Apprentissage de domaines HDDL à partir d'observations partielles et bruitées

Maxence Grand Damien Pellier Humbert Fiorino

Univ. Grenoble Alpes, LIG, 38000 Grenoble, France
maxence.grand@univ-grenoble-alpes.fr
damien.pellier@univ-grenoble-alpes.fr
humbert.fiorino@univ-grenoble-alpes.fr

Résumé

La planification HTN (Hierarchical Task Network) est largement utilisée pour décrire des domaines de planification. Contrairement à la planification STRIPS classique, qui ne requiert que la déclaration des actions, la planification HTN nécessite la déclaration des tâches et leur décomposition en sous-tâches, appelées méthodes. Par conséquent, l'encodage manuel des domaines HTN est considéré comme plus difficile, en particulier pour les experts applicatifs qui ne sont pas familiarisés avec les langages de planification. Pour relever ce défi, nous présentons une nouvelle approche appelée HierAMLSI, qui s'appuie sur l'induction grammaticale pour apprendre des domaines de planification HTN. Contrairement à d'autres méthodes d'apprentissage, HierAMLSI apprend, avec un haut degré de précision, les actions et les méthodes à partir de traces bruitées et partiellement observées.

Abstract

The Hierarchical Task Network (HTN) formalism is widely used for expressing planning domains. Unlike the classical STRIPS formalism, which only requires specifying actions, HTN demands the specification of tasks and their decompositions into subtasks, known as methods. Consequently, manually encoding HTN domains is considered more challenging and error-prone, particularly for application experts unfamiliar with planning languages. To address this challenge, we introduce a novel approach called HierAMLSI, which relies on grammar induction to learn HTN planning domain knowledge. Unlike other learning methods, HierAMLSI can learn with a high degree of accuracy both actions and methods from noisy and partially observed input data.

1 Introduction

La planification automatique repose sur des "modèles d'actions", c'est-à-dire des domaines de planification exprimés dans un langage déclaratif tel que PDDL (Planning Domain Description Language) [28] ou son équivalent hiérarchique HDDL (Hierarchical Domain Description Language) [18]. Contrairement à la planification classique, les modèles d'actions hiérarchiques intègrent des éléments supplémentaires pour représenter des "recettes" permettant de résoudre des "tâches". Par exemple, pour accomplir la tâche "préparer un repas", il peut être nécessaire de réaliser des sous-tâches telles que "préparer les entrées", "préparer le plat principal" et "préparer les desserts". En décomposant davantage la tâche "préparer les entrées", on peut avoir des sous-tâches telles que "préparer des cornichons", etc. Diverses "méthodes" peuvent être employées pour accomplir une tâche donnée, comme "préparer un repas végétalien" ou "utiliser des ingrédients sans gluten".

Cependant, cette expressivité s'accompagne de quelques inconvénients. L'encodage manuel de modèles d'actions hiérarchiques est souvent considéré comme une tâche difficile, fastidieuse et sujette aux erreurs, en particulier pour les experts applicatifs qui ne sont pas familiarisés avec les langages de planification. Pour relever ce défi, diverses techniques d'apprentissage automatique ont été proposées pour apprendre des modèles d'actions, qu'ils soient hiérarchiques ou non, à partir d'observations, c'est-à-dire des traces, des séquences de changements d'état du monde. Par exemple, certaines approches comme ARMS [38], LSONIO [29] et LOCM [3] se concentrent uniquement sur l'apprentissage de modèles d'actions non hiérarchiques. En revanche, d'autres, comme CAMEL [23], HTN-Maker [14, 12] et LHTNDT [30], fournissent des actions simples à l'algorithme d'apprentissage dans le but d'apprendre des

hiérarchies de tâches représentées sous la forme de méthodes. D'autres, comme HTN-Learner [39, 40], sont capables d'apprendre à la fois des modèles d'actions simples et des méthodes.

Malgré les progrès récents, l'apprentissage de modèles d'actions hiérarchiques présente encore plusieurs problèmes difficiles à résoudre. Tout d'abord, il nécessite une grande quantité de données d'apprentissage dont l'obtention peut s'avérer difficile et coûteuse dans le cadre d'applications réelles. De plus, les modèles appris ne sont souvent pas suffisamment précis pour être directement utilisés par des planificateurs. Une relecture par un expert humain est souvent nécessaire pour corriger les erreurs syntaxiques et garantir que le modèle appris fonctionne correctement. Enfin, peu d'algorithmes sont capables d'apprendre des modèles d'actions à partir d'observations partielles et bruitées.

Dans cet article, nous présentons HierAMLSI, un algorithme d'apprentissage spécialement conçu pour acquérir des domaines HDDL. HierAMLSI fait partie des rares algorithmes capables d'apprendre à la fois les actions et les méthodes, y compris leurs préconditions. Aussi, HierAMLSI apprend les domaines HDDL à partir de traces d'exécutions bruitées et partiellement observées.

Le reste de l'article est organisé comme suit. Dans la section 2, nous présentons le cadre formel. Dans la section 3, nous donnons une description de l'algorithme AMLSI [7, 8] sur lequel HierAMLSI est basé. La section 4 détaille l'algorithme HierAMLSI. Ensuite, la section 5 évalue les performances de HierAMLSI sur des benchmarks IPC [1]. Grâce à l'évaluation expérimentale, nous montrerons que HierAMLSI relève efficacement trois grands défis mis en évidence dans la littérature existante : (1) traiter des ensembles de données partielles et bruitées, (2) minimiser le volume de données requis pour l'apprentissage, et (3) obtenir des domaines très précis, réduisant ainsi la nécessité d'une relecture et d'une correction approfondies par des experts humains. Enfin, la section 6 présente l'état de l'art.

2 Cadre Formel

2.1 Planification STRIPS

Dans cette section, nous utilisons les définitions et les notations proposées par [17] et les adaptons au problème de l'apprentissage.

Un problème de planification STRIPS est un n-uplet $P = (L, A, S, s_0, g, \delta, \tau, \lambda)$, où L est un ensemble de propositions logiques décrivant les états du monde, S est un ensemble d'états, $s_0 \in S$ est l'état initial, et g est le but. La fonction $\lambda : S \to 2^L$ est une fonction d'observation qui attribue à chaque état l'ensemble des propositions logiques vraies dans cet état. A est un ensemble d'actions. Les préconditions, les effets positifs et négatifs des actions sont donnés par les fonctions $prec$, add et del incluses dans $\delta = (prec, add, del)$. La fonction $prec$ est définie comme $prec : A \to 2^L$. Les fonctions add et del sont définies de la même manière.

La fonction booléenne $\tau : A \times S \to \{\text{true}, \text{false}\}$ retourne vrai si une action est applicable dans un état, c'est-à-dire $\tau(a, s) \Leftrightarrow \text{prec}(a) \subseteq \lambda(s)$. Chaque fois que l'action a est applicable dans l'état s_i, la fonction de transition d'état $\gamma : S \times A \to S$ retourne l'état résultant $s_{i+1} = \gamma(s_i, a)$ tel que $\lambda(s_{i+1}) = [\lambda(s_i) \setminus \text{del}(a)] \cup \text{add}(a)$.

