29th International Conference on
Automated Planning and Scheduling

July 11 – 15, 2019, Berkeley (California), USA

HPlan 2019

Proceedings of the 2nd ICAPS Workshop on

Hierarchical Planning

Edited by:

Pascal Bercher, Gregor Behnke, Vikas Shivashankar, and Ron Alford

Organizing Committee

Pascal Bercher	Ulm University, Germany
Gregor Behnke	Ulm University, Germany
Vikas Shivashankar	Amazon Robotics, North Reading, Massachusetts, USA
Ron Alford	The MITRE Corporation, McLean, Virginia, USA

Program Committee

Ron Alford	The MITRE Corporation, McLean, Virginia, USA
Gregor Behnke	Ulm University, Germany
Pascal Bercher	Ulm University, Germany
Susanne Biundo	Ulm University, Germany
Rogelio E. Cardona-Rivera	University of Utah, USA
Humbert Fiorino	Université Grenoble Alpes, France
Daniel Höller	Ulm University, Germany
Héctor Muñoz-Avila	Lehigh University, Pennsylvania, USA
Damien Pellier	Université Grenoble Alpes, France
Felix Richter	Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, Stuttgart, Germany
Vikas Shivashankar	Amazon Robotics, North Reading, Massachusetts, USA
Lavindra de Silva	University of Cambridge, United Kingdom
Austin Tate	University of Edinburgh, Scotland

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. With this workshop, we bring together scientists working on many aspects of hierarchical planning to exchange ideas and foster cooperation.

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 issues required to tackle these – or further – problems in hierarchical planning are still unexplored.

The range of different problems in hierarchical planning is also reflected by the topics addressed by the submissions this year. While last year several submissions proposed novel planning systems (for different hierarchical problem classes), no such approaches are proposed this year. One paper applied and compared several existing planners in the computer game Minecraft to construct various complex structures. Two approaches are concerned with learning – one focusing on hierarchical task networks (HTNs), while the other focuses on hierarchical goal networks (HGNs) in non-deterministic environments. Another paper proposes an extension to the standard HTN formalism with semantic attachments. Further topics addressed are an efficient grounding procedure, a novel procedure for plan recognition and plan verification based on parsing, and a proposal for a standardized description language for HTN planning problems based on the well-known PDDL for classical, non-hierarchical problems.

This year – as several ICAPS workshops did – we tried out *openReview*, a system that allows other reviewers to take part in paper discussions, i.e., including those not officially assigned to the respective paper. Further, it allows publishing the reviews for a paper, plus the original submission context. We configured the system so that authors can decide whether they want to publish the reviews/original submission. Reviewers can also decide to hide their reviews. We thus invite you to visit openReview in case you are interested in the reviews behind some of the papers.

Pascal, Gregor, Vikas, and Ron
Workshop Organizers,
June 2019

Table of Contents

Construction-Planning Models in Minecraft

Julia Wichlacz and **Álvaro Torralba** and **Jörg Hoffmann**
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
{wichlacz,torralba,hoffmann}@cs.uni-saarland.de

Abstract

Minecraft is a videogame that offers many interesting challenges for AI systems. In this paper, we focus on construction scenarios where an agent must build a complex structure made of individual blocks. As higher-level objects are formed of lower-level objects, the construction can naturally be modelled as a hierarchical task network. We model a house-construction scenario in classical and HTN planning and compare the advantages and disadvantages of both kinds of models.

Introduction

Minecraft is an open-world computer game, which poses interesting challenges for Artificial Intelligence (Aluru et al. 2015; Johnson et al. 2016), for example for the evaluation of reinforcement learning techniques (Tessler et al. 2017). Previous research on planning in Minecraft focused on models to control an agent in the Minecraft world. Some examples include learning planning models from a textual description of the actions available to the agent and their preconditions and effects (Branavan et al. 2012), or HTN models from observing players' actions (Nguyen et al. 2017). Roberts et al. (2017), on the other hand, focused on online goal-reasoning for an agent that has to navigate in the minecraft environment to collect resources and/or craft objects. They introduced several propositional, numeric (Fox and Long 2003) and hybrid PDDL+ planning models (Fox and Long 2006).

In contrast, we are interested in construction scenarios, where we generate instructions for making a given structure (e.g. a house) that is composed of atomic blocks. Our long-term goal is to design a natural-language system that is able to give instructions to a human user tasked with completing that construction. As a first step, in the present paper we consider planning methods coming up with what we call a *construction plan*, specifying the sequence of construction steps without taking into account the natural-language and dialogue parts of the problem.

For the purpose of construction planning, the Minecraft world can be understood as a Blocksworld domain with a 3D environment. Blocks can be placed at any position having a non-empty adjacent position. However, while obtaining a sequence of "put-block" actions can be sufficient for an AI agent, communicating the plan to a human user requires more structure in order to formulate higher-level instructions like *build-row*, or *build-wall*. The objects being constructed (e.g. rows, walls, or an entire house) are naturally organized in a hierarchy where high-level objects are composed of lower-level objects. Therefore, the task of constructing a high-level object naturally translates into a hierarchical planning network (HTN) (Sacerdoti 1974; Tate 1977; Wilkins 1988; Erol, Hendler, and Nau 1994).

We devise several models in both classical PDDL planning (Bylander 1994; McDermott et al. 1998) and hierarchical planning for a simple scenario where a house must be constructed. Our first baseline is a classical planning model that ignores the high-level objects and simply outputs a sequence of place-blocks actions. This is insufficient for our purposes since the resulting sequence of actions can hardly be described in natural language. However, it is a useful baseline to compare the other models. We also devise a second classical planning model, where the construction of high-level objects is encoded via auxiliary actions.

HTN planning, on the other hand, allows to model the object hierarchy in a straightforward way, where there is a task for building each type of high-level object. The task of constructing each high-level object can be decomposed into tasks that construct its individual parts. Unlike in classical planning, where the PDDL language is supported by most/all planners, HTN planners have their own input language. Therefore, we consider specific models for two individual HTN planners: the PANDA planning system (Bercher, Keen, and Biundo 2014; Bercher et al. 2017) and SHOP2 (Nau et al. 2003).

Scenario Design

We consider a simple scenario where our agent must construct a house in Minecraft. We model the Minecraft environment as a 3D grid, where each location is either empty or has a block of a number of types: wood, stone, or dirt.

Figure 1 shows the hierarchy of objects of our construction scenario. For the high-level structure the house consists of four stone walls, a stone roof, and a door. The walls and the roof are further decomposed into single rows that need to be built out of individual blocks. The door consists of two gaps, i.e., empty positions inside one of the walls.

As our focus is on the construction elements we abstract low-level details away. For example, we avoid encoding the position of the agent and assume that all positions are always reachable. We also assume Minecraft's creative mode, where all block types are always available so we do not need to keep track of which blocks are there in the inventory.

This is a very simplistic model, where planning focuses simply on the construction actions (i.e. placing or removing blocks), of high-level structures. Nevertheless, it can still pose some challenges to modern planners, specially due to the huge size of the Minecraft environment.

$$\begin{tikzcd}
 & & \text{House} \arrow[dll] \arrow[d] \arrow[drr] & & \\
\text{Wall} \arrow[dr] & & \text{Roof} \arrow[dl] & & \text{Door} \\
 & \text{Row} \arrow[d] & & & \\
 & \text{Block} & & &
\end{tikzcd}$$

Figure 1: Object hierarchy of our construction scenario.

Classical Planning Model

Our first model is a classical planning model in the PDDL language that consists of only two actions: *put-block(?location, ?block-type)* and *remove-block(?location, ?block-type)* where there is a different location for each of the x-y-z coordinates in a 3D grid. The goal specifies what block-type should be in each location. As blocks cannot be placed in the air, the precondition of *put-block* requires one of the adjacent locations of *?location* to be non-empty. Other than that, blocks of any type can always be added or removed at any location. The goal is simply a set of *block_at* facts.

A limitation of this simple model is that it completely ignores the high-level structure of the objects being constructed. As there is no incentive to place blocks in certain order, a high-level explanation of the plan may be impossible. To address this, we introduce auxiliary actions that represent the construction of high-level objects. Figure 2 shows the auxiliary actions that represent building a wall. The attributes of the wall are specified in the initial state via attributes expressed by predicates *wall_dir*, *wall_length*, *wall_height*, *wall_type*, and *current_wall_loc*. In order to avoid the huge amount of combinations of walls that could be constructed of any dimensions and in any direction, the walls that are relevant for the construction at hand are specified in the initial state via these predicates. These three actions decompose the construction of a wall into several rows. Action *begin_wall* ensures that no other high-level object is being constructed at the moment and adds the fact *constructing_wall* to forbid the construction of any other wall (or roof) until the current wall has been finished.

