (L, \mathcal{T}, A, M, I, (T, \prec))$, a layered plan, the index i of the current layer π_i and τ the set of primitive tasks resolved by the actions in π_i. The initial call of CPFD is CPFD($P, \Pi = [[]], i = 0, \tau = \emptyset$). At each recursive call, CPFD checks if the list of tasks T of the task network is empty, i.e., no more tasks have to be resolved. If this condition is satisfied, the layered plan Π is a solution to P and Π is returned. Otherwise, a task $t \in T$ is non deterministically selected among the trailing tasks (tasks without predecessors with respect to precedence constraints), and a resolver is non deterministically chosen. As in the PFD procedure, there are two ways to resolve t depending on whether t is primitive or compound.

Case 1. (Primitive task) The resolvers of t are actions a whose preconditions are satisfied in the current state I and that are independent of all the actions already planned in the current layer π_i. Two cases are possible: either t has no resolvers and the current layer π_i is empty, meaning no action can solve t in the current state I, and CPFD returns FAILURE. Or t has a resolver but this resolver is not an independent action in the current layer: then CPFD moves to the next layer by applying to the current state all the actions already committed in the current layer. The idea is that t could be resolved by an action in a next state concurrently with other independent actions. Obviously, if t has an independent resolver a, a is added to the current layer π_i and t is added to the set of resolved primitive tasks τ.

Case 2. (Compound task) CPFD computes all the methods resolving the compound task t, i.e., whose preconditions are satisfied in the current state I. If there is no method, then t cannot be solved, and CPFD returns FAILURE. Otherwise, CPFD non deterministically chooses a method m decomposing t, update the task set and the ordering constraints accordingly.

CPFD(P, Π, i, τ) is then called recursively until the tasks to solve in P are emptied ($T = \emptyset$, line 2) or a failure condition is met (line 9 and 24).

Theorem 1 *Concurrent HTN is sound and complete.*

Proof sketch (Soundness) All produced plans come from a progression of the initial task network, thus there is a sequence of task decomposition that produced the primitive tasks is the solution plan. Furthermore, since a primitive task can be added to a layer if the corresponding node is unconstrained, all tasks preceding the one added have been planned on previous layers. Thus the ordering constraints in $\prec$ are satisfied in the solution plan. Thus output plans are sounds.

Proof sketch (Completeness) We will show that CPFD is complete based on the demonstration that PFD is complete. Let $P = (L, \mathcal{T}, A, M, I, tn)$ be a HTN problem and $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ a layered solution plan of P. Let us show that there is a sequence of recursive calls of CPFD outputting Π. First, let us note that given a concurrent layer $\pi = \{a_1, \ldots, a_k\}$, any linearization of that layer ($\langle a_{\gamma(1)}, \ldots, a_{\gamma(k)} \rangle$ where γ is a permutation function of $\{1, \ldots, k\}$) is a sequence of actions which can be applied to

[1]For more details about the method to compute the progression bound of the solution task network see (Alford et al. 2016)

Algorithm 1: CPFD(P, Π, i, τ)

```
{P = (L, T, A, M, I, (T, ≺, α)) is the current problem}
if T = ∅ then return Π
toSolve ← trail(tn) \ τ
π_i ← get(Π, i)
nondeterministically choose t ∈ toSolve
if t is primitive then
    resolvers ← {a ∈ A | task(a) = t, pre(a) ⊆ I and (∀b ∈ π_i, a independent of b)}
    if resolvers = ∅ then
        if π_i = ∅ then
            return Failure
        else
            I ← γ(I, π_i)  // Apply the layer effects
            Π ← Π + []  // Add a new empty layer
            i ← i + 1  // Index of the new empty layer
            T ← T \ τ // Update the task network
            ≺' = {(t', t'') ∈ ≺ | t'' ≠ t} ∪ ≺_m
              ∪ {(t'', t') ∈ T_m × T | (t, t') ∈≺}
            τ ← ∅  // Reset the resolved tasks set
        end
    else
        nondeterministically choose a ∈ resolvers
        π_i ← π_i ∪ {a}  // Add a to the current layer
        τ ← τ ∪ {t}  // Add t to the resolved tasks set
    end
else
    {t is compound}
    resolvers ← {m ∈ M | task(m) = t}
    if resolvers = ∅ then return Failure
    nondeterministically choose m ∈ resolvers
    {m = (T_m, ≺_m)}
    T ← (T \ {t}) ∪ T_m  // Decomposing tn with m
    ≺ ← {(t', t'') ∈ ≺ | t'' ≠ t} ∪ ≺_m ∪
        {(t'', t') ∈ T_m × T | (t, t') ∈≺}
end
return CPFD(P, Π, i, τ)
```

the same states as π. This is due to the *mutual independence* property of the actions within a concurrent layer. From there, any sequential plan produced by linearizing every layer of Π (by taking any permutation function on the layers) is a sound plan that also solves P. Let us consider the linearization Π_l defined by the n permutation functions $\gamma_1, \ldots, \gamma_n$. Since PFD is a complete algorithm, there is a sequence of recursion of PFD which outputs Π_l. We will show that there is an analogous sequence of CPFD recursions outputting Π. At each recursion, PFD and CPFD either solve an unconstrained abstract task, or an unconstrained primitive task. While they solve abstract tasks the same way, PFD solves a primitive task by a adding an action resolving it to the head of the plan, meanwhile CPFD adds the action resolving the task to the concurrent layer at the head of the plan. If the task can not be resolved, PFD returns a Failure while CPFD tries to add a new concurrent layer to the plan. Thus, for each recursive PFD call resolving an abstract task, the analogous call of CPFD is to solve the same abstract task. Each recursive call of PFD resolving a primitive task is analogous to a CPFD call adding the action to the current concurrent layer. However, CPFD requires extra recursive calls compared to PFD: it needs to select and try to resolve an unsolvable task after each layer. In conclusion, the analogous sequence of recursion of CPFD is the one solving the same abstract and primitive tasks than PFD but with the insertion of recursive calls trying to solve an unsovlable task after resolving the last action of a layer of Π. Thus there is a sequence of CPFD recursions outputting Π and CPFD is complete.

Example of CPFD application

Let us consider a simple example of application of the CPFD algorithm. Let us consider two propositions p_1 and p_2, an initial state $I = \{p_2\}$ and $tn_0 = (T_0, \emptyset, \emptyset)$ the initial task network with a single compound task T_0.

T_0 can be decomposed by a single method m_0 into three unordered primitive tasks t_1, t_2, and t_3. The task t_1 can be resolved by an action $a(t_1) = (t_1, \emptyset, \{p_1\}, \emptyset)$, then $a(t_2) = (t_2, \{p_1, p_2\}, \emptyset, \emptyset)$ and $a(t_3) = (t_3, \emptyset, \emptyset, \{f_2\})$. The only solution plan is the sequential plan $\langle a(t_1), a(t_2), a(t_3)\rangle$.

When solving this problem, CPFD would first non deterministically choose a trailing task among the initial task network. There is only one, T_0. Then a resolver is chosen, there is only one m_0 which would be applied to result in a new (unordered) task network $tn = (\{t_1, t_2, t_3\}, \emptyset, \emptyset)$. On the next iteration, three tasks can be non deterministically chosen. Choosing t_2 or t_2 would lead to *Failure* since they have no valid resolver in the current state and the current layer is empty. Choosing t_1 would offer a single valid resolver $a(t_1)$ which would be added to the current layer. On the next iteration, either t_2 or t_3 can be chosen, and choosing either one would lead to no resolver valid in the current state. However this time the current layer in not empty, so instead of returning a *Failure*, CPFD would switch to the next layer by applying the effects of $a(t_1)$. On the next iteration, the updated state offers a valid resolver for t_2, which can be added to the next layer and so on... In the end, we obtain the sequential plan $\langle a(t_1), a(t_2), a(t_3)\rangle$

Now let us consider the same example but with $a(t_3) = (t_3, p_2, \emptyset, \emptyset)$. In that case, after decomposing T_0, both t_1 and t_3 have a valid resolver in the current state. These resolvers are independent, so CPFD can choose to resolve t_1, then t_3 (or t_3 then t_1) in the first layer. Then the only remaining task would be t_2 which have no resolver in the current state.

So CPFD switches to the next layer before adding it to the second layer. In that case we produced a concurrent plan $\langle\{a(t_1), a(t_3)\}, \{a(t_2)\}\rangle$.

Planning the planning: Translating HTN to STRIPS

We present in this section the concrete STRIPS encoding of the compound algorithm previously presented, called Concurrent Task Holders Decomposition encoding (CTHD).

Taskholder Encoding

The CPFD procedure described previously resolves recursively unconstrained primitive tasks and compound tasks by modifying the initial task network of the problem until the task network contains only an empty set of tasks. To encode this process, we need first to model a task network with STRIPS. To achieve this, we use the concept of *taskholder* introduced first by (Alford et al. 2016). A taskholder is a STRIPS object that will act as a container for a task. By way of extension, a task network is modeled as a stack of taskholders: static ordering relationship between taskholders defines in which order the taskholders can be allocated to the tasks during the CPFD procedure and fluent relationship define the ordering constraints between the tasks of the task network. The number of taskholders should be fixed before translating the HTN problem. Similarly to HTN2STRIPS, CTHD requires at least as many taskholders as there are tasks in the largest explored task network, thus the number of taskholders in CTHD can be estimated the same way as HTN2STRIPS estimates the number of taskholders.

In addition, we order the taskholders into a stack which defines the order in which the taskholders can be used.

The current layer of the *layered plan* under construction is represented by a set of propositions, each proposition denoting the fact that an action $a \in A$ is planned in the current layer or not.

To fix ideas, let us consider the planning problem defined as a task decomposition graph (see Figure 2), a graphical representation of the initial structures of this problem, i.e., taskholders and current layer, is given Figure 6.

Encoding as Actions

The dynamics of the CPFD procedure (Alg . 1) is defined by three types of STRIPS actions: (1) the first type of actions resolves an unconstrained compound task and update the current task network according to a method decomposition (2) the second type of actions resolves a primitive task and add an action to the current layer; and (3) the last type of actions is the one switching layer, making the algorithm build the next layer of the solution plan. The planning process will choose one action among the three types. Applying one of this action is equivalent to one recursive call of CPFD procedure. The planning process ends when there is no tasks left in the task network, i.e., when all taskholders are empty. These actions are defined as follows:

1. **Actions for resolving compound tasks:** These actions reflect the decomposition of a compound task into subtasks according to a method. It corresponds to the lines 27 to 30 of CPFD procedure. In order to apply these actions, the following must be verified:
   - The taskholder containing the decomposed task is un-

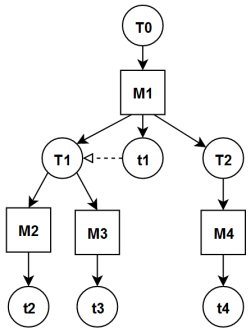


Figure 2: The task decomposition graph of $T0$. In this graph, each task node (represented with circles) is linked to the method resolving it (represented with squares). For instance, $T1$ can be decomposed into $t2$ by applying $M2$ or into $t3$ by applying $M3$. The ordering constraints are represented with the dotted arrows, so when decomposing $T0$ with $M1$, three subtasks are generated, $T1, t1$ and $T2$ where $t1$ must be planned before $T1$.

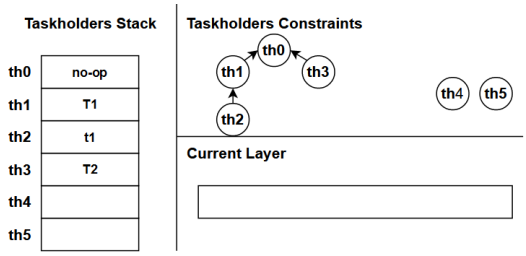


Figure 3: *T0* is decomposed into four tasks, two compound ones *T1* and *T2*, and two primitive ones *t1* an *no-op*. Since neither *T1*, *T2* or *t1* is a last task, a *no-op* action is inserted instead of *T0*. The constraint over the taskholders are represented on the top right graph: each directed edge represents a precedence constraint. So for instance, the task in $th2$ should planned before the one in $th1$.

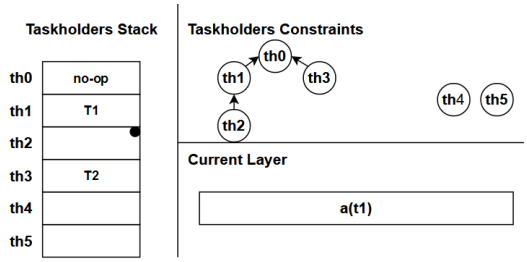


Figure 4: The second taskholder was unconstrained, and contains a primitive task. It is added to the plan step by removing the task from the taskholder, adding the action $a(t1)$ resolving $t1$ to the plan step, and marking the taskholder as resolved (represented by the black dot).

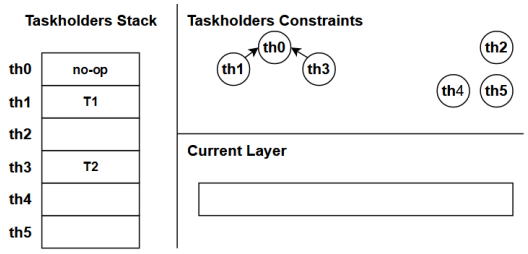


Figure 5: The plan step is terminated by emptying it, the constraints implied by the resolved taskholders are removed.

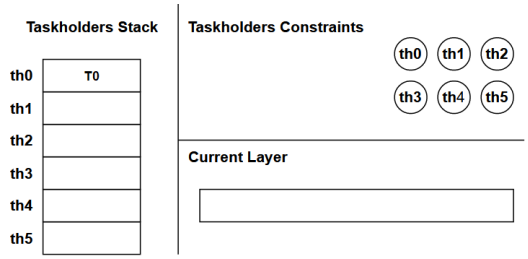


Figure 6: The problem is initialized by setting the initial task into the first taskholder.

constrained.

- There is enough taskholders remaining in the stack.

It works on our example as represented Figure 3. The substasks of the method M_1 decomposing T_0 are added to the stack of taskholders and the task T_0 is replaced by the last task of the current task network. As M_1 as no *last task*, the no-op action is used. Finally, the *ordering* constraints are set over the taskholders.

2. **Actions for resolving primitive tasks:** These actions resolve an unconstrained primitive task of the task network and add the resolver action $a \in A$ into the current layer (lines 21 to 23 of Alg. 1). In CPFD procedure, it translates to adding the action resolving a primitive task contained in an unconstrained taskholder to the current layer, removing the primitive task from the taskholder and marking the taskholder as resolved. These actions can be executed when the following conditions are verified:

   - The preconditions of the action are satisfied.
   - The taskholder containing the task is *unconstrained*.
   - All actions already planned at the plan step are *independent* with the task.

   An example of application is displayed on Figure 4. The taskholder $th2$ is unconstrained and contains a primitive task $t1$ that can be resolved by the action $a(t1)$. The task is resolved by adding $a(t1)$ to the current layer, $th2$ is emptied and marked as resolved.

   As in Alg. 1, the constraints implied by the resolved taskholder are not removed yet, they will when going to the next layer.

3. **Action for switching of plan layer:** This CTHD action corresponds to switching to the next layer. This action empty the *current layer*, setting up the next one in the solution plan. It also removes the constraint implying the resolved taskholders. This action can be applied in any situation and works as displayed on Figure 5. In this example, only $th2$ is marked as resolved. After the application of the action, the constraints implying $th2$ are removed, so the constraint between $th2$ and $th1$ is deleted, additionally $th2$ is unmarked as resolved and set as empty. Then the current layer is emptied by removing all actions in it.

Concurrent Taskholder Decomposition Encoding

In this section we present the STRIPS formulation of the translated problem. Similarly to HTN2STRIPS, our translation depends on an integer parameter denoted b representing the number of taskholders (i.e. the maximum number of tasks in the task network). We first define the predicates used to encode the problem, then we define the encoding of the three types of actions presented above. We end by presenting the initial state and goal state of the translated problem.

Let $P = (L, \mathcal{T}, A, M, I, tn)$ be a HTN problem and let $b \in \mathbb{N}$ the number of *taskholders*. CTHD encoding generates a STRIPS problem $CPFD(P, b) = (L \cup L', A', I \cup I', G')$.

The encoding generates the set propositions L' based on the following predicates:

- $(not\_constraint\ ?th1\ ?th2 - taskholder)$ represents the fluent constraints over the taskholders. The predicate is inverted for convenience, so when the proposition $(not\_constraint\ th1\ th2)$ is false, it means that the task contained in $th1$ must be planned before the task contained in $th2$.
- $(empty\ ?th - taskholder)$ represents the fact that a taskholder is empty. The proposition $(empty\ th)$ is true if the taskholder th does not contain a task.
- $(in\ ?t - task\ ?th - taskholder)$ is the predicate representing whether or not the task $?t \in \mathcal{T}$ is set in the parameter taskholder. The proposition $(in\ t\ th)$ is true if t is set in th.
- $(not\_planned\ ?a - action)$ is true if a is not planned in the current plan step.
- $(prec\_th\ ?th1\ ?th2 - taskholder)$ is a predicate defining the static relationship between the taskholders and defines the taskholders *stack*. So the proposition $(prec\_th\ th1\ th2)$ is true is $th1$ is *above* $th2$ in the stack. In the following, this order will be fixed and $\forall 0 \leq i, j < b$ the proposition $(prec\_th\ th_i\ th_j)$ will be true if and only if $i \leq j$.
- $(resolved\ ?th - taskholder)$ is a predicate representing either or not a taskholder have been resolved. So the proposition $(resolved\ th)$ is true if the taskholder th have been resolved.

The encoding generates the set of actions $A' = A_c \cup A_p \cup A_l$ where A_c is the set of actions for resolving compound tasks, A_p the set of actions for resolving primitive tasks and A_l the set of actions for switching of plan layer:

- **Actions for resolving compound tasks:**
  Let $m = (task(m), pre(m), tn(m))$ and $(task_1, task_2, \ldots, task_k)$ the subtasks in tn. We assume that $task_k$ is the *last task* of tn. For all methods $m \in M$ there is an action $a_m \in A_c$ with k parameters $(?th_1, ?th_2, \ldots, ?th_k)$ defined as follows:
  - $pre(a_m) = (in\ task(m)\ ?th_1) \wedge$
    $\bigwedge_{i=0}^{b-1}(not\_constraint\ th_i\ ?th_1) \wedge$
    $\bigwedge_{i=2}^{k}(prec\_th\ ?th_i\ ?th_{i+1}) \wedge$
    $\bigwedge_{i=2}^{k}(empty\ ?th_i)$
  - $add(a_m) = \bigwedge_{i=2}^{k}(in\ task_i\ ?th_i) \wedge (in\ task_k\ ?th_1)$
  - $del(a_m) = \bigwedge_{i=2}^{k}(empty\ ?th_i) \wedge$
    $\bigwedge_{task_i \prec task_j}(not\_constraint\ ?th_i\ ?th_j) \wedge$
    $\bigwedge_{i=2}^{k}(not\_constraint\ ?th_i\ ?th_0)$

  Note that the $k-1$ last taskholder parameters are required to be *ordered* according to the static relationship defined by the $prec\_th$ predicate. For instance, if three new taskholders are required to decompose the task in $th6$, $(th6\ th2\ th4\ th7)$ is a valid combination of parameters, while $(th6\ th2\ th5\ th3)$ is not.
- **Actions for resolving primitive tasks:**
  For all actions $a \in A$ there is an action $a_p \in A_p$ with one parameter: a taskholder containing p and denoted $?th$. The action is defined as follows:
  - $pre(a_p) = pre(a) \wedge (in\ task(a)\ ?th) \wedge$
    $\bigwedge_{i=0}^{b-1}(not\_constraint\ th_i\ ?th) \wedge$
    $\bigcup_{t \in nInd(p)}(not\_planned\ ?a)$
  - $add(a_p) = add(a) \wedge (empty\ ?th) \wedge (resolved\ ?th)$
  - $del(a_p) = del(a) \wedge (not\_planned\ ?a)$
- **Action for switching layer:**
  A_l is composed of one conditional action a_l with no parameter. This action has a non conditional part: emptying the current layer, and a conditional part: unconstraining and making available for a new use the resolved taskholders. It is defined as follows:
  - $pre(a_l) = \emptyset$
  - $add(a_l) = \bigwedge_{a \in A}(not\_planned\ ?a) \wedge$
    $\forall(?th - taskholder)$ when $(resolved\ ?th)$,
    $\bigwedge_{i=0}^{b-1}(not\_constraint\ ?th\ th_i)$
  - $del(a_l) = \emptyset$

Finally, let us define the encoding for the initial state and goal of the translated problem. The initial state is defined by setting the initial task into the first taskholder. The remaining taskholders are set as *empty* and ordered in a stack. As there is no constraint over the taskholder yet, all constraints predicate are initialized accordingly.

$I' = (in\ task_0\ th_0) \wedge \bigwedge_{1 \leq i < b}(empty\ th_i)$
$\wedge \bigwedge_{1 \leq i < j < b}(prec\_th\ th_i\ th_j) \wedge$
$\bigwedge_{0 \leq i,j < b}(not\_constraint\ th_i\ th_j) \wedge$
$\bigwedge_{a \in A}(not\_planned\ a)$

The problem is solved when all taskholders are empty, meaning that all task are planned. The goal state is defined accordingly.

$G' = \bigwedge_{i=0}^{p-1}(empty\ th_i)$

Differences and Similarities with HTN2STRIPS

If both CTHD and HTN2STRIPS encode HTN solving with taskholders, they aim at finding different kinds of solution plans. While CTHD aims at finding concurrent plans, HTN2STRIPS can only produce sequential ones. In the following, we will experimentally compare the two encodings but one has to keep in mind that CTHD produces plans with higher expressivity than the one produced by HTN2STRIPS.

In addition, while HTN2STRIPS taskholders are unordered, CTHD orders the taskholders into a *static stack* and imposes an order in which the taskholders parameters are used. The purpose of the stack is to reduce the number of valid operators in the translated problem: let us consider an example with four taskholders and a task T_1 set in the first taskholder $th1$. T_1 can be decomposed by a method m into four subtasks t_1, t_2, t_3 and t_4. The HTN2STRIPS translation, translates this method into 3! (equivalent) actions (one for each permutation of the three newly used taskholders). Meanwhile CTHD, by constraining the taskholder a fixed order, translates this method into a single action which corresponds to the HTN2STRIPS action where all taskholders are ordered. In the general case, if b is the progression bound, the number of translated actions generated from a method requiring k new taskholders is equal to the number of permutations of k elements among b for HTN2STRIPS. While the number of generated actions is equal to the number of *crescent* permutations for CTHD, which is much lower.

Experimentation and Results

The results of the experiments are displayed on Figure 7.

In this section we will present the experiments and results we used to demonstrate CTHD efficiency. We compare CTHD to the current best HTN to STRIPS translation HTN2STRIPS (Alford et al. 2016). This comparison will be made over IPC HTN benchmarks domains. These domains were chosen because they can express either totally ordered or partially ordered problems. Over each domain, the problems will be divided into totally ordered and partially ordered problems. While the solution plan of a totally ordered problem are necessarily sequential, the solution of a partially ordered problem can be concurrent.

Both HTN2STRIPS and CTHD can use the same *progression bound* to solve HTN problems. So in order to have a fair comparison, all problems were solved using the same (minimal) progression bound for both encodings. Note however that if there is no benefit for HTN2STRIPS to use a progression bound bigger that the minimal one, CTHD can explore bigger task networks and thus find solution plans with higher concurrency.

The comparison of HTN2STRIPS and CTHD will be made over five metrics:

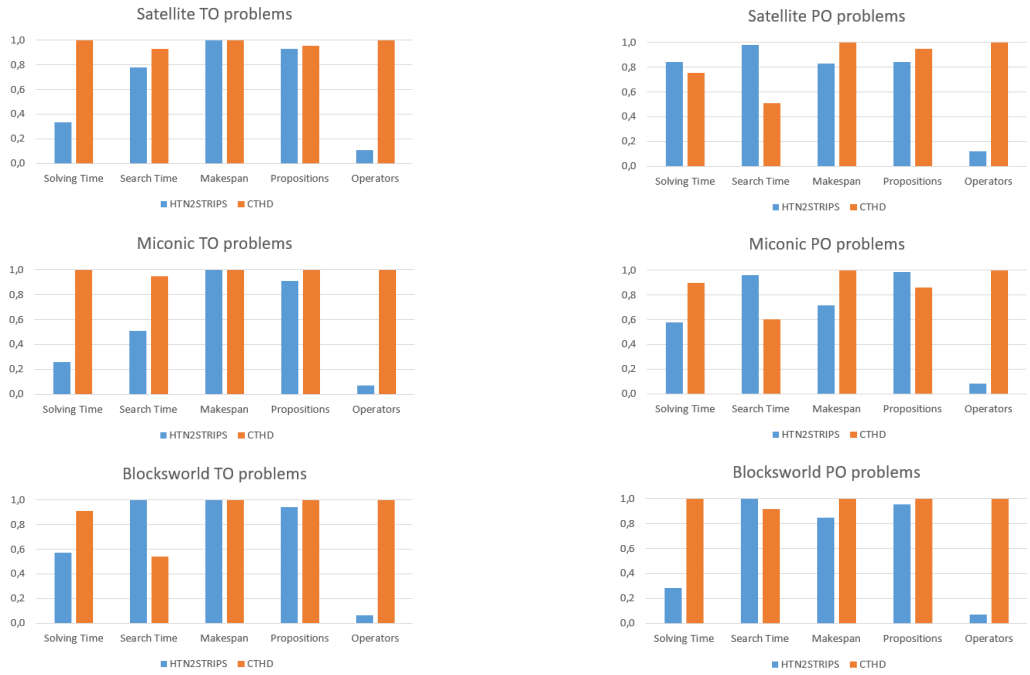


Figure 7: Results for the *Satellite*, *Miconic* and *Blocksworld* domains: TO (Totally ordered) and PO (Partially ordered)

- **Solving time:** This include the time spent to instantiate the problem and domain file and the time spent solving the instantiated problem.
- **Search time:** This corresponds solely to the time spent to solve the instantiated problem.
- **Solution makespan:** This corresponds to the "length" of the solution plan, meaning the number of actions within the plan for a sequential solution plan or the number of concurrent layers for a concurrent solution plan.
- **Number or proposition:** This corresponds to the number of proposition generated by instantiating the domain and problem file.
- **Number of operators:** This corresponds to the number of operators generated by instantiating the domain and problem file.

These five parameters will be evaluated by the IPC scoring metric, which is defined as follows:

$$IPC(k) = \frac{1}{|problems|} \sum_{i \in problems} \frac{min_{p \in planners}(cost(p))}{cost(k)}$$

We ran all experiments on a single core of a Intel Core i7-9850H CPU, using the Fast Downward library (Helmert 2011) with the Delfi 1 configuration (Katz et al. 2018), With a limit of 8GB of RAM over 600 seconds.

Over the three domains, CTHD consistently obtains a better score on the *operator* metric. This is due to the difference in scaling between the number of operators in the translated HTN2STRIPS problem and the CTHD one.

Proposition wise, both encodings are very similar. The discrepancies between domains can be explained by the number of concurrent actions within the domain: the more concurrency possible between actions, the more proposition are generated by HTN2STRIPS.

For the *Makespan* metric, on totally ordered problems HTN2STRIPS and CTHD obtained the same maximal score. This is due to the fact that since both encodings use the same minimal progression bound, they both find the shortest sequential plan. On the partially ordered problems, CTHD obtains a better score than HTN2STRIPS on every domain. Since CTHD can produce concurrent plans, on partially ordered problems it is able to fit several actions into a layer, thus producing plans with smaller makespan.

When it comes to *Search time* metric, the results are more mixed. Overall, HTN2STRIPS has a better *Search time* for the partially ordered problems but on the totally ordered ones, HTN2STRIPS outperforms CTHD only on the *Blocksworld* domain. These mixed results can be explained by the fact that a planner requires less steps to solve a HTN2STRIPS translated problem than a CTHD translated one: since HTN2STRIPS only produce sequential plans, going to the next layer is implied by resolving a primitive task. On the other hand, going to the next layer is a full solving step in a CTHD translated problem. However, we also saw that HTN2STRIPS generates more operators than CTHD, leading to higher *branching factor* for the planner solving the translated problem. In the end, there is a trade off between finding a shorter plan with a higher branching factor or finding larger plans with a lower branching factor.

Finally, the *Solving time* metric represents the total time spent by the planner to go from the parsing of the trans-

lated files to the solution plan. It is the sum of the time spent instantiating the files and the *Search time*. On this metric, CTHD obtains a better score than HTN2STRIPS on all domains except for the partially ordered *Satellite* one. However on all domains, CTHD improves its score going from *Search time* to *Solving time*. Once again, since CTHD requires much less operators, instantiating a CTHD translated domain file is much faster than a HTN2STRIPS one.

Overall, CTHD outperforms HTN2STRIPS both regarding the time spent to solve the translated problems and regarding the makespan of the solution plans.

Conclusion and future work

In this paper, we presented a new HTN procedure, CPFD, solving HTN problem with layered solution plan. We encoded this procedure as a STRIPS problem, thus producing a new HTN to STRIPS translation, CTHD. The translated problem, can be solved by any STRIPS planner. Then we showed experimentally that our translation outperforms the current best one HTN2STRIPS, both in terms of problem representation size, solving efficiency and quality of solution plans. This translation is a new way to solve HTN problems in a concurrent way, offering a new alternative to *plan-space* HTN algorithms. However, similarly to HTN2STRIPS, our translation still depends on an integer parameter the progression bound, which bounds the size of the explored task networks. In order to improve HTN to STRIPS translation, we feel that this last point is the one to focus on.

References

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016. Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problems. ICAPS'16, 20–28. ISBN 1577357574.

Alford, R.; Kuter, U.; and Nau, D. 2009. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. IJCAI'09, 1629–1634.

Behnke, G.; Pollitt, F.; HÃ¶ller, D.; Bercher, P.; and Alford, R. 2022. Making Translations to Classical Planning Competitive With Other HTN Planners. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022)*. AAAI Press.

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

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid Planning Heuristics Based on Task Decomposition Graphs. SoCS 2014, 35–43.

Bryce, D.; and Kambhampati, S. 2007. A Tutorial on Planning Graph Based Reachability Heuristics. *AI Magazine*, 28: 47–83.

Erol, K. 1996. Hierarchical task network planning: formalization, analysis, and implementation. In *Computer Science Departement, University of Maryland*, PhD. Thesis.

Erol, K.; Hendler, J.; and Nau, D. 2003. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence*, 18: 69–93.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3): 189–208.

Georgievski, I. 2020. HTN Planning Domain for Deployment of Cloud Applications. IPC'10, 34–36.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*.

Helmert, M. 2011. The Fast Downward Planning System. *CoRR*, abs/1109.6051.

Hoffmann, J. 2000. A Heuristic for Domain Independent Planning and Its Use in an Enforced Hill-Climbing Algorithm. In *ISMIS 2000*, volume 1932, 216–227.

Hoffmann, J.; and Nebel, B. 2011. The FF Planning System: Fast Plan Generation Through Heuristic Search. 14: 253–302.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online planner selection for cost-optimal planning. In *Ninth International Planning Competition (IPC-9)*, 55–62.

Menif, A.; Jacopin, E.; and Cazenave, T. 2014. SHPE: HTN Planning for Video Games. In *Third Workshop on Computer Games, CGW 2014*, 119–132. Prague, Czech Republic.

Milot, A.; Chauveau, E.; Lacroix, S.; and Lesire, C. 2021. Solving Hierarchical Auctions with HTN Planning. In *4th ICAPS workshop on Hierarchical Planning (HPlan)*. Guangzhou, China.

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

Pednault, E. 1994. ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, 4(5): 467–512.

Schreiber, D.; Pellier, D.; Fiorino, H.; and Balyo, T. 2019. Efficient SAT Encodings for Hierarchical Planning. ICAART 2019, 531–538. Prague, Czech Republic.

Vidal, V.; and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3): 298–335.

Chronicles for Representing Hierarchical Planning Problems with Time

Roland Godet,[1,2] **Arthur Bit-Monnot**[1]

[1] LAAS-CNRS, Université de Toulouse, INSA, CNRS, Toulouse, France
[2] ENS Paris-Saclay, Université Paris-Saclay, France
rgodet@laas.fr, abitmonnot@laas.fr

Abstract

In temporal planning, chronicles can be used to represent the predictive model of durative actions. Unlike the classical state-oriented representation, the usage of chronicles allows a rich temporal qualification of conditions and effects, beyond the mere start and end times of an action.

In this paper we propose an extension of the standard chronicle representation to support hierarchical problems. In particular, we show that the addition of temporally qualified subtasks to chronicles makes them suitable to represent not only primitive actions but also HTN methods.

We show how the set of solutions to a chronicle-based hierarchical problem can be quite naturally represented as a Constraint Satisfaction Problem (CSP). To associate semantics to this extended chronicle representation, we propose a set of rules that must hold for any solution to the hierarchical problem, specified as constraints on the associated CSP.

Introduction

In Artificial Intelligence, planning with Hierarchical Task Networks (HTNs) is an approach to automated planning where actions are hierarchically structured (Erol, Hendler, and Nau 1994; Höller et al. 2020) with compound tasks, which can be broken down into subtasks. This hierarchy of tasks allows the planning problem to be described at several levels, starting with more abstract tasks and ending with directly applicable primitive tasks.

In its simplest expression, an HTN (Erol, Hendler, and Nau 1994; Höller et al. 2020) planning problem consists of *(i)* an initial state, *(ii)* an initial task network describing the aim, *(iii)* a set of actions, *(iv)* a set of compound tasks, and *(v)* a set of methods.

A *state* defines the values of a set of *state variables*, each describing a specific attribute of the environment (e.g. the position of a truck).

A *task network* is a set of *tasks* and *constraints*. Each task describes a particular operation to be fulfilled (e.g. bring a package from A to B). It is composed of its name and a list of parameters, which may be variables or constants. There are two kinds of tasks: the *actions* (or *primitive* tasks), which can be directly executed, and the *compound* tasks, that the planner must refine into actions. The constraints might, e.g., restrict the value of some variables or the order of tasks.

An *action* consists of *(i)* a set of conditions over state variables, they characterize the set of states in which the action is applicable, and *(ii)* a set of effects on state variables, that update the state to reflect the consequences of the action.

A *method* is a pair $m = (t_c, tn)$, where t_c is a compound task and *tn* is a task network. It represents the fact that one way to perform t_c is to execute the tasks laid out in *tn*.

Given an initial state s_0, an initial task network *tn*, a set of actions, and a set of methods, a *plan* is a sequence of actions $\langle a_1, ..., a_n \rangle$. To produce a plan, a planner must systematically replace any compound task in the initial task network by the subtasks of a compatible method, repeating the process until only primitive tasks remain. Along with ordering, this approach defines the set of candidate plans as the ones that can be decomposed from the initial task network. A candidate plan is a solution of the planning problem if the corresponding action sequence is applicable in the initial state.

Approach Rather than this state-oriented view, another representation for planning follows a time-oriented view, as proposed with chronicles (Ghallab, Nau, and Traverso 2004; Bit-Monnot 2018) or timelines (Smith, Frank, and Jónsson 2000). Several works have considered the introduction of hierarchies for time-oriented planners, notably FAPE (Bit-Monnot et al. 2020) and CHIMP (Stock et al. 2015). The ANML language is also a very relevant proposal, but lacks clearly defined semantics (Smith, Frank, and Cushing 2007).

In this paper, we introduce *hierarchical chronicles*, an extension of the chronicles proposed by Bit-Monnot (2018) for generative planning. This is done by associating a chronicle with the task it achieves as well as the subtasks it requires. We show this to be sufficient for chronicles to represent HTN methods, in addition to HTN actions.

While conceptually simple, our formalization of the resulting problem as a CSP allows for advanced temporal features (e.g. intermediate conditions and effects) and closely relates to the time-oriented representation of scheduling problems which we hope will facilitate the application of scheduling techniques to hierarchical planning.

Hierarchical Chronicles

A *type* is defined by a set of values. It can be either a set of domain constants (e.g. the type *Truck* $= \{ R_1, R_2 \}$ defines two truck objects R_1 and R_2) or a discrete set of numeric

values. A type of particular interest for this formalization is the set of *timepoints* that we assume to be a discrete set of evenly spaced numeric values representing absolute times at which events can occur.[1] A decision *variable* is related to a type which defines its initial domain (i.e. possible values).

A *state variable* describes the progression of a specific attribute of the environment over time. It is generally parameterized by one or multiple domain objects. For instance $loc(R_1)$ denotes the evolution of the position of the truck R_1 over time. A state variable expression is often parameterized by variables, in which case the particular state variable it refers to depends on the value taken by its parameters, e.g., $loc(r)$ will correspond to $loc(R_1)$ or $loc(R_2)$ depending on the value taken by the variable r of type *Truck*.

A *task* denotes a particular operation to be fulfilled in the environment over time. It is generally parameterized by one or multiple domain objects. For instance $[5, 10]Go(R_1, L_1)$ denotes the operation of moving the truck R_1 to the location L_1 over the temporal interval $[5, 10]$. A task might be parameterized by variables, in which case the particular task it refers to depends on the value taken by its parameters.

A *chronicle* can be thought of as defining the requirements of a process in the planning problem, and in particular the process of executing an action or carrying out a method. A chronicle is a tuple $\mathcal{C} = (V, \tau, X, C, E, S)$ where:

- V is a set of **variables** appearing in the chronicle, partitioned into a set of temporal variables V_T whose domains are timepoints and a set of non-temporal variables V_O.
- τ is the **task achieved by the chronicle**. It is of the form $[s, e]task(x_1, \ldots, x_n)$ where s and e are temporal variables in V_T, $task(x_1, \ldots, x_n)$ is the parameterized task (with each $x_i \in V_O$). s and e respectively denote the start and end instants at which the chronicle is active, and we will refer to them as $start(\mathcal{C})$ and $end(\mathcal{C})$ respectively.
- X is a set of **constraints** over the variables in V.
- C is a set of **conditions**, Each condition is of the form $[s, e]var(p_1, \ldots, p_n) = v$ where s and e are variables in V_T, $var(p_1, \ldots, p_n)$ is a parameterized state variable (with each $p_i \in V_O$) and v is a variable in V_O. A condition states that the state variable $var(p_1, \ldots, p_n)$ must have the value v over the temporal interval $[s, e]$.
- E is a set of **effects**. Each effect is in the form of $[s, e]var(p_1, \ldots, p_n) \leftarrow v$ where s and e are variables in V_T, $var(p_1, \ldots, p_n)$ is a parameterized state variable (with each $p_i \in V_O$) and v is a variable in V_O. An effect states that the state variable $var(p_1, \ldots, p_n)$ takes the value v at time e. Over the temporal interval $]s, e[$, the state variable is transitioning from its previous value to v, and has an undetermined value.
- S is a set of **subtasks**. Each subtask has the form $[t_s, t_e]task(x_1, \ldots, x_n)$ where t_s and t_e are temporal variables in V_T and $task(x_1, \ldots, x_n)$ is a parameterized task (with each $x_j \in V_O$). It denotes a required task that must be achieved by another chronicle over $[t_s, t_e]$.

A planning problem defines a set of *chronicle templates* $\mathcal{T}$ where each template can be instantiated into a *chronicle instance* by substituting all variables in the template with fresh variables. We typically consider two types of chronicles templates: *action chronicles* that have effects but no subtasks, and *method chronicles* with subtasks but no effects.

Considering a method template *Deliver* $\in \mathcal{T}$ that delivers a package p from a position l_s to a position l_e with a truck r, it can be instantiated as the method chronicle $\mathcal{C}^1_{Deliver}$ where:

- $\tau = [t_s, t_e]$*Deliver*(p, l_e), i.e., this chronicle should result in delivering the package p to l_e over the temporal interval $[t_s, t_e]$.
- $C = \{\ [t_s, t_s]loc(p) = l_s, [t_s, t_s]loc(r) = l_s\ \}$, i.e., the package p and the truck r have to be at the starting location l_s at the beginning of the method.
- $E = \varnothing$, this is a method chronicle, with no direct effects.
- $S = \{\ [t^L_s, t^L_e]$ *Load*(p, r), $[t^M_s, t^M_e]$ *Move*(r, l_s, l_e), $[t^U_s, t^U_e]$ *Unload*$(p, r)\ \}$, i.e., to achieve this delivery action, the following tasks have to be done: loading the package in the truck, moving the truck from l_s to l_e, and unloading the package.
- $V_O = \{\ p, r, l_s, l_e\ \}$ are the parameters of the method (package, truck, start and end location) and $V_T = \{\ t_s, t_e, t^L_s, t^L_e, t^M_s, t^M_e, t^U_s, t^U_e\ \}$ where t_s, t_e are timepoints representing the start and the end of the method (from τ), and $t^L_s, \ldots, t^U_e$ are the corresponding start/end of subtasks.
- $X = \{\ t_e \leq t_s + 10, l_s \neq l_e, t^L_e \leq t^M_s, \ldots\ \}$, e.g., impose that the method should take no more than 10 units of time, the two locations must be different and the *Move* subtask has to be executed after the $Load$ subtask.

Considering the action template *Move* $\in \mathcal{T}$ that moves a truck r from l_s to l_e, it can be instantiated as a chronicle $\mathcal{C}_{Move}$ with the following effects: $E = \{\, [t_s, t_e]loc(r) \leftarrow l_e\,\}$, i.e., the truck r will be at the ending location l_e at the end of the action, but its position is unknown during the action execution. This chronicle $\mathcal{C}_{Move}$ achieves the eponymous task $[t_s, t_e]$*Move*(r, l_s, l_e).

We distinguish an *initial chronicle* $\mathcal{C}_0$ encoding the initial state as effects and the objectives of the planning problem as conditions and subtasks. It might also specify the anticipated evolution of the environment outside the planner's control, e.g., that a bus will pass at 6pm. This chronicle is the only one not associated to a meaningful task τ. For instance, the problem where the package P_1 is initially in location L_0 and must be brought to location L_1 or L_2 before time 50, can be described by the following initial chronicle $\mathcal{C}_0$:

- $V_O = \{\ l\ \}, V_T = \{\ t_s, t_e\ \}$
- $X = \{\ t < 50, l = L_1 \vee l = L_2\ \}$, constraints restricting the solution set.
- $C = \{\ [t_e, t_e]loc(R_1) = l\ \}$, specifying the goals.
- $E = \{\, [0, 0]loc(P_1) \leftarrow L_0, [0, 0]loc(R_1) \leftarrow L_0\,\}$, effects specifying the initial state.

[1] While this definition assumes a discrete time representation, it could be equally interpreted with a continuous time representation. Also note that discrete time is no less expressive when instantaneous changes are forbidden, as common in temporal planning in general and PDDL in particular (Cushing 2012). In general however the computational complexity might differ between a discrete and a continuous time representation.