Une séquence d'actions $(a_0 a_1 \dots a_n)$ est applicable dans un état s_0 lorsque chaque action a_i, avec $0 \leq i \leq n$, est applicable dans l'état s_i. Étant donné une séquence applicable $(a_0 a_1 \dots a_n)$ dans l'état s_0, $\gamma(s_0, (a_0 a_1 \dots a_n)) = \gamma(\gamma(s_0, a_0), (a_1 \dots a_n)) = s_{n+1}$. Il est important de noter que cette définition récursive de γ entraîne la génération d'une séquence d'états $(s_0 s_1 \dots s_{n+1})$. Un état s est un état but si $\lambda(g) \subseteq \lambda(s)$. L'état s satisfait g, c'est-à-dire $s \models g$, si et seulement si s est un état but. Une séquence d'actions est un plan solution pour un problème de planification P si et seulement si elle est applicable dans s_0 et entraîne un état but.

En langage formel, un ensemble de règles décrit la structure des mots valides, et le langage est l'ensemble de ces mots. Pour un problème de planification $P = (L, A, S, s_0, g, \delta, \tau, \lambda)$, ce langage est défini comme suit :

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

Nous savons que les problèmes de planification STRIPS génèrent un ensemble de langages réguliers [17]. En d'autres termes, un problème de planification STRIPS P engendre un langage $\mathcal{L}(P)$ équivalent à un automate fini déterministe (AFD) $\Sigma = (S, A, \gamma)$. S et A représentent respectivement les nœuds et les arcs de l'AFD, et γ est la fonction de transition.

Par conséquent, si un AFD Σ peut être appris à partir d'un ensemble d'observations $\Omega \subseteq \mathcal{L}(P)$ (voir la figure 1 pour un exemple d'AFD appris), alors il est possible d'apprendre le problème de planification STRIPS $P = (L, A, S, s_0, g, \tau, \delta, \lambda)$ à partir de ces observations. En particulier, il est possible d'apprendre la fonction δ définissant les actions de P et de la généraliser afin de l'exprimer dans un domaine PDDL.

2.2 Planification HTN

Un problème de planification HTN, selon les notations étendues de [15], est un n-uplet $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$. Les éléments L, S, $s_0 \in S$, g, λ, et δ sont similaires à ceux du problème de planification STRIPS.

A est l'ensemble des actions (ou tâches primitives) et C est l'ensemble des tâches composées (ou non primitives),

1. **I**nternatinnal **P**lanning **C**ompetition

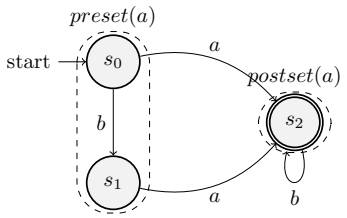


FIGURE 1 – Exemple d'AFD appris depuis un ensemble d'observations : $\Omega = \{\lambda(s_0) \xrightarrow{a} \lambda(s_2), \lambda(s_0) \xrightarrow{a} \lambda(s_2) \xrightarrow{b} \lambda(s_2), \lambda(s_0) \xrightarrow{b} \lambda(s_1) \xrightarrow{a} \lambda(s_2), \lambda(s_0) \xrightarrow{b} \lambda(s_1) \xrightarrow{a} \lambda(s_2) \xrightarrow{b} \lambda(s_2), \lambda(s_0) \xrightarrow{a} \lambda(s_2) \xrightarrow{b} \lambda(s_2) \xrightarrow{b} \lambda(s_2), \ldots$.
$preset(a)$ (resp. $postset(a)$) représente les *prédécesseur* (resp. *successeur*) de a.

avec $C \cap A = \emptyset$. Un réseau de tâches, noté ω, est une séquence de tâches [2]. Soit $T = C \cup A$, l'ensemble des tâches primitives et composées, un réseau de tâches est un élément de T^*, c'est-à-dire l'ensemble des tâches construit à partir de T. Les tâches composées sont décomposées en utilisant des méthodes. L'ensemble M contient toutes les méthodes. Les méthodes sont définies par la fonction $\sigma : M \rightarrow C \times T^*$. Ensuite, une tâche composée c est décomposable dans un état s en un réseau de tâches ω, si et seulement si il existe une méthode $m \in M$ telle que : $\sigma(m) = (c, \omega)$ et $prec(m) \subseteq s$. La fonction $\zeta : T^* \times S \rightarrow T^*$ est la fonction de décomposition. Cette fonction permet de décomposer la première tâche d'un réseau de tâches. Utilisée récursivement, elle décompose toutes les tâches d'un réseaux de tâches. Étant donné un réseau de tâches totalement ordonné $t\omega$ avec t comme première tâche, ζ est définie comme suit :

$$\zeta(t\omega, s) = \begin{cases} t\omega & \text{si } t \text{ est une tâche primitive.} \\ \omega^{'}\omega & \text{si } t \text{ est une tâche composée} \\ & \text{et est décomposable dans } s. \\ \emptyset & \text{sinon.} \end{cases}$$

Nous notons $\omega \rightarrow^* \pi$, avec $\pi \in A^*$, si ω peut être décomposé en π en utilisant récursivement ζ sur l'ensemble des tâches du réseaux ω. Enfin, ω_I est le réseau de tâches initial.

Une solution à un problème de planification HTN est un réseau de tâches π tel que :

1. $\omega_I \rightarrow^* \pi$, où π est obtenu en décomposant ω_I.
2. $\gamma(s_0, \pi) \models g$, où π est applicable dans s_0 et atteint un état but.

Il est maintenant possible de définir un problème de planification HTN $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$ en tant que langage formel comme suit :

2. Nous considérons uniquement les domaines totalement ordonnés

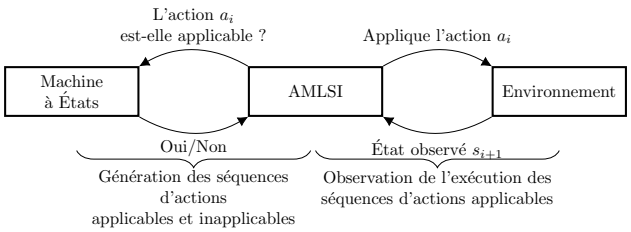


FIGURE 2 – AMLSI : Génération des observations.

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

Ainsi, apprendre les actions et les méthodes d'un problème HTN $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$ consiste à apprendre, à partir d'un ensemble donné d'observations $\Omega \subseteq \mathcal{L}(P)$, deux fonctions : δ, qui, comme pour la planification STRIPS, définit les actions, mais aussi σ, qui définit les méthodes.