Action *build_row_in_wall* ensures that a row of the given length will be built on the corresponding location and direction by adding predicates *(building_row)* and *(rest_row ?loc ?len ?dir ?t)*. Simultaneously, it updates the location for the rest of the wall to be built and decreases its height by one.

```
(:action begin_wall
  :parameters (?w - wall)
  :precondition (and (not (constructing_roof))
                     (not (constructing_wall))
  :effect (and (current_wall ?w) (constructing_wall)))

(:action build_row_in_wall
  :parameters (?w - wall ?loc ?locN - location
    ?len ?height ?heightN - number
    ?dir - direction ?t - blocktype)
  :precondition (and (current_wall ?w) (wall_dir ?w ?dir)
    (wall_length ?w ?len) (wall_height ?w ?height)
    (wall_type ?w ?t) (current_wall_loc ?w ?loc)
    (prev ?height ?heightN) (on_top ?loc ?locN)
    (not (building_row)))
  :effect (and (current_wall_loc ?w ?locN)
    (wall_height ?w ?heightN)
    (not (current_wall_loc ?w ?loc))
    (not (wall_height ?w ?height))
    (building_row) (rest_row ?loc ?len ?dir ?t)))

(:action finish_wall
  :parameters (?w - wall ?loc - location
    ?height - number ?dir - direction)
  :precondition (and (current_wall ?w) (is_zero ?height)
    (wall_height ?w ?height) (wall_dir ?w ?dir)
    (wall_initial ?w ?loc) (not(building_row)))
  :effect (and (wall_at ?w ?loc ?dir)
    (not (constructing_wall)) (not (current_wall ?w))))
```

Figure 2: Auxiliary PDDL actions to build a wall.

When the height is zero, the action *end_wall* becomes applicable, which finishes the construction of the wall.

In the goal we then use the predicates *wall_at* and *roof_at* that force the planner to use these constructions, instead of a set of *block_at* facts as we did in the simple model.

Hierarchical Planning Models

HTN models encode the construction of high-level objects in a straightforward way by defining tasks such as *build_house, build_wall* and *build_row*. These tasks will then be decomposed with methods until only primitive tasks will be left, in our case *place-block* and *remove-block*. We consider specific models for two individual HTN planners: the PANDA planning system (Bercher, Keen, and Biundo 2014; Bercher et al. 2017) and SHOP2 (Nau et al. 2003).

PANDA

PANDA uses an HTN formalism (Geier and Bercher 2011), which allows combining classical and HTN planning. The predicates describing the world itself, i.e. the relations between different locations remain the same as in the PDDL model, as do the *place-block* and *remove-block* primitive actions. On top of this, high-level objects are described as an HTN where each object corresponds to a task, without requiring to express their attributes with special predicates as we did in the PDDL model. Specifically, we defined tasks

```
(:method build_wall_1
  :parameters (?loc1 - location ?len ?hgt - numbers
    ?d - direction ?t - blocktype)
  :task (buildwall ?loc1 ?len ?hgt ?d ?t)
  :precondition (isone ?hgt)
  :subtasks (buildrow ?loc1 ?len ?d ?t))

(:method build_wall_2
  :parameters (?loc1 ?loc2 - location
    ?len ?hgt ?hgt2 - numbers
    ?d - direction ?t - blocktype)
  :task (buildwall ?loc1 ?len ?hgt ?d ?t)
  :precondition (and (not (isone ?hgt))
    (prev ?hgt ?hgt2)
    (on_top ?loc1 ?loc2))
  :ordered−subtasks (and
    (buildrow ?loc1 ?len ?d ?t)
    (buildwall ?loc2 ?len ?hgt2 ?d ?t)))
```

Figure 3: Methods for the build-wall task in the PANDA model.

that correspond to building a house, a wall, a roof, a row of blocks, and the door.

Figure 3 shows the methods used to decompose the task of building a wall. These methods work in a recursive fashion over the height of the wall. For walls with height one, the *build_wall_1* method is used to build them. For walls with larger height, the *build_wall_2* method decomposes the task of building them into building a row in the current location and building the rest of the wall (i.e., a wall of height-1) in the location above the previous one. These subtasks are ordered, so that walls are always built from bottom to top.

The methods for *buildrow* and *buildroof* work in the same fashion, while *buildhouse* only has one method decomposing the house into four walls, the roof, and the door. The task *builddoor* also has just one method stating which two blocks have to be removed to form a door. Choosing this way of modeling the door by first forcing the planner to place two blocks and later removing them again may seem inefficient, but for communication with a human user this may be preferable over indicating that these positions should remain empty in the first place.

SHOP2

The SHOP2 model follows a similar hierarchical task structure as the PANDA model, having methods for decomposing the house into walls, a wall into rows and rows into single blocks. Since one of the advantages of SHOP2 is that it can call arbitrary LISP functions, we can represent the locations using integers as coordinates and replace the predicates used in PANDA and PDDL to express their relations by simple arithmetic operations. This also allows us to compute the end point of rows of any given length in a given direction, which means we can construct the walls by alternating the direction of the rows. Based on this, we define two different recursive decompositions of walls as shown in Figure 4. In the first method we simply build the row starting in the current location, while in the second method we change the direction of the row we want to build and identify the position that would previously have been the end of the row by replacing the x-coordinate with $x + \mathit{length} - 1$. Since this computation is different for each direction, we need separate methods for them. Apart from this, the decomposition structure is the same as with PANDA, building the walls, roof, and rows incrementally using a recursive structure.

```
(:method (build-wall-east ?x ?y ?z ?length ?height ?dir)
zero-height
  ((call = ?height 0))
  ()

east-one
  ((up ?z1 ?z) (up ?height ?h1) (call = ?dir 1))
  (:ordered
    (:task build-row ?x ?y ?z ?length ?dir)
    (:task build-wall-east ?x ?y ?z1 ?length ?h1 ?dir)
  )
)

(:method (build-wall-east ?x ?y ?z ?length ?height ?dir)
zero-height
  ((call = ?height 0))
  ()

 east-two
  ((up ?z1 ?z) (up ?height ?h1) (call = ?dir 1))
  (:ordered
    (:task build-row (call -(call + ?x ?length) 1) ?y ?z ?length 2)
    (:task build-wall-east ?x ?y ?z1 ?length ?h1 ?dir)
  )
)
```

Figure 4: SHOP2 methods to build a wall in east direction.

Experiments

To evaluate the performance of common planners on our models[1], we scale them with respect to two orthogonal parameters: the size of the construction, and the size of the cubic 3D world we are considering. We use different planners for each model. For the classical planning models we use the LAMA planner (Richter, Westphal, and Helmert 2011). The PANDA planning system implements several algorithms, including plan space POCL-based search methods (Bercher, Keen, and Biundo 2014; Bercher et al. 2017), SAT-based approaches (Behnke, Höller, and Biundo 2018), and forward heuristic search (Höller et al. 2018). We use a configuration using heuristic search with the FF heuristic, which works well on our models. For SHOP2, we use the depth-first search configuration (Nau et al. 2003). All experiments were run on an Intel i5 4200U processor with a time limit of 30 minutes and a memory limit of 2GB.

In our first experiment, we scale the size of the house starting with a $3 \times 3 \times 3$ house and increasing one parameter (length, width, and height) at a time ($4 \times 3 \times 3, 4 \times 4 \times$

[1]Benchmarks are publicly available at: `https://doi.org/10.5281/zenodo.3239243`

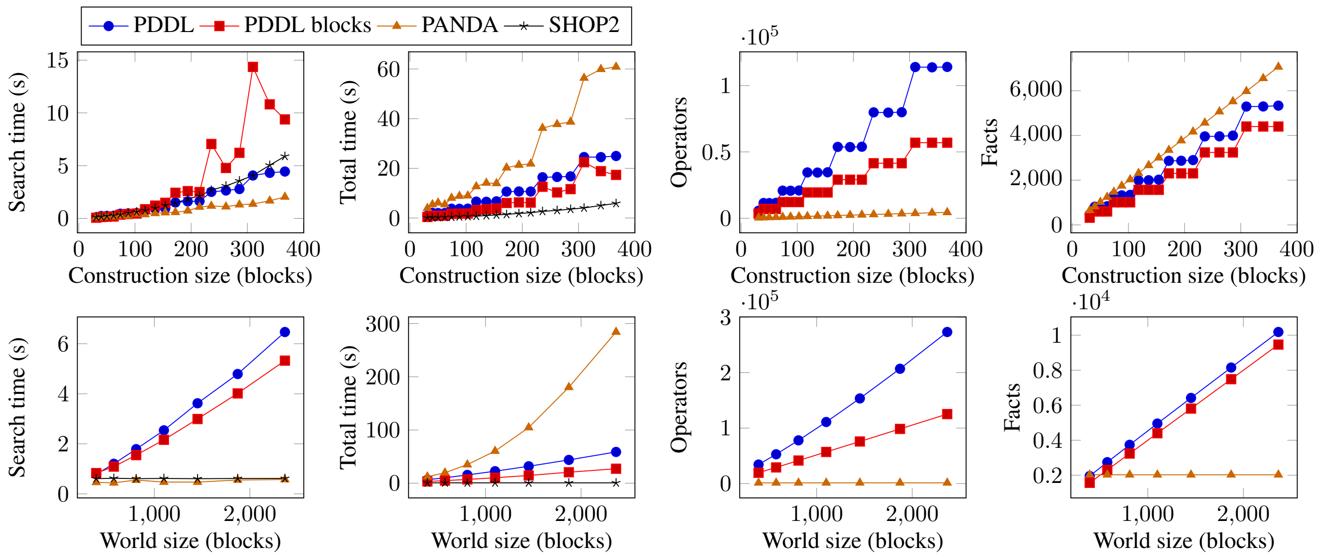


Figure 5: Search time, total time, number of operators, and facts of the grounded task to build a house with given number of blocks (above) or in a world with increasing size (below).