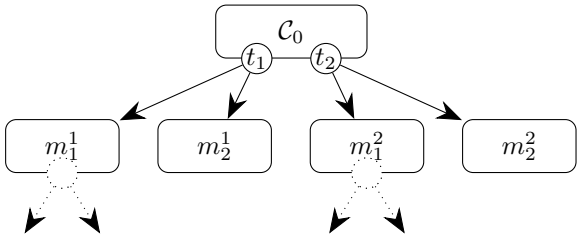


Figure 1: Decomposition graph resulting from the expansion of two tasks $t_1 : t(x)$ and $t_2 : t(y)$

- $S = \{\ [t_s, t_e]\textit{Deliver}(P_1, l)\ \}$, subtasks specifying specific tasks to be achieved by a solution plan.

A planning problem is the association $(\mathcal{C}_0, \mathcal{T})$ of an initial chronicle, defining the initial state and objectives, with a set of chronicle templates which can be instantiated, defining usable actions and methods.

Planning Problem as a CSP

Problem Instantiation

At this point we have defined the initial chronicle (representing the problem) and a set of chronicle templates (representing possible actions and methods). We now introduce the procedure to build a set of chronicle instances Π that can be used to represent a solution.

We initially set $\Pi = \{\ \mathcal{C}_0\ \}$, i.e., limited to the initial chronicle. Suppose the initial chronicle $\mathcal{C}_0$ contains two subtasks $t_1 : t(x)$ and $t_2 : t(y)$ and that the task $t(\cdot)$ can be achieved by one of two methods m_1 and m_2 (i.e. the task of m_1/m_2 are of the form $[\cdot, \cdot]t(\cdot)$). As the first task t_1 might be achieved by either m_1 or m_2, we add two fresh instantiations m_1^1 and m_2^1 to Π. We record the motivation for the introduction of both chronicles in a lookup-table $\mathit{decomposes}(m_1^1) \leftarrow t_1$ and $\mathit{decomposes}(m_2^1) \leftarrow t_1$. Similarly, we introduce two distinct chronicle instances m_1^2 and m_2^2 to represent the possible refinements of t_2. This process can be seen as creating a decomposition graph such as the one in Figure 1.

We refer to this process as *chronicle expansion*. More formally, for each subtask t of a chronicle instance $\mathcal{C} \in \Pi$, and each chronicle template $\mathcal{T}_i \in \mathcal{T}$ whose task is unifiable with t, we instantiate a new chronicle instance $\mathcal{T}_i^k$ and add it to Π. Each instantiated chronicle is uniquely associated to the task it was introduced to decompose with the $\mathit{decomposes}$ lookup table, effectively defining a decomposition tree. Any chronicle instance added to Π as a result of expansion will need to be expanded itself, making this a recursive process. In the case of cyclic HTN problems, the depth of resulting decomposition tree might not be bounded. For practical purpose – e.g. to encode the problem in a CSP solver – one might decide to bound the depth of the tree to obtain a finite number of chronicles.

Of course, it is not the case that each chronicle instance will be part of the solution plan. In Figure 1, the m_1^1 and m_1^2 chronicle instances are mutually exclusive as only one method can be used to decompose the t_1 task. To capture this fact, we associate each a chronicle instance $\mathcal{C} \in \Pi$ to a boolean variable $\textit{present}(\mathcal{C})$ that is true ($\top$) if $\mathcal{C}$ is present in the solution plan and false ($\bot$) otherwise. Note that the initial chronicle must always be present, so $\textit{present}(\mathcal{C}_0)$ is always true.

With this process, we have created a set of chronicle instances Π, representing all possible methods and actions that might appear in the plan. Each such chronicle instance $\mathcal{C} \in \Pi$ is associated with a boolean variable $\textit{present}(\mathcal{C})$ that represents whether it is part of a solution, and a task $\mathit{decomposes}(\mathcal{C})$ that it might decompose. We now ought to define what are the constraints that must hold for this set of optional chronicle instances to form a solution to the original planning problem. Before doing so, we introduce a synthetic representation for conditions, effects and tasks that will allow us to specify their behavior independently of the context in which they appear.

Core structures

Considering a planning problem Π, the core structures to express the constraints it must fulfill are described here.

Condition Token Given a chronicle instance $\mathcal{C} \in \Pi$, each condition in $\mathcal{C}$ is associated to a *condition token*:

$$\textit{present}(\mathcal{C}) : [s, e]var(p_1, \ldots, p_n) = v$$

This token states that, if $\mathcal{C}$ is present in the solution plan ($\textit{present}(\mathcal{C}) = \top$), then the state variable $var(p_1, \ldots, p_n)$ must have the value v over the temporal interval $[s, e]$. The set of condition tokens in Π is denoted C_Π.

Effect Token Given a chronicle instance $\mathcal{C} \in \Pi$, each effect in $\mathcal{C}$ is associated to an *effect token*:

$$\textit{present}(\mathcal{C}) : [s, e, t]var(p_1, \ldots, p_n) \leftarrow v$$

Note the new temporal variable $t \in V_T$. This token states that, if $\mathcal{C}$ is present in the solution plan ($\textit{present}(\mathcal{C}) = \top$), then the state variable $var(p_1, \ldots, p_n)$ is undefined over the temporal interval $]s, e[$ and has the value v over the temporal interval $[e, t]$. The expansion with the new timepoint t allows us to encode a minimal persistence of the effect until a later time t. The set of effect tokens in Π is denoted E_Π.

Task Token Given a chronicle instance $\mathcal{C} \in \Pi$, each subtask in $\mathcal{C}$ is associated to a *task token*:

$$\textit{present}(\mathcal{C}) : [s, e]task(x_1, \ldots, x_n)$$

This token states that, if $\mathcal{C}$ is present in the solution plan ($\textit{present}(\mathcal{C}) = \top$), then the task $task(x_1, \ldots, x_n)$ must be achieved over the temporal interval $[s, e]$. The set of task tokens in Π is denoted Γ_Π.

Token Characteristics Effect tokens here address the evolution of state variables over time. Each (present) effect token forces another value to its state variable, which is compelled to be maintained within a given temporal interval, thus encoding the state evolution. Condition tokens put requirements on the state evolution by imposing a state variable to have a given value over a temporal interval.

Each token (condition, effect, and task) is present in the solution plan if and only if its associated chronicle instance

is present in the solution plan. Given a token (condition, effect, or task) ω from a chronicle instance $\mathcal{C} \in \Pi$, we have $\textit{present}(\omega) = \textit{present}(\mathcal{C})$.

Constraints

Considering a planning problem Π, the constraints used to encode the consistency of a plan are described here.[2]

Coherence Constraint A state variable cannot take several values at the same time. This implies that for two distinct effect tokens ε and ε' in E_Π to be *coherent*, they may not concurrently impose a value or transition to the same state variable.

$$\text{Given } \begin{cases} \varepsilon = \langle p : [s, e, t] var(p_1, \dots, p_n) \leftarrow v \rangle \in E_\Pi \\ \varepsilon' = \langle p' : [s', e', t'] var(p'_1, \dots, p'_n) \leftarrow v' \rangle \in E_\Pi \end{cases}$$

the constraint $\textit{coherent}(\varepsilon, \varepsilon')$ is defined as:

$$p \wedge p' \implies t \leq s' \vee t' \leq s \vee p_1 \neq p'_1 \vee \dots \vee p_n \neq p'_n$$

By forcing two effect tokens to be non overlapping (over the presence p, time $[s, t]$ and the state variable $var(p_1, \dots, p_n)$ dimensions), this constraint ensures that a state variable will be given at most one value at any timepoint.

Support Constraint A condition token $\beta \in C_\Pi$ is said *supported by* an effect token $\varepsilon' \in E_\Pi$ if this effect establishes the value required by β and this value persists for the span of β.

$$\text{Given } \begin{cases} \beta = \langle p : [s, e] var(p_1, \dots, p_n) = v \rangle \in C_\Pi \\ \varepsilon' = \langle p' : [s', e', t'] var(p'_1, \dots, p'_n) \leftarrow v' \rangle \in E_\Pi \end{cases}$$

the constraint $\textit{supported-by}(\beta, \varepsilon')$ is defined by:

$$p' \wedge e' \leq s \wedge e \leq t' \wedge p_1 = p'_1 \wedge \dots \wedge p_n = p'_n \wedge v = v'$$

A condition token $\beta \in C_\Pi$ is said *supported* if it is supported by at least one effect token in E_Π. More formally, the constraint $\textit{supported}(\beta)$ is defined by:

$$\textit{present}(\beta) \implies \bigvee_{\varepsilon \in E_\Pi} \textit{supported-by}(\beta, \varepsilon)$$

Refinement Constraint A task token $\gamma \in \Gamma_\Pi$ is said *refined by* a chronicle instance $\mathcal{C} \in \Pi$ if *(i)* $\mathcal{C}$ was introduced as a potential decomposition of γ, *(ii)* it is part of the solution and *(iii)* the task it achieves is identical to the one of γ. Given

$$\gamma = \langle p : [s, e] task(x_1, \dots, x_n) \rangle \in \Gamma_\Pi$$

and $\mathcal{C} \in \Pi$ whose achieved task is of the form,

$$task(\mathcal{C}) = \langle [s', e'] task(x'_1, \dots, x'_n) \rangle$$

the constraint $\textit{refined-by}(\gamma, \mathcal{C})$ is defined by:

$$p \wedge \textit{decomposes}(\mathcal{C}) = \textit{id}(\gamma) \wedge s = s' \wedge e = e' \wedge x_1 = x'_1 \wedge \dots \wedge x_n = x'_n$$

where $id(\gamma)$ uniquely identifies the task associated to γ.

A task token $\gamma \in \Gamma_\Pi$ is said *refined* if it is refined by a chronicle instance in Π. More formally, the constraint $\textit{refined}(\gamma)$ is defined by:

$$\textit{present}(\gamma) \implies \bigvee_{\mathcal{C} \in \Gamma_\Pi} \textit{refined-by}(\gamma, \mathcal{C})$$

[2] Note that the coherence and support constraints are identical to the ones of Bit-Monnot (2018) and repeated here for completeness.

Motivation constraint For an HTN problem it is normally the case that a method or action is only allowed to be part of the solution if it derives from the initial task network. In a sense, it means that the presence of a chronicle must be motivated by the achievement of a task higher up in the hierarchy.

Consider a chronicle instance $\mathcal{C} \in \Pi$ that was introduced to decompose a task γ (i.e. $\textit{decomposes}(\mathcal{C}) = id(\gamma)$). Then the $\textit{motivated}(\mathcal{C})$ constraint is expressed as:

$$\textit{present}(\mathcal{C}) \implies \textit{present}(\gamma) \wedge \textit{refined-by}(\gamma, \mathcal{C}) \bigwedge_{\mathcal{C}' \in \textit{siblings}(\mathcal{C})} \neg \textit{present}(\mathcal{C}')$$

where $\textit{siblings}(\mathcal{C})$ is the set of all other chronicle instances $\mathcal{C}'$ that were introduced as a potential decomposition of the same task γ. This constraint effectively enforces that, if present, a chronicle instance uniquely achieves a task higher-up in the hierarchy.

Internal Chronicle Consistency Considering a chronicle $\mathcal{C} \in \Pi$, all its constraints X must be verified if $\mathcal{C}$ is part of the solution. It is represented by the constraint $\textit{consistent}(\mathcal{C})$:

$$\textit{present}(\mathcal{C}) \implies \bigwedge_{x \in X} x$$

Any requirement regarding a chronicle structure should be explicitly or implicitly encoded in the set X of chronicle constraints. In particular, a common requirement for hierarchical problems is that a method spans exactly the same time interval as its subtasks, i.e., that for any method chronicle $\mathcal{C}$ with a non-empty set of subtasks $subtasks(\mathcal{C})$:

$$start(\mathcal{C}) = \min_{st \in subtasks(\mathcal{C})} start(st)$$
$$end(\mathcal{C}) = \max_{st \in subtasks(\mathcal{C})} end(st)$$

Likewise, a method chronicle with no subtasks should be instantaneous (i.e. $start(\mathcal{C}) = end(\mathcal{C})$).

Formulation as a CSP Finally, a planning problem Π can be encoded as a CSP with variables V_Π and constraints X_Π where:

$$\begin{aligned} V_\Pi &= \{ V \mid (V, \tau, X, C, E, S) \in \Pi \} \\ &\cup \{ \textit{present}(\mathcal{C}) \mid \mathcal{C} \in \Pi \} \\ X_\Pi &= \{ \textit{coherent}(\varepsilon, \varepsilon') \mid (\varepsilon, \varepsilon') \in E_\Pi^2, \varepsilon \neq \varepsilon' \} \\ &\cup \{ \textit{supported}(\beta) \mid \beta \in C_\Pi \} \\ &\cup \{ \textit{refined}(\gamma) \mid \gamma \in \Gamma_\Pi \} \\ &\cup \{ \textit{motivated}(\mathcal{C}) \mid \mathcal{C} \in (\Pi \setminus \{ \mathcal{C}_0 \}) \} \\ &\cup \{ \textit{consistent}(\mathcal{C}) \mid \mathcal{C} \in \Pi \} \end{aligned}$$

Solution and Plan Extraction A solution to a hierarchical planning problem Π is an assignment to all variables in V_Π that satisfies all constraints in X_Π. The decision process involved in building the assignment will effectively impose the presence of methods and actions (through presence variables) as well as their instantiation (through parameter variables) and orderings (through temporal variables). We say that a chronicle $\mathcal{C} \in \Pi$ is present in the solution if its presence variable $\textit{present}(\mathcal{C})$ evaluates to true given the assignment.

From a solution assignment, it is straightforward to extract a solution plan: any action chronicle $\mathcal{C} \in \Pi$ present in the solution corresponds to an action in the plan. The value of its parameters as well as start and end times are given by the value of the corresponding variables in the assignment.

Likewise, the assignment encodes a full decomposition from the initial task network into the solution plan as the *refinement* and *motivation* constraints ensure that, for any subtask of a chronicle present in the solution, there is exactly one refining chronicle (action or method) present in the solution.

It should be noted that setting the presence variable of a chronicle to false has the same effect as not including it in the set Π of chronicles considered in the CSP. This suggests that bounding the depth of a decomposition tree should be interpreted as a decision restricting the set of solution. To maintain the completeness of a decision procedure, this bound should be reconsidered (i.e. increased) when it is shown that no solution exists within a given depth.

Discussion and Related Work

To the best of our knowledge all existing temporal HTN planners, including FAPE (Bit-Monnot et al. 2020), CHIMP (Stock et al. 2015) and SIADEX (Castillo et al. 2006), interpret HTN planning as a constructive process: they start from an initial task network that is iteratively expanded into a solution, each expansion bringing its own new subtasks, actions and additional constraints. This proposal adopts a different interpretation of hierarchical planning as a constraint satisfaction problem. While the two interpretations remain compatible, a key feature of the CSP interpretation is its proximity with the representation of scheduling problems. In particular, the notion of optional time intervals in scheduling solvers such as CPOptimizer (Laborie et al. 2018) brought many modeling capabilities that have a clear mapping with the structure of HTN problem (e.g. the *alternative* and *span* constraints of Laborie and Rogerie, 2008).

The closest formalism is the one used by FAPE (Bit-Monnot et al. 2020) that supports both hierarchical and generative planning. In FAPE this is allowed by annotating some actions as *task-dependent*, meaning that they can only be introduced as a refinement of an existing task. Unlike FAPE, this proposal focuses on pure hierarchical (HTN-like) planning allowing to remove this distinction. The addition of explicit temporal variables representing the end of the persistence of an effect is a subtle change that avoids an explicit handling of *causal-links* and *threats* (threats being particularly problematic for scalability as they might involve any combination of two effects and one condition, leading to a cubic number of constraints).

The notion of tokens (including the effect persistence) is inspired by the homonymous tokens of timeline-based planners such as Europa (Barreiro et al. 2012) or CHIMP (Stock et al. 2015). Unlike timeline-based planners however, we do maintain a distinction between conditions and effects and use an action-centric formalism.

Several hierarchical planners such as SIADEX (Castillo et al. 2006) or SHOP2 extensions (Goldman 2006) have enabled temporal features in a state-progression setting by timestamping states. Relationships with these state-oriented representations are less obvious as this paper adopts a time-oriented view where the evolution of each state variable is handled independently of the others.

Conclusion

In this paper, we introduced a constraint-based representation for hierarchical planning under temporal constraints. This encoding assumes a time-oriented view and a fully lifted representation. Its foundation on previous chronicle approaches allows leveraging their temporal expressiveness with a very limited increase in complexity.

While the paper is restricted to the discussion of the planning formalism, our current work is focused on the exploitation of this formalism and associated encoding in a constraint-based planner that leverages scheduling techniques.

References

Barreiro, J.; Boyce, M. E.; Do, M. B.; Frank, J. D.; Iatauro, M.; Kichkaylo, T.; Morris, P. H.; Ong, J. C.; Remolina, E.; Smith, T. B.; and Smith, D. E. 2012. EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization.

Bit-Monnot, A. 2018. A Constraint-Based Encoding for Domain-Independent Temporal Planning. *CP*.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. *ArXiv*, abs/2010.13121.

Castillo, L.; Fdez-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2006. Temporal Enhancements of an HTN Planner. In Marín, R.; Onaindía, E.; Bugarín, A.; and Santos, J., eds., *Current Topics in Artificial Intelligence*, 429–438.

Cushing, W. A. 2012. *When is Temporal Planning Really Temporal?* Ph.D. thesis, Arizona State University.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: Complexity and Expressivity. *AAAI*, 2.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Goldman, R. P. 2006. Durative Planning in HTNs. In *ICAPS*.

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. *AAAI*.

Laborie, P.; and Rogerie, J. 2008. Reasoning with Conditional Time-Intervals. In *FLAIRS Conference*.

Laborie, P.; Rogerie, J.; Shaw, P.; and Vilím, P. 2018. IBM ILOG CP Optimizer for Scheduling. *Constraints*.

Smith, D. E.; Frank, J.; and Cushing, W. 2007. The ANML Language.

Smith, D. E.; Frank, J. D.; and Jónsson, A. K. 2000. Bridging the gap between planning and scheduling. *The Knowledge Engineering Review*, 15: 47 – 83.

Stock, S.; Mansouri, M.; Pecora, F.; and Hertzberg, J. 2015. Hierarchical Hybrid Planning in a Mobile Service Robot. In *KI*.

Exploiting Solution Order Graphs and Path Decomposition Trees for More Efficient HTN Plan Verification via SAT Solving

Songtuan Lin[1], **Gregor Behnke**[2], **Pascal Bercher**[1]

[1] School of Computing, The Australian National University, Canberra, Australia
[2] ILLC, University of Amsterdam, Amsterdam, The Netherlands
{songtuan.lin, pascal.bercher}@anu.edu.au, g.behnke@uva.nl

Abstract

The task of plan verification is to decide whether a given plan is a solution to a planning problem. In this paper, we study the plan verification problem in the context of Hierarchical Task Network (HTN) planning. Concretely, we will develop a new SAT-based approach via exploiting the data structures *solution order graphs* and *path decomposition trees* employed by the state-of-the-art SAT-based HTN planner which transforms an HTN plan verification problem into a SAT formula. Additionally, for the purpose of completeness, we will also reimplement the old SAT-based plan verifier within an outdated planning system called PANDA$_3$ and integrate it into the new version called PANDA$_\pi$.

Introduction

Plan verification is the task of deciding whether a given plan is a solution to a planning problem. Research over the plan verification problem has drawn increasing attention in the last few years for its potential usages in numerous applications, e.g., in mixed initiative planning (see the work by Behnke, Höller, and Biundo (2017) for more details) and in International Planning Competition where a plan verifier is used to validate plans produced by participated planners.

We consider the plan verification problem in the context of HTN planning, which is a hierarchical approach to plan in which so-called abstract tasks are kept being refined until primitive ones (i.e., actions) are obtained. The HTN plan verification problem has been proved to be NP-complete (Behnke, Höller, and Biundo 2015; Bercher et al. 2016), and there exist three HTN plan verification approaches, namely, the SAT-based approach (Behnke, Höller, and Biundo 2017), the parsing-based approach (Barták, Maillard, and Cardoso 2018; Barták et al. 2020, 2021), and the planning-based approach (Höller et al. 2022) which transform a plan verification problem into a SAT problem, a language parsing problem, and an HTN planning problem, respectively. In this paper, we will develop a new SAT-based plan verification approach exploiting two data structures employed in the state-of-the-art SAT-based HTN planner (Behnke, Höller, and Biundo 2018, 2019a).

Specifically, we will adapt *solution order graphs* (SOGs) and *path decomposition trees* (PDTs) which are two data structures for formatting and storing refinement processes in an HTN planning problem. The core aspect of HTN plan verification, which is similar to solving an HTN planning problem, is to find a refinement process except that in plan verification, we demand that the refinement process must result in the given plan. These two data structures have been shown to be efficient in solving an HTN planning problem, but they are not exploited by the old SAT-based approach. Hence, we will adapt these data structures in our new SAT-based approach to see whether the performance can be improved. In order to distinguish our new SAT-based approach from the existing one, we call it the SOG-based approach and the existing one the DT-based approach where the term 'DT' refers to decomposition trees which we will introduce later on, and it is the core of the existing SAT approach. We emphasize 'SOG' because the majority of this paper aims to explain how to exploit SOGs in plan verification.

Apart from developing the new SOG-based plan verification approach, we will also reimplement the (old) SAT-based verifier. The old verifier is a part of an outdated HTN planning system called PANDA$_3$, which is written in JAVA and is now deprecated. Recently, a new version of PANDA called PANDA$_\pi$ has been developed which is written in C++. Thus, we would also like to rewrite the old verifier in C++ and integrate it into PANDA$_\pi$ for the purpose of completeness.

HTN Formalism

Before explaining the SOG-based approach, we first introduce the HTN formalism employed in the paper, which is an adoption of the one by Bercher, Alford, and Höller (2019). We start with the concept of task networks.

Definition 1. A task network tn is a tuple $(T, \prec, \alpha)$ where T is a set of task identifiers, $\prec \subseteq T \times T$ specifies the partial order defined over T, and α is a function that maps a task identifier to a task name.

Two task networks $tn = (T, \prec, \alpha)$ and $tn' = (T', \prec', \alpha')$ are said to be isomorphic, written $tn \cong tn'$, *iff* there exists a one-to-one mapping $\varphi : T \to T'$ such that for all $t \in T$, $\alpha(t) = \alpha'(\varphi(t))$, and for all $t_1, t_2 \in T$, if $(t_1, t_2) \in \prec$, $(\varphi(t_1), \varphi(t_2)) \in \prec'$.

Given a task network tn, the notations $T(tn)$, $\prec(tn)$, and $\alpha(tn)$ refer to the task identifier set, the partial order, and the identifier-name mapping function of tn, respectively. For convenience, we also define a restriction operation.

Definition 2. Let D and V be two arbitrary sets, $R \subseteq D \times D$ be a relation, $f : D \to V$ be a function and tn be a task network. The restrictions of R and f to some set X are defined by

- $R|_X = R \cap (X \times X)$
- $f|_X = f \cap (X \times V)$
- $tn|_X = (T(tn) \cap X, \prec(tn)|_X, \alpha(tn)|_X)$

Task names are further categorized as being primitive and compound. A primitive task name p, also called an action, is mapped to its precondition, add, and delete list by a function δ written $\delta(p) = (prec, add, del)$, where add and del are called the effects of p. On the other hand, a compound task name c can be refined (decomposed) into a task network tn by some method $m = (c, tn)$.

Definition 3. Let $tn = (T, \prec, \alpha)$ be a task network, $t \in T$ be a task identifier, c be a compound task name with $(t, c) \in \alpha$, and $m = (c, tn_m)$ be a method. We say m decomposes tn into another task network $tn' = (T', \prec', \alpha')$, written $tn \to_m tn'$, if and only if there exists a task network $tn'_m = (T_m, \prec_m, \alpha_m)$ with $tn'_m \cong tn_m$ such that

- $T' = (T \backslash \{t\}) \cup T_m$.
- $\prec' = (\prec \cup \prec_m \cup \prec_X)|_{T'}$ with $\prec_X = \{(t_1, t_2) \,|\, (t_1, t) \in \prec, t_2 \in T_m\} \cup \{(t_2, t_1) \,|\, (t, t_1) \in \prec, t_2 \in T_m\}$.
- $\alpha' = (\alpha \backslash \{(t, c)\}) \cup \alpha_m$.

An HTN planning problem is then defined as follows.

Definition 4. An HTN planning problem P is defined as a tuple (D, c_I, s_I) where D is called the domain of P. The domain D is a tuple (F, N_p, N_c, δ, M) in which F is a finite set of facts (i.e., propositions), N_p is a finite set of primitive task names, N_c is a finite set of compound task names with $N_c \cap N_p = \emptyset$, $\delta : N_p \to 2^F \times 2^F \times 2^F$ maps primitive task names to their preconditions and effects, and M is a set of (decomposition) methods. $c_I \in N_c$ is the initial task, which can be viewed as a task network consisting of solely one compound task, and $s_I \in 2^F$ is the initial state.

As mentioned in the introduction, the core aspect of solving an HTN planning problem (and solving a plan verification problem) is to find the decompositions which lead to a solution. Hence, before presenting the precise definition of a solution to an HTN planning problem, we would like to introduce the concept of *decomposition trees* (Geier and Bercher 2011) which capture such decomposition processes in an HTN planning problem.

Definition 5. Given a planning problem P, a decomposition tree $g = (V, E, \prec_g, \alpha_g, \beta_g)$ with respect to P is a set of labeled directed trees where V and E are the sets of vertices and edges respectively, $\prec_g$ is a partial order defined over V, $\alpha_g : V \to N_p \cup N_c$ labels a vertex with a task name, and β_g maps a vertex $v \in V$ to a method $(c, tn) \in M$.

A decomposition tree is *valid iff* for each $t \in T(tn_I)$, there exists a root vertex $r \in V$ labeled with $\alpha(tn_I)(t)$, and for each $v \in V$ with $\beta_g(v) = m$, $m = (c, tn)$, and $c \in N_c$, the following holds.

1) $\alpha_g(v) = c$.
2) tn is isomorphic to the task network induced by the children of v denoted as $ch(v)$, i.e.,
$$tn \cong (ch(v), \prec_g|_{ch(v)}, \alpha_g|_{ch(v)})$$
3) For any child v_c of v and any $v' \in V$, if $(v', v) \in \prec_g$, $(v', v_c) \in \prec_g$, and if $(v, v') \in \prec_g$, $(v_c, v') \in \prec_g$.
4) There are no other ordering constraints in $\prec_g$ except those demanded by 2) and 3).

The *yield* of a decomposition tree g, written $yield(g)$, is the task network $(T, \prec, \alpha)$ such that T is the set of all leafs of g, i.e., the set of all vertices which have no children, $(t_1, t_2) \in \prec$ *iff* $(t_1, t_2) \in \prec_g$ for any $t_1, t_2 \in T$, and $\alpha(t) = \alpha_g(t)$ for all $t \in T$.

Lastly, the solution criteria for HTN planning problems are defined as follows.

Definition 6. Let P be an HTN planning problem. A solution to P is a task network tn such that all tasks in it are primitive, there exists a valid decomposition tree g with respect to P such that $yield(g) = tn$, and it possesses a linearization of the tasks that is executable in the initial state.

Note that a solution to an HTN planning problem is a *partially ordered* task network, which is *not* a plan we refer to in practice, i.e., a plan is normally referred to as a *sequence* of actions (primitive tasks). Hence, we formally define the plan verification problem for HTN planning as follows.

Definition 7. Given a plan $\pi = \langle p_1 \cdots p_n \rangle$ ($n \in \mathbb{N}$) which is a sequence of primitive tasks and an HTN planning problem P, the plan verification problem for HTN planning is to decide whether there is a task network $tn = (T, \prec, \alpha)$ such that it is a solution to P, $|T| = n$, and it possesses a linearization $\overline{tn} = \langle t_1 \cdots t_n \rangle$ such that it is executable in the initial state of P, and for each $1 \leq i \leq n$, $\alpha(t_i) = p_i$.

Implementation of the SOG-based Approach

Having presented the HTN formalism, we now move on to introduce our SOG-based plan verification approach. We begin with introducing the data structures *path decomposition trees* (PDTs) and *solution order graphs* (SOGs) by Behnke, Höller, and Biundo (2019a).

Informally speaking, a PDT with depth K ($K \in \mathbb{N}$) stores *all* possible decomposition trees with depth at most K in an HTN planning problem.

Definition 8. A path decomposition tree $\mathcal{T}_K$ of a certain depth K ($K \in \mathbb{N}$) with respect to an HTN planning problem P is a labelled directed tree (V, E, γ) of depth K in which V is the set of vertices, E is the set of edges, $\gamma : V \to 2^{N_c \cup N_p}$ mapping each vertex to a set of task names, $\gamma(r) = \{c_I\}$ with $r \in V$ being the root of the tree (i.e., the vertex without ancestors), and for every inner vertex $v \in V$ which is neither the root nor a leaf (i.e., a vertex without children), it holds that for each $c \in \gamma(v) \cap N_c$ and every $m \in M$ with $m = (c, tn)$ and $tn = (T, \prec, \alpha)$, there exists a subset $S = \{v'_1, \cdots v'_{|T|}\}$ of v's children such that there exists a bijective mapping β_m from S to T which is also called a child arrangement function of v', and for every $v' \in S$, $\alpha(\beta_m(v')) \in \gamma(v')$.

We use $\mathcal{L}(\mathcal{T}_K)$ to refer to the set of all leafs of $\mathcal{T}_K$. From the definition, one might recognize that $\mathcal{L}(\mathcal{T}_K)$ stores the yields of all decomposition trees of depth smaller or equal to K. Hence, the idea of solving a plan verification problem in

terms of PDTs is straightforward, that is, after constructing a PDT with a certain depth K, we check whether we can *select* a decomposition tree from it whose yield possesses a linearization which is identical to the plan. Particularly, Behnke, Höller, and Biundo (2017) have shown that for any instance of the plan verification problem, we can calculate the upper bound K such that the given plan is a solution *iff* it can be obtained from a decomposition tree of depth smaller or equal to K.

The decision procedure can be captured by a SAT formula. Particularly, the SAT clauses for constructing a PDT with a certain depth and selecting a decomposition tree from the PDT have already been given by Behnke, Höller, and Biundo (2019a), which are still exploited in the context of plan verification. Consequently, in this paper, we focus on constructing the clauses expressing the constraint that the yield of the selected decomposition tree must possess a linearization that is identical to the plan to be verified.

To this end, we shall also introduce the data structure solution order graphs (SOGs) by Behnke, Höller, and Biundo (2019a), which can significantly reduce the number of clauses and state variables required in constructing SAT formulae.

Definition 9. The solution order graph $\mathcal{S}(\mathcal{T}_K) = (\hat{V}, \hat{E})$ of a PDT $\mathcal{T}_K$ of a certain depth K is a directed graph in which $\hat{V} = \mathcal{L}(\mathcal{T}_K)$ is the set of vertices, and an edge $(v_1, v_2) \in \hat{E}$ *iff* for the least common ancestor v of v_1 and v_2, every method $m = (c, tn)$ with $c \in \gamma(v) \cap N_c$ and $tn = (T, \prec, \alpha)$, and the child arrangement function β_m, there exist two children $\hat{v}_1, \hat{v}_2$ of v such that $(\beta_m(\hat{v}_1), \beta_m(\hat{v}_2)) \in \prec$.

Intuitively, the SOG of a PDT $\mathcal{T}_K$ contains the yields of all possible decomposition trees of depth smaller or equal to K, and for each such yield $(T, \prec, \alpha)$, $(t_1, t_2) \in \prec$ for some $t_1, t_2 \in T$ *iff* there is an edge from v_1 to v_2 where v_1, v_2 are two vertices corresponding to t_1 and t_2, respectively.

Having presented the definitions of SOGs and PDTs, we now introduce the SAT clauses encoding that the yield of a decomposition tree selected from a SOG must possess a linearization that is identical to a given plan. For this, we can assume that we already have the SOG $\mathcal{S}(\mathcal{T}_K)$ of a PDT $\mathcal{T}_K$ in hand, because the remaining parts, i.e., constructing the PDT, extracting the SOG, selecting the yield of a decomposition tree from the SOG, and the SAT clauses for encoding these processes, have all been described in the work by Behnke, Höller, and Biundo (2019a).

Further, since the selection of decomposition trees has already been encoded, we can simplify our goal as constructing SAT clauses to encode that for a SOG, there must be a subset of the vertex set such that there is a total order of this subset respecting the edges (i.e., we can view each edge as an ordering constraint), and this chain forms the given plan via selecting a task name for each vertex from its label set, i.e., the set of all possible task names assigned to the vertex.

Given a plan $\pi = \langle p_1 \cdots p_n \rangle$ and a SOG $\mathcal{S}(\mathcal{T}_K) = (\hat{V}, \hat{E})$ of a PDT $\mathcal{T}_K = (V, E, \gamma)$, we start by introducing the SAT variables used to construct the clauses. For each $v \in \hat{V}$ and every *task name* $t \in \gamma(v)$, we construct a state variable v_t indicating whether t in v is selected. For every $1 \leq i \leq n$ and $v \in \hat{V}$ with $p_i \in \gamma(v)$, the variable m_v^i indicates whether p_i is mapped to the vertex v, and the variable f_v^i indicates whether mapping p_j to v is *forbidden* for *every* $1 \leq j \leq i$. In other words, if f_v^i is set to *true*, then any p_j with $1 \leq j \leq i$ cannot be mapped to v. For convenience, for each p_i, $1 \leq i \leq n$, we use $\mathcal{V}(p_i)$ to refer to the set of all vertices v such that $p_i \in \gamma(v)$. Conversely, for every $v \in \hat{V}$, $\mathcal{V}^{-1}(v)$ refers to the set of all *integers* i with $p_i \in \gamma(v)$. For every $v \in \hat{V}$, a_v indicates whether the vertex v is activated. The activation of a vertex here is associated with the selection of a decomposition tree, i.e., if a vertex is activated, then it must be a leaf of the selected decomposition tree, see the work by Behnke, Höller, and Biundo (2019a) for more details.

We first construct the clauses $\mathcal{F}_1$ to enforce the constraint that for every task p_i ($1 \leq i \leq n$), if it is mapped to a vertex v with $v \in \mathcal{V}(p_i)$, then p_i in $\gamma(v)$ must be selected.

$$\mathcal{F}_1 = \bigwedge_{1 \leq i \leq n} \bigwedge_{v \in \mathcal{V}(p_i)} m_v^i \rightarrow v_{p_i}$$

Next, we construct the clauses to enforce that if a task p_i ($1 \leq i \leq n$) is forbidden to be mapped to a vertex $v \in \hat{V}$, then the mapping cannot happen.

$$\mathcal{F}_2 = \bigwedge_{1 \leq i \leq n} \bigwedge_{v \in \mathcal{V}(p_i)} f_v^i \rightarrow \neg m_v^i$$

Further, we shall encode the transition of forbiddenness.

$$\mathcal{F}_3 = \bigwedge_{2 \leq i \leq n} \bigwedge_{v \in \mathcal{V}(p_i)} f_v^i \rightarrow f_v^{i-1}$$

Another important constraint we have to deal with is that the mapping between the plan and the SOG must respect the edges (i.e., ordering constraints). For every $v \in \hat{V}$, we use $\mathcal{V}^+(v)$ to refer to the set of all predecessors of v, i.e., the set of all vertices that are reachable from v. This constraint is then expressed as follows.

$$\mathcal{F}_4 = \bigwedge_{2 \leq i \leq n} \bigwedge_{v \in \mathcal{V}(p_i)} \bigwedge_{v' \in \mathcal{V}^+(v)} m_v^i \rightarrow f_{v'}^{i-1}$$

Informally, the formula $\mathcal{F}_3$ enforce that for any primitive task p_i with $2 \leq i \leq n$, if it is mapped to a vertex v in the SOG, then any predecessor of v is not allowed to be mapped to p_{i-1} for the purpose of respecting ordering constraints.

The next formula encodes the constraint that every action in the plan must be mapped to *at least one* vertex in the SOG.

$$\mathcal{F}_5 = \bigwedge_{1 \leq i \leq n} \Big(\bigvee_{v \in \mathcal{V}(p_i)} m_v^i \Big)$$

Simultaneously, every action in the plan is allowed to be mapped to *at most one* vertex in the SOG. To encode this constraint, we adopt the encoding by Sinz (2005) which enforces that, given a set X of SAT variables, at most one of them can be set to *true*. To ease the notation, we use $\mathbb{M}(X)$ to refer to the encoding. Hence, the constraint in our context is expressed as follows.

	Transport	Woodworking	UM-Translog	Satellite	Monroe-Partially-Observable	PCP	Monroe-Fully-Observable
Total Instances	188	137	52	246	103	26	129
SOG-based	188 (100.00%)	137 (100.00%)	52 (100.00%)	246 (100.00%)	102 (99.03%)	26 (100.00%)	128 (99.22%)
DT-based	138 (73.40%)	95 (69.34%)	52 (100.00%)	246 (100.00%)	0 (0.00%)	25 (96.15%)	0 (0.00%)

Table 1: The number of solved instances in each domain. The header shows the name of each domain. The first row indicates the total number of instances in each domain. The last two rows indicate the number of solved instances by the two approaches.

$$\mathcal{F}_6 = \bigwedge_{1 \leq i \leq n} \mathbb{M}(\{m_v^i \mid v \in \mathcal{V}(p_i)\})$$

Lastly, for every vertex in the SOG, if it is activated, then exactly one action in the plan is mapped to it.

$$\mathcal{F}_7 = \bigwedge_{v \in \hat{V}} a_v \rightarrow \Big(\big(\bigvee_{i \in \mathcal{V}^{-1}(v)} m_v^i \big) \wedge \mathbb{M}(\{m_v^i \mid i \in \mathcal{V}^{-1}(v)\}) \Big)$$

The formula encoding that there exists a yield of a decomposition tree in the SOG whose linearization is identical to the given plan is thus the conjunction of the previous clauses.

Apart from implementing this SOG-based approach, we also reimplement the DT-based approach (Behnke, Höller, and Biundo 2017). Since the reimplementation is simply a translation from the original JAVA code to the C++ code, we omit the discussion of the technique details here. For more information, we refer to the original work by Behnke, Höller, and Biundo (2017).

Empirical Evaluation

We now compare the performance of our SOG-based approach with the reimplemented DT-based one. We used the benchmark set from the IPC 2020 on HTN Planning[1]. The benchmark set contains 1067 instances from 9 domains. However, two domains in the benchmark set, i.e., 'Rover' and 'Barman-BDI', feature so-called *method preconditions*, which are not yet supported by both SOG-based and DT-based (re)implemented in the paper. Consequently, our experiments are run on the remaining 7 domains which include 881 plan instances in total. For each instance, we gave it 10 minutes timeout and 8GB memory limit. All input planning problems were first grounded by the PANDA$_\pi$ grounder (Behnke et al. 2020).

The SOG-based approach successfully solved 879 instances and only failed in two (i.e., the two instances ran out of the timeout). The two failed instances are from the domains 'Monroe-Fully-Observable' and 'Monroe-Partially-Observable', respectively. In contrast, the reimplementation of the DT-based approach only solved 556 instances. Particularly, it failed in *all* instances from the domains 'Monroe-Fully-Observable' and 'Monroe-Partially-Observable', because the planning problems from these two domains contain too many methods and tasks which results in exceeding the memory limit. Tab. 1 shows the number of instances solved by these two approaches in *each* domain.

[1]https://github.com/panda-planner-dev/ipc-2020-plans/tree/master/po-plans

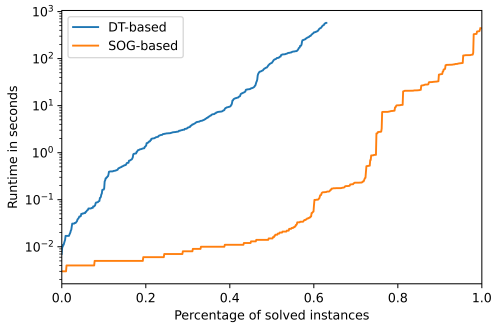


Figure 1: The runtimes against the percentages of solved instances by the SOG-based and the DT-based approach.

Further, Fig. 1 depicts the runtimes against the percentages of solved instances by the two approaches. From the figure, we can see that the new approach is significantly better than the old one. More concretely, over 70% of instances can be solved in one second by the new approach, whereas the number is only about 20% for the old approach.

Future Work and Discussion

In future work, we are going to implement an optimization in both approaches which discards the primitive tasks in a given planning problem which are *not* in a plan to be verified together with all compound tasks that can *only* be decomposed into them. The optimization is employed in the old SAT approach implemented in JAVA but not yet delivered by our reimplementation. One might recognize that the performance of our reimplementation is slightly worse than the old one in JAVA (the evaluation results are shown in the work by Höller et al. (2022)), and we assume that the underperformance is caused by the lack of the optimization.

Additionally, the optimization is also implemented in the planning-based approach (Höller et al. 2022) which transforms a plan verification problem into a planning problem. The planning-based approach happens to produce a SAT formula that is similar to the one produced by our SOG-based approach while the SAT *planner* is exploited. This is however not surprising because both our SOG-based plan verification approach and the SAT-based planner rely on PDTs and SOGs. Thus far, the performance of the planning-based approach with the SAT planner is also slightly better than our SOG-based approach which can solve all instances in the 7 domains on which we ran the empirical evaluation, and we believe that the reason for this is also the optimization.

More importantly, there are two functionalities that are

yet delivered in both approaches implemented in the paper. The first is to support method preconditions. Although many HTN planning formalisms presented in literature do not feature method preconditions, they occur quite often in practice, e.g., in almost all totally ordered (TO) HTN planning domain from the IPC 2020 on HTN planning. Hence, for the purpose of providing an independent plan verifier for IPC, supporting method preconditions is mandatory.

The second functionality we want to implement is calculating a tight bound for the maximal depth of a decomposition tree. As mentioned earlier, a plan is a solution to a planning problem if and only if there exists a decomposition tree of depth smaller or equal to a certain bound. Thus far, the two approaches implemented in the paper calculate such a bound in a loose way, i.e., the obtained bound is significantly larger than the optimal one. As a consequence, our implementations will be less efficient when verifying a plan that is *not* a solution, because in such a case, both approaches need to construct a PDT (DT) of depth up to the bound. Hence, in the future work, we will adapt the approach by Behnke, Höller, and Biundo (2019b) for calculating an optimal bound for a decomposition tree.

As a preliminary work, this paper only did the comparison between the two (re)implemented approaches. In future work, we would like to give a complete comparison between our approaches and other existing ones, e.g., the parsing-based one (Barták, Maillard, and Cardoso 2018; Barták et al. 2020, 2021) and the planning-based one (Höller et al. 2022). On top of that, we would also investigate some theoretical properties of our approaches, e.g., the size of a SAT formula obtained.