Cependant, contrairement aux problèmes STRIPS, le langage $\mathcal{L}(P)$, produit par un problème HTN, n'est pas nécessairement régulier [16] et ne peut pas être représenté par un AFD. Comme mentionné par [16, 15], $\mathcal{L}(P)$ est l'intersection de deux langages :

1. $\mathcal{L}_{\mathcal{A}}(P) = \{\pi \in A^* \mid \gamma(s_0, \pi) \models g\}$, qui est défini par le système de transition d'états défini par les préconditions et les effets des tâches primitives. $\mathcal{L}_{\mathcal{A}}(P)$ est régulier.
2. $\mathcal{L}_{\mathcal{T}}(P) = \{\pi \in A^* \mid w_I \rightarrow^* \pi\}$, qui est défini par la hiérarchie de décomposition, c'est-à-dire par les tâches composées et les méthodes. $\mathcal{L}_{\mathcal{T}}(P)$ n'est pas nécessairement régulier.

Le principe de notre approche est : (i) apprendre l'AFD correspondant au langage régulier $\mathcal{L}_{\mathcal{A}}(P)$ et apprendre la fonction δ, (ii) modifier l'AFD appris en ajoutant des transitions de tâches composées afin d'encoder $\mathcal{L}_{\mathcal{T}}(P)$ et d'approximer le langage $\mathcal{L}(P)$; enfin, (iii) apprendre σ. Une fois que les fonctions δ et σ ont été apprises, il est possible de les généraliser dans un domaine HDDL.

3 Aperçu de l'algorithme AMLSI

Dans cette section, nous donnons un aperçu de l'algorithme AMLSI sur lequel HierAMLSI est basé (pour plus de détails, voir [8, 7]).

L'idée principale d'AMLSI est qu'il est possible d'apprendre une machine à états modélisant un problème de planification $P = (L, A, S, s_0, g, \tau, \delta, \lambda)$ en testant ses transitions et en observant les états résultants. Cette machine à état est une boite noire complexe que AMLSI apprend et représente sous la forme d'un domaine PDDL. AMLSI suppose que L, l'ensemble de propositions décrivant le monde,

A, l'ensemble des actions (les étiquettes des transitions de la machine à états), et s_0, l'état initial de P, sont connus. La fonction d'observation λ est considérée comme partielle et bruitée. Aucune hypothèse n'est faite sur le but g de P et sur la fonction τ qui retourne vrai lorsque une action est applicable dans un état. L'objectif d'AMLSI est d'apprendre δ qui définit les actions de P, et de généraliser δ afin d'obtenir un domaine PDDL.

L'algorithme AMLSI se compose de 5 étapes : (1) génération des observations, (2) apprentissage de l'AFD correspondant aux observations, (3) inférence des actions à partir de l'AFD appris, (4) généralisation de δ pour l'exprimer comme un ensemble d'actions PDDL, et (5) enfin, raffinement des actions PDDL pour traiter les observations partielles et bruitées des états.

Étape 1 (Génération des observations). La figure 2 donne un aperçu de cette étape. AMLSI génère l'ensemble des observations Ω avec une marche aléatoire. Ω est composé de deux sous-ensembles tels que $\Omega = I_+ \cup I_-$ et $I_+ \cap I_- = \emptyset$ construits comme suit. Si une action est applicable dans l'état courant, la séquence d'actions est ajoutée à I_+, l'ensemble des exemples positifs. Sinon, la séquence est ajoutée à I_-, l'ensemble des exemples négatifs.

Une fois les exemples positifs et négatifs générés, AMLSI exécute les exemples positifs et observe un ensemble d'états. Ce sont ces observations d'états qui sont bruitées et partielles. Plus précisément, une observation partielle, est un état contenant un sous-ensemble de propositions logiques non observées. Une observation bruitée est un état contenant un sous-ensemble de propositions logiques mal-évaluées.

Étape 2 (Apprentissage de l'AFD). Pour apprendre l'AFD $\Sigma = (S, A, \gamma)$, AMLSI utilise une variante de RPNI [32], un algorithme classique d'inférence grammaticale régulière, pour exploiter à la fois I_+ et I_-.

Étape 3 (Inférence des actions). À cette étape, AMLSI infère, pour chaque action, δ à partir de l'AFD. Pour les préconditions $prec(a)$ de l'action a, AMLSI calcule les propositions logiques qui sont dans tous les états précédant a dans Σ.

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

Pour les effets positifs $add(a)$ de l'action a, AMLSI calcule les propositions logiques qui ne sont jamais présentes dans les états avant l'exécution de a, et présentes après l'exécution de a.

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

Et inversement, pour les effets négatifs $del(a)$ de l'action a, AMLSI calcule les propositions logiques qui sont toujours

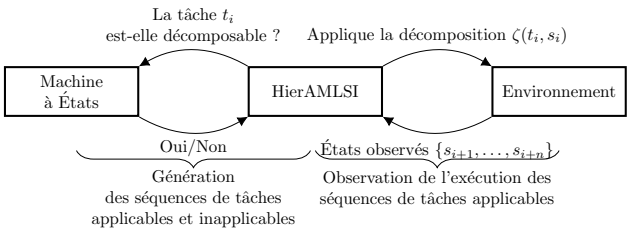


FIGURE 3 – HierAMLSI : Génération des observations.

présentes dans les états avant l'exécution de a, et absentes après l'exécution de a.

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

Étape 4 (Généralisation des actions). Une fois les préconditions et les effets inférés, les actions sont généralisées en actions PDDL. Les constantes dans les préconditions et les effets sont substitués par des variables. Ensuite, les préconditions et les effets des actions PDDL sont calculés comme l'intersection des préconditions et des effets de toutes les actions généralisées.

Étape 5 (Raffinement des actions). Pour traîter les observations partielles, AMLSI commence par raffiner les effets et les préconditions afin de garantir que chaque transition de l'AFD appris soit faisable. Pour ce faire, AMLSI ajoute tous les effets en veillant à ce que chaque transition dans le DFA soit faisable. Ensuite, AMLSI raffine les préconditions des actions. Comme dans [38], nous supposons que les effets négatifs d'une action doivent être présentes dans ses préconditions. Ainsi, pour chaque effet négatif, AMLSI ajoute les propositions correspondantes. Étant donné que le raffinement des effets dépend des préconditions, et que le raffinement des préconditions dépend des effets, AMLSI répète ces deux étapes jusqu'à convergence, c'est-à-dire jusqu'à ce que plus aucune précondition et plus aucun effet ne soit ajouté. Enfin, pour traîter les observations bruitées, AMLSI effectue une recherche Tabu [6]. Une fois que la recherche Tabu a atteint un optimum local, AMLSI répète les différentes étapes de raffinement jusqu'à convergence.