$3, \ldots, 9 \times 9 \times 9$.). The size of the 3D world is kept as small as possible to fit the house with some slack, so initially is set to $5 \times 5 \times 5$ and is increased by one unit in each direction every three steps, once we have scaled the house in all dimensions. The upper row of Figure 5 shows the search and total time of the planners on the different models. The construction size in the x-axis refers to the number of blocks that need to be placed in the construction. All planners scale well with respect to search time, solving problems of size up to $9 \times 9 \times 9$ in just a few seconds. The non-hierarchical PDDL planning model (PDDL blocks) that only uses the *place-block* and *remove-block* actions without any hierarchical information is the one with worst search performance. Moreover, it also results in typically longer plans that build many "support" structures to place a block in a wall without one of the adjacent blocks in the wall being there yet.

However, there is a huge gap between search and total time for the PANDA and PDDL models, mostly due to the overhead of the grounding phase. SHOP2 does not do any preprocessing or grounding so it is not impacted by this. For the PANDA and PDDL models, total time significantly increases every three problems, whenever the world size is increased. This suggests that, somewhat counterintuitively, the size of the world environment has a greater impact on these planners' performance than the size of the construction. In the PDDL based approaches, the number of operators and facts produced in the preprocessing shows a similar trend so the planner's performance seems directly influenced by the size of the grounded task. For PANDA, on the other hand, we observe a linear increase in the number of facts and only a comparatively small increase in the number of operators.

To test more precisely what is the impact of increasing the world size, we ran a second set of experiments where we kept the size of the house fixed at $5 \times 5 \times 5$ and just increased the size of the world. As shown in the bottom part of Figure 5 the performance of SHOP2 is not affected at all, since it does not require enumerating all possible locations. Search time for PANDA also stays mostly constant, but the overhead in the preprocessing phase dominates the total time. This contrasts with the number of operators and facts, which is not affected by the world size at all. The PDDL based models are also affected in terms of preprocessing time, due to a linear increase in the number of facts and operators with respect to world size, but to a lesser degree. However, search time increases linearly with respect to the world size due to the overhead caused in the heuristic evaluation.

Discussion

We have introduced several models of a construction scenario in the Minecraft game. Our experiments have shown that, even in the simplest construction scenario which is not too challenging from the point of view of the search, current planners may struggle when the size of the world increases. This is a serious limitation in the Minecraft domain, where worlds with millions of blocks are not unrealistic.

Lifted planners like SHOP2 perform well. However, it must be noted that they follow a very simple search strategy, which is very effective on our models where any method decomposition always leads to a valid solution. However, it may be less effective when other constraints must be met and/or optimizing quality is required. For example, if some blocks are removed from the ground by the user, then some additional blocks must be placed as auxiliary structure for the main construction. Arguably, this could be easily fixed by changing the model so that whenever a block cannot be placed in a target location, an auxiliary tower of blocks is built beneath the location. However, this increases the burden of writing new scenarios since suitable task decompositions (along with good criteria of when to select each decomposition) have to be designed for all possible situations.

This makes the SHOP2 model less robust to unexpected situations that were not anticipated by the domain modeler. PANDA, on the other hand, supports insertion of primitive actions (Geier and Bercher 2011), allowing the planner to consider placing additional blocks, e.g., to build supporting structures that do not correspond to any task in the HTN. This could help to increase the robustness of the planner in unexpected situations where auxiliary structures that have not been anticipated by the modeler are needed. However, this is currently only supported by the POCL-plan-based search component and considering all possibilities for task insertion significantly slows down the search and it runs out of memory in our scenarios. This may point out new avenues of research on more efficient ways to consider task insertion.

In related Minecraft applications, cognitive priming has been suggested as a possible solution to keep the size of the world considered by the planner at bay (Roberts and Hiatt 2017). In construction scenarios, however, large parts of the environment can be relevant so incremental grounding approaches may be needed to consider different parts of the scenario at different points in the construction plan.

Our models are still a simple prototype and they do not yet capture the whole complexity of the domain. We plan to extend them in different directions in order to capture how hard it is to describe actions or method decompositions in natural language. For example, while considering the position of the user is not strictly necessary, his visibility may be important because objects in his field of view are easier to describe in natural language. How to effectively model the field of vision is a challenging topic, which may lead to combinations with external solvers like in the planning modulo theories paradigm (Gregory et al. 2012).

Another interesting extension is to consider how easy it is to express the given action in natural language and for example by reducing action cost for placing blocks near objects that can be easily referred to. Such objects could be landmarks e.g. blocks of a different type ("put a stone block next to the blue block") or just the previously placed block (e.g., "Now, put another stone block on top of it").

References

Aluru, K. C.; Tellex, S.; Oberlin, J.; and MacGlashan, J. 2015. Minecraft as an experimental world for AI in robotics. 5–12. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – totally-ordered hierarchical planning through SAT. In McIlraith, S., and Weinberger, K., eds., *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 6110–6118. AAAI Press.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 480–488. AAAI Press/IJCAI.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In Edelkamp, S., and Bartak, R., eds., *Proceedings of the 7th Annual Symposium on Combinatorial Search (SOCS'14)*. AAAI Press.

Branavan, S. R. K.; Kushman, N.; Lei, T.; and Barzilay, R. 2012. Learning high-level planning from text. 126–135. The Association for Computer Linguistics.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1–2):165–204.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference of the American Association for Artificial Intelligence (AAAI'94)*, 1123–1129. Seattle, WA: MIT Press.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research* 27:235–297.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, 1955–1961. AAAI Press/IJCAI.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 114–122. AAAI Press.

Johnson, M.; Hofmann, K.; Hutton, T.; and Bignell, D. 2016. The malmo platform for artificial intelligence experimentation. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, 4246–4247. AAAI Press/IJCAI.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. Shop2: An htn planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Nguyen, C.; Reifsnyder, N.; Gopalakrishnan, S.; and Muñoz-Avila, H. 2017. Automated learning of hierarchical task networks for controlling minecraft agents. 226–231. IEEE.

Richter, S.; Westphal, M.; and Helmert, M. 2011. LAMA 2008 and 2011 (planner abstract). In *IPC 2011 planner abstracts*, 50–54.

Roberts, M., and Hiatt, L. M. 2017. Improving sequential decision making with cognitive priming. *Advances in Cognitive Systems*.

Roberts, M.; Piotrowski, W.; Bevan, P.; Aha, D.; Fox, M.; Long, D.; and Magazzeni, D. 2017. Automated planning with goal reasoning in minecraft. In *Proceedings of ICAPS workshop on Integrated Execution of Planning and Acting*.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115135.

Tate, A. 1977. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, 888–893. Cambridge, MA: William Kaufmann.

Tessler, C.; Givony, S.; Zahavy, T.; Mankowitz, D. J.; and Mannor, S. 2017. A deep hierarchical approach to lifelong learning in minecraft. In Singh, S., and Markovitch, S., eds., *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, 1553–1561. AAAI Press.

Wilkins, D. E. 1988. *Practical Planning*. San Francisco, CA: Morgan Kaufmann.

HDDL – A Language to Describe Hierarchical Planning Problems

D. Höller[*], G. Behnke[*], P. Bercher[*], S. Biundo[*], H. Fiorino[†], D. Pellier[†], and R. Alford[‡]
[*]Institute of Artificial Intelligence, Ulm University, 89081 Ulm, Germany
{daniel.hoeller, gregor.behnke, pascal.bercher, susanne.biundo}@uni-ulm.de
[†]University Grenoble Alpes, LIG, F-38000 Grenoble, France
{humbert.fiorino, damien.pellier}@imag.fr
[‡]The MITRE Corporation, McLean, Virginia, USA
ralford@mitre.org

Abstract

The research in hierarchical planning has made considerable progress in the last few years. Many recent systems do not rely on hand-tailored advice anymore to find solutions, but are supposed to be domain-independent systems that come with sophisticated solving techniques. In principle, this development would make the comparison between systems easier (because the domains are not tailored to a single system anymore) and – much more important – also the integration into other systems, because the modeling process is less tedious (due to the lack of advice) and there is no (or less) commitment to a certain planning system the model is created for. However, these advantages are destroyed by the lack of a common input language and feature set supported by the different systems. In this paper, we propose an extension to PDDL, the description language used in non-hierarchical planning, to the needs of hierarchical planning systems. We restrict our language to a basic feature set shared by many recent systems, give an extension of PDDL's EBNF syntax definition, and discuss our extensions, especially with respect to planner-specific input languages from related work.