Conclusion

In this paper, we developed a new SAT-based HTN plan verification approach which we call SOG-based approach and reimplemented an existing one (which we call DT-based approach). The empirical results show that the SOG-based one is significantly better than the reimplemented one. However, due to the lack of certain optimization techniques, the reimplementation of the DT-based approach slightly underperforms the original one in JAVA, and the SOG-based approach is also defeated by the state-of-the-art planning-based plan verification approach. Hence, we will further improve the (re)implementations in our future work.

References

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of Hierarchical Plans via Parsing of Attribute Grammars. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling, ICAPS 2018*, 11–19. AAAI.

Barták, R.; Ondrcková, S.; Behnke, G.; and Bercher, P. 2021. On the Verification of Totally-Ordered HTN Plans. In *Proceedings of the 33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021*, 263–267. IEEE.

Barták, R.; Ondrcková, S.; Maillard, A.; Behnke, G.; and Bercher, P. 2020. A Novel Parsing-based Approach for Verification of Hierarchical Plans. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020*, 118–125. IEEE.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the Complexity of HTN Plan Verification and Its Implications for Plan Recognition. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS 2015*, 25–33. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This Is a Solution! (... But Is It Though?) - Verifying Solutions of Hierarchical Planning Problems. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling, ICAPS 2017*, 20–28. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT - Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, 6110–6118. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing Order to Chaos - A Compact Representation of Partial Order in SAT-Based HTN Planning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence, AAAI 2019*, 7520–7529. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding Optimal Solutions in HTN Planning - A SAT-based Approach. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, 5500–5508. IJCAI.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On Succinct Groundings of HTN Planning Problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence, AAAI 2020*, 9775–9784. AAAI.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6267–6275. IJCAI.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2016. More than a Name? On Implications of Preconditions and Effects of Compound HTN Planning Tasks. In *Proceedings of the 22nd European Conference on Artificial Intelligence, ECAI 2016*, volume 285, 225–233. IOS.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, 1955–1961. AAAI.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2022. Compiling HTN Plan Verification Problems into HTN Planning Problems. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling, ICAPS 2022*. AAAI.

Sinz, C. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of the 11th Principles and Practice of Constraint Programming, CP 2005*, 827–831. Springer.

Learning Decomposition Methods with Numeric Landmarks and Numeric Preconditions

Morgan Fine-Morris[1], Michael W. Floyd[2], Bryan Auslander[2], Greg Pennisi[2], Kalyan Gupta[2], Mark Roberts[3], Jeff Heflin[1], Héctor Muñoz-Avila[1]

[1]Department of Computer Science, Lehigh University 113 Research Dr., Bethlehem, PA 18015, USA
[2]Knexus Research Corporation, 174 Waterfront Street, Suite 310, National Harbor, MD 20745 USA
[3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
[1]mof217@lehigh.edu, {heflin, munoz}@cse.lehigh.edu, [2]{first.last}@knexusresearch.com, [3]mark.roberts@nrl.navy.mil

Abstract

We describe an HTN method-learning system, which we call T2N, that learns hierarchical structure from plan traces in domains with numeric effects, where some subgoals are numeric. We investigate how different methods of preprocessing training data can impact the effectiveness of the learned methods. We test the learned methods by solving a set of 30 test problems in a simple numeric crafting domain based on the videogame Minecraft. Our results indicate that we can learn functional methods for domains with these characteristics and suggest that different preprocessing techniques lead to method sets with different strengths and weaknesses, with no preprocessing technique superior across all domain tasks.

1 Introduction

Hierarchical Task Network (HTN) planning is a planning paradigm where the planner takes advantage of user-provided domain information (called decomposition methods) to guide the search process. It decomposes complex tasks into progressively simpler ones, until it reaches the granularity of primitive tasks, accomplishable directly by the domain actions. It can provide significant speed up over classical planning techniques, at the cost of extra up-front knowledge engineering required to define the decomposition methods (Ghallab, Nau, and Traverso 2004; Nau et al. 2001) A major problem in applying HTN planning to new domains is the expense and difficulty of authoring correct and useful domain information. A previous system (Fine-Morris et al. 2020) for learning decomposition methods for domains with numeric preconditions was unable to deal with domains where all effects (and therefore goals) were numeric. In this work, we describe a system (called T2N, for Trace2NumericHTN) that, while based on similar design principles, can learn HTNs where changes in numeric variables dictate the structure of the network.

To create structure for our learned HTNs, we identify conditions, called bridge atoms (Gopalakrishnan, Munoz-Avila, and Kuter 2018; Fine-Morris et al. 2020), that mark switches in context within the traces and which we describe as "domain" landmarks. We use these conditions as subtasks into which our learned decomposition methods decompose the high-level tasks of a a domain. Hereafter we use bridge atom and landmark atom interchangeably.

We learn several sets of decomposition methods, each with a different set of domain landmarks as subtasks, then evaluate their performance by solving 30 planning problems. Our results indicate that the set of domain landmarks can impact planning efficiency and goal coverage, and that variations in the way we preprocess the traces before domain landmark learning can impact domain landmark selection, resulting in downstream effects on the learned methods. While more data is necessary to draw a final conclusion about the best way to select domain landmarks, this work shows that comparing method performance on a set of test problems can help us analyze the utility of different sets of domain landmarks.

2 Background

We will provide background information on HTN planning, planning and domain landmarks, extracting domain landmarks, and learning numeric preconditions using goal regression over numeric actions.

2.1 HTN Planning

An HTN planning problem, P, is a triple (D, s_0, T), where D is a domain description, s_0 is an initial state, and T is a list of tasks to accomplish. The domain description D is a tuple (O, M), where O is the set of domain operators and M are the decomposition methods. An operator $o \in O$ is a tuple (h, p, e), where h is the head (i.e., the task name and arguments), p are the preconditions, and e are the effects. A method $m \in M$ is a tuple (h, p, sub), where h and p are the same as for an operator, and sub is an ordered sequence of subtasks. Primitive tasks are those that are accomplishable using operators in O (ground versions of the head of an operator), while complex tasks must be further decomposed by the methods in M until they are reduced to a sequence of primitive tasks that can be performed to accomplish the complex task. What constitutes 'accomplishing' a task is not formally defined. In this work, we learn tasks that are concerned with achieving a goal and use them with an HTN planner. Therefore, we straddle two different hierarchical planning paradigms, HTN planning and Hierarchical Goal Network (HGN) planning, which is very similar to HTN planning, except that instead of decomposing tasks into subtasks, goals are decomposed into subgoals (Shivashankar et al. 2012).

2.2 Planning and Domain Landmarks

Planning landmarks are conditions which always occur in the solution of a given problem and have been addressed in many works (Hoffmann, Porteous, and Sebastia 2004; Porteous, Sebastia, and Hoffmann 2014). We are interested in finding a type of landmark that we call *domain landmarks*, common conditions for two or more problems in a domain as opposed to *planning landmarks*, which are conditions common to one problem in a domain. Hereafter any reference to "landmarks" instead of "planning landmarks" refer to domain landmarks.

Given the set of all problems in a domain, Φ, we define a ϕ-domain landmark as a common planning landmark for every problem $P \in \phi$ where $\phi \subseteq \Phi$. Confirming a planning landmark for a problem is PSPACE-complete problem (Hoffmann, Porteous, and Sebastia 2004) and confirming a domain landmark will increase the running time by the factor $|\phi|$. Therefore, we offer a 'pragmatic' definition of ϕ-domain landmarks: any condition that occurs in at least one solution to every problem in ϕ. In our work, ϕ consists of problems that have a similar initial state s_0 and/or list of tasks T, such that the problems will have common conditions for at least one solution for each problem in ϕ.

As in (Fine-Morris et al. 2020), we use these domain landmarks as subtasks for the tasks demonstrated in the training traces. We also use these domain landmarks to partition the traces into subtraces from which we learn methods for accomplishing each domain landmark. Unlike (Fine-Morris et al. 2020), in this work we learn a two level hierarchy of methods. The landmark methods decompose single top level tasks of the domain into subtasks based on the domain landmarks. These landmark tasks are decomposed by *subplan methods* into sequences of primitive tasks that achieve the landmark. For a generic example hierarchy learned from a trace containing three landmarks, see Figure 1. A trace may not contain all selected domain landmarks, but as in (Fine-Morris et al. 2020) we partition traces at most once per landmark, therefore the maximum number of partitions per trace is $|LM| + 1$ for the set of selected domain landmarks, LM.

Note that landmark methods derive their name from the use of landmarks to decompose the problem, **not** because they show the planner how to accomplish/achieve any individual landmark (except inasmuch as the achievement of later landmarks may be dependent on the achievement of earlier landmarks). They select which landmarks need to be achieved as subgoals of the final task. The role of determining *how* to achieve a landmark falls to the non-landmark methods which, unlike landmark methods, can have primitive subtasks.

2.3 Domain Landmark Extraction

We approximate the domain landmarks using a technique similar to the ones used in (Gopalakrishnan, Munoz-Avila, and Kuter 2018; Fine-Morris et al. 2020). This involves using the natural language processing algorithm Word2Vec (Mikolov et al. 2013) to learn a word embedding for each word (a condition atom or action) in the corpus (a set of traces). Word embeddings are vectors, each associated with a word, whose direction describes the context in which the word appears in the corpus. Therefore, two words with similar direction vectors can be assumed to co-occur frequently in the corpus. We use these vectors to determine how to split the words in the corpus into separate context regions. We use Hierarchical Agglomerative Clustering to form clusters of words, using cosine distance (a measure of the difference of the angle between two vectors) as the metric so that words which share contexts are clustered together. This creates separate context regions in the domain, so that we can probe the boundaries between the regions for condition atoms that signal a shift between the contexts. These condition atoms are our domain landmarks.

2.4 Learning Numeric Preconditions with Function Composition

In addition to learning the subtasks of methods we also must learn preconditions, which may contain numeric functions. We use a modified form of goal regression presented in Fine-Morris et al. (2020). Traditional goal regression involves inverting the effects of an action, so that the effects of the action are removed from a state, and the action preconditions are added to it. Previous work (Fine-Morris et al. 2020) describes how to augment traditional goal regression to regress actions with numeric fluents via function composition. For example, when learning a method that decomposes into a sequence of actions $a1, a2$ where the preconditions and effects of $a1$ and $a2$ are:

$$p^{a1} = [f(varX, \dots) > 1, \ \dots]$$
$$e^{a1} = [varX \leftarrow f(varX, \dots), \ \dots]$$
$$p^{a2} = [g(varX, \dots) > 2, \ \dots]$$
$$e^{a2} = [varX \leftarrow g(varX, \ \dots), \ \dots]$$

the methods preconditions would include,

$$f(varX, \dots) > 1, \ g(f(varX, \dots), \dots) > 2.$$

The inequalities from both actions occur in the new set of preconditions, but in addition both $g(\dots)$ and $f(\dots)$ are applied to $varX$ before the inequalities from $a2$ is checked, to ensure that whatever value $varX$ is, it will be > 2 after being updated according to both $f(\dots)$ and $g(\dots)$.

3 Example Domain

The domain used in this paper is a custom domain based on the crafting system of the game Minecraft. Base resources (wood, stone, coal, iron) can be gathered from the world, and used during crafting to make new items. The amount of a resource is described by terms in the form `value(item, amount)`, where `item` is a label for a counter of a particular type of item, and `amount` is a numeric value. When a gather action is used to collect a resource from the world, the world's resource counter is decreased by the specified amount, while the resource counter of the agent who performed the gather action increases by that amount.

Many crafting recipes require an item called a crafting table. Some resources – such as wood – can be gathered by hand or using a tool. Others – such as stone, coal, and

iron – can only be gathered using a specific type of tool, e.g., a pickaxe. The grade of the pickaxe dictates what resources it can collect and depends on the material it is made from. Pickaxes made from stronger materials can be used to mine harder resources. From weakest to strongest, the materials are: wood, (cobble)stone, iron. Mineable resources from softest to hardest are: stone, coal, iron. Wooden pickaxes can mine stone and coal. Stone and iron pickaxes can mine iron and anything softer. In typical Minecraft, pickaxes and axes have durability, and can only be used a certain number of times before they break. We elided this aspect of tool-use.

When an agent crafts an item, the value of the counters of the consumable ingredients are decreased according to the amounts called for by the crafting recipe. The agent's counter for the crafted item increases by the batch amount, which is usually 1 but can be more as in the case of the rail item, which can only be made in batches of 16.

Some items, called stations, are not consumable but are monopolized while they are required to craft a recipe. We included the crafting table, where argents craft complex items, and the furnace, which agents fuel with coal to smelt iron into iron ingots.

Some items are more expensive to craft from scratch than others, because they require more steps if the agent starts with an empty inventory. For example, to craft a wooden axe the agent only needs to gather wood and make a crafting table, sticks, and planks, while crafting a stone axe requires all the steps to craft a wooden pickaxe, which the agent must use to gather stone to form the blade of the stone pickaxe. To craft an item that has iron ingots as an ingredient (as in the case of our cart, rail, iron pickaxe, and iron axe goal items), the agent must have a furnace to smelt iron, which can only be gathered using a stone pickaxe. In terms of relative expense, wood items $<$ stone items $<$ iron items.

4 Approach

T2N has two phases: learning domain landmarks and learning methods. We next detail these phases and their steps.

4.1 Phase 1: Learn Domain Landmarks

Phase one has three steps. For a set of traces $\mathcal{T}$ (see Section 5.1 for trace generation): Step (1.1) formats $\mathcal{T}$ into three variations and discretizes numeric fluents, Step (1.2) learns word embeddings for $\mathcal{T}$ using Word2Vec, and Step (1.3) selects bridge atoms that partition $t \in \mathcal{T}$ into subtraces. Previously, Fine-Morris et al. (2020) used a similar technique with some differences in Steps 1.1 and 1.3. To summarize, this work better formalizes how to discretize numeric fluents, investigates the impact of formatting $\mathcal{T}$, and examines selecting domain landmarks for *numeric* subgoals.

Step 1.1: Preprocess Traces. To prepare traces for landmark learning, we first must format them in one of three styles, full (FL), precondition-effect (PE), and randomly-augmented (RA). Traces formatted in the FL style have full states (i.e., they contain static conditions carried over from the initial state). The states of traces formatted in the PE style include only the effects of the previous action and the preconditions of the next. Randomly-augmented traces are partway between the two, with states the contain all the atoms of the PE traces, plus a small set of randomly selected atoms that would be present in the FL version of the state but not the PE version. See Table 1 for examples of all three, illustrating these differences. In the FL example, the atom `value(agent_stone, 0)` is included in states before and after the action `gather(wood, 5)` despite its extraneousness. In the PE example, atoms which are neither preconditions nor effects of the surrounding actions are excluded, `value(agent_stone, 0)` is excluded from the state because it is neither a precondition nor effect of the actions `gather(wood, 5)` and `craft(plank)` which surround the state. In the randomly-augmented (RA) trace, atom `value(stone, 350)` which describes the amount of free (i.e., not held by an agent) stone in the world is included in the first state randomly, and `value(agent_stick, 0)` is included randomly in the second state, although they are not pertinent to either of the actions adjacent to them.

FL	PE	RA
value(a_w, 0)	value(a_w, 0)	value(a_w, 0)
value(w, 4000)	value(w, 4000)	value(w, 4000)
value(a_c, 0)		value(c, 350)
value(c, 350)		
...	...	...
gather(w, 5)	**gather(w, 5)**	**gather(w, 5)**
value(a_w, 5)	value(a_w, 5)	value(a_w, 5)
value(w, 3995)	value(w, 3995)	value(w, 3995)
value(a_c, 0)		value(a_s, 0)
value(c, 350)		
...	...	...
craft(plank)	**craft(plank)**	**craft(plank)**

Table 1: Trace formatting variations used for learning word embeddings (with the original numerics, i.e., 'unskolemized'). Actions are in bold. To save space, variable names have been shortened: 'a' replaces agent, 'w' replaces wood, 'c' replaces (cobble)stone, 's' replaces stick; therefore, variable a_c is a shortening of agent_stone. These shortening are not used in the real traces.

In FL styles, the vocab size is larger and states are much larger because all true conditions are included. This has repercussions when learning word embeddings, as the longer traces make Word2Vec more expensive and creates more connections between words, as static conditions occur with much greater frequency, and any conditions achieved early in the text are particularly effected. PE style traces limit relationships between words to try to ensure strong relationships form only between actions and conditions that are related by function. RA traces try to take advantage of the strengths of both.

In a process similar to skolemization (Bundy and Wallen 1984), we replace the numeric values with names for the value ranges. This ensures that landmarks from variables that take on many values correspond to any value within a range, instead of to a single value. To do this we collect the values for all variables across all traces and then

make a histogram for each variable. Then, for each instance of that variable in an trace, we replace the numeric value with a string `BINX` where `X` is the bin number in which the value is found. For example, if we have a variable for `agent_coal` and the values of this variable in the various traces are 15, 21, 23, 26, or 28, we might create the following histogram (bin label, range) pairs: (`BIN1`, [0-9]), (`BIN2`, [10-19]), (`BIN3`, [20-29]), and (`BIN4`, [30-∞]), such that `value(agent_coal, 21)` and `value(agent_coal, 23)` both become `value(agent_coal, BIN3)`. If we did not skolemize the numeric values, Word2Vec would interpret atoms that correspond to the same variable as completely different words when they have even slightly different numeric values. This would increase the vocabulary size of the corpus and make it more difficult to learn relationships concerning numeric atoms. By skolemizing, we ensure that values of the same state variable are recognized as the same word when they are within a certain range (i.e., the bin ranges). Previous similar work (Fine-Morris et al. 2020) skolemized all numerics to a single bin, but this erases all variation in numeric values, which is undesirable when trying to learn numeric landmarks.

Step 1.2: Learn Word Embeddings. We use Word2Vec with the skip-gram model to learn a set of word embeddings for the words in our corpus of traces. The properties of Word2Vec ensure that words that occur in the same context are clustered by cosine distance. A small cosine distance for two word embeddings indicates that the corresponding words frequently co-occur.

To learn word embeddings, we linearize traces such that each condition or action is treated as a word. For a trace $s_0, a_1, s_1, ..., s_N$, the linearized trace would be $c_{01}, ..., c_{0N}, a_1, c_{11}, ..., c_{1N}, ..., c_{N1}, ..., c_{NN}$, where all the conditions true in s_0 are represented by the set of conditions in the sub-sequence $c_{01}, ..., c_{0N}$ and each c is a condition true in state s_0. From a planning perspective, the ordering of the conditions within a state does not matter, and the conditions in a state can be reordered to create more input traces for Word2Vec.

Step 1.3: Select Landmark Atoms. The landmark learning process is as follows: we (1) learn word embeddings from the traces with Word2Vec, (2) form n=2 clusters of the word embeddings using a clustering algorithm and the cosine distance metric (we use Hierarchical Agglomerative Clustering), (3) score each atom according to the average cosine distance of the atom to those of the opposite cluster, (4) filter out non-effects atoms (i.e., any atom that isn't an effect of an action), (5) select atoms with scores less than $((max - min) \times .2) + min$ where max and min are the maximum and minimum scores of the effects atoms (i.e., any atom that is an effect of an action in at least one trace) and (6) replace all bin names with the associated numeric value range. The purpose of steps (2-3) are to split the atoms into a least two context groups and then find the atoms that are most in-between those two groups, i.e., the atoms in the corpus that has been assigned to one group but is as close as possible to another, possibly marking the boundary between the context groups. We choose two clusters semi-arbitrarily, as the best number of clusters will be domain-dependent and difficult to determine (we leave exploring this for future work).

4.2 Phase 2: Learn Methods

Phase two uses the landmark atoms to learn methods and consists of three steps. Step (2.1) partitions $\mathcal{T}$ using the landmark atoms, Step (2.2) learns subplan methods for each subtrace, and Step (2.3) learns landmark methods using the subplan methods.

In previous work (Fine-Morris et al. 2020), the final tasks of each trace were determined by finding which of a large set of user-provided possible final goals were present in the effects of the final action. In this work, each trace is annotated with a final task, which is not explicitly defined by a final goal condition, because this allowed tasks to be named more flexibly. Additionally, Fine-Morris et al. (2020) did not ground any of the numeric variables in the preconditions of the methods, while we selectively ground numeric variables to specific values when they are not already constrained by the calculations (otherwise values are constrained only by criteria inherited from actions).

During this phase, each trace is processed independently. Figure 1 shows how a hierarchy can be learned from one trace. The root contains the landmark method which has subtrees for each of its 4 subtasks: 1 for each of the three landmarks plus the final subtask method for final goal. The next level decomposes each landmark into a sequences of primitive tasks that accomplishes the landmark. All leaves in the tree are actions.

Step 2.1: Partition Traces. T2N accepts as input a set of traces, each annotated with the head of their final goal and a set of possible landmarks. We discard any traces which contain none of the landmarks because it is impossible to learn landmark methods from them. T2N partitions each trace, splitting at the states that contain the first instances of a landmark in the action effects. This results in a trace partitioned so that the last action of each subtrace achieves a landmark.

Step 2.2: Learn Subplan Methods. For each partition of the trace, T2N learns a method for accomplish the landmark that was accomplished by the subplan. The head of the subplan method comes from the landmark atom that occurs in the final state of the subtrace, modified so that any non-variable arguments are moved into the name. For example, a landmark such as `value(item, 1)` would become the method head `value_1(item)` because `item` denotes a variable but `1` does not.

We learn preconditions for a subplan method by performing goal regression using our modified regression technique over the subplan. We check to make sure that any variable that is updated in the subtrace and not already ground to a specific value in the preconditions have some calculation-based constraint applied to them in the precondition calculations. If not, we ground that variable according to its value in the initial state of the subtrace. We do this to ensure that variables which are unconstrained by the actions of the trace are still constrained by the context of the trace. We repeat this process for each remaining subtrace.

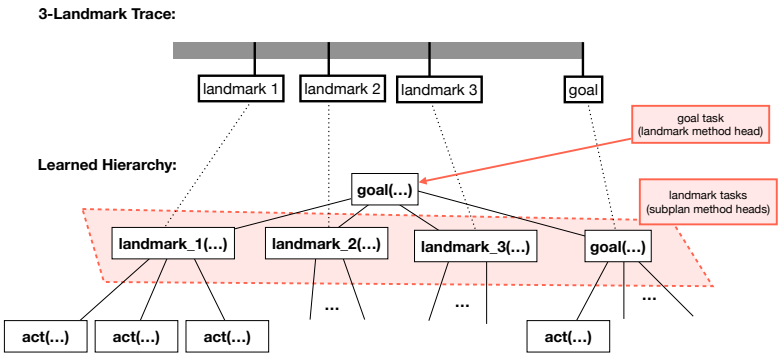


Figure 1: Example hierarchy showing the hierarchical structure learned from a trace with three landmarks. The structure learned from a trace is determined by the number of landmarks in the trace and the number of actions in each subtrace.

Each time we learn a method, we check to see if it can be merged with an existing method. Methods can be merged if they were learned from equivalent plans and if they are identical excepting their ground numeric variables. The merging process will ensure that any variables with numeric values will be transformed into a value range that includes both original values.

Step 2.3: Learn Landmark Method. Once we have methods for each (subtrace, subtask) pair, we can learn a landmark method. The head is the final trace goal, the subtasks are the landmarks plus the final trace goal. The preconditions are learned by via goal regression over the entire trace. As before, we ground any unconstrained numeric variables according to the first state of the trace.

An Example Hierarchy For a task such as `make stone_pickaxe(agent)`, the 3 landmarks of the example hierarchy of Figure 1 could be `value(agent_crafting_table, 1)`, `value(agent_wooden_pickaxe, 1)`, `value(agent_stone, 3)`, with `make stone_pickaxe(agent)` as the final goal. The head of the landmark method would be `make stone_pickaxe(agent)`, with 4 subtasks/subgoals: `value_1(agent_crafting_table)`, `value_1(agent_wooden_pickaxe)`, `value_3(agent_stone)`, `make stone_pickaxe(agent)`. The method for decomposing landmark_1 could contain a sequence of three primitive subtasks to gather wood, make planks, and make a crafting table.

5 Methods

All work for this paper was done on a 2020 MacBook Pro with a 2 GHz Quad-Core Intel Core i5 processor, 32 GB 3733 MHz LPDDR4X memory, running Big Sur (v. 11.6).

5.1 Training Problems and Solutions

To generate training traces, we first create randomized training problems, then use a planner to create solution plans (i.e., simulate plan demonstrations from a human). We use a customized version of the Pyhop planner with hand-crafted HTN methods, but as the plans are comprised of only state and action information and are not annotated with decomposition information, we believe that any planner will suffice. The training problems consist of initial states with counters for each gatherable item (wood, stone, iron, coal) indicating how many units of this resource are available for collection. The initial amounts of these counters were set randomly between 2000-5000 in steps of 100. Each agent (so far all traces have been single-agent) has a counter for each existing item type, all initialized to zero.

All tasks were to "make X" where X is some product/-goal item that requires a crafting table to produce. The goal items were: [wooden, stone, iron] axe, [wooden, stone, iron] pickaxe, furnace, rail, or cart.

The hand-crafted methods were designed to produce plans with variability, not to produce the most efficient plans, or to find plans more quickly. We prioritized plan variability because we believe that this is more advantageous for the landmark learner, but we have yet to do a formal comparison. For each goal item we generate 10 plans. From a single set of plans we generate traces in the three different formats (i.e., FL, PE, and RA). We generate 20 traces from each plan by reordering the conditions in each state randomly. Each training trace is annotated with the head of the corresponding task to be learned.

5.2 Word2Vec Hyperparameters

We learn word embeddings using the following hyperparameters (we leave probing the impact of hyperparameters on landmark selection for future work). For all trace formats we used alpha=0.001, min alpha=0.0001, and 1000 epochs, with vector dimensions calculated according to to $V/20$, where V is the size of the corpus vocab, and window size is $3C$, where C is the average number of conditions per state. For PE traces, the vector size was therefore 9, and for RA and FL it was 11. The window sizes were 462, 78, and 48 for FL, RA, and PE, respectively.

5.3 Test Problems

We select test problems to examine the effectiveness of landmark methods. Landmark methods that were applicable to achieving specific goals were not learned for all goals in all

method sets. Our test problems were selected to compare the performance of landmark methods, and by proxy the landmarks selected for each method set. If a particular landmark set results in learning a method set that doesn't cover certain goals, the failings of that set w.r.t. the other sets are obvious and don't require extensive testing.

Our selection procedure was designed with this in mind and is as follows: For each goal task, we generate a pool of 15 solvable test problems using the same technique for generating the training problems, but changing the starting ranges of the availiable gatherable items to a random number between 200-5000 in steps of 10. We confirmed that each problem was solvable by generating a solution plan using an HTN planner and the same hand-crafted methods used to generate the training traces. If the planner failed to generate a solution to test problem, we discarded and replaced the problem. Once we had a pool of solvable test problems for each goal task, we selected test problems via the following procedure for each method set: we (1) pool together all test problems for each goal task for which the method set has landmark methods and (2) select a set of 10 problems from the previously created multi-goal pools and add it to the set of selected methods. If a problem was selected for a previous method set, we selected a different problem (i.e. the final set of test problems should contain no duplicates).

5.4 Metrics

We used two metrics for comparing the four method sets: coverage and efficiency. **Coverage** measures how much of the domain is solvable by a particular method set, with subcomponents **goal coverage** and **problem coverage**. Goal coverage describes how many of the domain goals are address by at least one landmark method, regardless of how successful the problems with that goal are solved. If a method set has no landmark methods for a particular goal, it does not cover that goal. If a method set has landmark methods for a goal, but cannot solve any of the test problems with that goal in the time-limit, it still covers that goal. Problem coverage measures how successfully a method set solves test problems. **Efficiency**, the second metric, is concerned with how quickly a planner using a method set solves a particular test problem, or a set of test problems.

6 Results

Our results show that different ways of formatting the traces can produce very different sets of learned landmarks. We also find that landmark selection impacts which final goals are covered by a method set, and the efficiency with which plans for different goals are generated.

6.1 Landmarks

We learned three sets of landmarks (FL, PE, RA), one for each style of trace, and hand-selected a set of player-intuitive custom landmarks (CL) for a baseline (see Table 2). Of the landmarks learned by our algorithm, two were concerned with possessing a tool or station (wooden pickaxe in FL and furnace in RA), while most focused on gatherable resources (coal, iron ore, or cobblestone). The CL landmarks were more concerned with the possession of various types of pickaxes and two types of crafting stations (crafting table and furnace).

FL	value(agent_wooden_pickaxe, 1)
PE	value(cobblestone, [3896-4782])
	value(agent_cobblestone, 47)
RA	value(agent_coal, [10-18])
	value(agent_furnace, 1)
	value(agent_iron_ore, [10-18])
CL	value(agent_crafting_table, 1)
	value(agent_wooden_pickaxe, 1)
	value(agent_stone_pickaxe, 1)
	value(agent_furnace, 1)
	value(agent_iron_pickaxe, 1)

Table 2: All landmarks for each method set. Landmarks with a bin containing only a single value are specified with that discrete value, instead of the bin range (i.e., value(agent_crafting_table, 1)).

Goal- and Problem- Coverage. Both CL and FL method sets covered every goal in the set of test problems and demonstrated full problem coverage. Both RA and PE demonstrated full problem coverage for every covered goal, although neither covered the wooden_pickaxe goal and the former also did not cover the stone_pickaxe goal. In Table 3 we can see that only CL covers every final goal task. FL, PE, and RA are all missing landmark methods for "make wooden_axe". This is because none of them have landmarks pertinent to the task, so no landmark methods could be learned.

Goal Tasks	CL	FL	PE	RA
make cart	10	10	1	10
make furnace	7	7	3	7
make iron_axe	10	10	2	10
make iron_pickaxe	10	10	1	10
make rail	9	9	3	10
make stone_axe	8	8	3	2
make stone_pickaxe	6	6	3	0
make wooden_axe	5	0	0	0
make wooden_pickaxe	4	4	0	0

Table 3: Number of landmark methods in each method set for each task. Goals with a 0 are not covered by the method set of that column.

From these results, we can see that the formatting of the trace provided to Word2Vec to generate embeddings impacts the atoms selected as landmarks. This can impact the landmark methods the system can learn, because traces which contain none of the selected landmarks have to be discarded. If all traces demonstrating how to achieve a particular end goal are discarded, we cannot learn that goal.

6.2 Planning Duration

Figure 2 shows the amount of time the planner took to solve each test problem. Problems occur in the same order for all 4

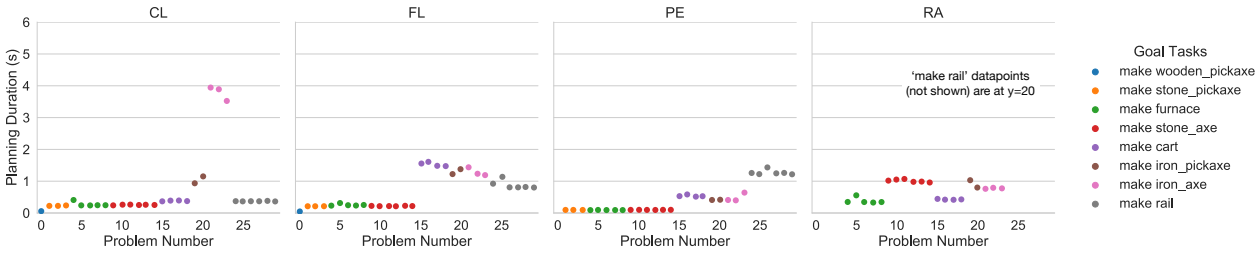


Figure 2: Number of seconds to find a correct plan using the four sets of learned methods.

graphs, and are grouped by goal (see legend). No method set showed obviously superior performance across all goal categories, however, the landmarks used during method learning can clearly have an impact on planning efficiency, as some of the method sets solve problems of a certain category faster than others.

In general, when problems are solved with the same set of methods, there is not much difference in the solving time of problems with the same goal. All attempted problems were solved in less than 30 seconds, but problems were only attempted if the goal of the problem was covered by a landmark method in the method set.

A primary purpose of these experiments was to determine how changes to the formatting of traces impacted the learning of landmark atoms, and therefore the learning of methods. We also sought to determine if learning methods and using them to solve problems would provide us with information on the utility of the learned landmarks, as it is otherwise difficult to judge the effectiveness of the landmark learning technique, including the Word2Vec hyperparameters, the best formatting for the traces input to Word2Vec, and the technique for selecting landmarks using word-embeddings.

It is difficult to declare any one set of method better. Each had their strengths and weaknesses. Problems solved with the CL method set were mostly completed in less than a half second. Goal categories that were outliers included those with iron_pickaxe and iron_axe goal products, which rose to about a second and about four seconds respectively. Notably, it was much more efficient on the rail problems than any other method sets.

For the PE method set, the solvable lower-cost goals (stone_pickaxe, furnace, stone_axe) were accomplished slightly more quickly than for any other method set, most-likely because the landmarks were all concerning cobblestone, an ingredient of stone axes/pickaxes and the furnace. The uptick in planning time for iron products is probably related to the slightly greater difficulty of deciding how much coal and iron to gather, and how much iron to smelt. Rails require the most iron, so they had the greatest time cost.

For FL, duration for the low-cost items are roughly the same as for CL and PE, but for the cart goal the duration jumps up by more than a second to nearly 1.5 seconds, and then decreases gradually for the remaining goals (iron_pickaxe, iron_axe, rail) stopping at slightly less than a second. This almost inverts the trend seen in the PE graph for the iron goal products, where rail is more expensive than cart. (Note that, because there is only one landmark atom selected for FL, the landmark methods can have only two subtasks: one for making the wooden pickaxe and one for making the final product.)

In some goal categories, RA performed worse than all other method sets (furnace and stone_axe) and demonstrated middle-of-the-road performance for other categories (cart, iron_pickaxe, iron_axe) and failed to cover both stone_pickaxe and wooden_pickaxe. Additionally, its performance on the rail goal was an extreme outlier at about 20 seconds while solutions for all other problems and method sets were achieved in less than 5 seconds. This is unexpected as the RA landmarks, which involve the furnace, iron_ore, and coal, should be useful for learning efficient solutions.

The comparatively poor performance on furnace and stone_axe is probably due to unnecessary resource gathering; the RA landmarks target resources that are not needed for either goal product, and therefore methods are only learned from traces for those goals when they do something inefficient (for example, like crafting a stone and iron_pickaxe in order to gather stone to make a stone_axe).

7 Related Work

Fine-Morris et al. (2020) uses a similar technique to learn methods with numeric preconditions, leveraging Word2Vec and clustering techniques. It then uses these landmarks to partition the traces and learn a similar hierarchy of methods with numeric preconditions of arbitrary complexity. The crucial difference is that T2N allows for goals to be numeric fluents. This enables the agent to solve problems where, for example, there is a minimum number of resources needed.

Word2HTN (Gopalakrishnan, Munoz-Avila, and Kuter 2018) uses a technique involving Word2Vec and clustering similar to the one described in this work, with similar inputs and outputs. A significant difference between our system and Word2HTN is that Word2HTN learns an exclusively binary hierarchy (every method has two subtasks) as opposed to our system, where the top-level methods decompose the task into arbitrarily-many subtasks, capturing complex underlying task structures beyond right-recursive task decompositions. Additionally, while Word2HTN can learn numeric preconditions and effects involving arithmetic operations only (e.g., addition, subtraction), our system can learn more complex (i.e., compound) functions via function composition. This has trade-offs, as Word2HTN can simplify its numeric expressions in ways our system cannot (for exam-

ple, it can combine two updates to $varX$, $+3$ and -2 into one $+1$ update. Because of this, the preconditions learned by our system can be very large.

Both HTN-Maker (Hogg, Muñoz-Avila, and Kuter 2016) and HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) learn task decomposition methods from traces and user-provided annotated task definitions comprised of (precondition, effects) pairs. HTNLearn uses constraint satisfaction to learn method preconditions, while HTN-Maker uses goal regression and can learn right-recursive subtasks. Crucially, in our work we are not giving the subtasks as input; while we annotate our traces with the goal-task, we identify the bounds of a task using the occurrence of learned landmarks in the effects of an action and did not require extensive user-provided subtask specifications. Also, neither handles numeric fluents.

Segura-Muros et al. (2015) learns HTN planning domains from plan traces using a combination of process mining and inductive learning. Process mining builds a behavioral model from a set of event logs. By treating plan traces as event logs the authors use process mining to learn the hierarchical structure of the plan traces. Once the structure is learned, they extract pre-state and post-state pairs from the plan traces for each action and method and utilize inductive learning to generate preconditions and effects. They show that their algorithm can learn a simple and straightforward domain capable of consistently solving test problems. However, they do not handle numerics.

ICARUS (Langley, Choi, and Rogers 2007) uses background knowledge and means-ends analysis to learn Teleoreactive logic programs from provided solution plans. The background knowledge includes concept definitions, which are composed together to form more complex concepts. The hierarchy of concepts this creates helps define the hierarchical structure of the programs. Since tasks are linked to the achievement of concepts which are built upon achieving goals, Teleoreactive logic programs basically encode HGNs. Our system requires less user-provided domain information to dictate hierarchical structure, instead inferring structure from the learned landmarks.

X-learn (Reddy and Tadepalli 1997) learns goal-subgoal relations. X-learn uses inductive learning to learn d-rules with preconditions and a sequence of fully-ground subgoals (similar to HGN decomposition methods or macro-actions) from increasingly-difficult exercises. X-learn is purely symbolic, unlike our system which can learn numeric goals.

8 Final Remarks

We have discussed our results from learning several sets of HTN methods using an automated learner, T2N, for a domain with numeric goals. We tested them on 30 test problems and discussed how the set of landmarks used to structure the learned methods can have an impact on which goals that method set covers. No consistent pattern exists in the efficiency of methods for all goals across method sets, but the set of landmarks impacts the efficiency with which different method sets solve problems for the same goal. Our results indicate that the selected landmarks can have an impact on the final goals that are covered by a method set, and that it is possible that different trace formats can impact landmark selection.

While it is difficult to draw firmer conclusions about the impact of trace formatting without more data, results thus far suggest that some method sets are complementary in terms of the problems and goals they solve. Although method sets may fall short of complete domain coverage, it is possible that the weakness of each could be remedied by using parallel planners, one for each method set. By taking the plan of the first planner to finish, we could cover the complete set of goal tasks. While this would be more computationally expensive, it would take advantage of the strengths of both learned method sets. We suggest this as an alternative to merging method sets, which might prove challenging to do without degrading performance.

The work presented here is preliminary, with possible future research including: (1) comparing the performance of landmarks learned using multiple bins with those learned from a single bin, (2) using clustering of numerics for assigning bin labels instead of discretely-sized bins, (3) performing the same experiments with different Word2Vec hyperparameters, (4) experimenting with more than 2 clusters during bridge atom selection, (5) trying to replicate the results in other domains, and (6) finding ways to prevent goals from going uncovered by landmark methods during learning due to landmark-less traces being skipped, for example, by introducing the ability to select fallback landmarks for landmark-less traces.

Acknowledgments. We thank NRL and ONR for funding this research and NSF's Independent Research and Development (IR/D) Plan. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Bundy, A.; and Wallen, L. 1984. Skolemization. In Bundy, A.; and Wallen, L., eds., *Catalogue of Artificial Intelligence Tools*, 123–123. Berlin, Heidelberg: Springer. ISBN 978-3-642-96868-6.

Fine-Morris, M.; Auslander, B.; Floyd, M. W.; Pennisi, G.; Muñoz-Avila, H.; and Gupta, K. M. 2020. Learning Hierarchical Task Networks with Landmarks and Numeric Fluents by Combining Symbolic and Numeric Regression. In *Proceedings of the 8th Annual Conference on Advances in Cognitive Systems*, 16.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier. ISBN 978-0-08-049051-9. Google-Books-ID: uYnpze57MSgC.

Gopalakrishnan, S.; Munoz-Avila, H.; and Kuter, U. 2018. Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. *AI Communications*, 31(2): 167–180.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research*, 22: 215–278.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2016. Learning Hierarchical Task Models from Input Traces.

Computational Intelligence, 32(1): 3–48. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/coin.12044.

Langley, P.; Choi, D.; and Rogers, S. 2007. Interleaving Learning, Problem Solving, and Execution in the Icarus Architecture. 27.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc.

Nau, D.; Muñoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-Order Planning with Partially Ordered Subtasks. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, IJCAI'01, 425–430. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-812-2.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2014. On the Extraction, Ordering, and Usage of Landmarks in Planning. In *Sixth European Conference on Planning*.

Reddy, C.; and Tadepalli, P. 1997. Learning Goal-Decomposition Rules using Exercises. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence.*, 843–843.

Segura-Muros, J. A. 2015. Learning HTN Domains using Process Mining and Data Mining techniques. *Workshop on Generalized Planning (ICAPS-17)*, 8.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A Hierarchical Goal-Based Formalism and Algorithm for Single-Agent Planning. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, 9. Valencia, Spain.

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

Learning Operational Models from Demonstrations: Parameterization and Model Quality Evaluation

Philippe Hérail, Arthur Bit-Monnot

LAAS-CNRS, Université de Toulouse, INSA, CNRS, Toulouse, France
philippe.herail@laas.fr, abitmonnot@laas.fr

Abstract

When acting in non-deterministic environments, autonomous agents must balance between long-term, complex goals with unpredictable events and reactive behavior. In this context, hierarchical operational models are attractive in that they allow the execution of complex behavior either in a purely reactive fashion or guided by a planning process. Just like for HTN models with which they share most characteristics, one key bottleneck in the exploitation of operational models is their acquisition.

In this paper, we introduce an algorithm for learning hierarchical operational models from a set of demonstrations. Given an initial vocabulary of tasks and some demonstrations of how they could be achieved, we present how each task can be associated to a set of methods capturing the operational knowledge of how it can be achieved. We present the structure of the learned models, the algorithm used to learn them as well as a preliminary evaluation of this algorithm.

Introduction

To allow an autonomous agent to operate in its environment, we can differentiate two family of approaches, split by the way they select their next action: reactive or deliberative. While a reactive approach may cover several distinct acting techniques, these often present the common characteristic of being fast to act but short-sighted. Meanwhile, deliberative approaches typically rely on planning to consider long-term impacts of a decision, at the cost of increased deliberation time. In order to find a compromise between both these approaches *hierarchical operational models* (Ghallab, Nau, and Traverso 2014) have been developed along with acting engines such as in (Patra et al. 2021). This combination allows to have models that distinguish between different ways to achieve a given task through methods. These hierarchical models allow the agent to act either reactively, selecting one method without any reasoning about the future, or to make more elaborate choices. Furthermore, the hierarchical aspect allows to reduce the possibilities that need to be considered at any given time, similarly as in hierarchical planning.