4 L'approche HierAMLSI

HierAMLSI est basé sur AMLSI. Il partage les mêmes hypothèses qu'AMLSI, c'est-à-dire qu'il est possible d'apprendre une machine à états modélisant un problème de planification HTN $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$ en testant ses transitions et en observant les états résultants. Comme pour AMLSI, cette machine à état est une boite noire complexe que HierAMLSI apprend et représente sous la forme d'un domaine HDDL. De plus, L, A et s_0 sont

$I_+ = \{ \quad \langle$*(pick-up a), (stack a b), (unstack a b), (put-down a)*$\rangle, \langle$*(pick-up b), (stack b a)*$\rangle$,
$\langle$*(pick-up b), (stack b a), (unstack b a), (put-down a), (pick-up b), (put-down b)*$\rangle, \ldots\}$
$I_- = \{ \quad \langle$*(pick-up a), (stack a b), (unstack a b), (put-down a), (put-down a)*$\rangle$,
$\langle$*(pick-up b), (stack b a), (stack b a)*$\rangle, \langle$*(stack b a)*$\rangle$,
$\langle$*(pick-up b), (stack b a), (unstack b a), (put-down a), (stack b a)*$\rangle, \ldots\}$

FIGURE 4 – Un exemple d'observations Ω contenant l'ensemble des séquences positives I_+ et l'ensemble des séquences négatives I_-.

connus, la fonction λ est partielle et bruitée, aucune hypothèse n'est faite sur le but de P. Dans HierAMLSI, la fonction de décomposition ζ est partiellement annotée. En d'autres termes, en ce qui concerne la décomposition, HierAMLSI observe seulement la tâche composée racine et n'a aucune connaissance sur les tâches composées intermédiaires. L'objectif de HierAMLSI est d'apprendre δ (les actions de P) et σ (les méthodes de P), et de généraliser ces fonctions dans leur représentation HDDL. L'approche HierAMLSI se compose de 7 étapes.

Étape 1 (Génération des observations). HierAMLSI produit un ensemble d'observations Ω avec une marche aléatoire en appliquant non seulement des tâches primitives comme dans AMLSI, mais aussi des tâches composées à partir de l'état initial de P. Cette différence est présentée dans §4.1.

Étape 2 (Apprentissage de l'AFD). HierAMLSI apprend l'AFD qui définit le système de transition du problème de planification P. Cette étape est similaire à l'étape 2 d'AMLSI.

Étape 3 (Inférence des actions). HierAMLSI infère le modèle d'actions δ à partir de l'AFD. Cette étape est également identique à l'étape 3 d'AMLSI.

Étape 4 (Annotation de l'AFD). HierAMLSI annote l'AFD appris en ajoutant des transitions pour les tâches composées afin d'approximer le langage $\mathcal{L}(P)$. Cette étape est détaillée dans §4.2.

Étape 5 (Inférence de méthodes). HierAMLSI infère σ qui définit les préconditions et les réseaux de tâches des méthodes observées à partir d'AFD annoté. L'inférence des préconditions des méthodes est effectuée en utilisant les techniques décrites à l'étape 3 d'AMLSI. L'inférence des réseaux de tâches des méthodes est spécifique à HierAMLSI. Nous détaillons ce point dans §4.3.

Étape 6 (Généralisation des méthodes et des actions). HierAMLSI généralise δ et σ en actions et méthodes HDDL. La méthode utilisée est celle décrite à l'étape 4 d'AMLSI.

Étape 7 (Raffinement des méthodes et actions HDDL). HierAMLSI utilise les techniques de raffinement décrites à l'étape 5 d'AMLSI pour affiner les actions et les méthodes HDDL afin de traiter les observations partielles et bruitées.

4.1 Génération des observations

La génération des observations (voir la figure 3) est similaire à celle utilisée dans AMLSI. Pour générer les observations Ω, HierAMLSI utilise des marches aléatoires en appliquant une tâche à partir de l'état initial du problème. La principale différence est que HierAMLSI peut choisir de manière aléatoire non seulement des tâches primitives (actions) mais aussi des tâches composées.

Si la tâche choisie est primitive et est applicable dans l'état courant, comme pour AMLSI, la séquence de tâches primitives de l'état initial à l'état courant est valide, et est ajoutée à I_+. Sinon, la séquence de tâches primitives est ajoutée à I_-. Si la tâche choisie est composée et est applicable dans l'état courant, la tâche composée est décomposée en une séquence de tâches primitives. La séquence de tâches primitives est alors ajouté à I_+. Sinon, la séquence dont la dernière tâche n'est pas applicable, est ajoutée à I_-. Les marches aléatoires sont répétées jusqu'à ce que I_+ et I_- atteignent une taille arbitraire.

Par exemple, supposons que nous ayons la séquence de tâches suivante générée à partir d'un problème simple de Blocksworld avec deux blocs a et b : $\langle$ *(do-put-on a b), (do-clear a)*$\rangle$, la séquence de tâches primitives générée et ajoutée à I_+ est : $\langle$ *(pick-up a), (stack a b), (unstack a b), (put-down a)*$\rangle$. Maintenant, supposons que nous ayons généré la séquence de tâches inapplicables suivante : $\langle$ *(do-put-on a b), (do-clear a), (put-down a)*$\rangle$, la séquence de tâches primitives ajoutée à I_- est : $\langle$ *(pick-up a), (stack a b), (unstack a b), (put-down a), (put-down a)*$\rangle$. La figure 4 donne un exemple d'un ensemble d'observations Ω.

Enfin, comme pour AMLSI, une fois les exemples positifs et négatifs générés, HierAMLSI exécute les exemples positifs et observe un ensemble d'états. Ce sont ces observations d'états qui sont bruitées et partielles.

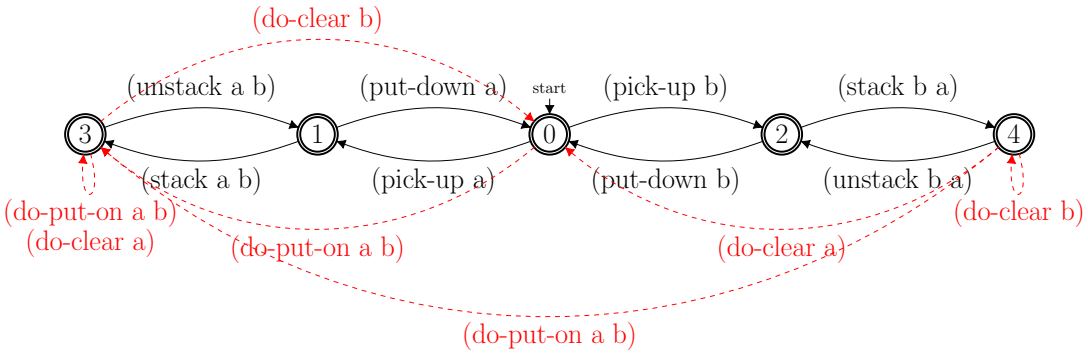


FIGURE 5 – Étapes d'apprentissage et d'annotation de l'AFD : l'AFD contenant uniquement les tâches primitives est représenté à l'aide des transitions noirs, et l'AFD comprenant les tâches composées est représenté avec les transitions rouges.