1 Introduction

Much progress has been made recently in the field of hierarchical planning. Novel systems based on the traditional, search-based techniques have been introduced (Bit-Monnot, Smith, and Do 2016; Shivashankar, Alford, and Aha 2017; Bercher et al. 2017; Höller et al. 2018), but also new techniques like the translation to STRIPS/ADL (Alford, Kuter, and Nau 2009; Alford et al. 2016a), or revisited approaches like the translation to propositional logic (Behnke, Höller, and Biundo 2018a; 2018b; 2019a; 2019b; Schreiber et al. 2019). In contrast to earlier systems, such systems can be considered to be domain-independent, i.e., they do not rely on hand-tailored advice to solve planning problems, but only on their solving techniques.

Even though the systems share the basic idea of being *hierarchical* planning approaches, the feature set supported by the different systems is manifold. Bit-Monnot, Smith, and Do (2016) focus, e.g., on advanced support for temporal planning, but lack the support for recursion; several systems are restricted to models that do not include partial ordering (Alford, Kuter, and Nau 2009; Behnke, Höller, and Biundo 2018a; Schreiber et al. 2019); and some, like the one by Shivashankar, Alford, and Aha (2017) even define an entirely new type of hierarchical planning problems.

Even systems restricted to the maybe best-known and most basic hierarchical formalism, called *Hierarchical Task Network* (HTN) planning, do not share a common input language, though the differences between the input languages are sometimes rather subtle, e.g. between the formalisms used by Alford et al. (2016a) and Bercher et al. (2017). To the best of our knowledge, the hierarchical language introduced for the first International Planning Competition (McDermott et al. 1998) is not supported by any recent system.

The lack of a common language has several consequences for the field. First, it makes the comparison between the systems tedious due to the translation process. Second – and even more important – it makes the use of hierarchical planning from a practical perspective laborious, because it is not possible to model a problem at hand and try which system performs best on it. Selecting the system in beforehand (if possible) requires much insights into the systems.

A common description language would make the comparison of the systems easier, it could foster a common set of supported features and result in a common benchmark set the systems are evaluated on.

In this paper, we propose the *Hierarchical Domain Definition Language* (HDDL) as common input language for hierarchical planning problems. It is widely based on the input language of PANDA, the framework underlying the planning systems by Bercher et al. (2017), Höller et al. (2018; 2019), and Behnke, Höller, and Biundo (2018a; 2019a; 2019b). We define it as an extension of the STRIPS fragment (language level 1) of the PDDL2.1 definition (Fox and Long 2003). To concentrate on a set of features shared by many systems, we restrict the language to basic HTN planning. However, we hope that the given definition is just the starting point for further language extensions like the first PDDL version in classical planning was.

We first introduce a lifted HTN formalism from the literature, before we define our language *by example*. We go through new language elements, introduce their syntax and meaning, discuss our design choices and differences to approaches from the literature, namely PDDL1.2 (McDermott et al. 1998), SHOP(2) (Nau et al. 2003), ANML (Smith, Frank, and Cushing 2008), HPDDL (Alford et al. 2016a), GTOHP (Ramoul et al. 2017), HTN-

PDDL (González-Ferrer, Fernández-Olivares, and Castillo 2009), and HATP (de Silva, Lallement, and Alami 2015).

We then give a full EBNF syntax definition[1] based on the definition of PDDL2.1 and discuss every extension and change. We conclude with a short outlook.

2 Lifted HTN Planning

In this section we formally define the problem class HDDL can describe, i.e., standard HTN planning in line with the text book description by Ghallab, Nau, and Traverso (2004). To define the formal framework we extend the formalization of Alford, Bercher, and Aha (2015a; 2015b).

Our *lifted* formalism is based upon a quantifier-free first-order predicate logic $\mathcal{L} = (P, T, V, C)$ with the following elements. P is a finite set of *predicate symbols*, each having a finite arity. The arity defines its number of parameter variables (taken from V), each having a certain type (defined in T). Thus, T is a finite set of *type symbols* as is also known from PDDL. V is a finite set of typed variable symbols to be used by the parameters of the predicates in P. C is a finite set of typed constants. They are the syntactic representation of the objects in the real world. Please be aware that a single constant can have several types, e.g. *truck* and *vehicle* to support a type hierarchy.

The basic data structure in HTN planning is a *task network*. Task networks are partially ordered multi-sets of tasks.

In contrast to classical (non-hierarchical) planning, there are two kinds of tasks in HTN planning: primitive and compound ones. Task networks can contain both primitive tasks (also called actions) and compound tasks (also called abstract). Each task (primitive or compound) is given by its name, followed by a parameter sequence. For instance, a (primitive) task for driving from a source location $?ls$ to a destination location $?ld$ is given by the first-order atom $drive(?ls, ?ld)$. We do not differentiate between the expressions *task* and *task names* – both are used synonymously.

Definition 1 (Task Network)**.** *A task network tn over a set of* task names *X (first-order atoms) is a tuple $(I, \prec, \alpha, VC)$ with the following elements:*

1. *I is a finite (possibly empty) set of* task identifiers*.*
2. *$\prec$ is a strict partial order over I.*
3. *$\alpha : I \to X$ maps task identifiers to task names.*
4. *VC is a set of variable constraints. Each constraint can bind two task parameters to be (non-)equal and it can constrain a task parameter to be (non-)equal to a constant, or to (not) be of a certain type.*

The task identifiers are arbitrary symbols which serve as place holders (or labels) for the actual tasks they represent. We need these identifiers because any task can occur multiple times within the same task network, but the partial order needs to be able to differentiate between them. We call a task network *ground* if all task parameters are bound to (or replaced by) constants from C.

Task networks can contain primitive and/or compound tasks. *Primitive tasks* are identical to actions known from classical planning. An *action* a is a tuple $(name, pre, eff)$ with the following elements: *name* is its *task name*, i.e., a first-order atom such as $drive(?ls, ?ld)$ consisting of the (actual) name followed by a list of typed parameter variables. *pre* is its *precondition*, a first-order formula over literals over $\mathcal{L}$'s predicates. *eff* is its *effect*, a conjunction of literals over $\mathcal{L}$'s predicates (that are often divided into the positive eff^+ and the negative effects eff^-). All variables used in *pre* and *eff* are demanded to be parameters of *name*. We also write *name(a)*, *pre(a)*, and *eff(a)* to refer to these elements. We also require that for each task name *name(a)* there exists only a single action using it as its name (this way, names can be used as unique identifiers).

A *compound task* is simply a task name, i.e., an atom. In contrast to primitive tasks its purpose is not to induce a state transition, but to reference a pre-defined mapping to one or more task networks by which that compound task can be refined. They do thus not use preconditions or effects. However, there are many hierarchical planning formalisms that do also feature preconditions and/or effects for compound tasks (Bercher et al. 2016), but they are not within the scope of this paper. The before-mentioned mapping from compound tasks to pre-defined task networks is given by a set of *decomposition methods* M. A decomposition method $m \in M$ is a tuple (c, tn, VC) consisting of a compound task name c, a task network tn, and a set of variable constraints VC. The variable constraints VC allow to specify (co)designations between the parameters of c and those of the task network tn.

Definition 2 (Planning Domain)**.** *A planning domain $\mathcal{D}$ is a tuple $(\mathcal{L}, T_P, T_C, M)$ defined as follows.*

- *$\mathcal{L}$ is the underlying predicate logic.*
- *T_P and T_C are finite sets of primitive and compound tasks, respectively.*
- *M is a finite set of decomposition methods with compound tasks from T_C and task networks over the names $T_P \cup T_C$.*

The domain implicitly defines the set of all states S, being defined over all subsets of all ground predicates.

Definition 3 (Planning Problem)**.** *A planning problem $\mathcal{P}$ is a tuple $(\mathcal{D}, s_I, tn_I, g)$, where:*

- *$s_I \in S$ is the initial state, a ground conjunction of positive literals over the predicates assuming the closed world assumption.*
- *tn_I is the initial task network that may not necessarily be ground.*
- *g is the goal description, being a first-order formula over the predicates (not necessarily ground).*

HTN planning is *not* about finding courses of action achieving a certain state-based goal definition, so it makes perfect sense to specify no goal formula at all. We added them anyway to be closer to the PDDL specification. Having a goal formula in the input specification is more convenient in case one actually wants to specify a goal, it has a clearly defined semantics, and (since it can be compiled away (Geier and Bercher 2011)) causes no problems to systems that do not support it directly.

We still need to define the set of solutions for a given problem. Informally, solutions are executable, ground, primitive

[1]Syntax definitions for the ANTLR and Bison parser generators can be found online at www.uni-ulm.de/en/in/ki/panda.

task networks that can be obtained from the problem's initial task network via applying decomposition methods, adding ordering constraints, and grounding.

Lifted problems are a compact representation of their ground instantiations that are, as in classical planning, up to exponentially smaller (Alford, Bercher, and Aha 2015a; 2015b). However, we define solutions based on their grounding. The semantics of such a lifted problem is thus defined in terms of the standard semantics of its ground instantiation. We assume that the reader is familiar with the grounding process and refer to the paper by Alford, Bercher, and Aha (2015a) for details about it. To the best of our knowledge there are currently only two publications devoted to grounding in more detail – by Ramoul et al. (2017)[2] and by Behnke et al. (2019b). We now give the required definitions based on a *ground* problem and domain. Note that we do not need to represent variable constraints anymore since their constraints are already represented within the groundings.