Even though these hierarchical models are a formalism that allows to act more efficiently while remaining interpretable by human engineers, it is cumbersome to design such models from scratch. This difficulty stems from the quickly exploding number of possible contexts that need to be considered when carrying out even basic tasks in a simple environment. To address this issue, we intend to allow the agent to learn such operational models from previously observed execution traces, and in particular the ones resulting from a tutor's demonstration.

The goal of such a learning system would be to be able to solve any previously demonstrated tasks through a solution of at least equivalent quality to the demonstrated one. It should also be able to generalize the demonstrations to solve new unseen tasks, or previously demonstrated tasks in a new environment. This is done by learning, for any given task in the considered domain, a set of methods that achieve the high-level objectives associated to the task. This set of methods should cover all possible ways of achieving this task with the exception of clearly suboptimal ways. Any method should be associated with a validity scope that define whether it is *applicable* in a given state. When applicable, it should achieve the task.

Intuitively, if a learned operational model has these desirable properties, an acting engine facing a task to achieve could pick any applicable method and have the guarantee that it will fulfill the objectives associated to the task. While this might lead to suboptimal behavior, it should be possible for an automated planner to guide the choice of the method to obtain the optimal behavior.

The objective of this paper is to present a method for building hierarchical operational models based on past experiences. The experience takes the form of execution traces that can be the result of the agent's own activity or of demonstrations by a tutor. The agent is assumed to have a fixed set of primitive capabilities and to act in a Fully Observable Non-Deterministic (FOND) environment.

Related Work

Over the years, several approaches have been developed to learn hierarchical models.

HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008) learns Hierarchical Task Networks (HTNs), for use in fully-observable deterministic domains from execution traces. This approach uses tasks annotated with preconditions and postconditions so as to extract sequences that allow to achieve the postconditions starting from a state where the preconditions hold. The method was later extended by Hogg, Kuter, and Muñoz-Avila (2009) to handle non-deterministic

domains, through the use of a right recursive structure instead of sequences. However, the models learned are restricted to a limited task and method structure, resulting in rather flat hierarchies, which limits the guidance offered to an agent using them for planning.

HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) also learns HTNs with similarly annotated tasks, through converting the learning problem into one of constraint satisfaction, building the constraints from, e.g., the ordering of tasks within the examples and the state preceding the application of a method. The problem is then solved using a MAXSAT approach and the solution converted back into an HTN model. While this approach does not handle nondeterminism, it supports partially observable states. A similar approach is even able to use partial, disordered input traces (Zhuo, Peng, and Kambhampati 2019).

Recently, the learning of Hierarchical Goal Networks (HGNs) structure instead of HTNs, for nondeterministic domains, has been proposed as a preliminary work by Fine-Morris and Muñoz-Avila (2019), leveraging a vector representation of the states and unsupervised learning procedures to learn such networks while limiting the burden of annotating demonstration data.

Due to their similarities with HTNs, some work aiming at learning grammars is relevant and in particular the work on learning Combinatory Categorial Grammars (CCGs) for plan and goal recognition (Geib and Goldman 2011; Kantharaju, Ontañón, and Geib 2019). While the learned CCGs are not always practically usable, the authors propose several ideas for extracting interesting patterns from a set of execution traces.

The algorithm for learning probabilistic primitive action models (i.e. not hierarchical) presented by Pasula, Zettlemoyer, and Kaelbling (2007) develop interesting solutions for handling the specificities of FOND environments.

Work has been done on the automated learning of Behavior Trees (BTs), a common framework for implementing hierarchical, reactive operational models. Colledanchise, Parasuraman, and Ögren (2018) and Zhang et al. (2018) develop techniques to efficiently apply genetic programming to these structures.

Learning Problem

Operational Model

For an agent acting in its environment, an operational model, as defined by Ghallab, Nau, and Traverso (2014), represents an agent's knowledge about how to carry out a given activity in its environment. In this work, we are specifically interested in nondeterministic, fully observable environments.

We define an operational model O as an HTN-like structure which can be written as a tuple $O = (T, A, M)$ where T is a set of abstract tasks, A a set of primitive actions and M a set of possible methods decomposing the tasks $t \in T$ into subtasks $\{t_d \mid t_d \in \{T \cup A\}\}$. We consider the tasks to be possibly annotated with postconditions, similarly as Hogg, Muñoz-Avila, and Kuter (2008).

A primitive action $a \in A$ models the basic acting capabilities of the agent, and represent directly executable primitives. They are represented using an identifier and a set of parameters, such as $a = \text{action\_name}(\text{arg}_1, \ldots, \text{arg}_n)$. We do not assume any knowledge on the preconditions and effects of a primitive action. Furthermore, as we consider a non-deterministic environment, there is no guarantee that applying an action twice in the same state will produce an identical result.

An abstract (or non-primitive) task $t \in T$ is defined as a tuple $t = (Post_t, M_t)$, where $Post_t$ are the postconditions of the task, that is the predicates that must hold after executing t for it to be considered a success. These are especially important in nondeterministic domains. A task without postconditions is considered successful whenever one of its method has been executed without failure. M_t is the set of methods decomposing t.

A method $m \in M_t$ is a tuple $m = (Pre_m, N_m)$, where Pre_m are the preconditions of the method, and N_m is a task network defining a way to decompose t into subtasks. This task network represents a way to advance the task t towards its intended effects, in the case of a task with postconditions, or a way to achieve t if $Post_t = \emptyset$. A method is applicable in a given state s iff its preconditions hold in this state. For the sake of simplicity, we will assume that the set N_m of subtasks is totally ordered.

We define an acting problem as an initial task network N_p, representing the activity we wish to carry out, as well as an associated environment and a starting state s_i described by a set of boolean state variables. When trying to solve an acting problem p, we associate it to an operational model O that will be used to select the actions to execute. We consider that at any instant, the current state s is fully observable and that it only evolves when a primitive action is executed (i.e. there are no exogenous events). This evolution may however be nondeterministic.

Acting with an Operational Model

To act with an operational model, abstract tasks are refined down to a sequence of executable primitive actions. This refinement is achieved through the methods associated with the tasks: each time the agent must choose an action to execute, the current best applicable method decomposing the task at hand is chosen. The process is iteratively repeated until a sequence of tasks starting with a primitive action is obtained. This sequence is then executed until a non-primitive task is encountered, at which point the same process is applied again. The choice of the best method may be done either reactively or deliberatively. In the first case, the best method is chosen greedily according to some metric, while in the second one an acting engine, such as the one described by Patra et al. (2021), may use planning techniques to select a method considering the long term implications of the actions.

During the refinement of a task t using the method m starting in a state s, several failure types may occur:

1. A primitive action fails to be executed, causing the whole parent method to be considered a failure.
2. A non-primitive descendant t' from m has no applicable method. Then, the parent method of t' is considered to

have failed.

3. t can be completely refined, but the postconditions of t do not hold in the final state. This case is obviously valid only for a task t such that $Post_t \neq \emptyset$.

In the first two cases, when a method m decomposing a task t is detected as having failed, then some retrial strategy must be used to continue acting. A simple strategy could be to try another applicable method m' applicable in the current state s. If no such method is available in the hierarchy, then the failure is to be propagated upwards to the parent task t, until we reach either a task with an applicable method or the root task, the latter case leading to a failure to solve the acting problem.

Learning of an Operational Model

Inputs to the Learning Problem For the learning problem itself, we consider as input a fixed set A of primitive actions as well as a vocabulary of non-primitive tasks T_I.

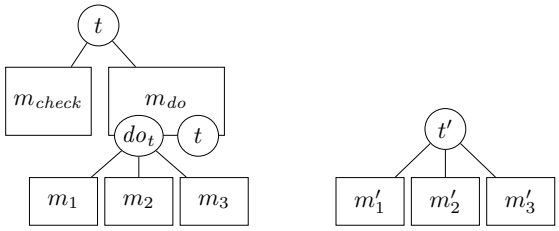
Recall that each primitive action has the form action_name(param$_1$, ..., param$_n$) and corresponds to a primitive directly executable by the agent. For a non-primitive task $t_I \in T_I$, we assume its postconditions $Post_{t_I}$ to be non-empty, as they are required to assess the success of the task execution in the agent's nondeterministic environment. Note that these postconditions might be learned independently of the methods, which is out of the scope of this paper.

Then, for each task $t_I \in T_I$, the agent is given a set D_{t_I} of demonstration traces from the tutor. Each trace $d \in D_{t_I}$ is an alternating sequence of states and tasks (either primitive or non-primitive), starting from a given initial state and ending in a final state in which the task t_I has been successfully achieved. d is considered optimal and maximally abstract with regard to the initial task vocabulary: for every demonstrated task, no other more abstract task from the initial vocabulary T_I may be used to abstract a subsequence of d, and each demonstration is optimal according to a chosen metric. For a case where actions are uniform in cost, one may naturally consider the total number of primitive actions required to achieve t_I as the optimality metric.

Learner Objectives Given a learned operational model and an acting problem, we say that the operational model solves the problem if, when used by the acting engine, it allows to refine a given task network into an executable sequence of primitives. Considering a successful solution to the problem, we define the soundness metric as the number of failed method execution (i.e. number of times we had to resort to recovery behavior) within the execution, to allow the evaluation of the quality of the learned preconditions of the methods. We also define the efficiency metric as the ratio between the cost of the executed behavior and the cost of executing an optimal model.

These metrics have been defined for a single acting problem. To assess the generalization capabilities of the model, we should consider a set of acting problems not encountered during learning. We define three metrics over this test set, namely the ratio of solved problems (coverage), the average number of method failures (average soundness), and the average efficiency.

Finally, to evaluate the algorithm as a whole, we should consider a set of demonstration and a set of acting problems. The performance of the algorithm is defined as the average performance of the model produced from the demonstrations on the acting problems, as defined in the previous paragraph.



(a) Task with postconditions. (b) Task without postconditions.

Figure 1: Structure of a task, with or without postconditions. The arguments of tasks and methods are omitted for clarity.

Approach to Model Learning

In our approach, we develop operational models designed to handle nondeterministic environments. To this end we distinguish tasks with and without postconditions, as shown in figure 1, presenting the structural differences a task t with postconditions and a task t' without.

In the case of t, ($Post_t \neq \emptyset$, figure 1a), we observe that t has two methods, m_{check} and m_{do}, and a right recursive structure. The former has no subtask, and its preconditions correspond to the postconditions of t, while the latter has no preconditions and two subtasks, do_t and t, recursively. The methods m_i of do_t encode different possible ways to act for the agent, with the intent to try and achieve t.

When decomposing t, m_{check} has priority over m_{do}, and is used to assess whether the goal associated with t has been achieved. We can note that t will thus be decomposed recursively until its associated goal is achieved, even in the case of unexpected events, provided there is a valid m_i to handle it, as in HTN-MAKER (Hogg, Kuter, and Muñoz-Avila 2009). Therefore, when decomposing t, in the case where its postconditions do not hold, we will choose one of the available method of do_t, refine and execute it, and then try to refine the original instance of t again, hoping to have achieved the intended effects. It should be noted that a method of a task may either represent a way to completely achieve it, or merely to advance it (or recover from mistakes), leveraging the recursive call to finally achieve the original endeavor.

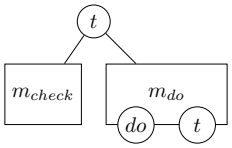
In the case of t', a task without postconditions ($Post_{t'} = \emptyset$, figure 1b), we have a task with a more standard structure, as a method will then always be considered successful if it is refined and executed to its completion. Indeed, it would not be possible to decide if the task should be recursively retried or not.

Requirements of Model Selection

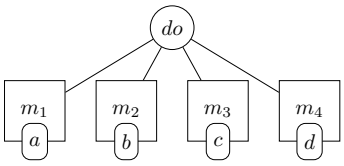
Let us now give an initial intuition about the shape of models that could be learned and the implication for the learn-

$t \rightarrow a\ b\ c$
$t \rightarrow a\ b\ d$

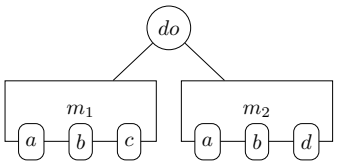
(a) Available demonstrations, showing that t was once achieved with the $a\ b\ c$ action sequence and once with the $a\ b\ d$ action sequence.



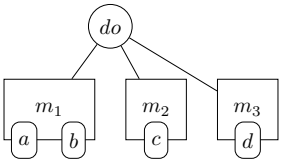
(b) Common right recursive structure, missing the refinement of the do task. A handful of possible models for the do task are shown in the subsequent figures.



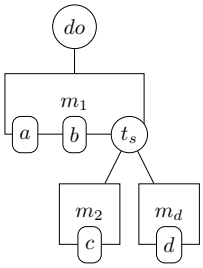
(c) Generic model where the actor might pick any of the primitive actions and rely on the recursive call in m_{do} to continue if needed.



(d) Model where each demonstration is fully encoded into a dedicated method.



(e) Intermediate model the common $a\ b$ sequence is grouped. It relies on the recursive call to t in m_{do} to produce a full sequence.



(f) Model where the $a\ b$ sequence is shared, requiring a synthetic task t_s

Figure 2: Illustration of the possible structures of the learned model for a simple learning task with two demonstration of how to perform a task t. Note that for conciseness the parameters or preconditions or the task and methods are omitted.

ing process. Figure 2 presents several possible models (figures 2c-2f) that could be generated based on two example sequences (figure 2a). All candidate models share the same recursive structure that we just saw (figure 2b) and only differ in the refinement of the do task.

The first one (2c) allows the choice of any of the four primitive actions $\{a, b, c, d\}$, each placed in a specific method. This model relies on the recursion to repropose the same choice until the task's postconditions are achieved. While this model allows building any sequence of actions it does not help the agent towards a meaningful sequence based on demonstrations. The second model (2d) takes the opposite approach and records each known trace into a method. This model is obviously strongly tied to the demonstration set and would fail to generalize to new problems. In between these two extreme, we have the models (2e) and (2f) that take different options to abstract common subsequences. The former encodes the repeated $a\ b$ sequence in a single method and relies on the recursive call to complete the sequence. The latter delays the choice between c and d to after the execution of a and b, using a synthetic task t_s.

These four models are just a handful of examples among the many possible models that could be generated. Denoting as Θ the set of possible models, the objective of a learning system is to find, or at least approach, the optimal model $\theta^* \in \Theta$

$$\theta^* = \arg\min_{\theta \in \Theta} cost(\theta)$$

where $cost(\theta)$ is a function that measures the cost of a particular model and should typically account for the size of the model as well as its capacity to solve both demonstrated and unseen problems. With this in mind we now turn our attention to the characterization of the set of possible models Θ. In a later section, we will propose a cost function to evaluate the models.

Generation of Candidate Operational Models

At a high level, the goal of the learning problem is to generate a model where some subtasks group together behaviors that happen repeatedly, with a sensible parameterization of methods depending on the current task, as well as reasonable preconditions to limit the search effort of the acting engine. It is easy to imagine an iterative process for this: we may extract some subsequence as a new task t, with a single method m_1 consisting of this subsequence. Once this is done, we may find another subsequence that achieves a similar goal, but using different tasks, thus allowing us to add a new method m_2 to t. Next, we may be able to extract a new task t' using t as a subtask of this method. This process is to be repeated until it produces a model that become too complex, not improving the score defined in the previous section.

Devising such an iterative learning process is however error-prone as it is easy to get stuck in a local minimum due to bad decision in the early stages of learning. Because the quality of a model depends not only on its structure, but also on its parameterization, in order to efficiently develop this latter part of the learning algorithm, we developed a method for generating a large number models so as to conduct a preliminary evaluation and relegate a more efficient exploration of the set of possible models to future work.

Considering a set of primitive actions and a set of demonstrations, it is easy to imagine a way to generate a number of operational models that can be used to achieve any of the given demonstration. For instance, we could start with a basic flat model, and add any number of possible subtasks whose methods correspond to arbitrary sequences of primitive actions, with possible duplications. This model can al-

ways fall back to choosing one of the single primitive actions if no subtask's method can be used, and therefore will always remain valid. As an infinite number of possible sequences can be used as subtasks, an infinite number of such models can be elicited, thus foregoing an exhaustive generation procedure. We therefore chose to restrict ourselves to a certain structure, described below, so that we could generate models exhaustively within this limited search space.

Considering an initial set of primitive actions $\{a, b, c, d\}$, we try and find all the ways to partition it, i.e. $P = (\{\{a\}\{b\}\{c\}\{d\}\}, \{\{a, b\}\{c\}\{d\}\}, \{\{a, b, c\}\{d\}\}, \dots)$. For each new set in P, we recursively apply the same process to each subpartition. The resulting sets will have a format such as $\{\{\{a\}\{b, c\}\}\{d\}\}$ each of which corresponding to a particular model (e.g. see the one presented in figure 3 for this particular example). To build such a model, we consider each level, starting from the first one (here, this first level contains the subpartitions $\{\{a\}\{b, c\}\}$ and $\{d\}$), considering that it shows different ways to decompose some task t that has been demonstrated. We then consider the subpartitions. If it is not further subpartitioned (as $\{d\}$ here), we create a method refining t down to the primitive action contained. Otherwise, we create a subtask st and a method that refines t into st and recursively apply the same procedure, considering that this subpartition shows ways to decompose st.

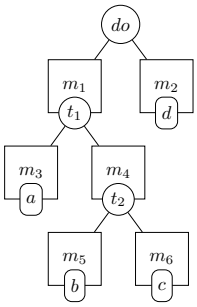


Figure 3: Model corresponding to the $\{\{\{a\}\{b, c\}\}\{d\}\}$ partitioning.

With this procedure, we will obtain models that are limited, as they will, e.g., never use the same primitive to decompose two different subtasks, nor will they contain methods that decompose as sequences of tasks. However, this generation process produces a number of models that scales exponentially with the number of primitive tasks given as input, due to the number of possible subpartitions of a set, therefore quickly making the use of the generated model set for evaluating our algorithm impractical. Adding more possible model structures would only worsen this issue, thus warranting the use of some local search procedure.

While this generation procedure allows us to generate new model structures, we still need to parameterize their tasks and methods to evaluate them properly on the existing demonstration set, and to eventually use the corresponding models for acting. This parameterization as well the extraction of methods' preconditions is detailed in the next section.

Parameters & Preconditions Extraction

For each method in the model, we need to identify the parameters that should be passed to its subtasks. When considering the abstract tasks, we need to distinguish two cases:

1. The set T_I of tasks given as the starting vocabulary for which arguments and postconditions are known.
2. The tasks that are inferred during learning, by grouping common subsequences for example, for which arguments must be extracted. This second type of tasks are called *synthetic tasks* and their set is called T_S. For these tasks, we do not consider the identification of intended effects in this work.

Process Overview Parameterization is a bottom-up, iterative process that selects a task or method whose subcomponents are already parameterized (i.e. their parameters are given or were previously identified). Those subcomponents are analyzed to identify the parameters of the current task/method, which in turn will enable the analysis of tasks/methods higher-up in the hierarchy.

1. We first consider any method m whose subtasks are all parameterized. For each such method we:
   - Identify its parameters based on the parameters of its subtasks
   - Extract its preconditions by considering each state that precedes an instance of the method in the demonstrations.
2. For any synthetic task $t \in T_S$ whose methods are all parameterized (as a result of the previous step), we identify the parameters of t based on the ones of its methods.
3. For any initial task $t \in T_I$ whose methods are all parameterized (as a result of the previous step), we associate the parameters of t to the parameters of its methods.

We repeat this process until all tasks and methods are parameterized, resulting in a bottom up identification of the parameters of the tasks and methods.

For this process, we need to know how our model would have been used if it had to generate the demonstrations, while not having yet access to its parameterization. To this end, we use the technique developed for HTN plan verification (Höller et al. 2021) to obtain such a trace from an HTN planner. To use this strategy with a candidate model O_c, we generate an HTN model by removing every argument and method precondition. We also replace each instance of a primitive action a by a new task a_{obs} whose methods each correspond to a single observation of an instance of a in the currently considered demonstration d. Each such method has a single precondition: being currently at the right point in the sequence. This means the method containing the 6th observation in the demonstration can only be selected between adding the 5th and 7th to the final plan.

Identifying the Arguments of a Method from its Subtasks When extracting the arguments of a method, we

leverage the fact that we know the arguments of its subtasks as well as, for each method demonstration instance, their mapping to the ground arguments.

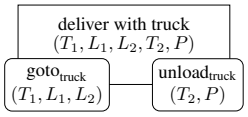
This allows us to unify the arguments of the subtasks as much as possible, by identifying the subtask arguments that are always bound to the same constant in each example. For instance, consider for a method m with $st_1(X_1, X_2)$ and $st_2(Y_1, Y_2, Y_3)$ as its subtasks. In this example, without any unification, the arguments of m could be $(X_1, X_2, Y_1, Y_2, Y_3)$, that is, the union of the arguments of its subtasks. If X_1 and Y_1 are bound to the same constant in each instantiation of the method (over all traces), we can unify them using a new variable Z_1. Therefore, the arguments of m become (Z_1, X_2, Y_2, Y_3), and its subtasks become parameterized as $st_1(Z_1, X_2)$ and $st_2(Z_1, Y_2, Y_3)$

To make the explanation clearer, let us consider an example *deliver with truck* method as presented in figure 4b, considering we also have two demonstration instances of this method, as presented in figure 4a. Since T_1 and T_2 are always bound to the same variable in each of the examples, we can introduce a new variable T, to replace each occurrence of T_1 or T_2. This leads to the method presented in figure 4c.

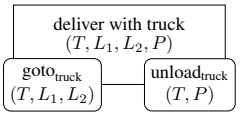
$$E_1 : \begin{cases} s_0^1 \rightarrow \text{goto}_{\text{truck}}(t_1, l_1, l_2) \\ \quad \hookrightarrow s_1^1 \rightarrow \text{unload}_{\text{truck}}(t_1, p_1) \rightarrow s_2^1 \end{cases}$$

$$E_2 : \begin{cases} s_0^2 \rightarrow \text{goto}_{\text{truck}}(t_2, l_3, l_4) \\ \quad \hookrightarrow s_1^2 \rightarrow \text{unload}_{\text{truck}}(t_2, p_1) \rightarrow s_2^2 \end{cases}$$

(a) Example traces associated to the method.



(b) Before unification.



(c) After unification.

$$\underbrace{\left\{\begin{array}{l} \text{package\_in}\,(t_1, p_1) \\ \text{truck\_at}(t_1, l_1) \\ \text{road}(l_1, l_2) \\ \text{sunny}(l_2) \\ \text{truck\_at}(t_2, l_2) \\ \begin{pmatrix} T \mapsto t_1 & P \mapsto p_1 \\ L_1 \mapsto l_1 & L_2 \mapsto l_2 \end{pmatrix} \end{array}\right\}}_{s_0^1} \quad \underbrace{\left\{\begin{array}{l} \text{package\_in}(t_2, p_1) \\ \text{truck\_at}(t_2, l_3) \\ \text{road}(l_3, l_4) \\ \text{cloudy}(l_2) \\ \text{truck\_at}(t_3, l_2) \\ \begin{pmatrix} T \mapsto t_2 & P \mapsto p_1 \\ L_1 \mapsto l_3 & L_2 \mapsto l_4 \end{pmatrix} \end{array}\right\}}_{s_0^2}$$

(d) State in the examples, with the mapping from method arguments to constants.

Figure 4: Example method structure before and after argument identification.

Extracting the Preconditions of a Method Having identified the arguments of a method m, we can extract its preconditions. As we know the mapping between method arguments and constants, we can easily filter the predicates in the states preceding every instance of m to keep only the ones fully specified by its arguments. Then, we take the intersection of these sets of predicates.

Consider the method *deliver with truck* from figure 4, and the starting states as defined in figure 4d. Then, using our knowledge of the mappings between the method parameters and the examples' constants, we can first exclude the last *truck_at* predicate from both examples, as at least one constant is not unified with a parameter. The preconditions of the method are therefore the intersection of the lifted and filtered states, i.e. $Pre = \{\text{package\_in}(T, P), \text{truck\_at}(T, L_1), \text{road}(L_1, L_2)\}$.

Associating the Arguments of a Method to the *Known* Arguments of the Task it achieves When learning methods for a task t in our initial vocabulary T_I, we have to map the (known) arguments of t to the arguments of its methods, which were extracted as described previously. These arguments are mapped in the same way as when unifying arguments for methods subtasks. Having some examples of decompositions of t, we follow the same logic as for the subtasks in a method: all parameters that are always bound the same value are unified.

Consider a task $t(Z_1, Z_2, Z_3)$ being decomposable with two methods, $m_1(X_1, X_2)$ and $m_2(Y_1, Y_2, Y_3, Y_4)$. Based on the traces of equation 1, we can replace X_1 and Y_1 with Z_1, X_2 with Z_2 and Y_3 with Z_3. We can therefore rewrite the methods' arguments as $m_1(Z_1, Z_2)$ and $m_2(Z_1, Y_2, Z_3, Y_4)$.

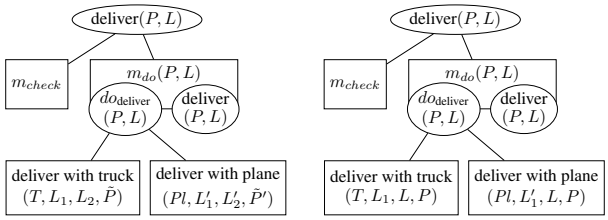
$$\begin{bmatrix} E_1 = t(a,b,c) : m_1(a,b),\ t(a,b,c) : m_2(a,c,c,d) \\ E_2 = t(x,y,x) : m_1(x,y) \\ E_3 = t(m,n,o) : m_2(m,n,o,q) \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} E_1 : \begin{cases} (X_1 \mapsto a), (X_2 \mapsto b) \\ (Y_1 \mapsto a), (Y_2 \mapsto c), (Y_3 \mapsto c), (Y_4 \mapsto d) \\ (Z_1 \mapsto a), (Z_2 \mapsto b), (Z_3 \mapsto c) \end{cases} \\ E_2 : \begin{cases} (X_1 \mapsto x), (X_2 \mapsto y) \\ (Z_1 \mapsto x), (Z_2 \mapsto y), (Z_3 \mapsto x) \end{cases} \\ E_3 : \begin{cases} (Y_1 \mapsto m), (Y_2 \mapsto n), (Y_3 \mapsto o), (Y_4 \mapsto q) \\ (Z_1 \mapsto m), (Z_2 \mapsto n), (Z_3 \mapsto o) \end{cases} \end{bmatrix} \quad (1)$$

To illustrate this with an example, let us consider the task described in figure 5a that represents the act of delivering a package P to a location L. It can be achieved either through a method delivering it via a truck (the same as described in the previous paragraph) or by a plane. Assuming we have some supporting examples, we can unify the arguments as presented in figure 5b.

Extracting the Arguments of a Synthesized Task from the Arguments of its Methods We now turn our attention to the identification of the arguments of synthetic tasks. This is done by leveraging the context of the instances of t to find method parameters that are always bound to the same constant in a given context.

Let us consider a new task $t \in T_S$, for which we do not know the arguments, and whose methods' arguments have been identified. We assume that t has a parent task t_p, possibly several levels higher in the hierarchy, whose arguments

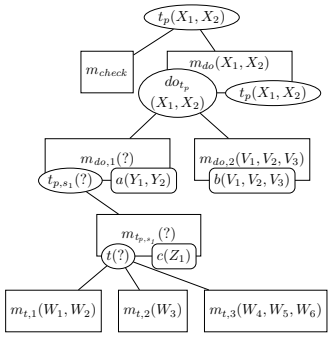


(a) Example task before unification.

(b) Example task after unification.

Figure 5: Example task before and after unification.

are known (i.e. $t_p \in T_I$). We note $\mathcal{C}_t$ the context surrounding an instantiation of t, defined as the variables bound from the instantiation of t_p and from all the definitive parts of the methods' instantiations on the path down to t. Figure 6a shows an example for such a task t, for which a context would be an instantiation of X_1, X_2, Y_1, Y_2, Z_1.



(a) Example task structure for contexts.

$$\text{args}(t) = \{a_1, a_2\}$$
$$a_1 \mapsto \{W_1, W_3\}$$
$$a_2 \mapsto \{W_4, W_5\}$$

(b) Example of a valid parameterization, assuming some supporting contexts.

Figure 6: Example method structure and parameterization.

After extracting all the contexts from the demonstrations, we wish to find a valid and relevant parameterization of t, such as the one of figure 6b. A valid parameterization is a mapping from $args(t)$ (the set of arguments of t) to $args(M_t)$ (the set of arguments of its methods) such that $\forall a_i \in args(t), \exists b_j \in args(M_t) : a_i \mapsto b_j$.

A single task parameter $a_i \in args(t)$ may map to several method arguments in $args(M_t)$. The core process for identifying the a_i arguments lies in identifying subsets of the methods' parameters that can be bound to the same task parameter. Intuitively, two methods parameters b_j and b_k can be bound to the same task parameter a_i iff:

- there is a context $\mathcal{C}_t$ where both b_j and b_k map to the same constant (positive example, noted $b_j \equiv_{\mathcal{C}_t} b_k$), and
- there is no context $\mathcal{C}'_t$ where b_j and b_k map to a different constant (counter-example, noted $b_j \not\equiv_{\mathcal{C}_t} b_k$).

More formally, a parameterization of a task can be seen as a set of grouped methods parameters $\mathcal{U} = \{U_0, \ldots, U_n\}$ where each U_i is the subset of $args(M_t)$ that the a_i parameter maps to. To be valid, a parameterization $\mathcal{U}$ must be such that:

$$b_j, b_k \in U_i \Rightarrow \left\{ \begin{array}{l} \exists \mathcal{C}_t : b_j \equiv_{\mathcal{C}_t} b_k \\ \nexists \mathcal{C}_t : b_j \not\equiv_{\mathcal{C}_t} b_k \end{array} \right\}$$
$$U_i \cap U_j = \emptyset \quad \text{if } i \neq j$$

Among all the possible valid parameterizations, the one leading to the smallest number of task arguments and containing the largest total number of method arguments is to be favored, in order to find relevant unifications. In practice, we implemented it by enumerating possible sets U_i, which we then try and combine together.

Evaluation of Operational Models

Now that we are able to generate a set of full operational models from an initial structure candidate, we ought to compare these in order to find the best one among all candidates.

The metric used to evaluate the operational model should compromise between models that generalize too much and models that overfit. Indeed, as described earlier, we wish to allow our agent to generate solutions to new acting problems, but we also want these solutions to be sound, i.e., with limited failures during execution.

To devise such a metric, we turn to the Minimum Description Length (MDL) principle (Grünwald 1996). This principle, coming from information theory, states that learning can be viewed as a form of data compression, as both intend to find some regularity in some source material. Therefore, in this framework, the best model is the one that can compress the data the most. Furthermore, as the total size of the compressed data includes the size of the model itself, used to reconstruct the source, this principle naturally guards against overly specific models.

Example uses of this technique range from learning Context Free Grammars (CFGs) of which HTNs, and thus our operational models, are close (Sapkota, Bryant, and Sprague 2012), to finding common graph patterns (Bariatti, Cellier, and Ferré 2020).

Framing our problem in the context of this MDL principle, we can say that a desirable model should be able to "compress" the demonstrations, while not being so specific that nothing else can be generated, thus limiting the complexity and therefore the size of this model.

Assuming we have generated an operational model candidate O_c as part of our search process, we define the description length of the operational model $L_{\text{opmod}}(O_c)$ and the description length of the demonstrations in $\mathcal{D}$ knowing O_c, written as $L_{\text{dem}}(\mathcal{D}|O_c)$. In order to compute the global length of the model and the demonstration set, we combine these two metrics, using a factor α to balance their relative importance, as defined in equation 2.

$$L(O_c, \mathcal{D}) = \alpha L_{\text{opmod}}(O_c) + L_{\text{dem}}(\mathcal{D}|O_c) \qquad (2)$$

To compute $L_{\text{opmod}}(O_c)$, we consider a simple (arbitrary) alphabet to describe our candidate model O_c and compute the length of an optimal encoding of this description of our model. Considering a random variable X_{O_c} that takes as possible values the symbols in our alphabet, information theory tells us that the entropy $H(X_{O_c})$ of this variable bounds the expected codeword length of our optimal code

as $H(X_{O_c}) \leq \mathbb{E}(L) \leq H(X_{O_c}) + 1$. Noting $x_1, \ldots, x_n$ the symbols of our alphabet, and their occurrence probability $P(x_1), \ldots, P(x_n)$, the entropy formula is: $H(X_{O_c}) = -\sum_{i=1}^{n} P(x_i) \log(P(x_i))$. As we know the frequency of each symbol in our model as well as their total number, we can therefore compute a bound on the optimal model encoding length as in the example below.

As an example, consider the structure of the simple model learned for performing a task t shown in figure 2c. Viewing our model as a grammar-like structure, we can describe it with the following rule: $do : a \mid b \mid c \mid d$;. The frequency table of each symbol is given in the table in figure 7a, giving us the entropy $H(X_{O_c})$. Given that there are 9 symbols occurrence in the rule, this bounds the optimal value of $L_{\text{opmod}}(O_c)$ as shown in the equation in figure 7b. As this value will only be used for comparison, we arbitrarily choose the lower bound as L_{opmod}.

Symbol	do	a	b	c	d	|	;	Total
Frequency	1	1	1	1	1	3	1	9

(a) Frequency table

$$H(X_{O_c}) = -\left(6\left(\frac{1}{9}\log\left(\frac{1}{9}\right)\right) + \frac{3}{9}\log\left(\frac{3}{9}\right)\right)$$
$$= 2.64 \text{ bits}$$
$$L_{\text{opmod}}(O_c) \in [9 \times 2.64, 9 \times 3.64] = [23.76, 32.76] \text{ bits}$$

(b) Entropy and model length bounds.

Figure 7: Model length calculation for the one presented in figure 2c.

To compute $L_{\text{dem}}(\mathcal{D} \mid O_c)$, we need to evaluate the cost of encoding a trace $d \in \mathcal{D}$ based on our operational model. The trace can be mapped to a sequence of refinements from the original task to the primitive sequence. When planning or acting, each refinement of a task constitutes a choice point at which the engine must select one method among the applicable ones in the current state. Recalling all choices, we can reconstruct the refinement sequence and the corresponding trace. We define $L_{\text{dem}}(\mathcal{D} \mid O_c)$ as the encoding size of all choice points, averaged over all examples. This leads us to equation 3, where $\mathcal{C}$ is the set of choices an optimal planner has to make to reconstruct d from O_c, and $M_{\text{app},cp}$ is the set of applicable methods at a choice point cp.

$$L_{\text{dem}}(\mathcal{D} \mid O_c) = \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} \sum_{cp \in \mathcal{C}} \log\left(|M_{\text{app},cp}|\right) \qquad (3)$$

As an example, we will study the set of demonstration presented in figure 2a using again the model of figure 2c, assuming the methods have no preconditions. For each of the sequences in the demonstration set, we know that we have to choose three times among four methods, therefore $L_{\text{dem}} = \frac{1}{2}\left(3\log(4) + 3\log(4)\right) = 6$ bits

To show how this metric can be used, table 1 presents the different values computed for the examples presented in figure 2, using $\alpha = 0.505$. In order to normalize the tree length and the sequence, based on the value obtained from the most basic model, i.e. the one presented in figure 2c. Using $\frac{1}{2}\alpha$ instead of α reduces the relative importance accorded to the model size compared to its efficiency at compressing the demonstrations.

O_c	$L_{\text{opmod}}(O_c)$	$L_{\text{dem}}(\mathcal{D}\|O_c)$	$L(O_c, \mathcal{D})$	
			α	$\frac{1}{2}\alpha$
Fig. 2c	23.76	12	24	18
Fig. 2d	24.57	2	14.4	8.2
Fig. 2e	22	6.34	13.1	11.89
Fig. 2f	29.21	2	16.75	9.37

Table 1: Description length in bits for the examples presented in figure 2.

As the previous paragraph should have made clear, we need to know the choices that are required while acting with our candidate model O_c, in order to generate a given demonstration trace d, for which we use again the technique presented by Höller et al. (2021). We can then recover the choices made and their associated state by analyzing the resulting trace. Using an optimal planner, the computed demonstration length will be the true value of $L_{\text{dem}}(\mathcal{D} \mid O_c)$, while a non-optimal one may lead to an overestimation.

Preliminary Evaluation

While we are still implementing the ideas presented in this paper, we have been able to obtain some preliminary results on the metric to evaluate a candidate model for which preconditions and parameters have been identified as previously discussed.

A preliminary evaluation using these models on a given set of demonstration traces show that our metric appears indeed to favor structures that allow to efficiently generate the demonstrations, while still being of limited complexity, i.e. not favoring the deepest hierarchies, nor ones that are completely flat.

While we have shown some results earlier in table 1, the examples were too simple to show the interest of the metric, favoring the "lookup" model. However, working with more realistic examples, even simple ones, show the interest of our metric. Table 2 shows the results from an example dataset from a nondeterministic logistics domain with six primitive actions and three demonstration sequences long of respectively 11, 18 and 23 tasks. In the same way as before, we compute $\alpha = 1.35$ to normalize the model length based on the flat model. As we can see, reducing the model size relative importance α allows our metric indeed to favor a balanced tree. Evaluation on datasets with smaller traces for which it is possible to analyze all the generated tree structures by hand shows that the best scoring tree is indeed the one most efficient to regenerate the demonstrations.

Conclusion and Future Work

Our current work is focused on implementing a basic reactive acting engine to evaluate the currently generated models

O_c	$L_{\text{opmod}}(O_c)$	$L_{\text{dem}}(\mathcal{D}\|O_c)$	$L(O_c, \mathcal{D})$	
			α	$\frac{1}{2}\alpha$
Flat	**24.57**	33.15	**66.3**	49.7
Lookup	141.2	**1.58**	192.2	96.9
Balanced	37.71	20.98	71.9	**46.4**

Table 2: Description length in bits for some real examples.

on a test set of acting problems. This will allow us to conduct a first partial evaluation of the performance of the higher scoring models on some unseen acting problems. With this done, we will implement a local search strategy in lieu of the present crude generation method, to generate more complex models by removing the current structural restrictions. The next step will then be to integrate these operational models in a deliberative acting engine in order to leverage its lookahead capabilities, to make a better evaluation of the real world applicability of our models.

While we are still working on the implementation of the algorithm presented in the previous parts, there are some limitations that we are aware of and intend on addressing when the main algorithm will have been implemented. First, the current approach for extracting method preconditions is limited in which part of the preceding states is considered, which could be improved by integrating deictic references (Pasula, Zettlemoyer, and Kaelbling 2007), and superfluous predicates may be removed with an adaptation of the approach of Martínez (2017). Second, we wish to extract the postconditions of synthetic tasks with possibly disjunct intended effects, which may be done similarly to the extraction of the set of changes of an action of Pasula, Zettlemoyer, and Kaelbling (2007). Finally, it would be interesting to try and relax the assumptions made on the demonstration traces format.

References

Bariatti, F.; Cellier, P.; and Ferré, S. 2020. GraphMDL: Graph Pattern Selection Based on Minimum Description Length. In *IDA 2020 - Symposium on Intelligent Data Analysis*. Konstanz, Germany.

Colledanchise, M.; Parasuraman, R.; and Ögren, P. 2018. Learning of Behavior Trees for Autonomous Agents. *IEEE Transactions on Games*, 11(2): 183–189.

Fine-Morris, M.; and Muñoz-Avila, H. 2019. Learning Domain Structure in HGNs for Nondeterministic Planning. In *Proceedings of the 2nd ICAPS Workshop on Hierarchical Planning (HPlan 2019)*, 22–30.

Geib, C. W.; and Goldman, R. P. 2011. Recognizing Plans with Loops Represented in a Lexicalized Grammar. In Burgard, W.; and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. *Automated Planning and Acting*. Cambridge: Cambridge University Press. ISBN 978-1-139-58392-3.

Grünwald, P. 1996. A Minimum Description Length Approach to Grammar Inference. In Carbonell, J. G.; Siekmann, J.; Goos, G.; Hartmanis, J.; Leeuwen, J.; Wermter, S.; Riloff, E.; and Scheler, G., eds., *Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing*, volume 1040, 203–216. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-60925-4 978-3-540-49738-7.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning Hierarchical Task Networks for Nondeterministic Planning Domains. In Boutilier, C., ed., *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 1708–1714.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, 950–956. Chicago, Illinois: AAAI Press. ISBN 978-1-57735-368-3.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2021. Compiling HTN Plan Verification Problems into HTN Planning Problems. In *Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)*, 8–15.

Kantharaju, P.; Ontañón, S.; and Geib, C. W. 2019. Extracting CCGs for Plan Recognition in RTS Games. In Guzdial, M.; Osborn, J. C.; and Snodgrass, S., eds., *Proceedings of the 2nd Workshop on Knowledge Extraction from Games Co-Located with 33rd AAAI Conference on Artificial Intelligence, KEG@AAAI 2019, Honolulu, Hawaii, January 27th, 2019*, volume 2313 of *CEUR Workshop Proceedings*, 9–16. CEUR-WS.org.

Martínez, D. 2017. *Learning Relational Models with Human Interaction for Planning in Robotics*. Ph.D. thesis, Universitat Politècnica de Catalunya.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning Symbolic Models of Stochastic Domains. *Journal of Artificial Intelligence Research*, 29: 309–352.

Patra, S.; Mason, J.; Ghallab, M.; Nau, D.; and Traverso, P. 2021. Deliberative Acting, Planning and Learning with Hierarchical Operational Models. *Artificial Intelligence*, 299: 103523.

Sapkota, U.; Bryant, B. R.; and Sprague, A. 2012. Unsupervised Grammar Inference Using the Minimum Description Length Principle. In Perner, P., ed., *Machine Learning and Data Mining in Pattern Recognition*, Lecture Notes in Computer Science, 141–153. Berlin, Heidelberg: Springer. ISBN 978-3-642-31537-4.

Zhang, Q.; Yao, J.; Yin, Q.; and Zha, Y. 2018. Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution. *Applied Sciences*, 8(7): 1077.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning Hierarchical Task Network Domains from Partially Observed Plan Traces. *Artificial Intelligence*, 212: 134–157.

Zhuo, H. H.; Peng, J.; and Kambhampati, S. 2019. Learning Action Models from Disordered and Noisy Plan Traces. *arXiv:1908.09800 [cs]*.

On the Efficient Inference of Preconditions and Effects of Compound Tasks in Partially Ordered HTN Planning Domains

Conny Olz[1] **and Pascal Bercher**[2]

[1] Ulm University
[2] The Australian National University
conny.olz@uni-ulm.de, pascal.bercher@anu.edu.au

Abstract

Recently, preconditions and effects of compound tasks based on their possible refinements have been introduced together with an efficient inference procedure to compute a subset of them. However, they were restricted to total-order HTN planning domains. In this paper we generalize the definitions and algorithm to the scenario of partially ordered domains.