4.2 Annotation de l'AFD

Une fois que l'AFD des tâches primitives a été appris à partir des observations des tâches primitives, HierAMLSI annote l'AFD en ajoutant des transitions pour les tâches composées en insérant des transitions dont les étiquettes sont des tâches composées. Par exemple, supposons que nous ayons observé dans Ω que la tâche composée *(do-put-on a b)* a été décomposée par les tâches primitives $\langle$*(pick-up a), (stack a b)*$\rangle$ depuis le nœud 0 et a atteint le nœud 3. Alors, nous ajoutons la transition suivante à l'AFD : $\gamma(0, \textit{(do-put-on a b)}) \rightarrow 3$. La figure 5 donne un exemple d'AFD contenant à la fois les tâches primitives et composées.

4.3 Inférence des méthodes

Après avoir appris et annoté l'AFD, HierAMLSI infère la fonction σ qui définit les méthodes du problème de planification HTN. L'inférence des préconditions des méthodes est effectuée à l'aide de la méthode d'inférence utilisée pour inférer les préconditions des actions (voir l'étape 3 d'AMLSI pour plus de détails). Nous nous concentrons dans cette section sur l'inférence des méthodes.

Dans le pire cas, $|T|.|I_+|^2$ méthodes peuvent être inférées, où $|T|$ représente le nombre de tâches du problème et $|I_+|$ représente le nombre de tâches primitives dans l'ensemble I_+. Pour réduire le nombre de méthodes et avoir une représentation compacte des méthodes, il est nécessaire de minimiser l'ensemble de méthodes tout en garantissant qu'elles décomposent toutes les tâches composées observées. Trouver cet ensemble minimal peut être réduit à une variante du problème de couverture par ensembles (Set Cover Problem) [24], qui est NP-Complet. Pour faire face à cette complexité et approximer l'ensemble minimal de méthodes, nous proposons une procédure itérative.

La première étape consiste à initialiser σ en ajoutant une méthode pour chaque transition de tâche composée dans l'AFD et pour chaque décomposition possible de la tâche composée observée. Par exemple, trois méthodes sont créées pour la tâche composée *(do-put-on a b)*, une pour chacune de ses décompositions possibles de tâches primitives (voir figure 6a). Initialement, σ est composé de 7 méthodes. Ensuite, à chaque itération et pour chaque tâche composée de σ qui n'a pas déjà été choisie lors d'une itération précédente, nous calculons l'ensemble minimal de méthodes qui permet de décomposer toutes les tâches. Ce calcul est effectué avec une approximation gloutonne (GA) [2]. GA est un processus itératif polynomial qui, à chaque étape, ajoute le réseau de tâches de la méthode couvrant le plus grand nombre de décompositions. GA s'arrête une fois que toutes les décompositions sont couvertes par l'ensemble de méthodes.

Supposons que nous voulions calculer l'ensemble minimal des méthodes de la tâche composée *(do-put-on a b)* en considérant la tâche composée *(do-clear a)*. À l'itération 1, nous pouvons réduire la tâche composée *(do-put-on a b)*, initialement avec 3 méthodes $\omega_1 = \emptyset$, $\omega_2 = \langle$*(pick-up a), (stack a b)*$\rangle$, $\omega_3 = \langle$*(unstack b a), (put-down b), (pick-up a), (stack a b)*$\rangle$ à une représentation plus compacte avec seulement 2 méthodes : $\omega_1 = \emptyset, \omega_2 = \langle$*(do-clear a), (pick-up a), (stack a b)*$\rangle$ (voir figure 6b). À la fin de chaque itération, σ est défini sur l'ensemble minimal calculé. La tâche composée utilisée pour obtenir le nouvel ensemble de méthodes σ ne peut pas être réutilisée pour l'itération suivante. Les itérations s'arrêtent lorsqu'il n'y a plus de tâche composée pouvant être utilisée pour calculer un ensemble plus petit de méthodes. Par conséquent, la procédure proposée effectue au plus k itérations où $k \leq |C|$ est le nombre de tâches composées dans σ lors de la première itération. L'ensemble des méthodes inférées est présenté dans la figure 6c.

(do-clear a) : $\omega_1 = \emptyset, \omega_2 = \langle$*(unstack b a), (put-down b)*$\rangle$
(do-clear b) : $\omega_1 = \emptyset, \omega_2 = \langle$*(unstack a b), (put-down a)*$\rangle$
(do-put-on a b) : $\omega_1 = \emptyset, \omega_2 = \langle$*(pick-up a), (stack a b)*$\rangle, \omega_3 = \langle$*(unstack b a), (put-down b), (pick-up a), (stack a b)*$\rangle$

(a) **Itération 0 :** Ensemble initial des méthodes inférées

(do-clear a) : $\omega_1 = \emptyset, \omega_2 = \langle$*(unstack b a), (put-down b)*$\rangle$
(do-clear b) : $\omega_1 = \emptyset, \omega_2 = \langle$*(unstack a b), (put-down a)*$\rangle$
(do-put-on a b) : $\omega_1 = \emptyset, \omega_2 = \langle$*(do-clear a), (pick-up a), (stack a b)*$\rangle$

(b) **Itération 1 :** Ensemble des méthodes inférées en considérant la tâche composée *do-clear a*.

(do-clear a) : $\omega_1 = \emptyset, \omega_2 = \langle$*(unstack b a), (put-down b)*$\rangle$
(do-clear b) : $\omega_1 = \emptyset, \omega_2 = \langle$*(unstack a b), (put-down a)*$\rangle$
(do-put-on a b) : $\omega_1 = \emptyset, \omega_2 = \langle$*(do-clear a), (do-clear b), (pick-up a), (stack a b)*$\rangle$

(c) **Itération k :** Ensemble des méthodes inférées à la dernière itération.

FIGURE 6 – Exemple d'itérations de la procédure d'approximation de la décomposition minimale des tâches utilisée pour inférer les méthodes.

5 Évaluation expérimentale

En pratique, l'évaluation des algorithmes d'apprentissage, en particulier ceux liés à l'apprentissage de domaines hiérarchiques, peut être complexe. La récente normalisation du langage HDDL en 2020 [18] complique davantage la comparaison avec les approches plus anciennes. De plus, les entrées fournies à ces algorithmes peuvent varier considérablement, et l'expressivité des domaines appris peut également différer (voir la section 6). Dans la littérature seules quelques approches [39, 40, 36] partagent la capacité de HierAMLSI a apprendre à la fois les actions et les méthodes. Cependant une trop grande disparité dans les entrées empêche toute comparaison entre ces méthodes et la nôtre. Par conséquent, il n'est pas possible, à notre connaissance, de sélectionner une approche comme référence dans la littérature pour mener une évaluation équitable avec HierAMLSI.

Notre évaluation consiste à évaluer la capacité de HierAMLSI à : (1) apprendre uniquement des actions [3] ; (2) apprendre uniquement des méthodes et (3) apprendre à la fois des actions et des méthodes. Pour chacun de ces cas, nous considérons quatre scénarios expérimentaux distincts : (1) observations complètes (100%) et non bruitées (0%) ; (2) observations complètes (100%) et bruitées (20%) ; (3) observations partielles (20%) et non bruitées (0%) ; et (4) observations partielles (20%) et bruitées (20%).