Given ground problems/models we can now define *executability* of task networks. Let A be the set of ground actions obtained from T_P. An action $a \in A$ is called executable in a state $s \in S$ if and only if $s \models pre(a)$. The state transition function $\gamma : S \times A \to S$ is defined as in classical planning: If a is executable in s, then $\gamma(s, a) = (s \setminus \mathit{eff}^-(a)) \cup \mathit{eff}^+(a)$, otherwise $\gamma(s, a)$ is undefined. The extension of γ to action sequences, $\gamma^* : S \times A^* \to S$ is defined straightforwardly.

Definition 4 (Executability). *A task network $tn = (I, \prec, \alpha)$ is called executable if and only if there is a linearization of its task identifiers $i_1, \ldots, i_n$, $n = |I|$, such that $\alpha(i_1), \ldots, \alpha(i_n)$ is executable in s_I.*

The means of transforming one task network into another to obtain executable task networks is decomposition.

Definition 5 (Decomposition). *Let $m = (c, (I_m, \prec_m, \alpha_m))$ be a decomposition method, $tn_1 = (I_1, \prec_1, \alpha_1)$ a task network, and $I_m \cap I_1 = \emptyset$ (the latter can be achieved by renaming). Then, m decomposes a task identifier $i \in I_1$ into a task network $tn_2 = (I_2, \prec_2, \alpha_2)$ if and only if $\alpha_1(i) = c$ and*

$$\begin{aligned}
I_2 &= (I_1 \setminus \{i\}) \cup I_m \\
\prec_2 &= (\prec_1 \cup \prec_m \cup \\
&\quad\ \{(i_1, i_2) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \cup \\
&\quad\ \{(i_1, i_2) \in I_m \times I_1 \mid (i, i_2) \in \prec_1\}) \\
&\quad\ \setminus \{(i', i'') \in I_1 \times I_1 \mid i' = i \text{ or } i'' = i\} \\
\alpha_2 &= (\alpha_1 \cup \alpha_m) \setminus \{(i, c)\}
\end{aligned}$$

Now we can formally define the solution criteria.

Definition 6 (Solutions). *Let $\mathcal{P} = (\mathcal{D}, s_I, tn_I, g)$ be a planning problem with $\mathcal{D} = (\mathcal{L}, T_P, T_C, M)$ and $tn_S = (I_S, \prec_S, \alpha_S)$. tn_S is a* solution to an HTN planning problem *P if and only if*

- *There is a sequence of decompositions from tn_I to $tn = (I, \prec, \alpha)$, such that $I = I_S$, $\prec \subseteq \prec_S$, and $\alpha = \alpha_S$*
- *tn_S is primitive and has an executable action linearization leading to a state $s \models g$.*

[2]Their procedure allows to delete effectless actions (Ramoul et al. 2017), which is not allowed in standard HTN planning and would e.g. invalidate the compilation for goal descriptions.

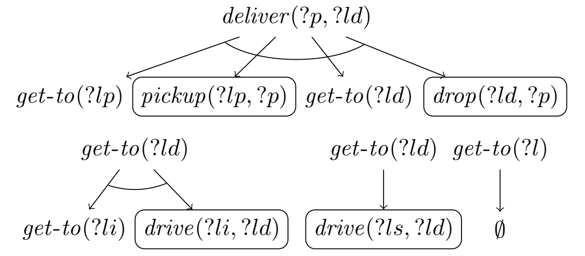


Figure 1: The method set of a simple transport domain. Actions are given as boxed nodes, abstract tasks are unboxed. All methods are totally ordered. There exists a smaller, equivalent model. However, the model has been created this way to illustrate the different language features.

3 HDDL by Example

In this section we explain our extensions to the PDDL definition based on a transport domain. To keep the example simple, the domain includes only a single transporter that has to deliver one or more packages. For each new language element we introduce its syntax and meaning and discuss the way it is modeled in other input languages.

The predicate and type definition is the same as in PDDL:

```
(define (domain transport)
  (:types location package - object)
  (:predicates
     (road ?l1 ?l2 - location)
     ...)
```

All other languages except for HATP (de Silva, Lallement, and Alami 2015) use the same theoretical model of objects and predicates as PDDL. HATP models its objects in an object-oriented way instead and further allows for SAS+ variables (Bäckström and Nebel 1995) in the input language.

The full method set of the domain is illustrated in Figure 1. Each method will be discussed in this section.

The domain contains two abstract tasks *deliver* and *get-to*. We propose to include an explicit definition of abstract tasks as it is the case for actions. HPDDL (Alford et al. 2016a) also defines abstract tasks explicitly, albeit with a slightly different syntax. Both ANML (Smith, Frank, and Cushing 2008) and HTN-PDDL (González-Ferrer, Fernández-Olivares, and Castillo 2009) require an explicit declaration of abstract tasks and their parameter types as well, but here the declaration is not separated from other elements of the domain as both declare methods together with their abstract tasks.

Some description languages for HTN problems define abstract tasks only in an implicit way by their use in methods. This includes the language used by SHOP and SHOP2 (Nau et al. 2003), PDDL1.2 (McDermott et al. 1998), HATP, as well as GTOHP (Ramoul et al. 2017). SHOP and GTOHP assume that any task that is used in a method, but is not declared to be an action is an abstract task. In contrast, PDDL1.2 assumes that every task that has no methods is primitive. This way of implicitly defining the set of compound tasks has also been chosen in some formal definitions of hierarchical problem classes (Alford, Bercher, and Aha 2015a; 2015b). However, this can be very cumbersome when debugging domains. If the modeler forgot to define a specific primitive task, the domain will still be valid, as it would be interpreted as an abstract task.

Another problem with such a definition is that the argument types are defined implicitly, namely as those with which the task can be instantiated via any method. The language of GTOHP further does not allow for using different types (that share a common ancestor in the type hierarchy) to be used for the same task. For example, there might be different methods for the *deliver* task, depending on the type of transported package. *deliver* might have two methods, one where the first argument is of type *regularPackage* and one where it is of type *valuablePackage*, the latter requiring an armored transporter. We assume that *regularPackage* and *valuablePackage* are disjunct types, but have a common super-type *package*, which would be the correct parameter type for *deliver*'s first argument. If its type is not declared explicitly, the planner can either reject the domain, as GTOHP does, or would have to infer the possible types of the arguments of an abstract task.

Declaring abstract tasks and their parameter types explicitly is also in line with the design choices of PDDL. Similar to abstract tasks, PDDL could omit the explicit definition of predicates as their types could be inferred from their usages. This is however discouraged from a modeling point-of-view.

Omitting the distinct definition of tasks and methods would also mean a significant deviation from the contemporary theoretical work on HTN planning. It could hinder further language extensions like annotating abstract tasks with constraints, e.g. preconditions and effects, as done by a couple of systems (see e.g. the survey by Bercher et al., 2016).

Here is the abstract task definition for the example:

```
6  (:task deliver :parameters (?p - package
       ?l - location))
7  (:task get-to :parameters (?l - location))
```

There is only a single method in the model to decompose deliver tasks (given at the top of Figure 1). It decomposes the task into four ordered sub-tasks: getting to the package, picking it up, getting to its final position, and dropping the package. The definition in HDDL could look like this:

```
8  (:method m-deliver
9    :parameters (?p - package
         ?lp ?ld - location)
10   :task (deliver ?p ?ld)
11   :ordered-subtasks (and
12     (get-to ?lp)
13     (pick-up ?ld ?p)
14     (get-to ?ld)
15     (drop ?ld ?p)))
```

The method definition starts with the name of the method that can e.g. be used to describe the decompositions needed to find a solution. We decided to give the method's parameters explicitly (line 9). This allows e.g. to restrict the types used in the subtasks and the decomposed task to subtypes of the original task parameters. Similarly, we can restrict the method to be applicable only to certain parameters of the abstract task it decomposes. To be correctly defined, we assume these parameters to be a superset of all parameters used in the entire method definition. The parameter definition is followed by the specification of the abstract task decomposed by the method as well as its parameters (line 10).

The same syntactical structure is used by HPDDL. In contrast, ANML, PDDL1.2, HATP and HTN-PDDL aggregate all decomposition methods belonging to a single abstract task, which have to be declared as part of the definition of an abstract task. As such, the variables that are declared as the arguments of an abstract task are automatically variables in a methods' task network. All of them type variables in methods explicitly.

In GTOHP's language, methods don't have names, but are identified via the abstract task they refine.

In SHOP, all variables inside a method are only defined implicitly by their usage as parameters of tasks and predicates inside the method. For example, the definition of a SHOP method starts with `:method` followed by an abstract task and its parameters – which if they are variables are automatically declared as new (untyped) variables. The same holds for variables that only occur as parameters of a method's subtasks. GTOHP and HTN-PDDL follow this pattern, but enforce that the parameters of the abstract task are typed, i.e., declared explicitly. Their languages however do not allow to specify the types of variables that occur in the method that are not parameters of the abstract task. Declaring the variables is, again, in line with the PDDL standard and e.g. done the same way in actions. We think it less error-prone. When the modeler explicitly defines the variables and their types, the system can check the compatibility of types and warn the modeler when undeclared variables are used (e.g. due to a spelling error).