Introduction

In Hierarchical Task Network (HTN) planning we refine an initial abstract task step-by-step into a more fine-grained description until an executable sequence of actions results (Erol, Hendler, and Nau 1996; Ghallab, Nau, and Traverso 2004; Bercher, Alford, and Höller 2019).

Compound tasks together with decomposition methods govern the refinement process. In many HTN formalizations one does not model or specify concrete preconditions or effects for compound tasks like for primitive ones. Instead, they are only given implicitly via the actions deeper down in the hierarchy. Recently, Olz, Biundo, and Bercher (2021) defined preconditions and effects of compound tasks that can be inferred based on the decomposition structure. Besides analyzing the computational complexity they also introduced a procedure to compute a subset of these in polynomial time but they were restricted to totally ordered (t.o.) HTN planning domains. We extend their work by generalizing the definitions and algorithm to also work with partially ordered (p.o.) domains.

As pointed out by Olz, Biundo, and Bercher (2021) the potential applications of such inferred preconditions and effects are manifold. In the context of modeling assistance they might reveal unintended modeling effects or errors and a study indicated that they can help to better comprehend a given domain model (Olz et al. 2021). More prominently, resemblances of the preconditions and effects considered by us were already exploited to speed up several kinds of planning systems (Tsuneto, Hendler, and Nau 1998; Nau et al. 2003; Clement, Durfee, and Barrett 2007; Waisbrot, Kuter, and Könik 2008; Goldman and Kuter 2019; Schreiber, Pellier, and Fiorino 2019; Magnaguagno, Meneguzzi, and de Silva 2021; Schreiber 2021). By extending the inference to p.o. domains we make them also available for planning systems solving p.o. problems like SHOP2 (Nau et al. 2003), FAPE (Dvořák et al. 2014), PANDA$_3$-POCL (Bercher et al. 2017), PANDA$_\pi$-SAT (Behnke, Höller, and Biundo 2019), PANDA$_\pi$-pro (Höller et al. 2020), SIADEX (Fernandez-Olivares, Vellido, and Castillo 2021), pyHiPOP (Lesire and Albore 2021), and PDDL4J (Pellier and Fiorino 2021). Their exploitation in p.o. systems should however be done with care as discussed later in the paper.

One further utilization especially for the p.o. case that we would like to bring up is that inferred preconditions and effects bear useful information for turning a p.o. domain or problem into t.o. while preserving specific properties. Planners can then make use of the special case to solve such problems more efficiently.

For an overview of related work concerning the inference of preconditions and effects we would like to refer to Olz, Biundo, and Bercher and add the work by Magnaguagno, Meneguzzi, and de Silva (2021) that has been published in the meantime. Their lifted planner HyperTensioN infers preconditions of compound tasks similarly to Olz, Biundo, and Bercher but is also restricted to t.o. domains.

HTN Planning Formalism

Our formalism is based on the one by (Bercher, Alford, and Höller 2019). A partially ordered (p.o.) HTN planning domain is a tuple $\mathcal{D} = (F, A, C, M)$, where F is a finite set of facts, A are *primitive tasks*, and C the set of *compound tasks*. Primitive tasks $a = (prec, add, del) \in A$ – also called actions – are described by their *preconditions* $prec(a) \subseteq F$ and their add and delete *effects* $add(a), del(a) \subseteq F$, resp. As in STRIPS planning, an action $a \in A$ is *applicable* in a state $s \in 2^F$ if $prec(a) \subseteq s$. The application of it in s (if applicable) produces the successor state $\delta(s, a) = (s \setminus del(a)) \cup add(a)$. Accordingly, the application of a sequences of actions $\bar{a} = \langle a_0 \dots a_n \rangle$ with $a_i \in A$ is possible in a state s_0 if a_0 is applicable in s_0 and for all $1 \leq i \leq n$, a_i is applicable in $s_i = \delta(s_{i-1}, a_{i-1})$. The second type of tasks are compound tasks, which serve as a collection of primitive and/or compound tasks organized in *task networks*. A task network is a triple $tn = (T, \prec, \alpha)$, where T is a (possibly empty) set of identifiers (ids) that are mapped to tasks by a function $\alpha : T \to A \cup C$. Therefore, a single task can be contained in a task network more than once. A set of ordering constraints $\prec : T \times T$ defines a partial order on the identifiers. *Decomposition methods* M specify how exactly

compound tasks were decomposed. A method $m \in M$ is a pair (c, tn) of a compound task $c \in C$ and a task network. It decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ into a task network $tn_2 = (T_2, \prec_2, \alpha_2)$ if $t \in T_1$ with $\alpha_1(t) = c$ and there is a task network $tn' = (T', \prec', \alpha')$ equal to tn but using different ids, so $T_1 \cap T' = \emptyset$. The task network tn_2 is defined as

$$\begin{aligned} tn_2 =& ((T_1 \setminus \{t\}) \cup T', \prec' \cup \prec_D, (\alpha_1 \setminus \{t \mapsto c\}) \cup \alpha') \\ \prec_D =& \{(t_1, t_2) \mid (t_1, t) \in \prec_1, t_2 \in T'\} \cup \\ & \{(t_1, t_2) \mid (t, t_2) \in \prec_1, t_1 \in T'\} \cup \\ & \{(t_1, t_2) \mid (t_1, t_2) \in \prec_1, t_1 \neq t \wedge t_2 \neq t\} \end{aligned}$$

So if a compound task c is decomposed, it is removed from the task network and the tasks of the chosen method's subnetwork were added together with ordering constraints that hold for c. When a task network tn can be decomposed into a task network tn' by applying the method m to a task with the identifier t, we write $tn \rightarrow_{t,m} tn'$; if it is possible using several methods in sequence, we write $tn \rightarrow tn'$.

An HTN planning problem $\Pi = (\mathcal{D}, s_I, tn_I, g)$ contains additionally an initial state $s_I \in 2^F$, an initial task network tn_I, and a goal description $g \subseteq F$. A solution to a Π is a task network $tn = (T, \prec, \alpha)$ if and only if

1. it can be reached via decomposing tn_I (i.e. $tn_I \rightarrow tn$),
2. all task are primitive ($\forall t \in T : \alpha(t) \in A$), and
3. there exists a sequence $\langle t_1 t_2 \dots t_n \rangle$ of the task identifiers in T that is in line with $\prec$, and the application of $\langle \alpha(t_1)\alpha(t_2)\dots\alpha(t_n) \rangle$ in s_0 results in a goal state $s \supseteq g$.

To ease notation we additionally define the following: A task network containing only one task $c \in A \cup C$, i.e., $(\{t\}, \emptyset, \alpha(t) = c)$, is denoted $\langle c \rangle$. A *t.o. refinement* of some compound task $c \in C$ is a sequence of primitive tasks $\bar{a} = \langle a_1 \dots a_n \rangle$ if and only if there exists a task network $tn = (T, \prec, \alpha)$, $\langle c \rangle \rightarrow tn$ and there exists a sequence $\langle t_1 \dots t_n \rangle$ of the ids in T that is in line with $\prec$ so that $\langle \alpha(t_1) = a_1 \dots \alpha(t_n) = a_n \rangle$. Lastly, by $c \in tn$ for some task $c \in A \cup C$ and task network $tn = (T, \prec, \alpha)$ we abbreviate that there exists a task identifier $t \in T$ so that $\alpha(t) = c$.

Preconditions and Effects of Compound Tasks

The definitions of preconditions and effects of compound tasks for totally ordered task networks by Olz, Biundo, and Bercher (2021) were based on sets of states that enable the execution of such tasks and the states in which an application of a refinement can result. We adapt these two definitions to p.o. domains in the following. Therefore, consider a domain $\mathcal{D} = (F, A, C, M)$ and a compound task $c \in C$.

The set of *executability-enabling states* of c is

$$E(c) = \{s \in 2^F \mid \exists \text{ t.o. refinement of } c \text{ applicable in } s\}.$$

The set of *resulting states* of c w.r.t. some state $s \in 2^F$ is

$$R_s(c) = \{s' \in 2^F \mid \exists \text{ t.o. refinement appl. in } s \text{ res. in } s'\}.$$

Based on these two updated definitions for p.o. domains, the definitions of preconditions and effects of compound tasks can be used without further adaptions. Olz, Biundo, and Bercher (2021) defined several types, e.g., depending on whether they are dependent or independent of a state and they differentiate between effects and postconditions. We repeat only those that are relevant for this paper.

State-independent positive and negative effects (cf. their Def. 4) of a compound task c are facts that are added or deleted, resp., by the successful execution of a refinement of c, independent of the state in which the task is executed, i.e.,

$$\begin{aligned} \mathit{eff}^+_*(c) &:= \Big(\bigcap_{s \in E(c)} \bigcap_{s' \in R_s(c)} s'\Big) \setminus \bigcap_{s \in E(c)} s \\ \mathit{eff}^-_*(c) &:= \bigcap_{s \in E(c)} \Big(F \setminus \bigcup_{s' \in R_s(c)} s'\Big) \end{aligned}$$

if $E(c) \neq \emptyset$, otherwise $\mathit{eff}^{+/-}_*(c) := \mathit{undef}$.

Possible state-independent effects (cf. Def. 5) of a compound task c are not guaranteed to hold (or not hold, resp.,) after *every* refinement of c but after at least one:

$$\begin{aligned} \mathit{poss\text{-}eff}^+_*(c) &:= \bigcup_{s \in E(c)} \Big(\bigcup_{s' \in R_s(c)} s' \setminus s\Big) \\ \mathit{poss\text{-}eff}^-_*(c) &:= \bigcup_{s \in E(c)} \Big(\Big(\bigcup_{s' \in R_s(c)} (F \setminus s')\Big) \cap s\Big) \end{aligned}$$

if $E(c) \neq \emptyset$ and $\mathit{poss\text{-}eff}^{+/-}_*(c) := \mathit{undef}$ otherwise.

Mandatory preconditions (cf. Def. 6) of c are facts that hold in every state for which there exists an executable refinement. So, they are required in every state in which a refinement of c shall be executed: $\mathit{prec}(c) := \bigcap_{s \in E(c)} s$ if $E(c) \neq \emptyset$ and $\mathit{prec}(c) := \mathit{undef}$ otherwise.

Important: These definitions for p.o. domains are actually not correct in the sense that in a given p.o. problem we do not consider c on its own but rather within a task network that usually contains further tasks unordered w.r.t. c. Those (or their subtasks) can interlock with the subtasks of c to enable the execution of some refinement. So the executability-enabling or resulting states of c (leaving open how exactly they are defined in such cases) can look totally different depending on which other tasks are present in a task network.

We introduced these definitions solely to define a weaker version that can be computed in polynomial time as also done by Olz, Biundo, and Bercher (2021) in the t.o. case. They showed that determining the preconditions and effects in a t.o. domain is computationally as hard as solving the respective planning problem, basically because one needs to check whether there is at least one executable refinement[1]. For practical exploitation this can often be too costly. Therefore, a relaxation has been introduced, which allows to find a subset of the original preconditions and effects efficiently. It is done by ignoring the primitive tasks preconditions as then only the tasks' ordering relation and occurrences need

[1]Note that not all complexity results can be transferred directly to p.o. domains because some proofs exploit the fact that only deterministic complexity classes $\mathcal{C}$ were considered, where it holds $\mathcal{C} = co\mathcal{C}$. This is not the case for all p.o. domains since, e.g., the plan existence problem for acyclic p.o. problems is NEXPTIME-complete (Alford, Bercher, and Aha 2015).

to be considered. So, the *precondition-relaxation* of a domain $\mathcal{D} = (F, A, C, M)$ is the domain $\mathcal{D}' = (F, A', C, M)$ with $A' = \{(\emptyset, add, del) \mid (prec, add, del) \in A\}$. Then, the *precondition-relaxed effects* $\mathit{eff}_*^{\emptyset+}(c)$, $\mathit{eff}_*^{\emptyset-}(c)$, $\mathit{poss\text{-}eff}_*^{\emptyset+}(c)$ and $\mathit{poss\text{-}eff}_*^{\emptyset-}(c)$ (cf. Def. 9) are defined just like the original ones but based on the precondition-relaxed variant of $\mathcal{D}$.

Analogue preconditions were defined slightly differently as removing them completely does not yield the expected result. A fact $f \in F$ is an *executability-relaxed precondition* of c if and only if for all t.o. refinements (ignoring executability) $\langle a_0 \dots a_n \rangle$ of c there exists an action a_i with $f \in prec(a_i)$ and there does not exist an action a_j with $j < i$ and $f \in add(a_j)$, where $i, j \in \{0 \dots n\}$ (cf. Def. 10).

The exploitation of the relaxed preconditions and effects is possible because they possess subset properties with regard to the actual ones so that they do not contain false candidates. However, one needs to pay attention to several small details: The sets $post_*^+(c) = \bigcap_{s \in E(c)} \bigcap_{s' \in R_s(c)} s' \subseteq \mathit{eff}_*^+(c) \cup prec(c)$, $post_*^+(c) = \mathit{eff}_*^-(c)$ are called *state-independent postconditions* (cf. Def. 5 by Olz, Biundo, and Bercher (2021)), which (in the case of positive ones) additionally contain facts that hold after the execution of every refinement but were not added explicitly. For t.o. domains it was shown that $prec^\emptyset(c) \subseteq prec(c)$ and $\mathit{eff}_*^{\emptyset+/-}(c) \subseteq post_*^{+/-}(c)$ if $E(c) \neq \emptyset$ (Olz, Biundo, and Bercher 2021). We would like to make two remarks on this. First, the precondition-relaxed effects can contain facts, which are also preconditions and therefore can be rather interpreted as postconditions than effects. As an example, consider a compound task c with only one method $(c, \langle(\{f\}, \{f\}, \emptyset)\rangle)$. Here f is contained in $\mathit{eff}_*^{\emptyset+}(c)$ but it is also needed to execute c. We still decided to call the sets $\mathit{eff}_*^{\emptyset+/-}(c)$ effects instead of postconditions since the definition is based on the actions' effects and getting also postconditions is more a byproduct than intention. Moreover, not all postconditions are captured by the sets $\mathit{eff}_*^{\emptyset+/-}(c)$. So, the definitions imply some counter-intuitive phenomena concerning postconditions and effects, however, we did not come up with a perfect solution that fixes every interpretation issue. Thus, one should consider carefully which properties are needed for the exploitation at hand and pick the right version accordingly. Second, if c does not have an executable refinement then $\mathit{eff}_*^{+/-}(c) = prec^\emptyset(c) = \mathit{undef}$ but the relaxed versions may contain facts. This can be seen, e.g., if c has only one method containing only the two actions $(\emptyset, \emptyset, \{f_1\})$ and $(\{f_1\}, \{f_2\}, \emptyset)$, which are ordered as given. Here, $\mathit{eff}_*^{\emptyset+}(c) = \{f_2\}$ but this sequence of tasks is never executable[2]. It is a direct consequence of reducing the reasoning complexity from EXPTIME (arb. t.o. domain) to P.

In the p.o. case one needs to keep in mind one more point when it comes to exploitation: As pointed out earlier there might be other tasks in a task network that can or *even must* be interleaved with the subtasks of c, which potentially add or delete the alleged preconditions or effects of c. So the sets $\mathit{eff}_*^{\emptyset+/-}(c)$ and $prec^\emptyset(c)$ can only be considered as preconditions and effects of c if no other tasks are ordered within the refinement of c.

[2]We thank the anonymous reviewer for providing the two examples.

Algorithm 1: Calculates the precondition-relaxed effects for all compound tasks

Input: $\mathcal{D} = (F, A, C, M)$, an HTN planning domain.
Output: The sets of precondition-relaxed effects of all compound tasks

$\mathit{poss\text{-}eff}_*^{\emptyset+}(c) = \mathit{poss\text{-}eff}_*^{\emptyset-}(c) = \mathit{eff}_*^{\emptyset+}(c) = \mathit{eff}_*^{\emptyset-}(c) = \emptyset$ for all $c \in C$
for all $f \in F$ **do**
  $\mathcal{D}' = \text{RESTRICTTOEFFECTS}(\mathcal{D}, f)$
  $C_\varepsilon = \text{COMPUTEEMPTYREFINEMENTS}(\mathcal{D}')$
  $\mathcal{D}'' = \text{SHORTENMETHODSFROMRIGHT}(\mathcal{D}', C_\varepsilon)$
  $M_R = \text{DECOMPOSITIONREACHABILITY}(\mathcal{D}'')$
  for all $c \in C$ **do**
    if $\exists\, (c', tn) \in M_R(c) \wedge a \in tn : f \in add(a)$ **then**
      $\mathit{poss\text{-}eff}_*^{\emptyset+}(c) = \mathit{poss\text{-}eff}_*^{\emptyset+}(c) \cup \{f\}$
    if $\exists\, (c', tn) \in M_R(c) \wedge a \in tn : f \in del(a)$ **then**
      $\mathit{poss\text{-}eff}_*^{\emptyset-}(c) = \mathit{poss\text{-}eff}_*^{\emptyset-}(c) \cup \{f\}$
    if $c \notin C_\varepsilon$ **then**
      if $f \in \mathit{poss\text{-}eff}_*^{\emptyset+}(c) \wedge f \notin \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$ **then**
        $\mathit{eff}_*^{\emptyset+}(c) = \mathit{eff}_*^{\emptyset+}(c) \cup \{f\}$
      if $f \notin \mathit{poss\text{-}eff}_*^{\emptyset+}(c) \wedge f \in \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$ **then**
        $\mathit{eff}_*^{\emptyset-}(c) = \mathit{eff}_*^{\emptyset-}(c) \cup \{f\}$
return $\mathit{poss\text{-}eff}_*^{\emptyset+}(c), \mathit{poss\text{-}eff}_*^{\emptyset-}(c), \mathit{eff}_*^{\emptyset+}(c), \mathit{eff}_*^{\emptyset-}(c)$ for all $c \in C$

Inference Algorithms

The proofs of Theorems 6 (on the poly-time decidability of possible effects) and Corollary 7 (guaranteed effects) as well as of Theorem 7 (on the poly-time decidability of preconditions) by Olz, Biundo, and Bercher (2021) essentially describe procedures to infer precondition-relaxed effects and executability-relaxed preconditions in t.o. domains in polynomial time. We now generalize these procedures so that they can also handle partially ordered task networks and present corresponding pseudo code.

Algorithm 1 is the main procedure to compute precondition-relaxed effects based on the textual description in the proof of Theorem 7 by Olz, Biundo, and Bercher (2021). The major modifications for p.o. domains affect solely subroutine SHORTENMETHODSFROMRIGHT. We consider one fact $f \in F$ after another and curtail the domain according to several subroutines, listed in Algorithm 2.

- We keep only primitive actions that add or delete f as all others are irrelevant. Therefore, the function RESTRICTTOEFFECTS$(\mathcal{D}, f)$ that takes as input a domain $\mathcal{D} = (F, A, C, M)$ and a fact $f \in F$ and outputs the domain $\mathcal{D}' = (\{f\}, A', C, M')$, where $A' = \{(prec(a) \cap \{f\}, add(a) \cap \{f\}, del(a) \cap \{f\}) \mid a \in A\} \setminus \{(\emptyset, \emptyset, \emptyset)\}$ and M' is obtained from M by restricting the task net-

Algorithm 2: Auxiliary Functions

1: ▷ *Returns $C_\varepsilon \subseteq C$, the set of compound tasks admitting an empty refinement.* ◁
2: **function** EMPTYREFINEMENTS($\mathcal{D} = (F, A, C, M)$)
3: $C_\varepsilon = \emptyset$; $M' = M$; $setChanged = true$
4: **for all** $m = (c, tn = (T, \prec, \alpha)) \in M$ **do**
5: **if** $T = \emptyset$ **and** $c \notin C_\varepsilon$ **then**
6: $C_\varepsilon = C_\varepsilon \cup \{c\}$
7: $M' = M' \setminus \{m\}$
8: **if** $\exists\, t \in T : \alpha(t) \in A$ **then**
9: $M' = M' \setminus \{m\}$
10: **while** $setChanged$ **do**
11: $setChanged = false$
12: **for all** $m = (c, tn = (T, \prec, \alpha)) \in M'$ **do**
13: **if** $c \notin C_\varepsilon$ **and** $\forall t \in T : \alpha(t) \in C_\varepsilon$ **then**
14: $C_\varepsilon = C_\varepsilon \cup \{c\}$
15: $M' = M' \setminus \{m\}$
16: $setChanged = true$
17: **return** C_ε

18: ▷ *Returns an updated domain, where only the right-most relevant tasks remain in all methods.* ◁
19: **function** SHORTENMETHODSFROMRIGHT($\mathcal{D}, C_\varepsilon$)
20: we assume that $\prec$ is minimal
21: let $\prec^+$ be the transitive closure of $\prec$
22: $M' = \emptyset$
23: **for all** $m = (c, tn = (T, \prec, \alpha)) \in M$ **do**
24: $T_{rem} = T_{check} = \{t \in T \mid \nexists\, t' : (t, t') \in \prec\}$
25: **while** $T_{check} \neq \emptyset$ **do**
26: select arbitrary $t \in T_{check}$
27: $T_{check} = T_{check} \setminus \{t\}$
28: **if** $\alpha(t) \in C_\varepsilon$ **then**
29: $T' = \{t' \in T \mid (t', t) \in \prec \wedge \nexists\, \tilde{t} \in T_{rem} : \alpha(\tilde{t}) \notin C_\varepsilon \wedge (t', \tilde{t}) \in \prec^+\}$
30: $T_{check} = T_{check} \cup (T' \setminus (T_{rem} \cap T'))$
31: $T_{rem} = T_{rem} \cup (T' \setminus (T_{rem} \cap T'))$
32: $\prec' = \{(t_1, t_2) \in \prec \mid t_1 \in T_{rem} \wedge t_2 \in T_{rem}\}$
33: $M' = M' \cup \{(c, (T_{rem}, \prec', \alpha|_{T_{rem}}))\}$
34: **return** $\mathcal{D}' = (F, A, C, M')$

works to tasks from $A' \cup C$ instead of $A \cup C$.

- The function EMPTYREFINEMENTS($\mathcal{D}'$) is called on the restricted domain that computes the set of compound tasks admitting an empty refinement, $C_\varepsilon \subseteq C$, i.e. they can be decomposed to an empty task network. If a compound task c can be refined into an empty task network, we know that f can only be a possible but not a mandatory (positive or negative) precondition-relaxed effect.
- Moreover, if the last/right-most task in a task network is primitive or does not admit an empty refinement then this task determines whether the fact gets added or deleted. In a partially ordered task network there are potentially several tasks that can be executed lastly. Therefore, the function SHORTENMETHODSFROMRIGHT($\mathcal{D}', C_\varepsilon$) identifies all these tasks for all decomposition methods and removes tasks that are ordered in front of them. In a t.o. task network we have a clear order of tasks and can go from right to left, stopping as soon as we encounter a task (primitive or compound) that does not admit an empty refinement. In our p.o. setting we consider initially all task that do not have a successor. If some of them admit an empty refinement we also consider their predecessors except of those that also precede another already selected task. The same applies for them until we reach a fix point.
- DECOMPOSITIONREACHABILITY($\mathcal{D}$) computes for all tasks $c \in C$ the set of methods that are still reachable via decomposition from c in the restricted domain.

Finally, the effects are determined task by task by analyzing all methods that are still reachable via decomposition from that task as described from line 7 to 16: If there is a reachable method containing an action a adding f then f is a possible positive precondition-relaxed effect because then there is a refinement of c containing a such that no other action adds or deletes f afterwards according to Olz, Biundo, and Bercher (2021). We can further conclude that f is even a guaranteed positive precondition-relaxed effect if it is a possible positive effect but not a possible negative one and c can not be refined into an empty refinement. The case for negative effects follows analogously.

To sum up, we can infer precondition-relaxed effects for compound tasks in p.o. domains like in t.o. domains with the difference in how to determine the relevant tasks that can be executed at the end. We found them after performing subroutine SHORTENMETHODSFROMRIGHT. Instead of computing and analyzing the set of reachable methods we could also perform some fix-point algorithm to propagate the effects up the hierarchy, i.e., we could annotate to each compound tasks whether f is added or deleted in its methods based on the primitive tasks (still for $\mathcal{D}''$). Afterwards we could do this again by taking also the annotated compound tasks into account. This can be repeated until there are no further annotations.

Proposition 1. *Algorithm 1 is sound and complete, i.e., it computes all and only precondition-relaxed effects of a compound task c given a domain $\mathcal{D} = (F, A, C, M)$ and $c \in C$.*

Proof. The proof by Olz, Biundo, and Bercher (2021) is based on the argument that by curtailing the domain as described we find and keep only those tasks in the task networks that can be at the last position adding or deleting a fact f in a linearization of the task network and also in a primitive refinement of c if they are still reachable through the hierarchy. We follow this idea but mainly concentrate on the modified part.

Primitive tasks that neither add nor delete f can be neglected so we remove them to ease notation and reasoning. The set C_ε then contains all compound tasks that can be decomposed into a refinement in which no action affects f. Now we want to determine the tasks (primitive and compound) that can be ordered at the last position in a linearization of a refinement of a task network as they determine whether f is a positive or negative effect, which is done in SHORTENMETHODSFROMRIGHT. If a compound task in a task network admits an empty refinement, then also its predecessors need to be considered. Consider some method's

task network $(c, tn = (T, \prec, \alpha))$ and a task that remained, i.e. some $t \in T_{rem}$. Either t has no successor tasks then it can clearly be ordered last or all (transitive) successors admit an empty refinement since otherwise the latter condition in line 29 would be violated. So if all those tasks were decomposed into an empty task network then t is again the last task. Therefore, for all other tasks $t' \in T \setminus T_{rem}$ it holds that they have primitive or compound successors that can not be refined into empty task networks, i.e., there is always a succeeding relevant primitive task, which makes t' irrelevant so that we delete it. Note that if t' is compound we thereby also cut off its subtasks. After ensuring this property for all methods we only need to check, which primitive tasks can be reached from c via decomposition as all of them can be inductively ordered last in some t.o. refinement. □

The procedure by Olz, Biundo, and Bercher (2021) restricted t.o. domains runs in polynomial time. As we only adapted SHORTENMETHODSFROMRIGHT$(\mathcal{D}', C_\varepsilon)$, which is still polynomial, we can conclude that our modified algorithms still has polynomial runtime.

The algorithm to compute executability-relaxed preconditions follows basically the same idea with small adaptions, which is why we do not include its pseudo code as well. Instead of the function RESTRICTTOEFFECTS$(\mathcal{D}, f)$ we now keep primitive tasks that contain f in their precondition or add effect list. Instead of SHORTENMETHODSFROMRIGHT$(\mathcal{D}', C_\varepsilon)$, we now shorten from left to right. Then, f is an executability-relaxed precondition of c if there does not exist a reachable method containing an action that adds f and c does not admit an empty refinement in the domain restricted just to actions that require f as precondition.

Conclusion

We defined preconditions and effects of compound tasks in p.o. domains and extended an existing inference algorithm for t.o. task networks operating in polynomial time to p.o. domains. This opens up the possibility to exploit such information for planning systems solving p.o. HTN planning problems as well as for modeling assistance or the linearization of partially ordered domains.

References

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

Behnke, G.; Höller, D.; and Biundo, S. 2019. Bringing Order to Chaos – A Compact Representation of Partial Order in SAT-based HTN Planning. In *AAAI*, 7520–7529. 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.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI*, 480–488. IJCAI.

Clement, B. J.; Durfee, E. H.; and Barrett, A. C. 2007. Abstract Reasoning for Planning and Coordination. *Journal of Artificial Intelligence Research (JAIR)*, 28: 453–515.

Dvořák, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and Acting with Temporal and Hierarchical Decomposition Models. In *ICTAI*, 115–121. IEEE.

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

Fernandez-Olivares, J.; Vellido, I.; and Castillo, L. 2021. Addressing HTN Planning with Blind Depth First Search. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 1–4.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In *Proc. of the 12th European Lisp Symposium (ELS 2019)*, 73–80. ACM.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN Planning as Heuristic Progression Search. *JAIR*, 67: 835–880.

Lesire, C.; and Albore, A. 2021. pyHiPOP – Hierarchical Partial-Order Planner. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 13–16.

Magnaguagno, M. C.; Meneguzzi, F. R.; and de Silva, L. 2021. HyperTensioN: A three-stage compiler for planning. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 5–8.

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

Olz, C.; Biundo, S.; and Bercher, P. 2021. Revealing Hidden Preconditions and Effects of Compound HTN Planning Tasks – A Complexity Analysis. In *AAAI*, 11903–11912. AAAI Press.

Olz, C.; Wierzba, E.; Bercher, P.; and Lindner, F. 2021. Towards Improving the Comprehension of HTN Planning Domains by Means of Preconditions and Effects of Compound Tasks. In *Proceedings of the 10th Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2021)*.

Pellier, D.; and Fiorino, H. 2021. Totally and Partially Ordered Hierarchical Planners in PDDL4J Library. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 17–18.

Schreiber, D. 2021. Lilotane: A Lifted SAT-Based Approach to Hierarchical Planning. *JAIR*, 70: 1117–1181.

Schreiber, D.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-Based Tree Explorationfor Efficient and High-Quality HTN Planning. In *ICAPS*, 382–390. AAAI Press.

Tsuneto, R.; Hendler, J.; and Nau, D. 1998. Analyzing External Conditions to Improve the Efficiency of HTN Planning. In *AAAI*, 913–920. AAAI Press.

Waisbrot, N.; Kuter, U.; and Könik, T. 2008. Combining Heuristic Search with Hierarchical Task-Network Planning: A Preliminary Report. In *Proc. of the 21st Int. Florida Artificial Intelligence Research Society Conference (FLAIRS 2008)*, 577–578. AAAI Press.

On Total-Order HTN Plan Verification with Method Preconditions – An Extension of the CYK Parsing Algorithm

Songtuan Lin[1], Gregor Behnke[2], Simona Ondrčková[3], Roman Barták[3], Pascal Bercher[1]

[1] School of Computing, The Australian National University, Canberra, Australia
[2] ILLC, University of Amsterdam, Amsterdam, The Netherlands
[3] Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
{songtuan.lin, pascal.bercher}@anu.edu.au, g.behnke@uva.ni, {ondrckova, bartak}@ktiml.mff.cuni.cz

Abstract

In this paper, we consider the plan verification problem for totally ordered (TO) HTN planning. The problem is proved to be solvable in polynomial time by recognizing its connection to the membership decision problem for context-free grammars. Currently, most HTN plan verification approaches do not have special treatments for the TO configuration, and the only one features such an optimization still relies on an exhaustive search. Hence, we will develop a new TOHTN plan verification approach in this paper by extending the standard CYK parsing algorithm which acts as the best decision procedure in general.

Introduction

The problem of plan verification is to decide, given a plan, whether it is a solution to a planning problem. The study of this problem has drawn increasing attentions in the last decade for its potential usages in benefiting the research on planning. For instance, an independent plan verifier is vital in International Planning Competition (IPC) for the purpose of verifying whether a plan produced by a participated planner is correct or not. Recently, several works have explored the possibility of employing plan verification technique in Human-AI interaction. For example, Behnke, Höller, and Biundo (2017) pointed out the connection between the plan verification problem and mixed-initial planning (Myers et al. 2003) where a planner shall iteratively adjust its output plan according to a user's change requests, and plan verification might also be seen as an approach for planning domain validation, i.e., deciding whether a planning domain is correctly engineered by a domain engineer, where a plan is given as a test case that is supposed to be a solution to a planning problem, and a failed verification indicates that there are some flaws in the domain.

In this paper, we consider the plan verification problem in Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Bercher, Alford, and Höller 2019). We particularly focus on a special class of HTN planning problems called totally ordered (TO) HTN planning problems which plays a prominent role in HTN planning, as evidenced by the fact that TO planning problem benchmarks significantly outnumber partially ordered (PO) ones in the IPC 2020 on HTN Planning. In spite of the significance, most existing HTN plan verifiers (Behnke, Höller, and Biundo 2017; Barták et al. 2020; Höller et al. 2022) have no special treatments for TO problems, and the only one having such an optimization is by Barták et al. (2021b).

The core of our TO plan verification approach is the CYK parsing algorithm (Sakai 1961), which can be employed here because a TOHTN planning problem is semantically equivalent to a context-free grammar (CFG) (Höller et al. 2014, Thm. 6), and hence, the TOHTN plan verification problem is essentially the parsing problem for CFG. However, that result by Höller et al. (2014) does not take into account so-called *method preconditions* which occur quite often in practice in many TOHTN planning benchmarks and thus also become an obstacle to directly applying the CYK algorithm to plan verification. Consequently, we will extend the standard CYK algorithm to adapt method preconditions.

The idea of viewing an HTN plan verification problem as a parsing problem is widely used. For instance, Barták, Maillard, and Cardoso (2018) and Barták et al. (2020) exploited the connection between HTN planning problems and attributed grammars and proposed a parsing-based plan verification approach for general HTN planning problems, which can also be used to correct flawed HTN plans (Barták et al. 2021a) and is then extended to have the special treatments for the TO setting (Barták et al. 2021b). Notably, their treatments for TOHTN planning problems still rely on an exhaust search and thus have several overheads, which is mandatory because the approach takes into account some additional state constraints. However, those state constraints are rare in many TOHTN planning benchmarks. For this reason, we are not concerned with such constraints, which allows us to fully exploit the connection between TOHTN planning problems and CFGs and thus develop a more efficient TO plan verification approach.

HTN Formalism

In order to explain how our TO plan verification approach works, we first introduce the HTN formalism employed in the paper. Since we only consider TOHTN plan verification in this paper, the formalism presented here is targeted specifically at the TO configuration, and it is an adaption of the one by Geier and Bercher (2011), by Behnke, Höller, and Biundo (2018), and by Bercher, Alford, and Höller (2019). We start by presenting the definition of TOHTN planning problems and explain in detail each component in the definition later.

Definition 1. A totally ordered HTN planning problem $\mathcal{P}$ is a tuple $(\mathcal{D}, tn_I, s_I)$ where $\mathcal{D} = (F, N_c, N_p, M, \delta)$ is called the domain of $\mathcal{P}$. F is a finite set of propositions, N_c is a finite set of *compound* task names, N_p is a finite set of *primitive* task names, M is a finite set of methods m with $m \in 2^F \times N_c \times (N_c \cup N_p)^*$, and $\delta : N_p \to 2^F \times 2^F \times 2^F$ is a function. $s_I \in 2^F$ and $tn_I \in (N_c \cup N_p)^*$ are called the initial state and the initial task network (or the goal task network) of $\mathcal{P}$, respectively.

We also define a $tn \in (N_c \cup N_p)^*$ as a task network, which is a sequence of task names.

In the definition presented above, task names are categorized as being primitive and compound. A primitive task name p, also called an action, is mapped to the respective precondition, add list, and delete list by the function δ, written $\delta(p) = (prec(p), add(p), del(p))$, where $prec(p)$, $add(p)$, and $del(p)$ respectively refer to the preconditions, add list, and delete list of p, each of which is a set of propositions. A primitive task name p is applicable in a state $s \in 2^F$, *iff* $prec(p) \subseteq s$, and we say that a state s' is a consequence of applying a primitive task p in a state s, written $s \to_p s'$, *iff* p is applicable in s, and $s' = (s \backslash del(p)) \cup add(p)$. Similarly, a state trajectory $\langle s_0 \cdots s_n \rangle$ is a consequence of applying a sequence of primitive task names $tn = \langle p_1 \cdots p_n \rangle$ with $n \in \mathbb{N}_0$, i.e., a primitive task network, in a state s *iff* $s_0 = s$, and for each $1 \leq i \leq n$, $s_{i-1} \to_{p_i} s_i$, and we say that the state s_n is obtained by applying tn in s, written $s \to^*_{tn} s_n$.

On the other hand, a compound task c in a task network could be rewritten as another task network tn by a method $m = (prec(m), c, tn)$ where $prec(m)$ refers to the precondition of m. We call this process the decomposition of c, written $c \to_m tn$. We will also omit the subscript m in the notation, i.e., $c \to tn$, to indicate that there *exists* some method which decomposes c into tn. m can be applied to decomposing c *if and only if* its precondition is satisfied. We will elaborate how to determine whether the precondition of a method is satisfied (i.e., the semantics of method preconditions) later on. The concept of decompositions can also be extended to task networks:

Definition 2. Let tn and tn' be two task networks where tn is of the form $tn = \langle tn_1\ c\ tn_2 \rangle$ with c being a compound task and tn_1 and tn_2 being two sequences of task names, each of which might be empty, and $m = (prec(m), c, \hat{tn})$ be a method. We say that tn is decomposed into tn' by m, written $tn \to_m tn'$, if $tn' = \langle tn_1\ \hat{tn}\ tn_2 \rangle$. Similarly, we write $tn \to tn'$ to indicate that there *exists* some method which decomposes tn into tn'. We also write $tn \to^*_{\overline{m}} tn'$ if tn is decomposed into tn' by a *sequence* $\overline{m}$ of methods and $tn \to^* tn'$ if there exists such a method sequence.

For any two task networks tn and tn' with $tn \to^* tn'$, a compound task c in tn is eventually decomposed into a continuous subsequence $\hat{tn}$ in tn' (Bartták et al. 2021b). Hence, we abuse the notation to let $c \to^* \hat{tn}$ denote that the compound task c in some task network is decomposed into the continuous subsequence $\hat{tn}$ of another task network by a sequence of methods, and we write $c \to^*_{\overline{m}} \hat{tn}$ if such a method sequence $\overline{m}$ is understood in the context.

Although a method sequence could capture the decomposition of a task network (or a compound task), it is ambiguous because it does not specify the correspondence between the methods and the compound tasks occurring in the decomposition process. In order to address this, we introduce the notion of *decomposition trees* based upon the one by Geier and Bercher (2011) which characterizes a decomposition process unambiguously.

Definition 3. Given a TOHTN planning problem $\mathcal{P}$, a decomposition tree $g = (\mathcal{V}, \mathcal{E}, \prec_g, \alpha_g, \beta_g)$ with respect to $\mathcal{P}$ is a labeled directed tree where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges, respectively, $\prec_g$ is a *total order* defined over $\mathcal{V}$, $\alpha_g : \mathcal{V} \to N_p \cup N_c$ labels a vertex with a task name, and β_g maps a vertex $v \in \mathcal{V}$ to a method $m \in M$. Particularly, g is *valid* if it is rooted at a vertex r with $\alpha_g(r) = c_I$, and for every inner vertex v whose children in the total order $\prec_g$ forms the sequence $\langle v_1 \cdots v_n \rangle$ ($n \in \mathbb{N}$), if $\alpha(v) = c$ for some $c \in N_c$, then $\beta_g(v) = m$ for some $m \in M$ with $m = (prec(m), c, tn)$ and $tn = \langle \alpha_g(v_1) \cdots \alpha_g(v_n) \rangle$.

Let $\langle l_1 \cdots l_n \rangle$ ($n \in \mathbb{N}$) be the leafs of g ordered in $\prec_g$. We define the yield of g, written $yield(g)$, as the task network $\langle \alpha_g(l_1) \cdots \alpha_g(l_n) \rangle$. For convenience, we will simply use $\mathcal{L}(g)$ to refer to the leafs of g ordered in $\prec_g$.

Having the definition of decomposition trees in hand, we can now define the semantics of method preconditions.

Definition 4. Let $\mathcal{P}$ be a TOHTN planning problem, g a valid decomposition tree g with respect to $\mathcal{P}$ where $\mathcal{L}(g) = \langle l_1 \cdots l_n \rangle$ and $yield(g) = \langle \alpha_g(l_1) \cdots \alpha_g(l_n) \rangle$ ($n \in \mathbb{N}$) consists solely of primitive tasks, and $m = (prec(m), c, tn)$ a method with $\beta_g(v) = m$ for some inner vertex $v \in \mathcal{V}$. The precondition of m is satisfied if and only if for the first vertex l_i ($1 \leq i \leq n$) in $\mathcal{L}(g)$ that is a descendant of v, it holds that $prec(m) \subseteq s_{i-1}$ with $s_I \to^*_{\overline{tn}'} s_{i-1}$ and $\overline{tn}' = \langle \alpha_g(l_1) \cdots \alpha_g(l_{i-1}) \rangle$. For $i = 1$, we define $s_0 = s_I$.

Lastly, we define the solution criteria for TOHTN planning problems.

Definition 5. Given a TOHTN planning problem $\mathcal{P}$, a solution to $\mathcal{P}$ is a task network tn consisting solely of primitive tasks such that tn is executable in s_I, i.e., $s_I \to^*_{tn} s$ for some $s \in 2^F$, and there exists a valid decomposition tree g with respect to $\mathcal{P}$ such that $yield(g) = tn$ and for every inner vertex v of g with $\beta_g(v) = m$ for some $m \in M$, the precondition of m is satisfied.

TOHTN Plan Verification

Having presented the TOHTN planning formalism, we now move on to introduce our CYK-based TOHTN plan verification approach. The basis for using the standard CYK parsing algorithm in TOHTN plan verification is that primitive tasks, compound tasks, and methods in TOHTN planning problems are respectively analogy to terminal symbols, nonterminal symbols, and production rules in CFGs. Consequently, the TOHTN plan verification problem is analogy to the membership decision problem for CFGs, which is what the CYK algorithm targeted at.

The CYK algorithm demands that an input CFG (*resp.* a TOHTN planning problem) should be in Chomsky Normal Form (Chomsky 1959) where every production rule (*resp.* a

Algorithm 1: The CYK-based plan verification approach. The lines without being numbered are the standard CYK algorithm, and those being numbered are the modifications for adapting method preconditions and 2RF.

Input: A plan $\pi = \langle p_1 \cdots p_n \rangle$
A planning problem $\mathcal{P}$ in 2RF
Output: True or false depending on whether π is a solution to $\mathcal{P}$
▷ Let $\langle s_0 \cdots s_n \rangle$ be the state sequence *s.t.* $s_0 = s_I$, and $s_{i-1} \rightarrow_{p_i} s_i$ for each $i \in \{1 \cdots n\}$
for $i \leftarrow n$ to 1
  $A[i,i] = \{c \mid c \rightarrow \langle p_i \rangle\} \cup \{p_i\}$
  for $j \leftarrow i$ to n
    for $k \leftarrow i$ to $j-1$
      for $m \in \left\{ m \;\middle|\; \begin{array}{l} m = (prec(m), c, tn), \\ tn = \langle c'_1\, c'_2 \rangle, c'_1 \in A[i,k], \\ c'_2 \in A[k+1,j] \end{array} \right\}$
        ▷ Checking the method precondition
8: **if** $prec(m) \subseteq s_{i-1}$
          $A[i,j] \leftarrow A[i,j] \cup \{c\}$
    ▷ Finding the unit productions
11: **for** $\overline{m} \in \left\{ \overline{m} \;\middle|\; \begin{array}{l} c' \rightarrow^*_{\overline{m}} \langle c \rangle, c' \in N_c, \\ c \in A[i,j] \end{array} \right\}$
12: **if** $prec(m) \subseteq s_{i-1}$ for each m in $\overline{m}$
13: $A[i,j] \leftarrow A[i,j] \cup \{c'\}$
if $c_I \in A[1,n]$ **return** true
else return false

method) decomposes a non-terminal symbol (*resp.* a compound task) into two non-terminal symbols or into a terminal symbol (*resp.* a primitive task). It then determines whether a string is in the language of the CFG (*resp.* whether a plan is a solution to the planning problem) by constructing parse trees (*resp.* decomposition trees) in a bottom-up manner.