Il est important de noter que les observations bruitées et partielles sont introduites exclusivement dans les séquences d'états. Ces perturbations sont introduites en supprimant ou en modifiant une partie des propositions, les propositions spécifiques étant sélectionnées de manière aléatoire.

Chaque scénario est testé sur un benchmark composé de 8 domaines HDDL [18, 19] (voir le tableau 1 pour les caractéristiques des domaines) de la compétition IPC 2020 : Blocksworld, Childsnack, Transport, Zenotravel, Gripper, Spanner, Elevator et Robot.

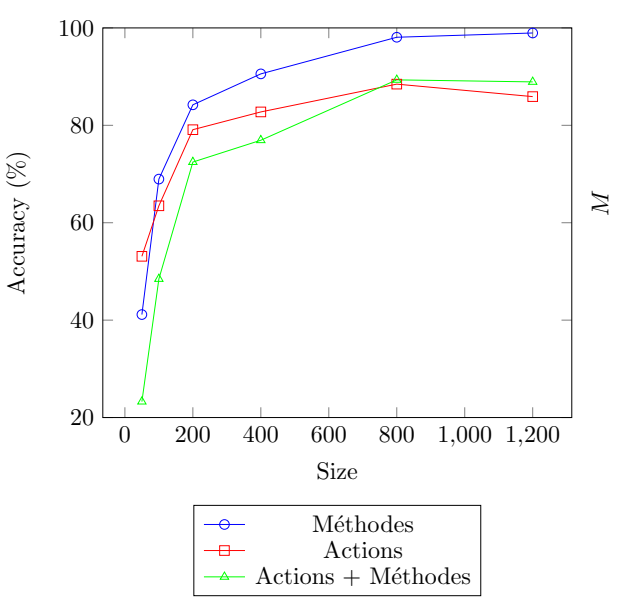
3. Ce scénario est équivalent à utiliser AMLSI seul

Domain	A	C	M	L
Blocksworld	4	4	8	5
Gripper	3	3	4	4
Zenotravel	4	2	5	7
Transport	3	4	5	5
Childsnack	6	1	2	12
Spanner	3	3	7	6
Elevator	2	4	5	7
Robot	4	4	7	8

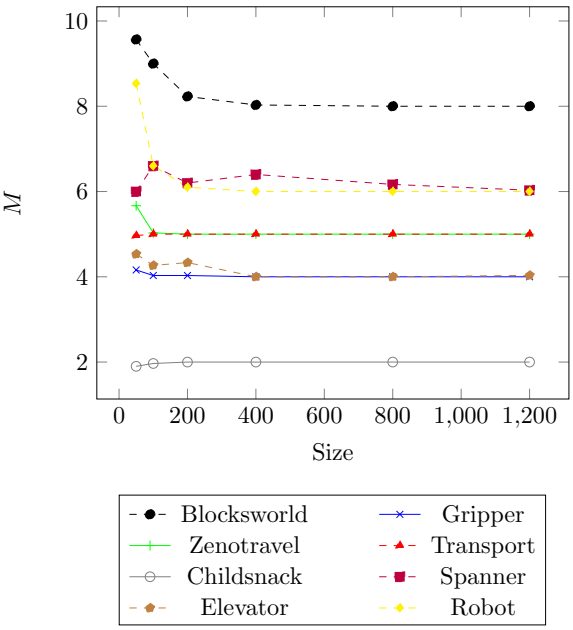
TABLE 1 – Charactéristiques du benchmark, de gauche à droite, le nombre de tâche primitives A, le nombre de tâches composées C, le nombre de méthode M et le nombre de prédicat L pour chaque domaine IPC.

De plus, HierAMLSI apprend un domaine HDDL à partir d'un seul problème. Cependant, pour éviter tout biais potentiel découlant du choix d'un problème spécifique, HierAMLSI est évalué avec trois problèmes distincts. Pour chaque problème, l'apprentissage est répété et évalué dix fois. Tous les tests ont été effectués sur un serveur Ubuntu 14.04 équipé d'un processeur Intel Xeon CPU E5-2630 multi-cœur fonctionnant à 2,30 GHz et de 16 Go de mémoire. Les données d'apprentissage ont été générés à l'aide de la bibliothèque PDDL4J [33].

Pour conclure l'évaluation de notre approche, nous démontrons comment HierAMLSI se comporte dans le scénario le plus difficile (scénario 4) en faisant varier le nombre de traces d'observations en entrée. Ce scénario vise à montrer que HierAMLSI peut apprendre efficacement des domaines HDDL même avec des données d'entrée limitées.



(a) Précision moyenne de HierAMLSI obtenu sur les 8 domaines de notre benchmark.



(b) Nombre moyen de méthodes inférées par HierAMLSI.

FIGURE 7 – Performances moyenne de HierAMLSI sur les 8 domaines de notre benchmark lorsque la taille des données d'apprentissage varie.

5.1 Critères d'évaluation

Traditionnellement, deux critères sont couramment utilisés pour évaluer les techniques d'apprentissage de domaines de planification : l'*erreur syntaxique* [42], qui quantifie la dissimilarité entre le domaine original et le domaine appris en termes de syntaxe, et la *précision* [41], qui évalue la performance du domaine appris lorsqu'il est utilisé pour résoudre de nouveaux problèmes.

Bien que l'erreur syntaxique soit fréquemment utilisée dans la littérature, il est important de noter que HierAMLSI est uniquement évalué en termes de précision. Il existe plusieurs raisons à ce choix. Tout d'abord, la précision est probablement la mesure la plus significative pour la planification, car elle mesure la capacité d'un domaine appris à résoudre de nouveaux problèmes. En pratique, même une légère divergence entre le domaine original et le domaine appris telle qu'une précondition ou un effet manquant, qui entraînerait une petite erreur syntaxique, peut rendre un domaine incapable de résoudre de nouveaux problèmes de planification. Deuxièmement, HierAMLSI ne possède pas de connaissances sur les noms des méthodes dans les domaines IPC nécessaires pour la comparaison avec les domaines appris. Il ne dispose que d'informations sur les noms des tâches. Par conséquent, le calcul de l'erreur syntaxique, qui repose sur la comparaison de la syntaxe des méthodes, n'est pas réalisable dans ce contexte.

Formellement, la précision $Acc = N/N^*$ est calculée comme le rapport entre N, qui représente le nombre de problèmes correctement résolus en utilisant le domaine appris, et N^*, qui représente le nombre total de problèmes à résoudre. Dans cette évaluation, la précision est calculée avec 20 problèmes. Ces problèmes sont résolus en utilisant le planificateur TFD, qui fait partie de la bibliothèque PDDL4J [33]. La validation des plans est effectuée avec VAL, l'outil de validation de la compétition IPC [20].