The subtasks of the method are given afterwards (starting in line 11). We decided to have two keywords to start the definition `:ordered-subtasks` (as given here) and `:subtasks` (which we will show in the next method). When the `:ordered-subtasks` keyword is used, the given list of subtasks is supposed to be totally ordered. HPDDL uses the keyword `:tasks`, which might cause errors if mixed up with the `:task` keyword. Since GTOHP does only support totally-ordered HTN planning problems, their language only allows for specifying sequences of actions with the keyword `:expansion`.

In the subtask section, all abstract tasks and actions defined in the domain can be used as subtasks (and only these). The variables defined in the method's parameter section and the constants defined in the domain may be used as parameters (and only these).

The *get-to* task from our example domain is again abstract and may be decomposed by using one of the three methods given at the bottom of Figure 1. We start with the left one that is used when there is no direct road connection. Then the transporter needs to go to the final location $?ld$ via some intermediate location $?li$. Therefore the method decomposes the task into another abstract *get-to* task, followed by a *drive* action with the destination location $?ld$.

```
16  (:method m-drive-to-via
17    :parameters (?li ?ld - location)
18    :task (get-to ?ld)
19    :subtasks (and
20      (t1 (get-to ?li))
21      (t2 (drive ?li ?ld)))
22    :ordering (and
23      (t1 < t2)))
```

Line 19 shows the aforementioned `:subtask` definition that allows for partially ordered tasks. The task definition contains IDs that can be used to define ordering constraints (line 22). They consist of a list of individual ordering constraints between subtasks. However, in the given example the resulting ordering is, again, a total order (and is just defined that way to demonstrate this kind of definition).

HPDDL uses the same keyword, but with a slightly different syntax so specify ordering constraints. The format omits the `and` and the `<` signs. We would argue that our notation is better readable to humans. As stated above, GTOHP cannot specify partial orders. ANML is primarily designed for temporal domains and uses a temporal syntax, e.g. `end(t1) < start(t2)`. SHOP2 and HTN-PDDL use a different approach to represent the task ordering. Instead of specifying individual ordering constraints, they require to specify the order as a single expression. This expression is a nested definition of the ordering, which can only contain two constructors: `ordered` and `unordered`. In SHOP2, e.g. `((:unordered (t1 t2) t3) t4)` corresponds to the ordering constraints `t1 < t2`, `t2 < t4`, and `t3 < t4`. Note that this construction cannot express all possible partially-ordered sets of tasks. Consider an ordering over five task identifiers `t1`, ..., `t5`, where `t1 < t4`, `t2 < t4`, `t2 < t5`, and `t3 < t5`. This ordering cannot be expressed with SHOP's nested ordered/unordered constructs. PDDL1.2 also uses this mode as a default, but does with an additional requirement also allow for an order specification as we and HPDDL do. Notably PDDL1.2 intertwines the definition of a method's subtasks and the definition of their order. The syntax of PDDL1.2 to specify the contents of methods and the order of tasks in them is somewhat convoluted and not easily readable. Thus, we have not adapted their syntax.

HATP uses a programming-language-style syntax for the encoding of methods. It further provides explicit means to determine the order in which groundings of methods should be explored during progression search. HATP's syntax for methods allows for specifying partial order, but its semantics is different from standard HTN planning. A HATP method containing partial order is interpreted as multiple totally-ordered method, one for each linearization of the given partial order. This allows for a more compact representation, but prohibits task interleaving.

HDDL – as HPDDL, SHOP2, HTN-PDDL, and ANML – only allow to specify a fixed *set* of ordering constraints. Notably, the HTN planner UMCP (Erol, Hendler, and Nau 1994) allows for *arbitrary formulae* that specify these orderings. E.g. they allow to specify an ordering $(t_1 \prec t_2) \implies (t_3 \prec t_4)$. We have not included such a generic means to formulate ordering constraints into HDDL as they do not seem to be used and supported by any current HTN planning system. In principle however, HDDL could be extended to support such complex ordering constraints.

A common feature of many HTN planning systems is the possibility of specifying state-based preconditions for methods as supported by the SHOP2 system. The feature is somewhat problematic. First, because it is (at least from our experience) usually used to guide the search and thus often breaks with the philosophy of PDDL to specify a model that does not include advice. The second problem is the way it is usually realized in the HTN planning systems: The systems introduce a new primitive task that holds the method's preconditions. It is added to the method and placed before all other tasks in the method's subtask network. Consider a totally ordered domain (i.e., the subtasks of all methods and the initial task network are totally ordered): here, the action is executed directly before the other subtasks of the method and the position where the preconditions are checked is fine. Now consider a partially ordered domain: here, the newly introduced action is not necessarily placed directly before the other subtasks, but we just know that it is placed somewhere before, i.e., the condition did hold at some point before the other tasks are executed, but may have changed meanwhile. However, though we are aware of these problems, the feature is often used and thus we integrated it and assume the standard semantics as given above.

The preconditions are defined as follows:

```
24  (:method m-already-there
25    :parameters (?l - location)
26    :task (get-to ?l)
27    :precondition (tAt ?l)
28    :subtasks ())
```

Here the method may be applied in a state where the transporter is already located at its destination. The given method has therefore no subtasks, but still has to assure that the transporter is at its destination.

Method preconditions are typically featured in languages expressing HTNs. HPDDL uses the same syntax we are proposing. GTOHP uses, as noted above, a separate `:constraints` section, where the method precondition has to be specified as a `before` constraint. This is (presumably) to allow for other state constraints later on. PDDL1.2 also features method preconditions, but they are specified as part of the task network. In ANML, there is no explicit means for writing down method preconditions, but they can be encoded into the state constraints allowed by ANML.

There is a strong contrast between what can be expressed in SHOP[3] and all other HTN formats. In SHOP, several methods for the same abstract task can be arranged in a single method declaration, each featuring its own method precondition. For the i^{th} method to be usable, it is not sufficient that its precondition is satisfied. In addition, the preconditions of all previous methods have to be not satisfied

[3]This potentially also applies to HTN-PDDL, as they use a similar syntax. Their description is unfortunately not explicit on the critical point in semantics (González-Ferrer, Fernández-Olivares, and Castillo 2009).

as well. Thus SHOP's method preconditions are in essence a chain of if-else constructs. This structure can be compiled into several individual methods with preconditions. In case one of the preconditions contains an existential quantifier (or in SHOP's case a free variable) this leads to universal quantified preconditions in the methods after it. Nevertheless we propose to drop the ability to use such if-else chains, most notably, since none of the newer languages supports it. Further, this kind of if-else is essentially a means to guide a depth-first search planner in an efficient way. Thus it does not constitute physics of the domain, but advice to the planner, which should not be part of the domain description language for a domain-independent planner.

In addition to method preconditions, HPDDL features method effects, which are modeled after SHOP2's assert and retract functionality. Method effects are executed in the state in which the method preconditions are evaluated. As far as we know, their formal semantics is not defined in any publication. We propose to drop this feature (at least for the given definition intended to be the core language). It is not commonly used and might be difficult to use for newcomers to HTN planning. Note that even without method effects in the description language, we can still simulate them with additional actions in the methods' definitions.

Sometimes it might be useful to define constraints in a method, e.g. on its variables or sorts. This is demonstrated in the following example where the transporter's source position must be different from its destination.

```
29   (:method m-direct
30     :parameters (?ls ?ld - location)
31     :task (get-to ?ld)
32     :constraints
33       (not (= ?li ?ld))
34     :subtasks (drive ?ls ?ld))
```

We are aware that PDDL allows for variable constraints in the precondition of actions. Due to consistency we also argue to allow this when method preconditions are specified. However, many HTN models are defined without methods that have preconditions and we think it not intuitive to specify a `precondition` section solely to define variable constraints. Furthermore, we think that other constraints apart from simple variable constraints might be added to the standard. These might, e.g., be constants that certain state features must hold between two tasks, or directly before some task. Therefore we integrated a constraint section to the method definition (line 32f) though our current definition only allows for equality and inequality constraints.

HPDDL places the variable constraints of a method into the method's preconditions. In addition to equality and inequality it features type constraints, where e.g. `(valuablePackage ?p)` is the constraint that `?p` belongs to the type `valuablePackage`. GTOHP allows for equality and inequality constraints that are also within the `:constraints` section, but are located in a separate `before` block. In SHOP's syntax, variable constraints have to be compiled into method preconditions referring to predicates for the individual types and an explicitly declared `equals` predicate. ANML also allows for variable constraints that can be declared freely inside a method.