More concretely, given a string (*resp.* a plan) $\langle p_1 \cdots p_n \rangle$ ($n \in \mathbb{N}$), the ultimate goal of the CYK algorithm is to find, for each subsequence $\pi^i_j = \langle p_i \cdots p_j \rangle$ ($1 \leq i \leq j \leq n$), the set $A[i,j]$ of all possible non-terminal symbols c such that $c \rightarrow^* \pi^i_j$, i.e., c can be decomposed into π^i_j by a sequence of production rules (methods). Mathematically, this goal can be accomplished via the following recursion formula:

$$A[i,j] = \begin{cases} \{c \mid c \rightarrow \langle p_i \rangle\} & \text{if } i = j \\ \left\{ c \;\middle|\; \begin{array}{l} c \rightarrow \langle c'_1\, c'_2 \rangle, i \leq k < j \\ c'_1 \in A[i,k], c'_2 \in A[k+1,j] \end{array} \right\} & \text{if } i < j \end{cases}$$

The interpretation of the formula is that, for each $1 \leq i \leq n$, a non-terminal symbol c is in the set $A[i,i]$ if it can be decomposed into the terminal symbol p_i by some production rule, and for each i, j with $1 \leq i < j \leq n$, $A[i,j]$ has a non-terminal symbol c if c can be decomposed into two other non-terminal symbols c'_1 and c'_2 by some production rule such that there exists a k with $i \leq k < j$, $c'_1 \in A[i,k]$, and $c'_2 \in A[k+1,j]$, i.e., $c'_1 \rightarrow^* \pi^i_k$ and $c'_2 \rightarrow^* \pi^{k+1}_j$. Notably, the recursion holds because we make the restriction that the input CFG must be in CNF.

In the CYK algorithm, the recursion is implemented via dynamic programming where a two dimension table is constructed to memorize each entry $A[i,j]$ ($1 \leq i \leq j \leq n$), and the table is filled in a right-left, bottom-up order. The implementation is shown by Alg. 1 in which the lines without being numbered are the code for the standard CYK algorithm, and we substitute every component in CFGs (i.e., terminal/non-terminal symbols, production rules, etc.) with its counterpart in TOHTN planning problems.

In order to adapt the CYK algorithm in TOHTN plan verification, we have to deal with method preconditions whose counterpart does not exist in CFGs. This is however trivial because we can simply check whether a method's precondition is satisfied when filling the table, see Alg. 1, line 8.

Though the procedure presented above can already serve as a mature TOHTN plan verification approach, it relies on the strict constraint that an input planning problem must be in CNF. It is unfortunately not trivial to transform an arbitrary TOHTN planning problem into CNF. Similar to how such transformation is done for CFGs (Hopcroft, Motwani, and Ullman 2007; Lange and Leiß 2009), transforming a TOHTN planning problem into CNF usually requires four steps ordered as follows:

1) *binarization*: splitting every method such that it contains *at most* two subtasks,
2) *deletion*: deleting all methods and tasks which will result in the empty task network,
3) *elimination*: eliminating all unit productions, and
4) *termination*: enforcing that for any method, if it contains only one subtask, then the task is a primitive one.

As pointed out by Lange and Leiß (2009), the four steps (the third one in particular) for transforming a CFG into CNF will lead to a quadratic explosion of the size of the grammar, which is also the case for a TOHTN planning problem. This is a significant overhead for TOHTN plan verification because usually a planning problem already contains an enormous number of methods. Further, due to the existence of method preconditions, the four-step transformation on TOHTN planning problems need to modify method preconditions accordingly.

In ordered to avoid these overheads, we only apply the first step *binarization* to an input TOHTN planning problem and result in the planning problem being in so-called 2-regularation Form (2RF) (Behnke and Speck 2021), i.e., in which every method contains *at most* two subtasks (could be either primitive or compound). Note that 2RF also has its counterpart in CFGs called 2-normal Form (2NF) (Lange and Leiß 2009). The advantage of adapting 2RF is that we could avoid the quadratic explosion of the size of a problem and the additional modifications to method preconditions.

The price for adapting 2RF instead of CNF is that we have to merge the remaining three transformation steps into the plan verification procedure. That is, after computing an entry $A[i,j]$ in the standard CYK algorithm, we shall also search for all compound tasks $c' \in N_c$ such that $c' \rightarrow^* \langle c \rangle$ for some $c \in A[i,j]$, and the precondition of every method occurring in the decomposition process is satisfied. This is equivalent to finding *all* method sequences $\overline{m}$ such that the precondition

of each method in it is satisfied, and $c' \rightarrow^*_{\overline{m}} \langle c \rangle$ for some $c' \in N_c$ and $c \in A[i,j]$, and such compound tasks c' should then also be included in $A[i,j]$ (Alg. 1, lines 11 to 13).

For this purpose, we can do the following two things. First, we want to find *all* compound tasks c and *all* method sequences $\overline{m}$ such that $c \rightarrow^*_{\overline{m}} \varepsilon$ where ε refers to the empty task network. We call such a c a nullable task which is analogy to a nullable symbol in CFGs. We can directly apply the procedure for finding all nullable symbols in a CFG to finding all nullable tasks in a TOHTN planning problem together with all method sequences that decompose them into the empty task network. For the details about how this procedure works, we refer to the work by Hopcroft, Motwani, and Ullman (2007). Second, we will construct a graph $G = (V, E)$ with $V = M$, i.e., the vertices are the methods of the planning problem, and an edge $(m', m) \in E$ with $m' = (prec(m'), c', tn')$ and $m = (prec(m), c, tn)$ *iff* either $tn' = \langle c \rangle$ or $tn' = \langle t_0\, t_1 \rangle$ such that there exists an $i \in \{0, 1\}$ with $t_i = c$ and t_{1-i} being a nullable task. We name such a graph as a *unit production graph*.

Having identified all nullable tasks in a planning problem, the unit production graph can be constructed by simply iterating through all methods in the planning problem. As an example about how to construct the graph, consider a method $m = (prec(m), c, \langle c'_0\, c'_1 \rangle)$ where c'_0 is a nullable task and $c'_1 \in N_c$. For each method m' that can decompose c'_1, we add the edge (m, m') to the graph (see the appendix for more details about the construction).

For finding all method sequences $\overline{m}$ such that $c' \rightarrow^*_{\overline{m}} \langle c \rangle$ for some $c', c \in N_c$. We observe one key property of the unit production graph G in which, for any two compound tasks c' and c, $c' \rightarrow^* \langle c \rangle$ *iff* there exists a path in G from m' to m such that m' and m respectively decompose c' and c. Consequently, we only need to conduct several depth-first search in the *reverse* graph of G (i.e., reversing the direction of each edge in G), each of which starts from a vertex m which can decompose c and ends at a vertex m' which decomposes c'. For each found method sequence $\overline{m} = \langle m_1 \cdots m_k \rangle$, we shall also check whether the precondition of each m_i $(1 \leq i \leq k)$ in it is satisfied. Notably, if m_i contains a subtask t which is nullable, then we must also check whether there exists a method sequence $\overline{m}'$ such that $t \rightarrow^*_{\overline{m}'} \varepsilon$ and the precondition of every method in it is satisfied. This is trivial because for each nullable task, we have already found all method decomposing it into the empty task network.

Taking together, Alg. 1 summarizes the procedure of our TOHTN plan verification approach, given a planning problem in 2RF. We first implement the standard CYK algorithm for computing each table entry $A[i,h]$, and then for each such entry $A[i,j]$, we find all method sequences $\overline{m}$ such that $c' \rightarrow^*_{\overline{m}} \langle c \rangle$ for some $c' \in N_c$ and $c \in A[i,j]$ and check whether all method preconditions in the sequence are satisfied. If so, we then add c' to $A[i,j]$ (see the appendix for more implementation details).

Empirical Evaluation

We ran the experiments on a Xeon Gold 6242 CPU. For each instance, each verifier was given 10 minutes of runtime and 8 GB of RAM. The experiments were done both on the TO benchmark set which have method preconditions and on the one which does not. The benchmark set with method preconditions are from the IPC 2020 on HTN Planning which contain 12367 plan instances from 24 domains where 10961 instances are valid, i.e., those are solutions to some planning problems, and the remaining 1406 instances are invalid. The benchmark set without method preconditions is again from the IPC 2020 on HTN Planning, and it is obtained by discarding method preconditions in original planning problems. This set again contains 12367 instances where 11264 are valid, and 1103 are invalid (note the increasing number of valid instances after removing method preconditions).

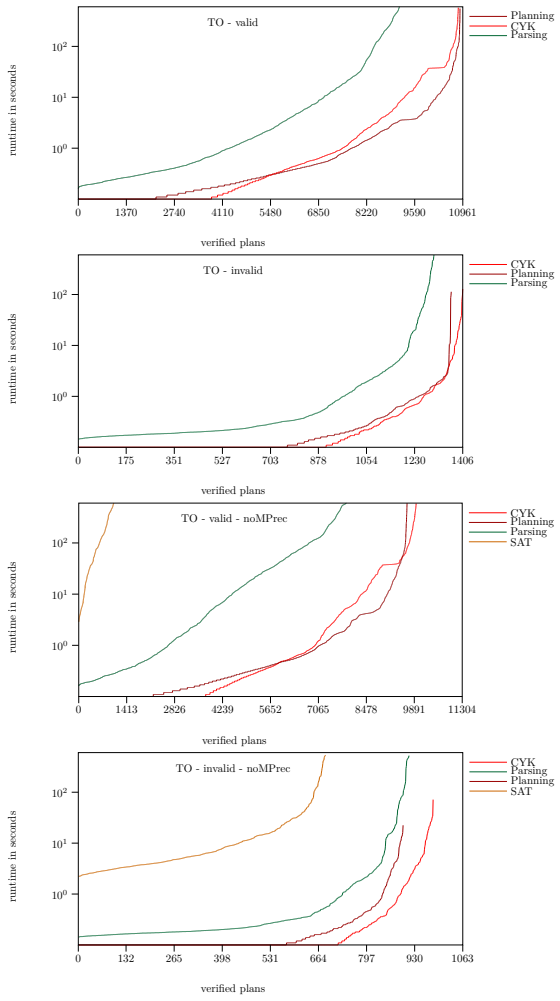


Figure 1: The number of solved instances against runtimes.

Experiment Results

We compared our CYK-based approach with the parsing-based one by Barták et al. (2021b), which is the current state-of-the-art TO plan verifier, and with two general (i.e., PO) plan verifiers which can also be employed in verifying TO plans, i.e., the SAT-based one by Behnke, Höller, and Biundo (2017) and the planning-based one by Höller et al. (2022), which respectively transform a verification problem

Benchmark	Instances	Parsing-based		Planning-based		SAT-based		CYK-based (Ours)	
to-val	10961	9158	(83.55)	**10881**	(**99.27**)	*no support*		10832	(98.82)
to-inval	1406	1301	(92.53)	1364	(97.01)	*no support*		**1406**	(**100.00**)
to-val-no-mprec	11304	7889	(69.79)	9679	(85.62)	1036	(9.16)	**9946**	(**87.99**)
to-inval-no-mprec	1063	915	(86.08)	898	(84.48)	684	(64.35)	**981**	(**92.29**)

Table 1: Table comparing runs of multiple approaches for plan verification. For each verifier, the number in each row indicates the number of solved instances in the corresponding benchmark set, and the respective percentage indicates the coverage rate.

into a SAT problem and an HTN planning problem.

In the experiments on the benchmark set with method preconditions, we did not consider the SAT-based verifier because it does not support method preconditions. For the valid instances, the planning-based verifier achieve the best performance by solving 10881 instances (99.27%). Our approach slightly underperforms it by solving 10832 instances (98.82%) and beats the parsing-based one which solves 9158 instances (83.55%). For the invalid instances, our approach solved all 1406 instances (100%) compared with the planning-based one and the parsing-based one which solve 1364 instances (97.01%) and 1301 instances (92.53%), respectively. The results are summarized in Tab. 1 where the rows to-val and to-inval respectively indicate the valid and invalid instances.

In the experiments on the benchmark set without method preconditions, we included the SAT-based verifier. Our approach outperforms the others in solving both valid and invalid instances. Specifically, our verifier solved 9946 valid instances (87.99%) and 981 invalid instances (92.29%). The planning-based one solved 9679 valid instances (85.62%) and 898 invalid instances (84.48%), and the parsing-based one solved 7889 valid instances (69.79%) and 915 invalid instances (86.08%). The SAT-based verifier has the worst performance, which only solved 1036 valid instances (9.16%) and 684 invalid ones (64.35%), see the last two rows in Tab. 1 for the summary.

Further, Fig. 1 depicts the number of solved instances (the x-axis) against the runtimes (the y-axis), i.e., how many instances can be solved in a specific runtime, for the evaluations of both valid and invalid instances on the two benchmark sets. One might observe that in solving the instances with method preconditions, our approach has the similar performance compared with the planning based one and outperforms the parsing based one. For those without method preconditions, our approach clearly beats the others.

Discussion

We now give some discussion over our CYK-based plan verification approach compared with others, i.e., the parsing-based, the SAT-based, and the planning-based approach.

According to the experiment results, our approach outperforms the parsing-based one (Barták et al. 2020, 2021b) which is the only one by now having the special treatments for the TO configuration. We believe that the major reason for the underperformance of the parsing-based approach is that the approach does not restrict the number of subtasks in each method. As a consequence, the parsing-based approach, which, like our CYK-based approach, try to find all possible compound tasks that can be decomposed into a subsequence of a given plan, relies on an exhaustive search for that purpose.

For example, in our approach, in order to decide whether a compound task c can be decomposed into a subsequence $\pi^i_j = \langle p_i \cdots p_j \rangle$ via a method $m = (prec(m), c, \langle c'_1\, c'_2 \rangle)$, we only have to check whether $c'_1 \in A[i,k]$ and $c'_2 \in A[k+1,j]$ for some $i \leq k < j$. In contrast, in the parsing-based approach, checking whether c can be decomposed into π^i_j via a method which has k ($k \in \mathbb{N}$) subtasks $\langle c'_1 \cdots c'_k \rangle$ requires deciding whether π^i_j can be divided into k subsequences $\pi^i_j = \langle \pi'_1 \cdots \pi'_k \rangle$ such that $c'_r \rightarrow^* \pi'_r$ for each $1 \leq r \leq k$. The latter one is clearly more computationally expensive.

Notably, the parsing-based approach does not restrict the number of subtasks in a method for the purpose of supporting an additional state constraint imposed by the method called the *between-constraint* which must hold *between* the start and the end of the subsequence of the plan obtained from the method. Although it is possible to transform a TOHTN planning problem into 2RF (or CNF) while maintaining these additional constraints, it might cause an unavoidable quadratic explosion of the problem size, which is another significant overhead. Further, despite that the parsing-based approach support such an additional constraint, the benchmark set on which we did the empirical evaluation does not feature it, and hence, this extra functionality will not incur overheads to the approach in the experiments.

For the planning-based approach (Höller et al. 2022), it outperforms our approach in the experiment of verifying valid plan instances with method preconditions and underperforms ours in the remaining three experiments. We hypothesize that the outstanding performance of the planning-based approach in verifying valid plans is due to the heuristics employed by the TOHTN planner which solves the planning problem transformed from a plan verification problem. Particularly, the heuristics might rule out some methods in advance whose preconditions are not satisfiable and henceforth significantly reduce the search space, as evidenced by its underperformance in solving instances without method preconditions. On the other hand, the heuristics might be less powerful when confronting an unsolvable planning problem (i.e., verifying an invalid plan), which might be the reason for why it underperforms the CYK-based approach in verifying invalid plans (with or without method preconditions). Generally speaking, we argue that our CYK-based approach as a decision is still better than the planning-based approach.

Lastly, the SAT-based approach has the worst performance compared with others. We hypothesize that this is

because phrasing a plan verification problem as a SAT formula is already computationally expensive, and solving a SAT problem is $\mathbb{NP}$-hard as well.

Future Works

The TOHTN plan verification approach developed in the paper is based on the CYK algorithm, which is in the family of the so-called *chart parsers*. Some parsing algorithms in the family, e.g., the LR parsing algorithm, have been proved to be more efficient than the CYK algorithm when an input CFG (*resp.* a planning problem) has certain properties, and some, e.g., the Earley parsing algorithm (Earley 1970), do not require any special format of input CFGs while still maintaining reasonable time complexity. Thus, in future works, we would like to explore the possibility of employing other chart parsers in TOHTN plan verification and henceforth make the connection between TOHTN planning and formal languages more strong.

Conclusion

In this paper, we developed a totally ordered HTN plan verification approach that are tailored to method preconditions by extending the standard CYK parsing algorithm. The empirical evaluation results show that our approach significantly outperforms another parsing-based plan verification approach by Barták et al. (2020; 2021b) which is also the only approach by now features the special treatments for the TO configuration. Further, though the approach slightly underperforms the state-of-the-art plan verifier by Höller et al. (2022) when input plans are indeed solutions, it has better performance when an input plan is invalid. Additionally, our approach always has better performance when method preconditions are not considered independent of whether an input plan is valid or not. We thus still regard our approach as a better decision procedure.

Acknowledgment

Simona Ondrčková is (partially) supported by SVV project number 260 575 and by the Charles University project GA UK number 280122.

References

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of Hierarchical Plans via Parsing of Attribute Grammars. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling, ICAPS 2018*, 11–19. AAAI.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021a. Correcting Hierarchical Plans by Action Deletion. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021*, 99–109. IJCAI.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021b. On the Verification of Totally-Ordered HTN Plans. In *Proceedings of the 33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021*, 263–267. IEEE.

Barták, R.; Ondrčková, S.; Maillard, A.; Behnke, G.; and Bercher, P. 2020. A Novel Parsing-based Approach for Verification of Hierarchical Plans. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020*, 118–125. IEEE.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This Is a Solution! (... But Is It Though?) - Verifying Solutions of Hierarchical Planning Problems. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling, ICAPS 2017*, 20–28. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT - Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, 6110–6118. AAAI.

Behnke, G.; and Speck, D. 2021. Symbolic Search for Optimal Total-Order HTN Planning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021*, 11744–11754. AAAI.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6267–6275. IJCAI.

Chomsky, N. 1959. On Certain Formal Properties of Grammars. *Information and Control*, 2(2): 137–167.

Earley, J. 1970. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2): 94–102.

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

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, 1955–1961. IJCAI.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014*, 447–452. IOS.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2022. Compiling HTN Plan Verification Problems into HTN Planning Problems. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling, ICAPS 2022*. AAAI.

Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2007. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Lange, M.; and Leiß, H. 2009. To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. *Informatica Didactica*, 8: 1–21.

Myers, K. L.; Jarvis, P.; Tyson, M.; and Wolverton, M. 2003. A Mixed-initiative Framework for Robust Plan Sketching. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling, ICAPS 2003*, 256–266. AAAI.

Sakai, I. 1961. Syntax in Universal Translation. In *Proceedings of the International Conference on Machine Translation and Applied Language Analysis*, 593 – 608.

Appendix

In the appendix, we would like to elaborate more technical details about our CYK-based TOHTN plan verification approach. In our approach (as well as the CYK algorithm), when computing an entry $A[i,j]$, we have to find all methods m (*resp.* production rules) such that $c \rightarrow_m \langle c'_1\, c'_2\rangle$ for some $c \in N_c, k \in \{i \cdots j-1\}$, $c'_1 \in A[i,k]$, and $c'_2 \in A[k+1,j]$. Most literature about the CYK algorithm in the context of formal languages accomplish this step via iterating through *all* production rules. This is however *not* efficient in the context of plan verification. The reason for this is that in a CFG, the number of production rules is considered to be relatively smaller than the length of a string, whereas this is not the case in plan verification. For instance, some TOHTN planning problem could have more than 10 thousands methods compared with the length of an input plan which is normally below one thousand.

Thus, in order to eliminate this overhead, we maintain two mappings $\varphi_1 : N_p \rightarrow M$ and $\varphi_2 : N \times N \rightarrow M$ where $N = N_p \cup N_c$. Specifically, given a $p \in N_p$, $\varphi_1(p) = m$ for some $m \in M$ *iff* m decomposes some compound task into $\langle p\rangle$, and similarly, given $t_1, t_2 \in N$, $\varphi_2(t_1, t_2) = m$ *iff* m decomposes a compound task into $\langle t_1\, t_2\rangle$. Consequently, given two entries $A[i,k]$ and $A[k+1,j]$ (or one single entry $A[i,i]$), we can quickly find all methods which decompose a compound task into two (or one) subtask(s) that are (is) in the respective entries (entry) by visiting the mapping(s).

Next, we would like to give a recursive procedure for finding all nullable tasks together with the respective method sequences that decompose them into the empty task networks. The procedure is not given in the main paper. We instead refer to the work by Hopcroft, Motwani, and Ullman (2007) for the procedure. The procedure works as follows:

Basis: If $c \rightarrow_m \varepsilon$ for some $m \in M$, then c is a nullable task, and we mark $\langle m\rangle$ as a method sequence that decomposes c into the empty task network.

Induction: If $c \rightarrow_m \langle t_1\, t_2\rangle$ (or $c \rightarrow_m \langle t\rangle$) for some $m \in M$ and t_1, t_2 (or t) are (is) nullable, then c is also nullable, and for *any* two method sequences $\overline{m}_1$ and $\overline{m}_2$ that respectively decompose t_1 and t_2 into the empty task network (or any $\overline{m}$ with $t \rightarrow^*_{\overline{m}} \varepsilon$), $\langle m\ \overline{m}_1\ \overline{m}_2\rangle$ (or $\langle m\ \overline{m}\rangle$) together with any permutation of it is marked as a method sequence that decomposes c into ε.

Having identified all nullable tasks in a planning problem, we can then construct the *unit production graph*. The procedure for constructing the graph is described as follows: For each method $m \in M$ with $m = (prec(m), c, tn)$,

- if $tn = \langle t_0\ t_1\rangle$ for some $t_0, t_1 \in N$ ($N = N_c \cup N_p$), and there exists an $i \in \{0,1\}$ such that t_{1-i} is nullable, then for *each* m' that can decompose t_i, we add the edge (m, m') to the graph, or
- if $tn = \langle t\rangle$ for some $t \in N_c$, then for each method m' that decomposes t, we add the edge (m, m') to the graph.

Now that we have clarified how to construct a unit production graph, we move on to prove the most important property of such a graph, that is, for any two compound tasks c and c', c can eventually be decomposed into c' *iff* there exists a path connecting two methods m and m' in the graph which respectively decomposes c and c'.

Theorem 1. *Let $c, c' \in N_c$, $c \rightarrow^* \langle c'\rangle$ if and only if there is a path in the unit production graph $G = (V, E)$ from m to m' such that m decomposes c and m' decomposes c'.*

Proof. ($\Longrightarrow$): We prove this by induction on the number of steps in decomposing c into c'. The base case is $c \rightarrow \langle c'\rangle$. In this case, a path (m', m) with c' being decomposed by m' exists by the construction of the graph G.

Now suppose that $c \rightarrow^* \langle c'\rangle$ in k steps ($k > 1$), it follows that there must exist a method m which decomposes c into a task network tn such that either tn containing only one subtask task $\hat{c}$ that is in N_c or tn consisting two subtasks where one is nullable, and the other $\hat{c}$ is decomposed into c', because otherwise, c cannot be decomposed into c'. For both cases, we have that $\hat{c} \rightarrow^* \langle c'\rangle$ in $k-1$ steps. By the induction hypothesis, there exists a path from $\hat{m}$ to m' in the graph such that m' decomposes c' and $\hat{m}$ decomposes $\hat{c}$. Further, by the construction of the graph, $(m, \hat{m}) \in E$, and hence, there is a path in G from m to m'.

($\Longleftarrow$): We prove this by induction on the length of the path from m to m'. The base case is that $(m, m') \in E$. By construction, m decomposes a compound task c into a task network tn such that either $tn = \langle c'\rangle$ or $tn = \langle t_0\ t_1\rangle$ in which there exists an $i \in \{0,1\}$ with $t_i = c'$ and t_{1-i} is nullable. For the former, $c \rightarrow c'$ holds naturally, and for the latter, since $t_{1-i} \rightarrow^* \varepsilon$ (because t_{1-i} is nullable), it follows immediately that $c \rightarrow^* \langle c'\rangle$.

For the case where a path from m to m' has length k ($k > 1$), the path can be divided as two parts: a path from $\hat{m}$ to m' of length $k-1$ and an edge $(m, \hat{m}) \in E$. By the induction hypothesis, there exist $\hat{c} \in N_c$ with $\hat{c}$ being decomposed by $\hat{m}$ such that $\hat{c} \rightarrow^* \langle c'\rangle$. Further, by the construction of the graph G, the presence of the edge $(m, \hat{m})$ implies that m decomposes a compound task c into a task network tn in which either $\hat{c}$ is the only subtask, or tn contains two subtasks where one is $\hat{c}$ and the other is nullable. For both cases, we have $c \rightarrow^* \langle \hat{c}\rangle$ and henceforth $c \rightarrow^* \langle c'\rangle$. □

The correctness of Alg. 1 thus follows immediately.

Theorem 2. *A plan $\pi = \langle p_1 \cdots p_n\rangle$ is a solution to a planning problem $\mathcal{P}$ if and only if Alg. 1 returns true.*

Lastly, we would like to discuss the time complexity of our CYK-based plan verification approach. For an input plan $\langle p_1 \cdots p_n\rangle$, one can easily recognize that the time required for visiting all entries $A[i,j]$ ($1 \le i \le j \le n$) is $\mathcal{O}(n^3)$. Further, when computing each entry $A[i,j]$, we need to visit at most all $|M|$ methods for finding all $c \in N_c$ with $c \rightarrow^* \langle c'\rangle$ and $c' \in A[i,j]$. Therefore, the time complexity of the CYK-based plan verification approach is $\mathcal{O}(|M| \times n^3)$.

Theorem 3. *Alg. 1 has the time complexity $\mathcal{O}(|M| \times n^3)$.*

Note that the time complexity of the CYK-based approach also emphasize the importance of maintaining the mappings φ_1 and φ_2 because, as said, $|M|$ is normally larger than $|n|$ in plan verification, and hence, if we visit all methods in each iteration like what is done in most literature, the actual time complexity in practice would be $\mathcal{O}(n^4)$.

T-HTN: Timeline Based HTN Planning for Multiple Robots

Viraj Parimi[1], **Zachary B. Rubinstein**[2], **Stephen F. Smith**[2]

[1] Massachusetts Institute of Technology, [2] Carnegie Mellon University

Abstract

Effective coordinated actions by a team of robots operating in close proximity to one another is an important requirement in many emerging applications, ranging from warehousing and material movement to the conduct of autonomous housekeeping and maintenance of deep space habitats during unmanned periods. Yet, such multi-robot planning problems remain a significant challenge for contemporary planning technologies, due to several complicating factors: goals must be assigned to robots and accomplished over time in the presence of complex temporal and spatial constraints in a manner that optimizes overall team performance, attention must be given to the durational uncertainty inherent in robot task execution, and planning must be responsive to changing and unexpected execution circumstances. In this paper, we present T−HTN, a novel planner that attempts to overcome this challenge by coupling the structure and efficiency of Hierarchical Task Network (HTN) models with the flexible scheduling infrastructure of timeline-based planning systems. We present initial results on a simple set of multi-robot problems that show the potential of T−HTN in comparison to a state-of-the-art PDDL-style temporal planner.

Introduction

Generation of multi-robot plans that optimize overall team performance in the presence of tight temporal-spatio constraints remains a significant challenge for contemporary automated planning frameworks. On one hand, heuristic, action-based temporal planners (both PDDL-style (Coles et al. 2010; Do and Kamhbampati 2003; Eyerich, Matmuller, and Roger 2009) and HTN-style (Qi et al. 2017)) typically achieve tractability by constraining the plan generation process to forward state-space search (i.e., expanding plans in a strict time-forward order). This assumption can be quite awkward for satisfying certain types of temporal constraints (e.g., task deadlines), and more generally can be quite limiting with respect to optimizing team performance objectives across a set of goals (e.g., minimizing makespan, maximizing the number of goals satisfied). On the other hand, timeline-based planners (e.g., (Muscettola et al. 1992, 1998; Fratini, Pecora, and Cesta 2008; Verfaillie and Lematre 2003; Umbrico et al. 2017) provide a flexible, Simple Temporal Network (STN)-based infrastructure (Dechter, Meiri, and Pearl 1991) that allows planning actions to be inserted opportunistically at various time points across an agent's planning horizon during plan generation so as to better optimize global properties of the plan (such as its makespan). But timeline-based systems lack general principles for global search control, and tend to be driven by use of hand-crafted, domain specific heuristics that do not transfer easily to new problems.

Drawing inspiration from previous work in constraint-based scheduling (Smith, Becker, and Kramer 2004; Rubinstein, Smith, and Barbulescu 2012), wherein global search control revolves around allocation of resources to instantiated task networks, this paper proposes a multi-agent planning and scheduling framework that attempts to combine the strengths of both action-based and timeline-based planning approaches, and describes its initial implementation in a prototype planning system called T−HTN. To overcome the inefficiency and idiosyncrasy of reasoning about actions at the individual component level, T−HTN couples timeline-based reasoning with an HTN planning model designed for resource allocation. By introducing hierarchical structure that organizes task decomposition around allocation of resources to tasks and placement of their constituent actions on resource timelines, T−HTN is able to achieve both search efficiency and flexibility, while retaining the representational expressiveness and generality of state-of-the-art temporal planning frameworks.

Some previous efforts have attempted to exploit the structure imposed by an HTN planning model within a rich temporal reasoning infrastructure. In (Qi et al. 2017), the SHOP2 planner was extended to include a temporal reasoning component and provide the ability to generate plans with durative actions. However SHOP2's reliance on forward plan-space search was retained to avoid the added complexity of timeline-based reasoning, and hence the issue of plan quality in the case of multiple goals and actors was not addressed. On the timeline-based planning front, recent work by (Umbrico et al. 2017) has also proposed the use of hierarchical structure as a means of directing search control. However, this framework is rooted in the dynamical systems origin of the timeline-based planning paradigm of modeling and controlling a system of physical components over time, and hence their notion of hierarchy focuses only on aggregate physical system structure. Perhaps closest in spirit to our approach is the recently introduced FAPE planner (Bit-Monnot et al. 2020), which also aims at integrating

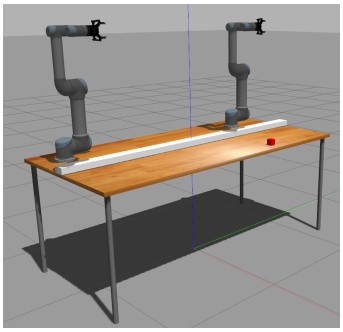


Figure 1: Consider two UR5 robotic arm manipulators mounted on shared railway network who are tasked to move the red cube from its initial location to a target location within a specified deadline.

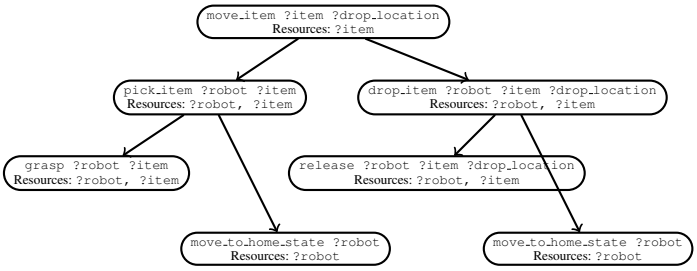


Figure 2: Compiled HTN decomposition tree for the working example as described in Figure 1

HTN and timeline-based planning to gain search efficiency. However, FAPE is designed to accommodate a mix of generative and HTN planning. It incorporates a mix of search control strategies to this end, and in the case of a fully specified HTN model (our interest in this paper), FAPE falls back on the same forward heuristic search algorithm that limits other HTN planners. T−HTN, in contrast, retains the flexibility to opportunistically expand the developing plan and centers plan expansion around allocation of resources to achieve search efficiency.

In this paper, we summarize the T−HTN planner, and present initial experimental results on a set of simple multi-robot planning problems that show its potential in comparison to POPF (Coles et al. 2010), a state-of-the-art action-based temporal planner.

Running Example

Figure 1 introduces a simple problem, motivated by our research interest in robotic systems for autonomous maintenance of deep space habitats, that we will use as a running example throughout the paper. In this scenario, two UR5 robotic manipulators are mounted on a shared railway network. The network is divided into *rail blocks* and two ensure safe operation, only one robot can occupy a given rail block at a time. Three attributes are used to model the state of each robot at any point in time: the rail block it is occupying, the position of its arm and the state of its gripper. Initially, each robot arm starts in the "home" (contracted) position, which is required for rail travel, and each arm's gripper state is empty. Each arm has primitive actions for travelling across rail blocks (*rail_move*), for grasping (*grasp*) and releasing (*release*) objects, and for returning the arm to its home position (*move_to_home_state*). Figure 2 shows a *move-item* HTN for the example problem.

Technical Approach

Like all timeline-based planning systems, T−HTN replaces the classical planning representation of current state as a forward-rolling collection of facts with an explicit characterization of its evolution over time, where various aspects of state (referred to as *state variables*) are represented as sequences of facts (or *state values*) that are true over time. By definition, this change results in a greatly expanded search space vis a vis traditional HTN planning formalisms, as states corresponding to specific intervals across the planning horizon must now be examined to situate actions in the plan. To gain search efficiency in this expanded search space, T−HTN introduces and exploits additional problem structure. Specifically, we ascribe special status to objects declared as resources and assume that all primitive (leaf) nodes in a HTN plan (referred to as *actions*) will ultimately be allocated to and scheduled on the specific resource(s) that will carry them out. Accordingly each declared resource has an associated state variable that represents the resource's availability over time, referred to as its *timeline*. When an action in the plan is scheduled on a resource r, an *action token* is inserted onto r's timeline for the determined interval. This action token indicates all aspects of the action's state that are true at the start, end or throughout the scheduled interval, and the key assumption is that these aspects of state will change only as new actions are subsequently inserted before or after this action token on r's timeline. The insertion of an action token on r's timeline may also result in derivative state changes to other timelines (e.g., the effects of a *grasp* action will dictate the start of a change to the location of the grasped item). To allow for specifying this additional resource structure, T−HTN uses an extended version of HDDL (Holler et al. 2020) for representing the domain and problem inputs.

Language Extensions

Using a model similar to the one used in the Action Notation Modeling Language (ANML) (Smith, Frank, and Cushing 2007) for representing resources and complex objects, T−HTN uses types to designate objects as resources. In this paper we restrict attention to resources of type discrete_reusable_resource. The aspects of state that are stored with an action token on a given resource's timeline are specified as an optional second form on the type definition that denotes variable-type pairs (delimited by matching curly brackets). When objects of a resource type are created, relevant aspects of state are initialized by providing an optional second argument (also delimited by curly brackets and in the form of *predicate* = *value*). As new action tokens are inserted onto a resource timeline, the preconditions and ef-

fects of the corresponding actions will dictate how relevant aspects of state will change. The following snippet is an example definition of `robot` and `rail_block` resource types and initialization of a pair of robots:

```
Definition:
    (:types
        robot {?block - rail_block
            ?arm-position - state
            ?gripper-state - item} -
                discrete_reusable_resource
                    )
 Initialization:
    (:objects
        ur5A, ur5B - robot)
    (:object-instances
        ur5A {block = blockA,
            arm-position = home, gripper
                -state = empty}
        ur5B {block = blockD,
            arm-position = home, gripper
                -state = empty})
```

A second extension that `T-HTN` makes to HDDL involves specification of temporal constraints. In this case, we borrowed directly from PDDL 2.1, incorporating its `:duration` field and the temporal qualifiers (`atstart`, `atend`, `overall`) used in preconditions and effects into the action definition of HDDL. For example:

```
(:action move_to_home_state
    :parameters (?r - robot)
    :duration (= ?duration 10)
    :precondition (and
        (atstart (unsafe state
            ?r.arm-position)))
    :effect (and
        (atend (= ?r.arm-position
            home)))
)
```

By definition, any resource that is allocated to an action is unavailable `overall` of the action's scheduled interval.

A third extension to HDDL allows for the use of specialized algorithms for handling specific planning sub-problems efficiently. In our running example, algorithms for solving the multi-agent path-finding problem (MAPF) are relevant to the movement of robots along the rail network in service of tasks in a collision-free manner. Rather than modeling the movement possibilities as adjacent blocks on the rail and trying all combinations until a goal location is found, application of a specialized planner can quickly determine an efficient route by coupling a shortest-path algorithm with a collision-avoidance strategy. This extension is achieved in `T-HTN`with two additional constructs: (1) the definition of a functional predicate with `f-predicate` and (2) the definition of functional methods with `f-method` keywords. Using these constructs, one can tie specialized external planners to the domain representation based on their unique name identifiers. Functional methods are slightly different in their representation than standard methods in terms of their defined task network. Since they are connected to a specialized external planner, we provide a simple wrapper function whose name is restricted to be the same as the corresponding functional method. Its purpose is to first convert the input from the parsed format to the API that the specialized planner supports and then to convert the output of the specialized planner into a sequence of actions and temporal constraints that it can be linked into the overall task network. The following snippet shows a specification of these constructs for the MAPF planner in our example scenario:

```
Predicate:
    (:f-predicates
        (clear ?from ?to - rail_block)
    )

Method:
    (:f-method m_clear_and_move
        :parameters (?r - robot
            ?to - rail_block)
        :task (clear_and_move ?r ?to)
        :precondition (and
            (atstart (not (clear
                ?r.block ?to))))
        :effect (and
            (atend (clear ?r.block ?to))
                )
    )
```

A final extension to HDDL allows for specifying a broader set of temporal constraints between tasks. HDDL allows specification that tasks must be done in a particular order for example, but does not allow specification that a task must be done immediately after another task completes. To address this limitation, `T-HTN` introduces the `:sync-constraints` field to the `:method` construct, in which any set of pairwise constraints can be added. Using built-in constraint types, this construct supports general specification of Allen temporal relations between tasks. In the following example using `:sync-constraints`, *meets* denotes that `task1` must start immediately after `task0` completes.

```
(:method m_pick_item
    :
    :ordered-subtasks (and
        (task0 (grasp ?r ?i))
        (task1 (move_to_home_state ?r)))
    :sync-constraints (and
        (task0 meets task1))
)
```

Core Search Procedure

Leveraging the domain representation, `T-HTN` employs an incremental algorithm for generating and feasibly inserting a new task plan into the current global multiagent plan/schedule. It first enumerates all possible decompositions of the incoming request and then instantiates a task network for each decomposition that is tied to an underlying STN. Each possible task network instantiation still needs to be grounded with a specific set of resource assignments, and the choice of resources can significantly affect overall plan quality. Alternative sets of resource assignments for a given instanti-

ated task network are explored by applying a backward timeline scanning procedure to each. For a given set of resource choices, the scanning procedure is applied to determine the set of slots on relevant resource timelines where actions can be feasibly scheduled. We believe this ability to (1) organize the search around alternative sets of resource choices and (2) exploit timeline structure to determine feasible options is key to achieving overall planning/scheduling efficiency in multi-agent domains.

Enumerating Decompositions As mentioned before, T-HTN uses the standard HTN task decomposition process by methods to produce an AND/OR HTN that represents alternative solution paths. Once the path decomposition tree is generated, the next step is to enumerate all the possible decompositions, by combinatorically expanding the existing OR nodes. Algorithm 1 provides an efficient, high-level, recursive algorithm that enumerates all paths in the tree while guaranteeing that no potential alternatives are skipped. This algorithm follows from a straightforward depth-first search procedure where, at each level, we keep track of the choices made thus far and then recursively maintain a cartesian product of all such decisions. The worst-case time complexity of Algorithm 1 is dominated by the size of the cartesian product, which is computed in Line 15.

Algorithm 1: Algorithm to return all the possible instantiations of a given path decomposition tree.

```
1:  procedure ENUMERATE DECOMPOSITIONS(vertex)
2:      Initialize empty leafs vector
3:      if vertex is a leaf then
4:          Add vertex to leafs
5:          return leafs
6:      else if vertex is OR then
7:          for all children c of vertex do
8:              leafs +=
                    ENUMERATE DECOMPOSITIONS(c)
9:          end for
10:     else
11:         Initialize empty op vector
12:         for all children c of vertex do
13:             op += ENUMERATE DECOMPOSITIONS(c)
14:         end for
15:         leafs = cartesian_product(op)
16:     end if
17:     return leafs
18: end procedure
```

Instantiating Alternative Task Networks For each possible alternative decomposition in turn, T-HTNinstantiates a task network that is tied with the underlying STN and contains tokens corresponding to constituent tasks and actions. Each token designates a start and end time point for a particular task/action, and, if the token corresponds to an action, the corresponding duration constraint is enforced between the start and end time points. Plans are generated for a task network by scheduling its action tokens on the timelines of compatible resources. An action token is scheduled on a compatible resource timeline by searching for *slots*, i.e., temporal intervals of availability between tokens on the timeline, in which the action token can be feasibly inserted.

Algorithm 2: Algorithm to return an instantiated task network (connected to the underlying STN) that enforces all pre-specified temporal constraints.

```
1:  procedure EXPAND(vertex, tree, STN)
2:      Create a token for vertex
3:      Connect token to tree
4:      if vertex is not leaf then
5:          for all children c of vertex do
6:              EXPAND(c, tree)
7:          end for
8:          Add synchronization constraints to the STN
9:          Add contains constraints to the STN
10:     end if
11:     return tree
12: end procedure
13: procedure INSTANTIATE TASK NET(search, STN)
14:     Instantiate an empty tree
15:     tree ← EXPAND(search.root, tree, STN)
16:     Add the release time constraint to search.root
17:     Add the due date constraint to search.root
18:     return tree
19: end procedure
```

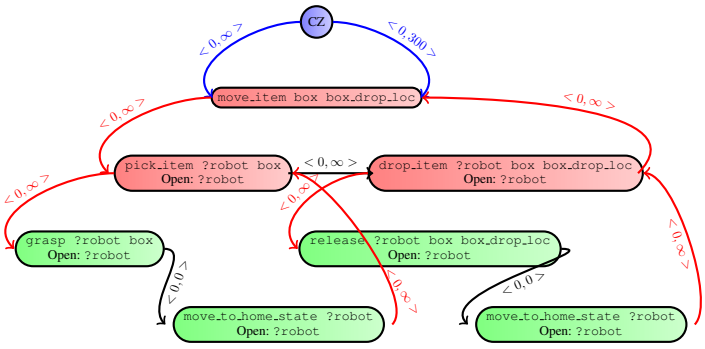


Figure 3: Instantiated task network for the working example as described in Figure 1. Nodes in red represent the high-level tasks, and the nodes in green represent the action primitives. The blue edges correspond to the release time and due date constraints, while the red edges correspond to the contains constraint. The black edges correspond to the pre-specified synchronization rules.