5.2 Discussion

La tableau 2 présente la précision obtenue par HierAMLSI sur les domaines de notre benchmark pour tous les scénarios expérimentaux. Indépendamment du scénario, HierAMLSI apprend systématiquement des domaines précis avec une précision variant de 57% dans le pire des cas à 100% dans le meilleur des cas. Plus précisément, lorsque les actions sont connues, HierAMLSI atteint généralement une précision élevée dans l'apprentissage des domaines ($> 95\%$) dans tous les scénarios expérimentaux. Il est à noter que les performances diminuent légèrement lorsque HierAMLSI doit apprendre à la fois les actions et les méthodes. Cependant, même pour les scénarios les plus difficiles, avec des niveaux élevés de bruits et d'observations partielles, les domaines appris maintiennent une bonne précision.

Il est important de noter que, même dans le scénario le plus difficile, seules 200 tâches sont nécessaires pour apprendre des domaines précis (comme indiqué dans la figure 7a). Cela démontre que HierAMLSI peut produire des

Observabilité		100%		20%	
Bruit		0%	20%	0%	20%
Blocksworld	A	100%	86.7%	100%	76.7%
	M	100%	100%	100%	100%
	A + M	100%	86.7%	76.7%	83.2%
Gripper	A	100%	100%	100%	100%
	M	100%	100%	100%	100%
	A+M	100%	100%	100%	100%
Zenotravel	A	100%	100%	100%	100%
	M	100%	100%	100%	100%
	A+M	100%	100%	100%	100%
Transport	A	100%	100%	100%	100%
	M	100%	100%	100%	100%
	A+M	100%	100%	100%	100%
Childsnack	A	100%	74.8%	96.7%	82.5%
	M	100%	100%	100%	100%
	A+M	100%	80.5%	96.7%	82.5%
Spanner	A	100%	94.7%	83%	54.7%
	M	96.7%	96.7%	96.7%	96.7%
	A+M	96.7%	91.3%	96.7%	82.7%
Elevator	A	100%	100%	100%	100%
	M	95%	95%	95%	95%
	A+M	95%	95%	94.8%	95%
Robot	A	83.5%	90%	86.8%	73.3%
	M	100%	100%	100%	100%
	A+M	83.3%	90%	86.7%	73.3%

TABLE 2 – Précision de HierAMLSI obtenue sur les 8 domaines de notre benchmark. Les données d'apprentissage sont composées de 30 séquences de 40 tâches. 3 variantes sont testées : (A) seules les actions sont apprises, (M) seules les méthodes sont apprises et (A+M) à la fois les actions et les méthodes sont apprises

domaines de bonne qualité avec un minimum de données, soulignant son efficacité dans l'apprentissage de domaine.

Enfin, la figure 7b montre le nombre de méthodes inférées par HierAMLSI. Pour la plupart des domaines, à l'exception de Spanner et Elevator, HierAMLSI a tendance à inférer un nombre similaire de méthodes à celles des domaines IPC (voir Table 1). Cependant, pour les domaines Spanner et Elevator, le nombre de méthodes inférées est inférieur à celui des domaines IPC. Dans ces cas, les réseaux de tâches inférés au sein des méthodes sont trop généraux. Cela entraîne la génération de plans solution excessivement longs ou incorrects.

6 État de l'art

De nombreuses approches ont été développées pour apprendre des domaines de planification HTN. Ces approches peuvent être catégorisées en fonction du type de données d'entrée utilisées dans le processus d'apprentissage et de la sortie qu'elles produisent. La sortie peut inclure le modèle d'actions, l'ensemble des méthodes, et peut ou non inclure les préconditions des méthodes. Les données d'entrée peuvent comprendre des plans générés en résolvant un ensemble de problèmes de planification, des plans annotés, des arbres de décomposition, ou des marches aléatoires. De plus, les données d'entrée peuvent incorporer des états observés en plus des tâches, et ces états peuvent être entièrement observables, partiellement observables, ou bruités.

Ces approches se concentrent principalement sur les aspects structurels de la décomposition des tâches. Notamment, des approches telles que [37, 13, 25, 10, 11, 1, 27, 26, 35] reposant sur des traces de plans comme données d'entrée. Certaines approches, telles que HTN-Maker [14, 12], LHTNDT [30] et l'approche proposée par [4], utilisent également des plans annotés, segmentés en différentes tâches et leurs décompositions intermédiaires, mais l'acquisition de tels exemples annotés demande un effort humain important.

Ces approches présentent plusieurs limitations. Elles se concentrent uniquement sur l'apprentissage des méthodes. De plus, les préconditions des méthodes ne sont pas toujours apprises. Seules les approches proposées par [23, 21, 22, 11, 31, 26, 35, 4] apprennent les préconditions des méthodes.

Enfin, seules quelques approches [39, 40, 36] sont capables d'apprendre à la fois des modèles d'actions et des méthodes. Cependant, aucune de ces approches n'est robuste aux observations bruitées et les domaines appris ne sont pas suffisamment précis pour être utilisées directement, c'est à dire sans la moindre correction humaine, par un planificateur.

7 Conclusion

Dans cet article, nous avons présenté l'algorithme d'apprentissage de domaines HDDL HierAMLSI. HierAMLSI est parmi les rares algorithmes capables d'apprendre à la fois des actions et des méthodes, et il détient la capacité unique d'apprendre à partir d'observations partielles et bruitées. Notre évaluation expérimentale a démontré que HierAMLSI peut apprendre de manière efficace des domaines de planification précis même dans des scénarios avec des niveaux élevés de bruit, et ce, avec un minimum de données d'entrée. De plus, HierAMLSI montre la capacité à dériver une représentation concise des méthodes, le nombre de méthodes apprises se rapprochant de celui des domaines d'origines.

Malgré ces résultats prometteurs, l'expressivité des domaines appris par HierAMLSI reste limitée. Une piste d'amélioration serait d'augmenter l'expressivité des domaines appris. Plus précisément, des travaux récents ont montré des résultats prometteurs pour l'apprentissage de domaines temporels non hiérarchiques [9, 5]. Une piste d'amélioration serait d'étendre notre approche pour apprendre des domaines temporels hiérarchiques en tirant parti des efforts en cours de la communauté pour incorporer des caractéristiques temporelles dans le langage HDDL [34].

Références

[1] Kevin Chen, Nithin Shrivatsav Srikanth, David Kent, Harish Ravichandar, and Sonia Chernova. Learning hierarchical task networks with preferences from unannotated demonstrations. In *Proc. of CoRL*, pages 1572–1581, 2021.

[2] Vasek Chvátal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3) :233–235, 1979.

[3] Stephen Cresswell, Thomas Leo McCluskey, and Margaret Mary West. Acquiring planning domain models using *LOCM*. *Knowl. Eng. Rev.*, 28(2) :195–213, 2013.