We left the action definition unchanged compared to the PDDL standard we build on. Therefore we included only the following action into our example.

```
35   (:action drive
36     :parameters (?l1 ?l2 - location)
37     :precondition (and
38       (tAt ?l1)
39       (road ?l1 ?l2))
40     :effect (and
41       (not (tAt ?l1))
42       (tAt ?l2)))
43   ...)
```

The problem file is slightly adapted to represent the additional elements necessary for HTN panning (line 6).

```
1  (define (problem p)
2   (:domain transport)
3   (:objects
4    city-loc-0 city-loc-1 city-loc-2 -
        location
5    package-0 package-1 - package)
6   (:htn
7    :tasks (and
8     (deliver package-0 city-loc-0)
9     (deliver package-1 city-loc-2))
10   :ordering ()
11   :constraints ())
12  (:init
13   (road city-loc-0 city-loc-1)
14   (road city-loc-1 city-loc-0)
15   (road city-loc-1 city-loc-2)
16   (road city-loc-2 city-loc-1)
17   (at package-0 city-loc-1)
18   (at package-1 city-loc-1)))
```

The section starts with a keyword that specifies the problem class. In this example, it starts with `:htn` to define a standard HTN planning problem. However, there are several other problem classes in hierarchical planning. An example for such a class is HTN planning with task insertion, where the planner is allowed to insert tasks apart from the hierarchy. An overview of hierarchical problem classes can be found in the survey by Bercher, Alford, and Höller (2019). Some of the described problem classes are even syntactically equivalent to standard HTN planning problems and only differ in their solution criteria. By making the specification of the problem class explicit, extensions to the language can easily add new classes.

The definition of the initial task network is nested in this section. It has the same form as the methods' subtask networks. The other description languages for HTN planning also allow for a similar definition of the initial plan. Again, all of them use a slightly different syntax to describe them.

In the given example, the planning process is started with two *deliver* tasks, one for each package. These initial tasks are not ordered with respect to each other, i.e., their subtasks may be executed interleaved.

In the original PDDL standard, the domain designer has to specify a state-based goal. HTN planning problems do not require such a goal and thus often do not specify one. Therefore we made its definition optional.

4 Full Syntax Definition

We defined our syntax as close as possible to the STRIPS part (i.e., language level 1) of the PDDL 2.1 language definition of Fox and Long (2003). Wide parts of the following definition are **identical** to their definition. Changes and extensions are discussed in the following.

The domain definition has been extended by definitions for compound tasks (line 6) and methods (line 7).

```
1 <domain> ::= (define (domain <name>)
2     [<require-def>]
3     [<types-def>]:typing
4     [<constants-def>]
5     [<predicates-def>]
6     <comp-task-def>*
7     <method-def>*
8     <action-def>*)
```

The definition of the basic elements is nearly unchanged.

```
9  <require-def> ::=
       (:requirements <require-key>+)
10 <require-key> ::= ...
11 <types-def> ::= (:types <types>+)
12 <types> ::= <typed list (name)>
       | <base-type>
13 <base-type> ::= <name>
14 <constants-def> ::=
       (:constants <typed list (name)>)
15 <predicates-def> ::=
       (:predicates <atomic-formula-skeleton>+)
16 <atomic-formula-skeleton> ::=
       (<predicate> <typed list (variable)>)
17 <predicate> ::= <name>
18 <variable> ::= ?<name>
19 <typed list (x)> ::= x+ - <type>
       [<typed list (x)>]
20 <primitive-type> ::= <name>
21 <type> ::= (either <primitive-type>+)
22 <type> ::= <primitive-type>
```

The only change concerns the definition of `<types-def>` (lines 11 and 13) in combination with the definition of `<typed list (name)>` (line 19). In the PDDL2.1 standard, this can be realized by a list of names, e.g. in an untyped way. Our intention was to enforce a typed model and therefore allow for untyped elements only in the type definition. There, it is necessary to define the base type(s). In every other definition that includes `<typed list (name)>` (e.g. parameter and constant definitions), we wanted to enforce a typed list.

Abstract tasks are defined similar to actions.

```
23 <comp-task-def> ::= (:task <task-def>)
24 <task-def> ::= <task-symbol>
       :parameters (<typed list (variable)>)
25 <task-symbol> ::= <name>
```

In a standard HTN setting, methods consist of a parameter list (line 27), the abstract task they decompose (line 28), and the resulting task network (line 30). The parameters of a method are supposed to include all parameters of the abstract task that it decomposes and those of the tasks in its network of subtasks.

By setting the `:htn-method-prec` requirement, one might use method preconditions (line 29).

```
26 <method-def> ::= (:method <name>
27     :parameters (<typed list (variable)>)
28     :task (<task-symbol> <term>*)
29     [:precondition <gd>]:htn-method-prec
30     <tasknetwork-def>)
```

The definition of task networks is used in method definitions as well as in the problem definition to define the initial task network. It contains the definition of sub-tasks (line 32), ordering constraints (line 33), and variable constraints (line 34) between any method parameters.

When the key `:ordered-subtasks` is used, the network is regarded to be totally ordered. In the other cases, ordering relations may be defined explicitly. This is done by including ids into the task definition that can then be referenced in the ordering definition.

```
31 <tasknetwork-def> ::=
32     [:[ordered-][sub]tasks
           <subtask-defs>]
33     [:order[ing] <ordering-defs>]
34     [:constraints <constraint-defs>]
```

We use the same syntax definition for method subnetworks and the initial task network. Here, the keyword `subtasks` would seem odd. Therefore the syntax also allows for the keys `tasks` and `ordered-tasks` (line 32) that are supported to be used in the initial task network.

The subtask definition may contain one or more subtasks. A single task consists of a task symbol and a list of parameters. In case of a method's subnetwork, these parameters have to be included in the method's parameters, in case of the initial task network, they have to be defined as constants in s_0 or in a dedicated parameter list (see definition of the initial task network, line 82). The tasks may start with an id that can be used to define ordering constraints.

```
35 <subtask-defs> ::= () | <subtask-def>
       | (and <subtask-def>+)
36 <subtask-def> ::= (<task-symbol> <term>*)
       | (<subtask-id> (<task-symbol> <term>*))
37 <subtask-id> ::= <name>
```

The ordering constraints are defined via the task ids. They have to induce a partial order.

```
38 <ordering-defs> ::= () | <ordering-def>
       | (and <ordering-def>+)
39 <ordering-def> ::=
       (<subtask-id> "<" <subtask-id>)
```

So far we only included variable constraints into the constant section, but the definition might be extended in further language levels, of course.

```
40 <constraint-defs> ::= () | <constraint-def>
       | (and <constraint-def>+)
41 <constraint-def> ::= ()
       | (not (= <term> <term>))
       | (= <term> <term>)
```

The original action definition of PDDL has been split to reuse its body in the task definition.

```
42 <action-def> ::= (:action <task-def>
43     [:precondition <gd>]
44     [:effects <effect>])
```

We restricted the definition of preconditions and effects to level 1, i.e., the STRIPS part of the overall language.

```
45 <gd> ::= ()
46 <gd> ::= <atomic formula (term)>
47 <gd> ::=:negative-preconditions <literal (term)>
48 <gd> ::= (and <gd>*)
49 <gd> ::=:disjunctive-preconditions (or <gd>*)
50 <gd> ::=:disjunctive-preconditions (not <gd>)
51 <gd> ::=:disjunctive-preconditions (imply <gd> <gd>)
52 <gd> ::=:existential-preconditions
       (exists (<typed list (variable)>*) <gd>)
53 <gd> ::=:universal-preconditions
       (forall (<typed list (variable)>*) <gd>)
54 <gd> ::= (= <term> <term>)
55 <literal (t)> ::= <atomic formula(t)>
56 <literal (t)> ::= (not <atomic formula(t)>)
57 <atomic formula(t)> ::= (<predicate> t*)
58 <term> ::= <name>
59 <term> ::= <variable>
60 <effect> ::= ()
61 <effect> ::= (and <c-effect>*)
62 <effect> ::= <c-effect>
63 <c-effect> ::=:conditional-effects
       (forall (<variable>*) <effect>)
64 <c-effect> ::=:conditional-effects
       (when <gd> <cond-effect>)
65 <c-effect> ::= <p-effect>
66 <p-effect> ::= (not <atomic formula(term)>)
67 <p-effect> ::= <atomic formula(term)>
68 <cond-effect> ::= (and <p-effect>*)
69 <cond-effect> ::= <p-effect>
```

The problem definition includes as additional element the initial task network (line 74). Since a state-based goal definition is often not included in HTN planning, we made it optional (line 76).

```
70 <problem> ::= (define (problem <name>)
71     (:domain <name>)
72     [<require-def>]
73     [<p-object-declaration>]
74     [<p-htn>]
75     <p-init>
76     [<p-goal>])
77 <p-object-declaration> ::=
       (:objects <typed list (name)>)
78 <p-init> ::= (:init <init-el>*)
79 <init-el> ::= <literal (name)>
80 <p-goal> ::= (:goal <gd>)
```

The initial task network contains the definition of the problem class (line 81). In this first definition we only included standard HTN planning.

```
81 <p-htn> ::= (<p-class>
82     [:parameters (<typed list (variable)>)]
83     <tasknetwork-def>)
84 <p-class> ::= :htn
```

Our overall definition includes two new requirement flags:

- `:htn` requires the applied system needs to support HTN planning at all, so this can be seen as the basic requirement for the language defined here.
- `:htn-method-prec` requires the applied system needs to support method preconditions.

5 Discussion

We consider the language proposed in this paper as a first step towards a standardized language for hierarchical planning problems and hope that it helps to find a minimal set of features supported by the diverse systems. However, this basic feature set as well as many design options are still open and have to be discussed in the research community.