T-HTN enforces a *contains* temporal constraint between each higher-level token and its corresponding decompositions so that the two levels are temporally linked. Such constraints ensure that any temporal constraints imposed on an aggregate task are also applied to its constituent sub-tasks. If the constituent sub-tasks of an aggregate task are known to be *ordered*, then the *contains* constraint need only be enforced between the start time point of the parent task to the start time point of the first task in ordered decomposition

and the end time point of the last sub-task in the ordered decomposition to the end time point of the parent task. Absolute temporal constraints, such as release times and due dates, are asserted to corresponding root task and propagated down the network via the contains constraints. To efficiently compute all resource and parameter assignments, we segregate choices based on whether they have been already made or not. Decisions that have been made prior are moved to a *closed* set, while the decisions that remain to be made are moved to the *open* set. A cartesian product is then computed on all the possible assignments of *open* set parameters and these are used to ground the generated task network for scheduling. Algorithm 2 provides a high-level overview of how task network generation proceeds. Assuming that there can be V nodes in the generated tree, the worst-case time complexity of the algorithm is $O(V)$ since each vertex needs to be visited at least once so that it can be part of the task network. Figure 3 shows the generated task network for the working example as described in Figure 1.

Finding slots Once the instantiated task network is built, it acts as a template for all possible resource and parameter assignments. For each possible combination of assignments, T−HTN determines the set of feasible slots by a backward timeline scanning process that repeatedly attempts to backward schedule the action tokens in the instantiated task network at each possible start point. The timeline scanning process iterates through all required resource timelines in reverse order, identifying sequential pairs of tokens currently on the timeline that delineate potential slots. The search of slots pivots around resources of type *robot*, while considering other required resources (e.g., rail_blocks) as dependent resources. The action tokens of the instantiated task network are only scheduled on *robot* and *rail-block* timelines.

At the same time, the effects of an inserted action may also imply changes to other, dependent state variable timelines. For each combination of slots identified on a resource timeline and its dependent state variable timelines, T−HTN queries the underlying STN to confirm the feasibility of that slot based on current temporal bounds. This process helps prune infeasible combinations of slots early on, and thereby speeds up the scanning process. Following these checks, T−HTN constructs a coherent world state of the environment by iteratively modifying the initial world state with the effect literals encapsulated by the currently considered slots. This updated world state is utilized to check the preconditions of actions, and assuming that all preconditions are verified, the temporal constraints imposed on the action token corresponding to the action are enforced in the underlying STN. This includes retraction of the sequencing constraint between the two tokens on the timeline surrounding the current slot, and the posting of two new sequencing constraints to insert the new action token into this slot. If all of these constraints can be consistently asserted, the slot is a feasible assignment for the action token. T−HTN uses this procedure to generate sets of feasible slot assignments (options) for all action tokens in an instantiated task network by trying to schedule each of the possible combinations of assignments in their Cartesian product. The maximum number of options generated before terminating the search can be restricted by setting a customizable variable to the preferred number. Finally, once all options have been generated, they are evaluated according to some set of objective criteria (e.g., minimize overall task makespan, complete task as early as possible, reduce disruption to plans of other robots), and T−HTN commits to the best option.

Algorithm 3: Algorithm to find a set of feasible slots for a given instantiated task network while attempting to satisfy any failing precondition literals.

```
1:  procedure SATISFY PRECONDITION(lit, task_net, STN)
2:      Find the satisfying task tk
3:      Identify tk's required resources and parameter assignments
4:      if tk is an action primitive then
5:          Create token corresponding to tk
6:          Add token to task_net
7:          Add temporal constraints of token to STN
8:      else
9:          tokens ← Call the specialized external planner
10:         for all token in tokens do
11:             Add token to task_net
12:             Add temporal constraints of token to STN
13:         end for
14:     end if
15: end procedure
16: procedure FIND SLOTS(task_net, STN)
17:     Instantiate an empty slots data structure
18:     for all Leaf tasks tk in task_net.leafs do
19:         Collect tk's required resource R assignments
20:         for all Slots s over R do
21:             Check temporal bounds of s using STN
22:             Compute the world state ws using s
23:             for all Precondition literal p over tk do
24:                 if p fails against ws then
25:                     SATISFY PRECONDITION(p, task_net, STN)
26:                 end if
27:             end for
28:             Enforce temporal constraints of tk onto the STN
29:             slots += s
30:         end for
31:     end for
32:     return slots
33: end procedure
```

Note that sometimes it is hard to encapsulate the entire precondition check within single or multiple literals. By utilizing the functional predicate constructs introduced earlier, we use specialized algorithms to compute such prerequisite checks efficiently and optimally where appropriate[1]. If, however, any precondition literal fails either via a functional

[1]In our current context, this refers to the MAPF solver mentioned earlier.

predicate call or via a violation of a constant literal in the process of checking preconditions against an updated world state, we attempt to satisfy such failing preconditions in two different ways. First, we iterate through the list of actions and identify potential actions whose effects match the failing precondition. If we find such an alternative, the newly instantiated tokens are then added to the same task network, and the procedure continues normally. Second, we iterate through the list of functional methods and identify potential solutions whose effects match the failing precondition. If we find such a method, we call the specialized planner associated with that functional method. We update the task network by adding all newly generated action tokens, after which the procedure continues normally. Whenever T−HTN finds an alternative action or method, it validates that alternative by comparing its set of preconditions with the updated world state. If there is any violation, T−HTN continues to look for other alternatives until they are exhausted. Since this process can potentially lead to infinite recursion, we employ a conservative approach where the act of satisfying preconditions is terminated after the first recursive level. Algorithm 3 provides a high-level overview of the outlined search procedure. Assuming that there are T leaf actions to be scheduled and at most N potential slots for each such task, then the worst-case complexity of the algorithm is $O(TN)$ which is going to be heavily dominated by the size of slots since as each leaf gets scheduled $N \gg T$.

To summarize, our developed framework, T−HTN combines Algorithms 1, 2 and 3 to satisfy any incoming request given a set of timelines tied to an underlying STN. It first parses the incoming request and generates a corresponding path decomposition tree that gets processed by Algorithm 1 to generate all possible alternative decompositions. Each decomposition is then passed to Algorithm 2 to form a corresponding task network that enforces all the relevant temporal constraints in the underlying STN. The generated task network is then passed to Algorithm 3, which finds a set of feasible slots on the required resource timelines while attempting to satisfy any failing precondition literals. Since Algorithms 2 and 3 are repeated for all possible decompositions, the overall complexity of T−HTN also depends on the total number of such possible decompositions as defined by the input HTN. Assuming that there are at most D such decompositions, the worst-case complexity of our approach is $O(DTN)$ which is heavily dominated by Algorithm 3.

Looking back at the working example, assuming the robot parameter assignment in the corresponding task network shown in Figure 3 was UR5A, it is clear to observe that UR5B must move out of the way for the actions to take place successfully. This means that when T−HTN tries to schedule the instantiated task network with UR5A as the robot parameter assignment, a satisfying precondition procedure is triggered that calls the m_clear_and_move functional method. This method is internally linked to the specialized MAPF solver, which computes the best joint rail moves for both robots. Moreover, to pick the box, UR5A is first expected to be close to the object before attempting the grasp. This prerequisite condition also fails, triggering another reachable functional method that is also tied to

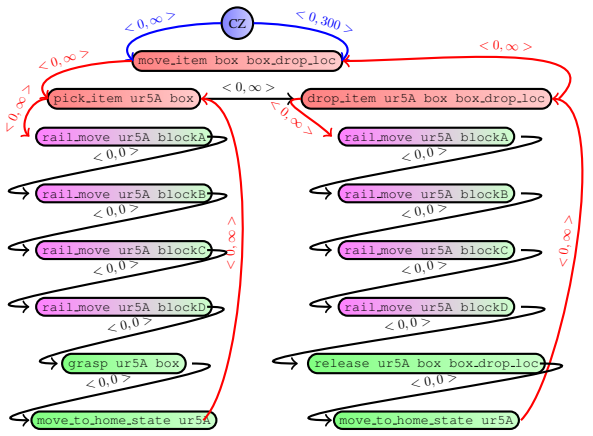


Figure 4: Updated instantiated task network for the working example as described in Figure 1. Nodes in red represent the high-level tasks, and the nodes in green represent the action primitives. The nodes in magenta were added to the original task network by a specialized planner who was triggered by T−HTN to satisfy a failing precondition. The blue edges correspond to the release time and due date constraints, while the red edges correspond to the contains constraint. The black edges correspond to the pre-specified synchronization rules.

the MAPF solver. In this case the path of the calling robot to the required destination is computed and installed in the plan. This results in the generation of an updated task network, which is shown in Figure 4. Figure 5 provides a final snapshot of the timelines generated by T−HTN as a result of scheduling the problem scenario outlined in the working example.

Continual Multiagent/Multi-Robot Planning

The core search procedure just summarized is repeatedly applied to incrementally generate, extend and manage multiagent/multi-robot plans over time as new pending requests and unexpected execution results that require replanning are received. New tasks are allocated to specific resources and integrated into the overall plan as new requests are received. In some cases, the remaining temporal flexibility in the current plan/schedule (or equivalently the continuing availability of required resources) will seamlessly accommodate additional requests. In other, more resource-constrained situations, the addition of new tasks may result in the delay or removal of some less important, previously scheduled tasks. In re-planning settings, it may also be the case that some previously planned/scheduled actions may no longer be relevant and can be retracted to create resource availability for performing corrective tasks. This inherently incremental search approach to planning and scheduling is well suited to such continual planning problems.

Experiments

To benchmark T−HTN's performance, we designed a multi-request variant of the original scenario, which involves mov-

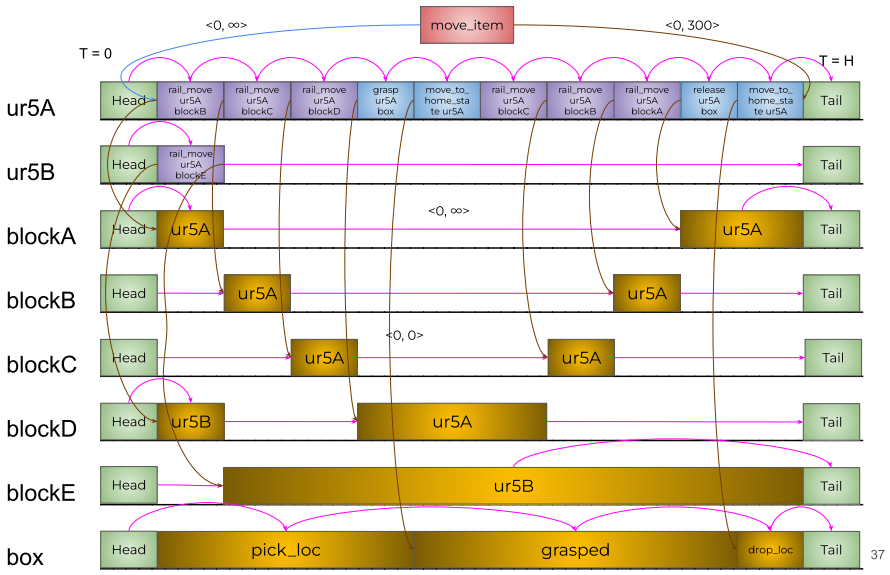


Figure 5: Final snapshot of the timelines generated by T−HTN in response to scheduling the working example outlined in Figure 1. The red node encapsulates the task network, which was shown in Figure 4 and the blue arrows signify the release time and due date constraints. The pink arrows relate to the $\langle 0, \infty \rangle$ sequencing constraint. In contrast, the brown arrows mark the $\langle 0, 0 \rangle$ contains constraint that joins the tokens on the *robot* and corresponding dependent timelines. The head and tail tokens on all the timelines act as auxiliary tokens, which do not have any significance apart from helping in a coherent token insertion procedure.

ing random objects from one location to another in the presence of a rail network that acts as a shared global resource constraint between the two robotic manipulators. Each request specifies the movement of a distinct, unique object from its initial location to another pre-specified destination location. All requests were given the same release-time and deadline constraints to facilitate comparative analysis. For the experiments, T−HTN utilizes an objective metric to prioritize plans that minimize makespan. The entire T−HTN planner, including the domain representation parser, was built in C++.

To evaluate the potential of the T−HTN framework, we compare its performance to another state-of-the-art planner that can be configured to optimize for makespan called POPF (Coles et al. 2010). POPF is a forward-chaining temporal planner built on the foundations of grounded forward search in combination with linear programming to handle continuous linear numeric change. Within the POPF domain specification, we specify one global deadline to be enforced on all requests, which is consistent with the common release and due dates specified in T−HTN input requests. We compare the two planners based on two metrics: computational cost to generate the plan and the resulting plan makespan. The experiments vary in the number of rail blocks, increasing the number of resources that must be managed, and the number of requests, which increases the size of the overall plan. We consider an experimental design that varies both the number of rail blocks and the number of requests from 5 to 25 in increments of 5. A time limit of 10 minutes was imposed for solution of any problem instance. All experiments were run on a Dell machine with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz to ensure a fair comparison.

Tables 1 and 2 show the results obtained relative to both performance objectives.With respect to makespan (Table 1), it can be seen that T−HTN outperforms POPF on the majority of problem instances solved by both techniques.

The more significant results are shown in Table 2. T−HTN dominates with respect to computational cost across all experiments and the differential increases significantly as problem size increases. Notably, POPF increasingly times out before generating a solution as the size of the problem grows. A natural question to ask with respect to computational cost is what impact the specialized MAPF algorithm had on comparative computational cost. To provide some insight, we conducted additional comparative experiments on problems involving just a single request that required multiple rail block moves, and in these experiments, it was found that T−HTN and POPF produced comparable compute times. Their average compute times over 10 different problem instances of single requests were 0.073 and 0.005 seconds respectively. Hence, it appears that the combinatorics of ordering multiple task requests dominates the computational cost of POPF solutions.

Summary

In this paper, we have presented a multiagent / multi-robot planning framework that combines the structural advantages of an HTN representation with the expressiveness and flexibility of timeline-based planning frameworks. We have argued that by emphasizing resource allocation as the basic decision-making focus, it is possible to overcome the complexity of the resulting expanded search space and efficiently

Requests	5		10		15		20		25	
Blocks	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**
5	620	**600**	1320	**1040**	1960	**1500**	2400	**2000**	3320	**2580**
10	**800**	**800**	2480	**1340**	Timeout	**2120**	Timeout	**2300**	Timeout	**3000**
15	1220	**1020**	**1900**	2060	Timeout	**2320**	Timeout	**3620**	Timeout	**5040**
20	**960**	1620	3600	**2560**	Timeout	**3320**	Timeout	**4620**	Timeout	**5780**
25	Timeout	**1320**	2280	**2200**	Timeout	**3620**	Timeout	**5020**	Timeout	**5260**

Table 1: Comparison of POPF and T-HTN with respect to the makespan of the generated plan.

Requests	5		10		15		20		25	
Blocks	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**
5	**0.22**	0.23	3.3	**0.54**	15.14	**2.03**	51.32	**3.43**	196.26	**10.61**
10	100.72	**0.37**	255.90	**0.87**	Timeout	**2.99**	Timeout	**5.63**	Timeout	**5.78**
15	2.64	**0.50**	94.00	**1.62**	Timeout	**3.61**	Timeout	**7.42**	Timeout	**11.56**
20	0.92	**0.91**	327.62	**2.84**	Timeout	**4.43**	Timeout	**17.29**	Timeout	**15.93**
25	Timeout	**0.76**	73.69	**2.12**	Timeout	**6.10**	Timeout	**10.88**	Timeout	**14.28**

Table 2: Comparison of POPF and T-HTN with respect to their computational times in seconds for generating a valid plan.

produce high quality multiagent plans.

To demonstrate this claim, we have developed the T-HTN planner/scheduler. Starting with the HDDL domain representation language, we introduced extensions to give resources and resource timelines special status, to include durative actions and incorporate complex temporal constraints between them, and to enable the use of specialized algorithms to solve well understood planning sub-problems. We then presented a core search algorithm that exploits these representational extensions to generate multiagent plans efficiently. Initial comparative experiments carried out in multi-robot scenarios involving two UR5 robot arms mounted on a shared rail network provided evidence in support of our overall design hypothesis.

Future Work

One immediate direction for future research is more extensive experimentation and analysis of T-HTN's performance characteristics. The results we have presented are preliminary and restricted to relatively simple sets of two-robot, object movement scenarios. We would like to expand experimentation to include other International Planning Competition (IPC) domain problems of general interest to the planning community. One complication here is mapping these domains and problems into the T-HTN's extended HDDL representation.

With regard to further development of T-HTN, a number of simplifying assumptions were made in its initial implementation that provide focal points for future research. First, resource timelines have been realized exclusively as single capacity resources (i.e., resources capable of doing just one task at a time). Although it appears straightforward, one short-term extension will be to extend resource timeline representations to accommodate multi-capacity robotic systems that can simultaneously accomplish multiple tasks (e.g., perform a visual inspection while carrying out a move-object request).

A second related simplification made in T-HTN was to restrict any agent (resource) from interleaving the execution of multiple task requests. Although interleaved accomplishment of multiple requests could lead to more efficient overall behavior in some circumstances, it also runs the risk of search space explosion. How to selectively relax this assumption is a longer term resource challenge.

Acknowledgements: This research was funded in part by the NASA Space Technology Research Institute for Deep Space Habitat Design under grant #80NSSC19K1052, and the CMU Robotics Institute.

References

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: A Constraint-based Planner for Generative and Hierarchical Temporal Planning. *ARXiv Preprint*.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'10, 42–49. AAAI Press.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence*, 49(1): 61–95.

Do, M.; and Kamhbampati, S. 2003. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research*, 20: 155–194.

Eyerich, P.; Matmuller, R.; and Roger, G. 2009. Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. In *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, 130–137. AAAI Press.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences*, 18(2): 231–271.

Holler, 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 *Proceedings AAAI 2020*, 9883–9891. New York, NY.

Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. 1998. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2): 5–47.

Muscettola, N.; Smith, S.; Cesta, A.; and D'Aloisi, D. 1992. Coordinating Space Telescope Operations in an Integrated Planning and Scheduling Framework. *IEEE Control Systems*, 12(1).

Qi, C.; Wang, D.; Muoz-Avila, H.; Zhao, P.; and Wang, H. 2017. Hierarchical Task Network Planning with Resources and Temporal Constraints. *Knowledge-Based Systems*, 133(C): 17–32.

Rubinstein, Z.; Smith, S.; and Barbulescu, L. 2012. Incremental Management of Oversubscribed Vehicle Schedules in Dynamic Dial-A-Ride Problems. In *Proceedings AAAI 2012*. Toronto,.

Smith, D. E.; Frank, J.; and Cushing, W. 2007. The ANML Language.

Smith, S.; Becker, M.; and Kramer, L. 2004. Continuous Management of Airlift and Tanker Resources: A Constraint-Based Approach. *Mathematical and Computer Modeling*, 39(6-8): 581–598.

Umbrico, A.; Cesta, A.; Orlandini, A.; and Mayer, M. 2017. PLATINUm: A New Framework for Planning and Acting. In *16th International Conference of the Italian Association for Artificial Intelligence*.

Verfaillie, C., G.and Pralet; and Lematre, M. 2003. How to model planning and scheduling problems using constraint networks on timelines. *Knowledge Engineering Review*, 25(3): 319–336.

Teaching an HTN Learner

Ruoxi Li[1], Mark Roberts[2], Morgan Fine-Morris[2,3], Dana Nau[1]

[1]Dept. of Computer Science and Institute for Systems Research, Univ. of Maryland, College Park, MD, USA
[2]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
[3]Department of Computer Science, Lehigh University, Bethlehem, PA 18015, USA
rli12314@cs.umd.edu, mark.roberts@nrl.navy.mil, nau@umd.edu, mof217@lehigh.edu

Abstract

We describe Teachable-HTN-Maker, a modified version of the well-known HTN-Maker algorithm that learns Hierarchical Task Network (HTN) methods. Instead of learning methods from all subsequences of a solution plan as HTN-Maker does, Teachable-HTN-Maker learns from a curriculum consisting of examples that are presented in a meaningful order. We compare Teachable-HTN-Maker against HTN-Maker in two planning domains, and observe that it learns fewer methods and better ones.

1 Introduction

Curriculum learning (Bengio et al. 2009) is a training strategy for machine learning. It was inspired by the observation that humans and animals learn much better when the examples are not randomly presented but organized in a meaningful order that starts by illustrating simple concepts and gradually introduces more complex ones. In this paper we investigate how to apply this strategy to improve HTN-Maker (Hogg, Muñoz-Avila, and Kuter 2016).

HTN-Maker learns methods for annotated tasks from subsequences of a solution plan. However, it tries to learn methods for *all* of the annotated tasks from *all* of the subsequences. Many of the methods learned by are not useful because they have undesirable preconditions or decomposition strategies. As a result, an HTN planner that uses methods learned by HTN-Maker may not perform efficiently.

In this paper we make the following contributions:

- We describe Teachable-HTN-Maker, a modified version of HTN-Maker. Instead of examining every subsequence of a solution plan, Teachable-HTN-Maker examines only the subsequences that we tell it to examine in an order that we specify. This modification makes it possible for Teachable-HTN-Maker to learn from curricula.
- We compare Teachable-HTN-Maker and HTN-Maker on two sets of planning problems. One is to move a stack of n blocks in the Blocks World, maintaining their order (which requires moving the stack twice). The other is to deliver n packages in the Logistics domain. In our experiments, Teachable-HTN-Maker learned fewer methods and better ones than HTN-Maker, and did so with less running time. A planner using the methods learned by Teachable-HTN-Maker solves more problems with lower runtime than with the methods learned by HTN-Maker.

2 Background and Related Work

Automated planning systems typically require that a domain expert provide knowledge about the dynamics of the planning domain. In classical planning, the domain knowledge includes semantic descriptions of actions. In Hierarchical Task Networks (HTNs), the domain knowledge includes structural properties and potential hierarchical problem-solving strategies. A significant knowledge engineering burden for a domain expert is required to write HTN decomposition methods. HTN-Maker (Hogg, Muñoz-Avila, and Kuter 2016) overcomes this burden, in part, by learning HTN methods after analyzing the semantics of a solution plan for planning problems.

Several other works have investigated ways to learn HTN methods (Lotinac and Jonsson 2016; Zhuo, Munoz-Avila, and Yang 2014; Xiao et al. 2020). Furthermore, Choi and Langley (2005) have investigated how to learn hierarchical logic programs that are analogous to HTN methods. However, none of those investigations used curricula.

Algorithm 1 describes the high-level operation of HTN-Maker. Its input includes the domain $\mathcal{D}$, initial states from a planning problem $\mathcal{P}$ in a planning domain, an execution trace $\mathcal{E}$ (which can be a plan produced by a planner), a set $\mathcal{T}$ of *annotated tasks* to be accomplished, and the Boolean choice *p* of whether pruning is enabled. Each task's annotations include preconditions that need to be true to accomplish the task, and effects that must be true after accomplishing the task (see Figures 2 and 3 for examples).

During the learning process, if *pruning* is enabled, newly learned methods from the following two categories will be pruned by (i.e., removed from the set of learned methods): 1) subsumed methods, where method m_1 subsumes method m_2 if there exists a substitution that may be applied to m_2 such that both have the same head and subtasks and the precondition of the m_2 implies the precondition of m_1; and 2) unneeded methods: if the preconditions of a subtask are fulfilled, the subtask could just be called directly.

The procedure *LearnMethods* performs the analysis for τ on the subtrace $\mathcal{E}[start, end]$. HTN-Maker analyzes all $O(k^2)$ subtraces for an $\mathcal{E}$ of length *k*, and often learns many methods with undesirable preconditions or decomposition strategies. To address these issues, we modify HTN-Makerf (distribution version *ch-htn-tools-1.1*) to use a curriculum to guide the learning process.

Algorithm 1: A high-level description of HTN-Maker.

Input: domain $\mathcal{D}$, problem $\mathcal{P}$, solution trace $\mathcal{E}$, Annotated tasks $\mathcal{T}$, Pruning enabled p
Output: A set of HTN methods $\mathcal{M}$

```
1: M = ∅
2: for end ← 1 to |E| do
3:     for start ← end down to 1 do
4:         for τ in T do
5:             LearnMethods(start, end, τ, D, P, E, M, p)
6: return M
```

Algorithm 2: Teachable-HTN-Maker.

Input: domain $\mathcal{D}$, problem $\mathcal{P}$, solution plan $\mathcal{E}$, curriculum $\mathcal{C}$
Output: A set of HTN methods $\mathcal{M}$

```
1: M = ∅
2: for (start, end, τ) in C do
3:     LearnMethods(start, end, τ, D, P, E, M, p)
4: return M
```

3 Teachable HTN-Maker

Suppose we want to teach an HTN method learner how to solve some task τ. A curriculum would start by teaching the learner how to solve very simple subtasks of τ, then increasingly complicated subtasks, until we teach it how to solve τ itself. If the learner learns from plan traces, then the plan traces for the subtasks of τ will be subtraces of the plan trace for τ. More specifically, if $\mathcal{E}$ is a plan trace for τ, then the plan trace for each subtask τ_i is a subtrace $\mathcal{E}[start_i, end_i]$ of $\mathcal{E}$. Thus we can represent our curriculum as a sequence of triples of the form $(start_i, end_i, \tau_i)$.

Teachable-HTN-Maker is a modified version of HTN-Maker that takes such triples as input, and analyzes only these triples rather than analyzing every subsequence of $\mathcal{E}$. The pseudocode is in Algorithm 2.

4 Experimental Setup

To examine whether a curriculum can improve upon the methods learned by HTN-Maker, we compared Teachable-HTN-Maker and HTN-Maker in the Blocks World domain and the Logistics domain from the 2nd International Planning Competition (IPC-2). Although these domains are conceptually simple, large problems remain a challenge for planners. For our comparisons, we measured the total number of methods each system learned (with or without pruning) and the time they took to learn those methods, and we evaluated the methods' planning performance.

Blocks World Domain The first domain includes a number of blocks sitting on a table (possibly on top of each other), and a robotic hand that can grasp one block at a time. The objective is to learn methods to move a stack of *n* blocks using the robotic hand, keeping the top-to-bottom order of the blocks the same as in the original stack.

For example, let τ_1 be the task of moving a stack of two blocks (A and B) from block C onto the table, while maintaining their order (i.e. A on B). Let π_1 be the following plan for that task:

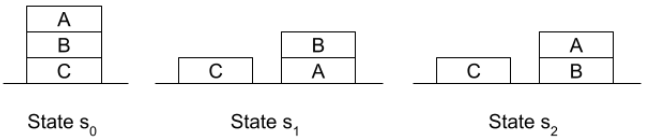


Figure 1: To move two blocks A and B from block C to the table while maintaining their order, the plan π_1 inverts their order (state s_1), then inverts it again (state s_2).

```
(:task Make-1Pile
 :parameters
   (?a)
 :precondition
   (and)
 :postcondition
   (and
     (on-table ?a)
     (clear ?a)))
```

```
(:task Make-2Pile
 :parameters
   (?a ?b)
 :precondition
   (and)
 :postcondition
  (and (on-table ?b)
       (on ?a ?b)
       (clear ?a)))
```

Figure 2: Example annotated tasks in the Blocks World.

Action 1: unstack(A,B)	Action 5: unstack(B,A)
Action 2: putdown(A)	Action 6: putdown(B)
Action 3: unstack(B,C)	Action 7: pickup(A)
Action 4: stack(B,A)	Action 8: stack(A,B)

Figure 1 shows what the plan does.

To teach how to accomplish τ_1, we can use the curriculum shown below. It consists of seven subplans of π_1, starting with simpler ones and combining them into progressively harder ones. For each subplan, the curriculum includes the annotated task (see Figure 2 for examples) that the subplan accomplishes.

	Subplan of π_1	Annotated Task
1.	Actions 1 and 2	Make-1Pile
2.	Actions 3 and 4	Make-2Pile
3.	Actions 1 through 4	Make-2Pile
4.	Actions 5 and 6	Make-1Pile
5.	Actions 7 and 8	Make-2Pile
6.	Actions 5 through 8	Make-2Pile
7.	Actions 1 through 8	Make-2Pile

The preconditions of the methods learned from this curriculum include cases where part of the stack has already been moved, but not cases where a block is held in the robot hand.

Logistics Domain The objective in the Logistics domain is to move packages among locations in various cities, using trucks within cities and airplanes between cities. Let τ_2 be the task of moving three packages p1, p2, p3 within a city, from locations p1s, p2s, p3s to p1d, p2d, p3d, respectively. Let π_2 be the following plan for τ_2:

Action 1: drive-to(p1s)	Action 7: drive-to(p2d)
Action 2: load(p1)	Action 8: unload(p2)
Action 3: drive-to(p1d)	Action 9: drive-to(p3s)
Action 4: unload(p1)	Action 10: load(p3)
Action 5: drive-to(p2s)	Action 11: drive-to(p3d)
Action 6: load(p2)	Action 12: unload(p3)

Here is a curriculum to teach how to perform τ_2. As before, each curriculum entry includes a subplan of π_2 and an annotated task (see Figure 3) that the subplan accomplishes:

	Subplan of π_2	Annotated Task
1.	Actions 1 through 4	Deliver-1Pkg
2.	Actions 5 through 8	Deliver-1Pkg
3.	Actions 1 through 8	Deliver-2Pkg
4.	Actions 9 through 12	Deliver-1Pkg
5.	Actions 1 through 12	Deliver-3Pkg

Methods First, we randomly generate problems with corresponding solution traces for moving a stack of *n* blocks in the Blocks World domain and delivering *n* packages in the Logistics domain. Then we compare the average number of methods learned (with and without pruning) as well as the time (ms) taken by HTN-Maker and our Teachable-HTN-Maker. After we learn HTN methods for each problem, we evaluate the methods by using them to solve the planning problems using an HTN planner: HTN-Maker's implementation of the SHOP (Nau et al. 1999) planning algorithm. Finally, we compare the average length of the plan generated by the HTN planner as well as the running time (ms) for those problems. For each stage of experiments (method learning or planning with the learned methods), for each problem domain (Blocks World or Logistics), as well as for each configuration of learning approaches (HTN-Maker or Teachable-HTN-Maker) and pruning strategies (with or without pruning), we allow a limit of 2 hours of running time on each test suite.

5 Results and Discussion

The results of the method learning experiments (Figure 4a) show that Teachable-HTN-Maker learns significantly fewer methods in less time than HTN-Maker (both with and without pruning). HTN-Maker's run time increases with problem size, such that it cannot solve problems with larger than a certain amount of blocks or packages. The evaluation results (Figure 4b) show that the HTN planner takes significantly less time to solve more problems using the methods learned by Teachable-HTN-Maker (both with and without pruning).

In the Logistics domain, the number of methods learned increases exponentially with the number of packages. This is caused by the existence of alternative ways to bind variable names to object names when learning methods for DelivernPkg tasks. More specifically, when learning methods for the task Deliver-2Pkg from a solution plan that first has package A then package B delivered to the destination, there are 3 possible ways to bind the object names in the domain to the variable names in the annotated task: 1) o1 to A and o2 to B, 2) o1 to B and o2 to A, or 3) both o1 and o2 to B. We have not yet implemented a solution to prevent unwanted name binding or to prune the unwanted methods caused by unwanted name binding. Nevertheless, Teachable-HTN-Maker still learns fewer methods with the same deficiency.

HTN-Maker nondeterministically chooses subtask groupings to form methods when there are several possibilities. The implementation tested in the evaluations caused the algorithm to make deliberate choices when a method decompositions is learned from right to left, such that the right

```
(:task
    Deliver-1Pkg
  :parameters
  (?o - obj
   ?d - location)
  :precondition
  (and)
  :effect
  (and (at ?o1 ?d)))
```

```
(:task Deliver-2Pkg
  :parameters
  (?o1 - obj
   ?o2 - obj
   ?d - location)
 :precondition
 (and)
  :effect
 (and (at ?o1 ?d)
(at ?o2 ?d)))
```

Figure 3: Example annotated tasks in the Logistics domain.

subtask (if any) always corresponds to a previously learned method instance that extends over the largest subplan (if there are multiple ones). For the Blocks World example problem described previously (Figure 1), the method learned from the 7th curriculum step (Figure 5) effectively moves a stack of blocks *b* and *a* from above *c* onto table while maintaining the order by dividing the task into two subtasks that respectively reach state s_1 and s_2. The learned method has `((MAKE-2PILE ?a ?b) (MAKE-2PILE ?b ?a))` as subtasks (Figure 5). Respectively, the original HTN-Maker learns a method that has `((UNSTACK ?a ?b) (MAKE-2PILE ?b ?a))` as subtasks, where the subtask `(MAKE-2PILE ?b ?a)` takes the remaining 7 out of 8 total actions. The method learned with the curriculum is conceptually more desirable.

In the Blocks World domain, the plans produced by the planner using methods learned by HTN-Maker are slightly shorter than the plans produced by the planner using methods learned by Learnable-HTN-Maker. For example, to move a stack of 2 blocks A and B (over C) onto the table while maintaining the order (Figure 1), it takes 6 actions: unstack A from B, put down A, unstack B from C, put down B, pick up A, and stack A on B. On the contrary, the plan produced by the planner using methods learned by Teachable-HTN-Maker has 8 actions. As the number of blocks increases, the length difference between the plans decreases by percentage. However, it takes significantly longer time to find the plans using methods learned by HTN-Maker.

In the Logistics domain, the planner couldn't solve problems with more than 3 packages using the methods learned by HTN-Maker, and with the pruned methods couldn't solve any of the problems. In contrast, when using methods learned by Teachable-HTN-Maker (with or without pruning), the planner always found a solution. In Figure 4(b), notice that when the HTN planner used Teachable-HTN-Maker's methods (without pruning) for 5 packages, its average running time was relatively large. In another set of experimental runs (not shown here) on the same problems, its average running time was much smaller. We believe the variation in running time was because the HTN planner randomized its choices among applicable HTN methods. We will further investigate this in the future.

6 Conclusions

We have described Teachable-HTN-Maker, a modified version of HTN-Maker learns using a curriculum. Our prelim-

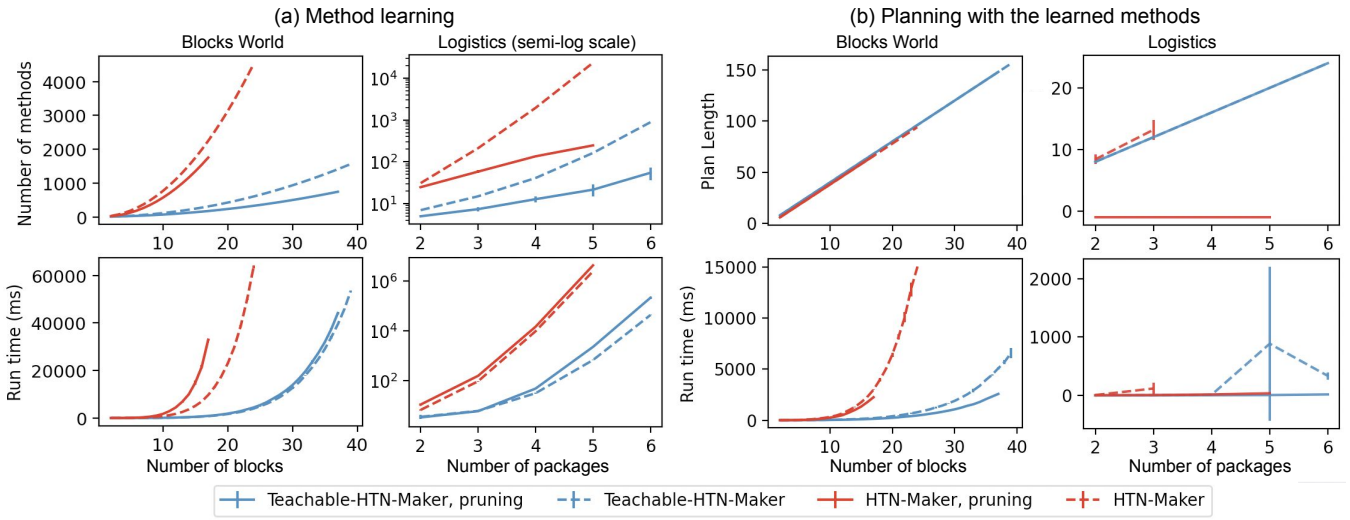


Figure 4: The plots show (a) average number of methods learned and running time for HTN-Maker and Teachable-HTN-Maker, both with and without pruning; and (b) the HTN planner's average plan length and running time on the same problems using the learned methods. In the Blocks World problems, each problem is to move a stack of n blocks, maintaining their order; and each data point is the average of 20 randomly generated problems of size n ($2 \leqslant n \leqslant 40$). In the Logistics problems, each problem is to deliver n packages, and each data point is the average of 5 randomly generated problems of size n ($2 \leqslant n \leqslant 10$).

On some of the larger problems, no results are shown for HTN-Maker or the HTN planner using the HTN-Maker methods because our 2-hour time limit (see "Methods" in the main text) was exceeded before reaching those problems.

In (b), the zero values for HTN-Maker with pruning in the Logistics problems mean that the HTN planner could not solve the problems using the learned methods.

```
(:method MAKE-2PILE
 :parameters (?b - BLOCK ?a - BLOCK)
 :vars (?c - BLOCK)
 :precondition(and (not (= ?b ?a))
 (ON ?b ?c) (ON ?a ?b) (not (= ?a ?c))
 (not(= ?b ?c)) (CLEAR ?a) (HAND-EMPTY))
 :subtasks ((MAKE-2PILE ?a ?b)
   (MAKE-2PILE ?b ?a)))
```

Figure 5: Final method learned for Figure 1.

inary experiments in the Blocks World domain and Logistics domain show that it learns significantly fewer and better methods with less computational effort.

Future Work In future work, we will evaluate our approach in more sophisticated problems with more varieties of annotated tasks and in more domains, e.g., the Depots, Zeno Travel, and Satellite domains, and domains from the 2020 IPC for Hierarchical Planning (Höller et al. 2020). It is both a challenge and an opportunity to figure out how to generate optimal curriculum as the planning problem gets more sophisticated. We will also consider evaluating how different curricula influence the performance of Teachable-HTN-Maker, and evaluate the methods in different problems of the same kind instead of only evaluating them in the exactly same problem where they were learned.

The annotated tasks input to HTN-Maker do not specify any precondition, this allows HTN-Maker to learn methods for the same annotated task from subtraces with different starting positions. The learned methods would respectively have different preconditions, including the ones we are not interested in. A straightforward augmentation to HTN-Maker is to specify preconditions in the annotated tasks and let those preconditions to be checked when learning methods from a subtraces. We would like to compare our approach with the augmented version.

Method instances learned from different subtraces can be generalized into the same lifted method (e.g., the method instances learned from step 1 and 4 of Blocks World curriculum). However, HTN-Maker cannot recognize the semantic equivalence among those subtraces, and therefore spends unnecessary computational resources on learning the same lifted methods from such subtraces. We hope to significantly strengthen the system by making it learn each *lifted* method only once. This is analogous to DreamCoder (Ellis et al. 2020), which learns *concepts* incrementally.

Acknowledgments. At UMD, this work has been supported in part by ONR grant N000142012257 and NRL grants N0017320P0399 and N00173191G001. At NRL, Mark Roberts thanks ONR and NRL for funding this research. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

Bengio, Y.; Louradour, J.; Collobert, R.; and Weston, J. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, 41–48.

Choi, D.; and Langley, P. 2005. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, 51–68. Springer.

Ellis, K.; Wong, C.; Nye, M.; Sable-Meyer, M.; Cary, L.; Morales, L.; Hewitt, L.; Solar-Lezama, A.; and Tenenbaum, J. B. 2020. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2016. Learning hierarchical task models from input traces. *Computational Intelligence*, 32(1): 3–48.

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 *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*, 9883–9891. AAAI Press.

Lotinac, D.; and Jonsson, A. 2016. Constructing hierarchical task models using invariance analysis. In *ECAI 2016*, 1274–1282. IOS Press.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, 968–973.

Xiao, Z.; Wan, H.; Zhuo, H. H.; Herzig, A.; Perrussel, L.; and Chen, P. 2020. Refining HTN Methods via Task Insertion with Preferences. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 10009–10016.

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

Urban Modeling via Hierarchical Task Network Planning

Michael Staud

StaudSoft UG, Ravensburg , Germany
michael.staud@staudsoft.com

Abstract

In this paper we present a new method for city modeling based on hierarchical task network planning. The planner creates actions that are executed in a city simulation. These actions generate step by step a city model within the simulation.

The advantage of this approach is that it takes into account that real cities are not designed on a drawing board, but have a history of development. By simulating this development, economic aspects can be taken into account. The result is a much more realistic urban model.

An urban simulation is an extremely complex planning domain for a planner. Therefore, we have developed a new domain-independent hierarchical task network planning algorithm that divides the planning problem into smaller planning problems. Our planning algorithm is sound and complete. We give preliminary results on its performance.

1 Introduction

Cities play a very important role in our daily lives. Most of our life takes place in a city, cities have shaped our social development. Therefore, it is important to be able to create models of cities quickly and efficiently. This field is called urban modeling. It involves either creating virtual cities based on existing cities or creating virtual cities based on parameters (Kelly and McCabe 2007). These can then be used in movies, games or artificial intelligence (Formichella, Lucien 2020; Glockner, Cyrill 2018).

Virtual cities today are generated through *procedural generation* which is used to mimic the work of artists (Freiknecht and Effelsberg 2017; Van Der Linden, Lopes, and Bidarra 2013). Procedural generation itself is not limited to generating cities, but can also be used to create textures, sounds, models or even stories (Togelius et al. 2013). A popular approach is to use a *modified L-system*(Rozenberg and Salomaa 1980) to create the road network of an urban environment. Buildings, also generated using an L-system, are then placed between the roads (Parish and Müller 2001). Other approaches include agent-based simulations (Lechner, Watson, and Wilensky 2003), template-based generation (Sun et al. 2002) or grid layouts (Kelly and McCabe 2006).