[4] Morgan Fine-Morris, B Auslander, Michael W Floyd, Greg Pennisi, Héctor Muñoz-Avila, and EDU Kalyan Moy Gupta. Learning hierarchical task networks with landmarks and numeric fluents by combining symbolic and numeric regression. In *Proc. of the 2020 Conference on Advances in Cognitive Systems*, 2020.

[5] Antonio Garrido and Sergio Jiménez. Learning temporal action models via constraint programming. In *Proc. of ECAI*, pages 2362–2369, 2020.

[6] Fred W. Glover and Manuel Laguna. *Tabu Search*. Kluwer, 1997.

[7] M. Grand, H. Fiorino, and D. Pellier. Amlsi : A novel and accurate action model learning algorithm. In *Proc. of KEPS Workshop*, 2020.

[8] M. Grand, H. Fiorino, and D. Pellier. Retro-engineering state machines into pddl domains. In *Proc. of ICTAI*, pages 1186–1193, 2020.

[9] Maxence Grand, Damien Pellier, and Humbert Fiorino. Tempamlsi : Temporal action model learning based on STRIPS translation. In *Proc. of ICAPS*, pages 597–605, 2022.

[10] Bradley Hayes and Brian Scassellati. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *Proc. of ICRA*, pages 5469–5476, 2016.

[11] Philippe Hérail and Arthur Bit-Monnot. Leveraging demonstrations for learning the structure and parameters of hierarchical task networks. In *Proc. of FLAIRS*, 2023.

[12] Chad Hogg, Ugur Kuter, and Héctor Munoz-Avila. Learning hierarchical task networks for nondeterministic planning domains. In *Proc. of IJCAI*, 2009.

[13] Chad Hogg, Ugur Kuter, and Hector Munoz-Avila. Learning methods to generate good plans : Integrating htn learning and reinforcement learning. In *Proc. of AAAI*, volume 24, 2010.

[14] Chad Hogg, Héctor Munoz-Avila, and Ugur Kuter. Htn-maker : Learning htns with minimal additional knowledge engineering required. In *Proc. of AAAI*, pages 950–956, 2008.

[15] Daniel Höller. Translating totally ordered htn planning problems to classical planning problems using regular approximation of context-free languages. In *Proc. of ICAPS*, volume 31, pages 159–167, 2021.

[16] Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. Language classification of hierarchical planning problems. In *Prof. of ECAI*, pages 447–452, 2014.

[17] Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of ICAPS*, page 158–165, 2016.

[18] Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. Hddl : An extension to pddl for expressing hierarchical planning problems. In *Proc. of AAAI*, pages 9883–9891, 2020.

[19] Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ronald Alford. Hierarchical planning in the IPC. In *Proc. of HPlan Workshop (ICAPS)*, 2019.

[20] Richard Howey and Derek Long. Val's progress : The automatic validation tool for pddl2. 1 used in the international planning competition. In *Proc. of IPC Workshop*, pages 28–37, 2003.

[21] Okhtay Ilghami, Héctor Muñoz-Avila, Dana S. Nau, and David W. Aha. Learning approximate preconditions for methods in hierarchical plans. In *Proc. of ICML*, pages 337–344, 2005.

[22] Okhtay Ilghami, Dana S. Nau, and Héctor Muñoz-Avila. Learning to do HTN planning. In *Proc. of ICAPS*, pages 390–393, 2006.

[23] Okhtay Ilghami, Dana S. Nau, Héctor Muñoz-Avila, and David W. Aha. Camel : Learning method preconditions for HTN planning. In *Proc. of ICAPS*, pages 131–142, 2002.

[24] Richard M. Karp. Reducibility among combinatorial problems. In *Proc. of a symposium on the Complexity of Computer Computations*, pages 85–103, 1972.

[25] Nan Li, William Cushing, Subbarao Kambhampati, and Sungwook Yoon. Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences. *ACM Trans. Intell. Syst. Technol.*, 5(2) :1–32, 2014.

[26] Ruoxi Li, Mark Roberts, Morgan Fine-Morris, and Dana Nau. Teaching an htn learner. In *Proc. of HPlan*, pages 68–72, 2022.

[27] Damir Lotinac and Anders Jonsson. Constructing hierarchical task models using invariance analysis. In *Proc. of ECAI*, pages 1274–1282, 2016.

[28] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL-the planning domain definition language, 1998.

[29] Kira Mourão, Luke S. Zettlemoyer, Ronald P. A. Petrick, and Mark Steedman. Learning STRIPS operators from noisy and incomplete observations. In *Proc. of UAI*, pages 614–623, 2012.

[30] Fatemeh Nargesian and Gholamreza Ghassem-Sani. Lhtndt : Learn htn method preconditions using decision tree. In *Proc. of ICINCO-ICSO*, pages 60–65, 2008.

[31] Conny Olz, Susanne Biundo, and Pascal Bercher. Revealing hidden preconditions and effects of compound htn planning tasks–a complexity analysis. In *Proc. of AAAI*, pages 11903–11912, 2021.

[32] Jose Oncina and Pedro García. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis : Selected Papers from the IVth Spanish Symposium*, volume 1, pages 49–61. World Scientific, 1992.

[33] D. Pellier and H. Fiorino. PDDL4J : a planning domain description library for java. *J. Exp. Theor. Artif. Intell.*, 30(1) :143–176, 2018.

[34] Damien Pellier, A. Albore, Humbert Fiorino, and R. Bailon-Ruiz. HDDL 2.1 : Towards defining a formalism and a semantics for temporal htn planning. In *Proc. of the HPlan Workshop*, 2023.

[35] Greg Pennisi, Morgan Fine-Morris, Michael W Floyd, Bryan Auslander, Héctor Munoz-Avila, Jeff Heflin, and Kalyan Moy Gupta. Htn learning via transfer learning of domain landmarks. In *Proc. of the 2021 International Florida Artificial Intelligence Research Society Conference*, 2021.

[36] José A Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. Learning htn domains using process mining and data mining techniques. In *Proc. of Generalized Planning Workshop*, 2017.

[37] Zhanhao Xiaoa, Hai Wan, Hankui Hankz Zhuoa, Andreas Herzigb, Laurent Perrusselc, and Peilin Chena. Learning htn methods with preference from htn planning instances. *Proc. of HPlan Workshop*, page 31, 2019.

[38] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artif. Intell.*, 171(2-3) :107–143, 2007.

[39] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Hector Muñoz-Avila. Learning HTN method preconditions and action models from partial observations. In *Proc. of IJCAI*, pages 1804–1810, 2009.

[40] Hankz Hankui Zhuo, Héctor Muñoz-Avila, and Qiang Yang. Learning hierarchical task network domains from partially observed plan traces. *Artificial intelligence*, 212 :134–157, 2014.

[41] Hankz Hankui Zhuo, Tuan Anh Nguyen, and Subbarao Kambhampati. Refining incomplete planning domain models through plan traces. In *Proc. of IJCAI*, pages 2451–2458, 2013.

[42] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, 174(18) :1540–1569, 2010.