The problem with all of the approaches mentioned so far is that they only attempt to mimic the structure of a city. They do not create the structure of a city, nor do they deal with growth and development over time. However, a city grows depending on its economic structure. Businesses or factories grow only where they are profitable. Residential areas grow only when jobs are available. In addition, our approach generates a lot of other information that is useful for a credible simulation (building type, traffic routes, ...).

1.1 Our Approach

We therefore propose a new way to model cities by formulating the complex interactions in a city as a planning domain. To create a city, a planning problem is defined that contains information about the goal (e.g., to create a city of a certain size) and the environment. The environment consists of the terrain, which is defined as a graph and contains information about whether a node is buildable, whether there is water, and what resources are available (see Section 5).

One problem we faced in our approach was that such a planning problem is too large to be handled by current planners. To solve this problem, we use a new type of domain-independent hierarchical task network planning (HTN planning) (Ghallab, Nau, and Traverso 2004). We defined a new type of abstract tasks that are defined by the domain designer. We call them *separable abstract tasks* (see Section 3). When our planning algorithm encounters such a task, it generates a new independent planning problem from it and tries to solve it independently of the main problem. This process can be recursive. We call this a planning process (see Section 4.2).

To make this work, we use what we call a *hierarchical world state*. Several abstraction layers are defined in the domain. The number of layers is stored in the variable n. Each layer contains the same information, but in a more abstract form. Layer one is the most abstracted. The layer n contains only non-abstracted information (see Section 3).

Our contribution is:

- A new sound and complete domain-independent HTN planning algorithm (see Section 4).
- The application of HTN planning to urban modeling (see Section 5).

In the next section, we will first describe HTN planning in detail and then our new approach. Then we will describe the domain we used to create an urban simulation. After that, we will present some preliminary results.

2 Hierarchical Task Planning

We define the set of all constants as C and the set of all variables as V. A literal is an atom or its negation. The set of all atoms is denoted by A. An atom is a predicate applied to a tuple of terms. A term can be a constant or a variable.

The following definitions are adapted from Bercher, Keen, and Biundo (2014). We use total-order hierarchical task network planning (Ghallab, Nau, and Traverso 2004, 238). A hierarchical planning domain is a tuple $D = (T_a, T_p, M)$ containing 3 finite sets. T_a is the set of abstract tasks, T_p is the set of primitive tasks, and M is the set of methods. Primitive and abstract tasks are also tuples of the form $t(\overline{\tau}) = \langle prec_t(\overline{\tau}), eff_t(\overline{\tau})\rangle$. Each task has a precondition $prec_t(\overline{\tau})$ and an effect $eff_t(\overline{\tau})$. Moreover, each task has a set of parameters $\overline{\tau}$. If all parameters of a primitive task are equal to atoms, the task is grounded and is called an action. A method is a tuple $m = \langle t_a(\overline{\tau}_m), P_m\rangle$, where $t_a(\overline{\tau}_m)$ is the abstract task it can replace, P_m is a set of plan steps and $\overline{\tau}_m$ are the parameters of the method. A plan step is a uniquely labeled task $l : t(\overline{\tau})$. A plan is also a sequence of plan steps. A solution is a plan in which each task is an action and its preconditions are satisfied at each step. In addition, the solution transforms the initial state into the goal state. A task can be executed in a particular world state only if its preconditions are satisfied. The preconditions, which are a set of literals, are satisfied if every positive literal is in the current world state and if every negative literal is not in the current world state. The effects of a task are also a set of literals. Positive literals add atoms to the world state when the task is "executed". Negative literals remove an atom from the world state. The function $\sigma : \mathcal{P} \times T_p \to \mathcal{P}$ applies the effects of a primitive task to a world state.

A plan step $l : t(\overline{\tau})$ associated with an abstract task can be decomposed by a method $m = \langle t_a(\overline{\tau}_m), P_m\rangle$, where the plan step l in the current plan is replaced by the plan steps in P_m. A world state w is a set of atoms. A *derived predicate* (Edelkamp and Hoffmann 2004) is defined by a rule containing a logical formula with variables. If the rule of a predicate evaluates to true, it is added to the world state. Otherwise, it is removed. The logical formula may contain quantifiers.

A problem is a tuple $P = \langle init_P, goal_P, PS_P\rangle$, where $init_P$ is the initial state consisting of atoms, the goal $goal_P$ is a set of literals. The initial plan is stored in PS_P.

3 Hierarchical World State

The world state is divided into n layers. Each predicate and task has an associated *hierarchical layer* l. The layer 1 has the highest level of abstraction (see Section 5 for an example). The layer with the highest index n stores the actual, non-abstracted facts of the simulation. A task can only change predicates on the same layer. To jump from layer l to layer $l + 1$ during the planning process, we introduce a new type of abstract tasks, the *separable abstract tasks*. They are not directly decomposed into a method if they do not appear in the initial plan $init_P$. Instead, they give the planner a hint that this abstract task needs to be decomposed in another planning process that operates at layer $l + 1$. It should be noted that a separable abstract task can occur in the initial state of one planning process and by decomposition in another planning process. This is the reason why we do not simply use normal abstract tasks in the initial state. How the planning processes operates in detail is described in Section 4.2.

3.1 Information Flow to the Layer $l + 1$

The following rules enforce the separation of layers:

- A task of layer i can only use predicates in its precondition of layer $l \leq i$.
- A task can only have predicates in its effects of the same layer as itself.
- Primitive tasks are allowed only in the highest layer n.
- Normal abstract tasks cannot have effects.
- Separable abstract tasks may only occur in the layers $l < n$.
- Normal abstract tasks of layer l can only be decomposed into plan steps with tasks of layer l.
- Separable abstract tasks of layer l can only be decomposed into plan steps with tasks of layer $l + 1$.

Therefore, the initial tasks belong to layer 1 and it must be ensured by the domain developer that they can be decomposed into the primitive tasks across the n layers.

3.2 Information Flow to the Layer $l - 1$

To pass information from a layer l to a layer $l - 1$, all predicates on the layers $l < n$ must be derived predicates unless they never occur in an effect of a task. A derived predicate of layer l can only use predicates in its formula that come from layer $l+1$. Derived predicates of layer l' are only updated in the world state of the main system or in a planning instance of layer l if $l' < l$ holds (see Section 4). If the predicates have the same layer as the planning instance, they are not treated as derived predicates. Thus, effects that change derived predicates do not cause inconsistencies in a planning process of layer l, since a task of layer l can only change predicates of layer l. And there is also no inconsistency in the world state of the main system, since only primitive tasks are applied to it and they cannot change derived predicates.

4 Planning Algorithm

The general idea is to split the planning process into many smaller processes. This can increase the performance by an exponential factor (Korf 1987). The system consists of 2 main modules (see Figure 1):

- **Main System**: stores the *main plan* that is created during the planning process. It also stores the world state w, which is initialized by the initial state $init_P$ of the problem. The main system always contains the entire world state (see Algorithm 1).
  The main plan is a series of plan steps of primitive tasks. When the planning algorithm is finished, it will contain the plan which will transform the initial state into the goal state, if this is possible.
- **Planning Processes**: These are small planning problems that are solved during the planning process and operate on a particular layer l of the world state (but they can contain predicates of a level $l' < l$). At each step of the

main system, all planning processes are invoked. When not suspended, a process takes the current world state of the main system as input to provide the next action to be added to the main plan. However, it does not use the entire world state for planning, only the horizon (see Section 4.1). Similar to adversarial search, no complete plan is created, only the next best action is returned (though the other possible actions are stored to allow backtracking).

If it is not possible to generate a next action, the entire system (main system and planning processes) will backtrack. This means that the action added to the main plan is removed, planning processes can be removed or deleted. This process continues until a step is found where another choice point can be selected in the search tree of a planning process. In our practical experiments, backtracking was never used because each planning process always found a solution that was valid in the world state of the main system.

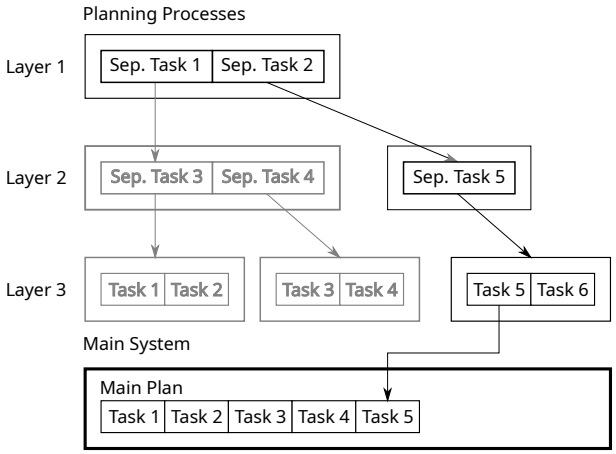


Figure 1: Example of a planning process in a domain with 3 layers. The grey processes are already completed.

4.1 Horizon

The horizon is a set containing all atoms that can be used in a planning problem. By "used" it is meant that they may appear in a precondition or effect of a task during the planning process. Formally, this means that the set contains exactly those atoms that occur in each task of the task decomposition graph of a planning process (Bercher, Keen, and Biundo 2014). When constructing the task decomposition graph, separable abstract tasks are treated in the same way as in the planning process. They are not decomposed unless they occur in the initial plan of the planning problem. When derived predicates occur in the graph, the predicates on which they depend are added to the horizon only if their level l is equal to or less than the level of the planning process (see Section 4.2). All this makes the horizon much smaller than the original problem and increases planning performance (both in terms of time and memory).

4.2 Planning Processes

When a planning process plans, it treats separable abstract tasks as primitive tasks unless they occur in the initial plan of the planning problem. If they do occur in the initial plan, they are decomposed like a normal abstract task. Similar to primitive tasks, separable abstract tasks are passed to the main system. However, they are then handled differently. Instead of adding them to the main plan, a new subplanning process is started by the main system and added to the list of planning processes. The planning process that tried to execute the separable abstract task is suspended until the new subplanning process finishes its task. The subordinate planning process can in turn create new subordinate processes. This is illustrated in Figure 1.

- **Primary Planning Process**: This process directly tries to solve the given problem. This means that its *initial state* and *goal* match with the one defined in the problem. However, it does so only within its horizon (see Section 4.1) using derived tasks and separable abstract tasks. It is initially generated by the main system and not by a separable task. Layer 1 is assigned to it. Therefore, the goal $goal_P$ can only contain predicates associated with layer 1.
- **Instanced Planning Process**: This is created whenever a separable abstract task of layer l is returned by a planning process as the next task. The *goal* of the planning process is equal to the effects of the separable abstract task. The *initial plan* contains only the separable abstract task. It takes the world state of the main system at each step and then determines the task that will be returned to the main system. So the *initial state* is equal to the current world state of the main system. The layer $l+1$ is assigned to it.

A single planning process is much easier to solve than the full planning problem because it plans only within its own horizon (see Section 4.1).

The algorithm of the main system can be seen in Algorithm 1. The initial problem is the tuple $P = \langle init_P, goal_P, PS_P \rangle$. A planning process is a tuple $p_p = (H_s, G, P_T, S, l)$, where H_s is the current task stack containing which primitive tasks to execute and which abstract tasks to decompose. G stores the goal of the planning process. P_T is the parent planning process that produced this process. For the primary planning process, it holds $P_T = \bot$. Whether the process is suspended because it is waiting until another planning process is finished is stored in $S \in \{\top, \bot\}$. The layer of the planning process is stored in l. The $getNextTask$ function triggers the planning algorithm of a process. It returns $\bot$ if the process was suspended or if failed. Otherwise, it returns a separable abstract task or a primitive task. Abstract tasks are never returned. The derived predicates are updated in $update\_hierarchical\_state(w)$.

Our planning system uses the Monte Carlo tree search algorithm (Kocsis and Szepesvári 2006) in the planning processes to determine the next action to add to the main system. It uses forward decomposition (Ghallab, Nau, and Traverso 2004, 238). We use the H0 heuristic (Ghallab, Nau, and Traverso 2004, 201) to estimate the distance to the target in the playouts.

Algorithm 1: The algorithm in the main system. The main plan is stored in the sequence m.

$P_l = \{(PS_P, G, \bot, \bot, 1)\}$
$w = init_P$
$m = ()$
repeat
  for $p_p = (H_p, G_p, P_p, S_p, l_p) \in P_l$ **do**
    $t = getNextTask(w, p_p)$
    if $t = \bot$ **then**
      continue or backtrack if p_p failed
    end if
    if $t \in T_p$ **then**
      $w = \sigma(w, t)$, $m = (m_1, \ldots, m_{|m|}, t)$
      $w = update\_hierarchical\_state(w)$
      if $finished(p_p)$ **then**
        $P_l = P_l \setminus \{p_p\}$
        $unsuspend(P_p)$
      end if
    else
      $p_{new} = ((t), eff_t, p_p, \bot, l_p + 1)$
      $P_l = P_l \cup \{p_{new}\}$
      $suspend(p_p)$
    end if
  end for
until $isGoal(w, G)$

4.3 Theoretical Properties

The algorithm is sound because it only performs valid actions in the domain. Otherwise, it will backtrack. The algorithm finds a solution if the problem does not allow infinite decompositions of abstract tasks. In this case, the algorithm is also complete since the search space is finite. It should be noted that an HTN planning algorithm can be made complete on more unrestricted domains by performing additional checks (Nau et al. 2001).

5 Urban Simulation Domain

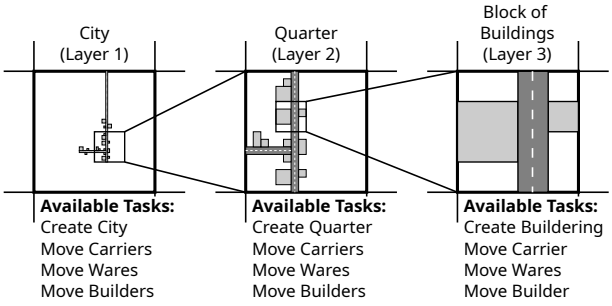


Figure 2: Hierarchical structure in the urban simulation domain.

Our planning algorithm is used for planning in an urban simulation. The rules for the simulation are declared in HDDL (Höller et al. 2020). Units and buildings are placed on a grid. Units can move on the grid. Buildings can be built at specific positions and cannot move. Each unit and building has unique capabilities. The builder unit can construct buildings. Roads are special buildings. The carrier unit can move goods between buildings. The main building can create new builder or carrier units. The woodcutter building can cut down trees and create wood. We tested an environment with a grid size of 64×64.

The domain contains separable abstract tasks for performing certain actions at different layers of the hierarchical world state. For example, creating a city, creating a neighborhood, or moving a building unit (see Figure 2). The source code of the domain is available on the Internet (Staud 2022).

6 Results

We measure how many states our algorithm (Separable Hierarchical Task Network Algorithm, SHTN for short) needs to solve multiple given goals in our urban simulation domain when using separable abstract tasks and compare it with to Forward Decomposition (Ghallab, Nau, and Traverso 2004, 238) Monte Carlo tree search (FDMCTS). This algorithm differs from ours only in that separable abstract tasks are decomposed like normal abstract tasks and thus no additional planning processes are generated. There is only the main planning instance and the hierarchical world state rules (see Section 3) are not applied.

The results can be seen in Table 1. For both of the Monte Carlo algorithms, we used a fixed number of iterations in each call to $getNextTask(w, p_p)$. In the playouts, we explored the search space in the playouts until all normal abstract tasks were randomly decomposed. Our algorithm then requires much fewer states because it does not need to explore the decomposition of separable abstract tasks. In addition, we measured the size of the horizon of the different planning instances and compared it to the size of the original planning problem. These results can be found in Table 2.

Goal	SHTN	FDMCTS
Create City	6925	6542100
Create Carrier	10716	8060040
Builder Move	38441	33479127

Table 1: Performance comparisons: Total number of states visited in the search space to reach the specified goal.

7 Conclusions

We have presented a novel HTN planning algorithm that is capable of planning in an urban simulation. The proof-of-concept shows that this approach is successful. And our tests show that it can also solve larger and more problems, as it performed very well in our experiments. Our next goal is to use Deep Learning to better predict the actual effects of separable abstract tasks. And then we want to find out how well this algorithm can perform in an adversarial environment. An opposing player in this case can be, for example, the crime rate, an enemy power in a war, or environmental influences can be modeled as opponents. These can be

Primary	Initial Atoms
Full Problem	120325
Primary Problem	338
Create City (Layer 2)	935
Create Quarter (Layer 3)	2803
Create Carriers (Layer 2)	936
Create Carriers (Layer 3)	2804
Move Builders (Layer 2)	1353
Move Builders (Layer 3)	4567

Table 2: Size of the horizons of various planning processes created by separable abstract tasks. The primary problem is the amount of atoms in the primary planning process. The full problem contains all the atoms of the problem without a horizon. The size of each horizons is on average 100 times smaller than the original planning problem.

floods, fires, or heavy rains. By modeling these as adversaries rather than random effects, the algorithm is forced to consider these effects in every part of the urban environment.

References

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid Planning Heuristics Based on Task Decomposition Graphs. In *SoCS 2014*, 35–43. AAAI Press.

Edelkamp, S.; and Hoffmann, J. 2004. PDDL 2.2: The Language for the Classical Part of IPC-4. In *Int. Planning Competition*.

Formichella, Lucien. 2020. Fourteen Groundbreaking Movies That Took Special Effects to New Levels. https://www.insider.com/most-groundbreaking-cgi-movies-ever-created-2020-1. Accessed: 2022-02-16.

Freiknecht, J.; and Effelsberg, W. 2017. A Survey on the Procedural Generation of Virtual Worlds. *Multimodal Technologies and Interaction*, 1(4): 27.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier.

Glockner, Cyrill. 2018. Simulators: The Key Training Environment for Applied Deep Reinforcement Learning. https://towardsdatascience.com/simulators-the-key-training-environment-for-applied-deep-reinforcement-learning-9a54353f494f. Accessed: 2022-02-16.

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 *Proc. of the AAAI Conference on AI*, volume 34, 9883–9891.

Kelly, G.; and McCabe, H. 2006. A Survey of Procedural Techniques for City Generation. *ITB Journal*, 14(3): 342–351.

Kelly, G.; and McCabe, H. 2007. Citygen: An Interactive System for Procedural City Generation. In *5th Int. Conf. on Game Design and Technology*, 8–16.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *ECML*, 282–293. Springer.

Korf, R. E. 1987. Planning as Search: A Quantitative Approach. *AI 87*, 33(1): 65–88.

Lechner, T.; Watson, B.; and Wilensky, U. 2003. Procedural City Modeling. In *1st Midwestern Graphics Conf.*

Nau, D.; Munoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-Order Planning with Partially Ordered Subtasks. In *IJCAI*, volume 1, 425–430.

Parish, Y. I.; and Müller, P. 2001. Procedural Modeling of Cities. In *Proc. of SIGGRAPH-01*, 301–308.

Rozenberg, G.; and Salomaa, A. 1980. *The Mathematical Theory of L-Systems*. Academic Press.

Staud, M. 2022. Urban Simulation HDDL Domain. https://www.staudsoft.com/urbansimulation.html. Accessed: 2022-05-29.

Sun, J.; Yu, X.; Baciu, G.; and Green, M. 2002. Template-Based Generation of Road Networks for Virtual City Modeling. In *Proc. of VRST-01*, 33–40.

Togelius, J.; Champandard, A. J.; Lanzi, P. L.; Mateas, M.; Paiva, A.; Preuss, M.; and Stanley, K. O. 2013. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Artificial and Computational Intelligence in Games*, 61–75. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Van Der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1): 78–89.

Towards Hierarchical Task Network Planning as Constraint Satisfaction Problem

Tobias Schwartz, Michael Sioutis, Diedrich Wolter

University of Bamberg
An der Weberei 5
Bamberg
{tobias.schwartz, michail.sioutis, diedrich.wolter}@uni-bamberg.de

Abstract

In recent years, propositional logic encodings for HTN planning have seen many improvements and resulted in competitive planners. Modeling all kinds of features and constraints imposed by the task hierarchy, however, is very challenging in propositional logic, and has recently led to including preprocessing steps before creating the SAT formulas. Instead of using propositional logic, classical planning problems have previously been encoded as constraint satisfaction problems (CSPs), which are more expressive. Indeed, CSPs allow a more natural and convenient way of representing all constraints of the task hierarchy, yet only little work on using constraint solving methods in HTN planning exist. Hence, in this paper, we outline first ideas for encoding an HTN planning problem into a single CSP. Our motivation lies in obtaining constraint networks for HTN planning that we can solve with state-of-the-art solvers.

Introduction

Many real-world tasks deal with some form of constraints. As such, constraint satisfaction techniques have proven successful as an underlying framework for solving problems of various domains. Constraints, in this context, can express a large bandwidth of problem features, from simple ordering relations to modeling complex numerical resource allocations. Despite its success in many fields, constraint satisfaction methods have been applied only sparsely in the field of AI planning, especially in hierarchical task network (HTN) planning. Instead, recently, HTN planning problems have often been encoded as a sequence of propositional satisfiability (SAT) problems (Behnke, Höller, and Biundo 2018, 2019a; Schreiber et al. 2019; Schreiber 2021). Since HTN planning is generally undecidable, one cannot simply compile the problem into propositional logic, but rather fix the number of possible actions and then employ SAT techniques to verify whether there exists a plan of a particular length. This process may be repeated for a series of compilations of increasing length (Bercher, Alford, and Höller 2019).

Compared to a SAT compilation, formulating a planning problem as a constraint satisfaction problem (CSP) can be characterized as more natural or convenient (Nareyek et al. 2005). This is mostly attributed to the fact that CSPs support higher-level constraints that are able to directly represent domain-specific knowledge. In SAT, for example, quantitative information in the form of discrete numbers may be captured in a propositional formula by using a variable for each value of the discrete domain, but doing so will result in loosing the natural ordering information of the numbers.

As noted by Barták and Toropila (2008), early constraint models for planning had their origins in SAT and thus were only using Boolean variables and constraints in form of logical formulas. Then, they proposed multiple constraint models for classical planning that rely on more sophisticated constraint structures, clearly improving on their logical counterparts in terms of both stronger constraint propagation and faster runtime.

In contrast to classical planning, HTN planning imposes additional structural constraints of task hierarchies that must be satisfied. Those additional constraints increasingly motivate the use of constraint programming. Unfortunately, early reports of ongoing work in this direction (Surynek and Barták 2005) have apparently not been followed through. Stock et al. (2015) provide the only successful application of CSPs for solving HTN planning problems, known to us. They, however, rely on a more complex architecture with multiple constraint networks used independently of one another, each handling a different form of knowledge (e.g., causal, temporal, spatial), called meta-CSP (Mansouri and Pecora 2016).

Instead, in this paper, we outline first ideas for encoding HTN planning as one single CSP. Our motivation lies in obtaining constraint networks for HTN planning that we can solve with state-of-the-art off-the-shelf solvers, and not relying on tailored solution to this problem.

Preliminaries

We start by briefly presenting some frameworks that are relevant to our work and we will be referring to in this paper.

Qualitative Constraint Satisfaction Problems

In the following, we focus on qualitative constraint satisfaction problems (QCSPs), which are typically used to represent and reason about qualitative temporal (or spatial) information. They are defined analogously to classical CSPs (Russell and Norvig 2020), but allow variables to be of infinite domains. QCSPs are often tackled via the use of a qualitative constraint graph, called *Qualitative Constraint Network* (QCN), which is defined as follows.

Definition 1 (QCN). *A* QCN *is a tuple* (V, C) *where:*

- $V = \{v_1, \ldots, v_n\}$ *is a non-empty finite set of variables, each representing an entity of an infinite domain* D*;*
- *and* C *is a mapping* $C : V \times V \rightarrow 2^{\mathsf{B}}$ *such that* $C(v, v) = \{\mathsf{Id}\}$ *for all* $v \in V$ *and* $C(v, v') = C(v', v)^{-1}$ *for all* $v, v' \in V$.

Let $\mathcal{N} = (V, C)$ be a QCN, then a *solution* of $\mathcal{N}$ is a mapping $\sigma : V \rightarrow \mathsf{D}$ such that $\forall v, v' \in V$, $\exists b \in C(v, v')$ such that $(\sigma(v), \sigma(v')) \in b$, and $\mathcal{N}$ is *satisfiable* (or consistent) iff it admits a solution (Ligozat 2013; Dylla et al. 2017).

We assume our constraint language to be defined like the well-known Interval Algebra (Allen 1983), which is a first-order theory for representing and reasoning about temporal information. For now, we make use of its equality (eq), inequality ($neq = \{\mathsf{B} \setminus eq\}$), and ordering constraints ($\{<, >\}$).

Hierarchical Task Network Planning

Hierarchical planning extends classical planning by introducing a task hierarchy. Instead of only using the notion of applicable actions, it essentially differentiates between primitive and compound tasks. Primitive tasks are hereby comparable to the actions in classical planning. Compound tasks describe a more abstract notion of a set of actions. This grouping can impose additional restrictions that might not be easily achievable using only preconditions and effects of actions. For example, an imposed ordering constraint can be easily encoded in a compound task and drastically improve efficiency of the planner. In fact, ordering tasks according to a partial order can be seen as the motivation behind hierarchical task network (HTN) planning, perhaps the most basic hierarchical formalism (Bercher, Alford, and Höller 2019). In what follows, we briefly recall the definitions for lifted HTN planning as recently defined in the hierarchical domain definition language (HDDL) (Höller et al. 2020).

The basis for HTN planning is the so-called *task network*, which essentially imposes a strict partial order on a finite set of tasks. A set of variable constraints may constrain certain task parameters to be (non-)equal to other task parameters or constants, or to (not) be of a certain type. A task network is called *ground* if all parameters are bound to constants.

An HTN planning domain D defines the sets of all primitive tasks T_P, compound tasks T_C, and decomposition methods M. A method $m \in M$ is a triple (c, tn, VC) of a compound task name $c \in T_C$, a task network $tn \in T_P \cup T_C$ and some variable constraints VC over the parameters of c and tn. An HTN planning problem P is a tuple (D, s_I, tn_I, g), where D is the planning domain, $s_I \in S$ is the initial state, tn_I is the initial task network, and g optionally defines a goal description.

Although a goal description can be defined, the objective in HTN planning is not to achieve a certain state-based goal. Instead, a *solution* to a given HTN planning problem is a final task network tn_S which is reachable from s_I by only applying methods and compound tasks. In the process, all compound tasks need to be decomposed into primitive actions, such that tn_S does not contain compound tasks anymore. The enforced task hierarchy directly restricts the set of possible solutions to only those that can be obtained by task decomposition (Bercher, Alford, and Höller 2019).

Listing 1: Action `drive` in HDDL

```
(:action drive
  :parameters (?l1 ?l2 - location)
  :precondition (and
    (tAt ?l1)
    (road ?l1 ?l2))
  :effect (and
    (not (tAt ?l1))
    (tAt ?l2)))
...)
```

HTN Planning Constraint Model

Similar to Barták and Toropila (2008), we employ a multi-valued representation of the planning problem. That is, instead of grounding every single fact using enumeration, we create *state variables* for different fragments of the world state, where the domains of values represent exclusive options. For example, given a service robot, the robot may only be at one particular location at any given time. Instead of now generating all combined facts of a robot being at a particular location, all potential locations represent the domain for the state variable of the robots location. Using such a multi-valued representation instead of a purely propositional, fact-based encoding, the number of variables decreases, whereas the size of the domains increases. This is generally recommended for constraint modeling, as opposed to the other way around (many variables of small domains) (Barták and Toropila 2008).

We follow the constraint model proposed by Ghallab, Nau, and Traverso (2004), coined the *straightforward model* by Barták and Toropila (2008). For now, we only consider HTN features defined by the *Hierarchical Domain Definition Language* (HDDL) (Höller et al. 2020).

A CSP denoting the problem of finding a plan of length n, consists of $n+1$ incrementally changing constraint networks $\mathcal{N}$, where the k^{th} network $\mathcal{N}_k^i$ represents the state s_k^i after performing $k-1$ planning operations and i incremental task decompositions. Since the plan length n, i.e., the sequence of primitive actions in the final plan, is generally unknown in advance, we dynamically grow the list of constraint networks $\mathcal{N}$ whenever we perform a primitive action in state s_k^i which results in a new successor state s_{k+1}^i. Describing states as constraint networks $\mathcal{N}$, is a variation in presentation from Barták and Toropila (2008), as they describe states as sets of v multi-valued variables.

In the following, we elaborate on the constraint network design and point out how the features of HDDL can be expressed within this notation.

Variable representation: We model the variables in a special constraint network, which ensures a mapping of each state variable to exactly one object of the problem domain. To illustrate this, consider the action `drive` from the transport domain presented by Höller et al. (2020), given in Listing 1. The corresponding problem file further specifies the

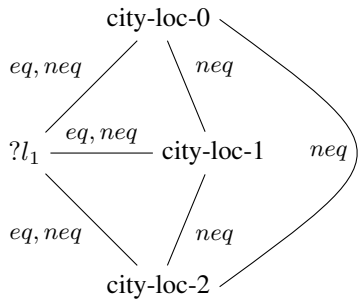


Figure 1: Grounding of location $?l_1$ as qualitative constraint network $\mathcal{N}$.

relevant location objects as city-loc-0, city-loc-1, or city-loc-2. We can formulate a mapping from the location $?l_1$, given as parameter, to a particular object specified by the problem, as a constraint network (Figure 1), where $?l_1$ is either equal (eq) or not equal (neq) to any of the objects. By preferring eq over neq in the search, and ensuring that all objects are different (i.e., neq), we guarantee a mapping to at most one of the objects. Indeed, once we commit to any one of the equality relations, constraint propagation will force all other relations to become neq, due to the neq constraint between all of the objects. We can easily extend this formulation with other variables from the same domain (such as $?l_2$), by adding them as new nodes to the network and creating the same eq, neq constraints to all of the objects. Note that all such variables by default are independent in terms of constraint propagation and thus mapping one variable to an object does not influence the other variable mappings. Obviously, we can change this behavior by adding additional constraints (such as neq) between variables if desired. Furthermore, note that by default this constraint network approach postpones all variable mappings without explicit constraints until eventually the CSP solver is called. For example, imagine that for action `drive` we could have multiple vehicles available at $?l_1$ of which any particular one could be used to drive to $?l_2$. If no direct constraints are imposed, initially we only set the eq, neq constraints which postpone this decision to the latest point in time.

Action representation: Action application works just like in classical planning, where for a given state s_k^i action variable $A^{s_k^i}$ acts as a logical constraint, leading from one constraint network $\mathcal{N}_k^i$ to the next $\mathcal{N}_{k+1}^i$. Clearly, all required preconditions of the action need to be satisfied in $\mathcal{N}_k^i$, then after its execution all effects of the action hold in $\mathcal{N}_{k+1}^i$. We follow the formulation by Barták and Toropila (2008) and model this relation using logical implications, i.e. for any action variable $A^{s_k^i}$ in state s_k^i we have,

$$A^{s_k^i} = a \rightarrow \textit{Pre(a)}^{s_k^i}, \forall a \in Dom(A^{s_k^i}),$$

$$A^{s_k^i} = a \rightarrow \textit{Eff(a)}^{s_{k+1}^i}, \forall a \in Dom(A^{s_k^i})$$

where $\textit{Pre(a)}^{s_k^i}$ is a conjunction of equalities changing the required relations within the constraint network to reflect the preconditions of action a in state s_k^i. Similarly, $\textit{Eff(a)}^{s_{k+1}^i}$ expresses the effects of a in the successor state s_{k+1}^i. Barták and Toropila (2008) additionally introduce constraints for the *frame axioms*, i.e. constraints to ensure that any state variable unaffected by the action remains unchanged. In our representation this naturally holds as we consider state s_{k+1}^i as a copy of its predecessor s_k^i and only change the variables affected by the action. A final refinement, e.g., variable binding $?l_1 \xrightarrow{eq}$ city-loc-0, which is not directly affected by the action, can later not be changed through constraint propagation but would only lead to inconsistency.

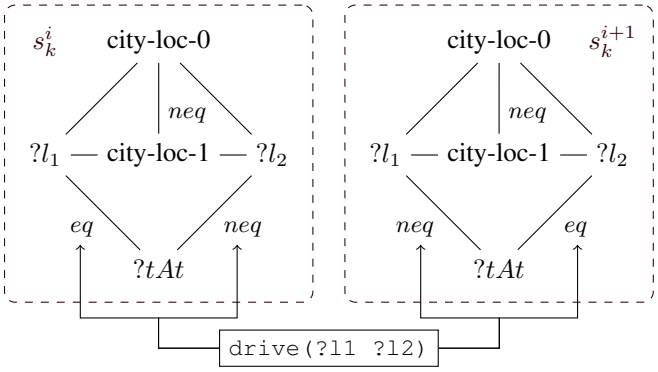


Figure 2: Applying action `drive` (Listing 1) in state s_k^i (left) to derive at s_{k+1}^i (right).

Note that action $a \in Dom(A^{s_k^i})$ may only be one of many choices possible in the given state s_k^i. However, the effect of the action and hence the successor state s_{k+1}^i depends on which action was chosen. To this end, we employ the same implication structure as outlined above, requiring the constraint solver to handle such instances. Figure 2 illustrates this notion on a simple example of applying action `drive`. Formulating this implication directly within the realms of the constraint language can eliminate the need for extending the state-of-the-art solvers that we have today for solving qualitative constraint networks and is part of future work.

Abstract task representation: In HDDL, abstract tasks are defined explicitly in the domain. They represent a form of abstraction from the specific method used to fulfill a certain task, already defining the parameters and their respective input types. We can use this information to establish the same eq, neq constraints to all variables of the domain indicated by the type. We hereby convey the information that each parameter should be linked to exactly one variable of its domain, without making this link explicit yet.

Method representation: As described above, methods are always linked to an abstract task. However, they may define further parameters beyond the ones already stated in the abstract task definition.

Generally, methods describe a fixed number of subtasks that have to be fulfilled in order to complete the task. Some HTN planning systems, in particular those employing SAT compilation techniques (e.g., Behnke, Höller, and Biundo (2018); Schreiber et al. (2019); Schreiber (2021)), rely on

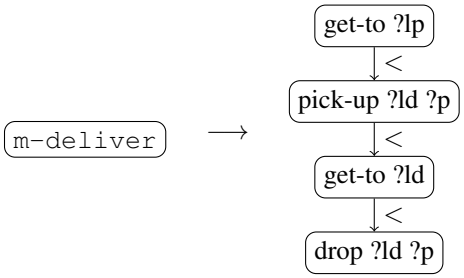


Figure 3: Applying method `m-deliver` to decompose task `deliver` in state s_k^i (left) to derive at s_k^{i+1} (right). For full method and task descriptions, see Höller et al. (2020).

a totally ordered set of those subtasks. By contrast, HDDL also supports partially ordered subtasks. Conveying the relative ordering of subtasks has been described as the main difficulty for a SAT encoding (Behnke, Höller, and Biundo 2019a). Where Behnke, Höller, and Biundo (2019a) rely on a preprocessing step, conducting reasoning on the order before creating the actual SAT formula, CSPs naturally allow the representation of ordering constraints.

Where an action progresses the state one step from k to $k+1$ within a single layer of the decomposition tree, a task decomposition introduces an incremental change along the hierarchy of the decomposition tree, moving from layer i to $i+1$. Given a task variable $T^{s_k^i}$ in state s_k^i, we then have,

$$T^{s_k^i} = m \rightarrow \textit{Dec(m)}^{s_k^{i+1}}, \forall m \in Dom(T^{s_k^i})$$

where *Dec(m)* is the decomposition effect of the method m, adding all subtasks to the constraint network $\mathcal{N}_k^{i+1}$. Any ordering relation between those subtasks can simply be established by introducing ordering constraints in the network.

Similarly to the incremental SAT encoding by Schreiber et al. (2019), we must ensure that for finding a plan of length n, the final increment of all constraint networks does not contain any task or method variables.

Discussion and Related Work

Mali and Kambhampati (1998) were the first to propose a propositional logic encoding for HTN planning problems. However, these encodings were restricted to non-recursive domains. Only in recent years, development of new SAT encodings overcame this restriction and resulted in competitive performance. Initially, these encodings were only applicable to the subset of totally-ordered HTN planning problems (Behnke, Höller, and Biundo 2018; Schreiber et al. 2019). By now, SAT encodings have been extended to partially-ordered problems and used to find optimal plans with respect to the plan length (Behnke, Höller, and Biundo 2019a,b).

As also summarized by Schreiber (2021), all these SAT encodings operate similarly. Their encodings are iteratively extended along the depth of the hierarchy, instead of the length of a final plan (as done in classical planning). Additionally, they all require a prior grounding procedure. While it has been shown that grounding often improves subsequent search algorithms, on some planning domains, grounding suffers from intrinsic scaling problems. Thus, Schreiber (2021) developed a lifted SAT planner that omits grounding, but instead is limited to totally-ordered problems.

A CSP encoding, as discussed in this paper, by design avoids the need for grounding. Furthermore, additional constraints can essentially be modeled for free, which make this approach quite comfortable in dealing with partially-ordered HTN planning problems.

The advantages of a CSP encoding have previously been explored by Stock et al. (2015). But in contrast to our work, they rely on a sophisticated reasoning framework, called meta-CSP (Mansouri and Pecora 2016), as underlying architecture. This framework has the advantage that it allows reasoning with knowledge of different forms (such as temporal, spatial, causal, resources). It comes, however, with the drawback that finding consistent solutions is generally slow, since each form of knowledge is dealt with in a separate constraint network and finding a consistent solution requires all networks to be consistent at once. This requires special solvers that allow interaction among each other. Instead, we are motivated to find an encoding which uses only one single constraint network architecture, such that we can employ state-of-the-art solvers that we have today for solving qualitative constraint networks. This may be seen as an approach in between a pure propositional logic encoding on one side and a quite complex constraint-based encoding on the other side. We argue that this allows us to combine the advantages of both worlds.

For now, a few challenges remain that may impact the success of the proposed CSP encoding. First, we assume that dynamically creating new constraint networks both within one hierarchy for primitive action application and following the hierarchy for compound task decomposition is feasible without computational blow-up. We here expect that state-of-the-art qualitative constraint solvers can help to drastically reduce the state space as inconsistent configurations can be pruned early, as done traditionally in CSP search. Second, action variables are mapped to their respective domain indicated by the variable type. We currently do not consider the case where the type of a variable may change or is unavailable. Without type information a mapping to all ground objects can be done, albeit inefficiently. While a variable is not yet ground, changing its type is simple, as this just changes the possible mappings and consequently disallows all others (by use of neq constraints).

Conclusion and Future Work

In this paper, we have outlined first steps towards constructing a constraint satisfaction problem (CSP) encoding for HTN planning problems. CSPs have been used in classical planning before and allow for a more natural representation compared to an encoding in propositional logic (Nareyek et al. 2005). Using such SAT encodings has recently led to very successful results in HTN planning. Especially the constraints imposed by the task hierarchy present in HTN planning problems motivate the use of more sophisticated constraint satisfaction techniques. To our surprise, only little work has been conducted in this direction. In fact, Stock

et al. (2015) provide the only approach we are aware of, using CSPs in the context of HTN planning. Their approach is based on modeling the planning problem in a more complex reasoning framework, called meta-CSP (Mansouri and Pecora 2016), which allows them to represent different forms of information in separate CSPs independently.

We avoid the overhead of combining multiple CSPs by aiming to encode the HTN planning problem directly into one single qualitative constraint network architecture. Our encoding draws inspiration from recent SAT encodings for expressing the task hierarchy in an incremental fashion and restricting the depth of the decomposition tree instead of the length of the plan (Schreiber 2021). Action encodings follow the structure proposed previously for classical planning (Barták and Toropila 2008), using an implication constraint. We additionally introduce a novel binding mechanism, based on preferring equality-relations over all others. State-of-the-art solvers that we have today for solving qualitative constraint networks can be extended to handle those implication constraints and follow the required preference when solving the constraint problems.

Future work will be further refining our encoding, such that state-of-the-art solvers can be applied directly without the need of adaptations. We believe that the Interval Algebra (Allen 1983) already allows us to model several challenges of encoding HTN planning as CSP, as the relations defined within this qualitative constraint language, such as *during* or *meets*, intuitively describe properties present in HTN planning. Finally, we are interested in actually implementing our encoding, verifying its correctness and comparing its performance with current state-of-the-art HTN planners.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. This research is partially supported by BMBF AI lab dependable intelligent systems.

References

Allen, J. F. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM*, 26(11): 832–843.

Barták, R.; and Toropila, D. 2008. Reformulating Constraint Models for Classical Planning. In *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference*, 525–530. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT - Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing Order to Chaos – A Compact Representation of Partial Order in SAT-Based HTN Planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7520–7529. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding Optimal Solutions in HTN Planning - A SAT-based Approach. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 5500–5508.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 6267–6275.

Dylla, F.; Lee, J. H.; Mossakowski, T.; Schneider, T.; van Delden, A.; van de Ven, J.; and Wolter, D. 2017. A Survey of Qualitative Spatial and Temporal Calculi: Algebraic and Computational Properties. *ACM Comput. Surv.*, 50(1): 7:1–7:39.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning*. Elsevier Science & Technology. ISBN 1558608567.

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 *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9883–9891. AAAI Press.

Ligozat, G. 2013. *Qualitative Spatial and Temporal Reasoning*. ISTE Ltd and John Wiley & Sons, Inc. ISBN 978-1-84821-252-7.

Mali, A. D.; and Kambhampati, S. 1998. Encoding HTN Planning in Propositional Logic. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS)*, 190–198. AAAI.

Mansouri, M.; and Pecora, F. 2016. A robot sets a table: a case for hybrid reasoning with different types of knowledge. *J. Exp. Theor. Artif. Intell.*, 28(5): 801–821.

Nareyek, A.; Freuder, E. C.; Fourer, R.; Giunchiglia, E.; Goldman, R. P.; Kautz, H. A.; Rintanen, J.; and Tate, A. 2005. Constraints and AI Planning. *IEEE Intell. Syst.*, 20(2): 62–72.

Russell, S. J.; and Norvig, P. 2020. *Artificial Intelligence - A Modern Approach, Fourth Edition*. Pearson Education. ISBN 78-0-13-461099-3.

Schreiber, D. 2021. Lilotane: A Lifted SAT-based Approach to Hierarchical Planning. *Journal of Artificial Intelligence Research*, 70: 1117–1181.

Schreiber, D.; Pellier, D.; Fiorino, H.; and Balyo, T. 2019. Tree-REX: SAT-Based Tree Exploration for Efficient and High-Quality HTN Planning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*, 382–390. AAAI Press.

Stock, S.; Mansouri, M.; Pecora, F.; and Hertzberg, J. 2015. Online task merging with a hierarchical hybrid task planner for mobile service robots. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 6459–6464. IEEE.

Surynek, P.; and Barták, R. 2005. Encoding HTN Planning as a Dynamic CSP. In *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *LNCS*, 868. Springer.