First of all, we think it is important to remain as close as possible to PDDL and to reuse its features to allow domain modelers to create both hierarchical and non-hierarchical problems with minimal learning effort. Then, we must decide which features have to be at the core of the language, and which ones are secondary and possibly could be ignored. This is especially important to establish a competition to compare the performance of different systems (see the proposal by Behnke et al. (2019a)).

A feature that was present in the early HTN formalisms (see e.g. the formalism by Erol, Hendler, and Nau, (1994)) is the possibility to define more elaborated constraints in task networks. Recent work in hierarchical planning was not based on such a rich definition language, but on rather minimalistic formalisms like the one introduced by Geier and Bercher, (2011). In this first definition we only included the very basic constraints: ordering constraints, variable constraints, and method preconditions. However, we think that a constraint set as given in PDDL3 might be a nice extension beneficial for domain designers. When the community wants to foster application in real world domains, it may be necessary to integrate support for numbers and time into the planning systems. Since our definition builds upon the PDDL2.1, at least the extension of the syntax in that direction could easily be done. Another possible extension is the support for preconditions and effects in the definition of abstract tasks (see Bercher et al. (2016) for an overview of that feature).

Beside new features, it might be interesting to include new problem classes like *HTN planning with task insertion*, *decompositional planning*, or *HGN planning*, which comes with the ability to decompose not tasks, but also goals (Shivashankar et al. 2012) and that even has been combined with task decomposition (Alford et al. 2016b).

6 Conclusion

We propose a common description language for hierarchical planning problems. We argue that the core feature set underlying many hierarchical planners from the last years is that of HTN planning and introduced its elements as an extension of PDDL. We defined the language in a way that can easily be extended by further features as has been done in PDDL. We introduced our novel language elements "by example" and discussed our design choices, the syntax used in related work, and the proposed meaning. We gave a full syntax definition afterwards and discussed the extensions and changes to the PDDL standard. We hope that a common input language may foster the cooperation between groups working in hierarchical planning, the comparison of different hierarchical planning systems, and the application on real problems, because it enables an easy exchange of the planning system used for a given problem.

Acknowledgements

This work was partly funded by the technology transfer project "Do it yourself, but not alone: Companion-Technology for DIY support" of the SFB/TRR 62 funded by the German Research Foundation (DFG). The industrial project partner is the Corporate Research Sector of the Robert Bosch GmbH.

References

Alford, R.; Bercher, P.; and Aha, D. 2015a. Tight bounds for HTN planning. In *Proc. of the ICAPS*.

Alford, R.; Bercher, P.; and Aha, D. 2015b. Tight bounds for HTN planning with task insertion. In *Proc. of the IJCAI*.

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016a. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proc. of the ICAPS*.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016b. Hierarchical planning: Relating task and goal decomposition with task sharing. In *Proc. of the IJCAI*.

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proc. of the IJCAI*.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11(4):625–656.

Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2019a. Hierarchical planning in the IPC. In *Proc. of the Workshop on the IPC (WIPC)*.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2019b. More succinct grounding of HTN planning problems – Preliminary results. In *Proc. of the ICAPS Workshop on Hierarchical Planning*.

Behnke, G.; Höller, D.; and Biundo, S. 2018a. totSAT – Totally-ordered hierarchical planning through SAT. In *Proc. of the AAAI*.

Behnke, G.; Höller, D.; and Biundo, S. 2018b. Tracking branches in trees – A propositional encoding for solving partially-ordered HTN planning problems. In *Proc. of the ICTAI*.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proc. of the AAAI*.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proc. of the IJCAI*.

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – One abstract idea, many concrete realizations. In *Proc. of the 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 *Proc. of the ECAI*.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In *Proc. of the IJCAI*.

Bit-Monnot, A.; Smith, D. E.; and Do, M. 2016. Delete-free reachability analysis for temporal and hierarchical planning. In *Proc. of the ECAI*.

de Silva, L.; Lallement, R.; and Alami, R. 2015. The HATP hierarchical planner: Formalisation and an initial study of its usability and practicality. In *Proc. of the IROS*, 6465–6472.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. of AIPS*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of the IJCAI*.

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

González-Ferrer, A.; Fernández-Olivares, J.; and Castillo, L. 2009. JABBAH: A java application framework for the translation between business process models and HTN. In *Proc. of the Int. Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proc. of the ICAPS*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019. On guiding search in HTN planning with classical planning heuristics. In *Proc. of the IJCAI*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

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

Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN planning domain. *International Journal on Artificial Intelligence Tools* 26(5):1–24.

Schreiber, D.; Balyo, T.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proc. of the ICAPS*.

Shivashankar, V.; Alford, R.; and Aha, D. W. 2017. Incorporating domain-independent planning heuristics in hierarchical planning. In *Proc. of the AAAI*.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. of the AAMAS*.

Smith, D.; Frank, J.; and Cushing, W. 2008. The ANML language. In *Proc. of the Workshop on KEPS*.

HTN Planning with Semantic Attachments

Maurício Cecílio Magnaguagno, Felipe Meneguzzi
School of Computer Science (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre - RS, Brazil
mauricio.magnaguagno@acad.pucrs.br
felipe.meneguzzi@pucrs.br

Abstract

Hierarchical Task Networks (HTN) generate plans using a decomposition process guided by extra domain knowledge to guide search towards a planning task. While many HTN planners can make calls to external processes (e.g. to a simulator interface) during the decomposition process, this is a computationally expensive process, so planner implementations often use such calls in an ad-hoc way using very specialized domain knowledge to limit the number of calls. Conversely, the few classical planners that are capable of using external calls (often called *semantic attachments*) during planning do so in much more limited ways by generating a fixed number of ground operators at problem grounding time. In this paper we develop the notion of *semantic attachments* for HTN planning using semi co-routines, allowing such procedurally defined predicates to link the planning process to custom unifications outside of the planner. The resulting planner can then use such co-routines as part of its backtracking mechanism to search through parallel dimensions of the state-space (e.g. through numeric variables). We show empirically that our planner outperforms the state-of-the-art numeric planners in a number of domains using minimal extra domain knowledge.

Introduction

Planning in domains that require numerical variables, for example, to drive robots in the physical world, must represent and search through a space defined by real-valued functions with a potentially infinite domain, range, or both. This type of numeric planning problem poses challenges in two ways. First, the description formalisms (Fox and Long 2003) might not make it easy to express the numeric functions and its variables, resulting in a description process that is time consuming and error-prone for real-world domains (Strobel and Kirsch 2014). Second, the planners that try to solve such numeric problems must find efficient strategies to find solutions through this type of state-space. Previous work on formalisms for domains with numeric values developed the Semantic Attachment (SA) construct (Dornhege et al. 2009) in classical planning. Semantic attachments were coined by (Weyhrauch 1981) to describe the attachment of an interpretation to a predicate symbol using an external procedure. Such construct allows the planner to reason about fluents where numeric values come from externally defined functions. In this paper, we extend the basic notion of semantic attachment for HTN planning by defining the semantics of the functions used as semantic attachments in a way that allows the HTN search and backtracking mechanism to be substantially more efficient. Our current approach focused on depth-first search HTN implementation without heuristic guidance, with free variables expected to be fully-ground before task decomposition continues.

Most planners are limited to purely symbolic operations, lacking structures to optimize usage of continuous resources involving numeric values (Gerevini, Saetti, and Serina 2008). Floating point numeric values, unlike discrete logical symbols, have an infinite domain and are harder to compare as one must consider rounding errors. One could overcome such errors with delta comparisons, but this solution becomes cumbersome as objects are represented by several numeric values which must be handled and compared as one, such as points or polygons. Planning descriptions usually simplify such complex objects to symbolic values (e.g. *p25* or *poly2*) that are easier to compare. Detailed numeric values are ignored during planning or left to be decided later, which may force replanning (Şucan and Kavraki 2011). Instead of simplifying the description or doing multiple comparisons in the description itself, our goal is to exploit external formalisms orthogonal to the symbolic description. To achieve that we build a mapping from symbols to objects generated as we query semantic attachments. Semantic attachments have already been used in classical planning (Dornhege et al. 2009) to unify values just like predicates, and their main advantage is that new users do not need to discern between them and common predicates. Thus, we extend classical HTN planning algorithms and their formalism to support semantic attachment queries. While external function calls map to functions defined outside the HTN description, we implement SAs as semi co-routines (Dahl, Dijkstra, and Hoare 1972), subroutines that suspend and resume their state, to iterate across zero or more values provided one at a time by an external implementation, mitigating the potentially infinite range of the external function.

Our contributions are threefold. First, we introduce SAs for HTN planning as a mechanism to describe and evaluate external predicates at execution time. Second, we introduce a symbol-object table to improve the readability of symbolic descriptions and the plans generated, while making it easier to handle external objects and structures. Finally, we empirically compare the resulting HTN planner with a modern classical planner (Ilghami and Nau 2003) in a number of mixed symbolic/numeric domains showing substantial gains in speed with minimal domain knowledge.

Background

Classical Planning

Classical planning algorithms must find plans that transform properties of the world from an initial configuration to a goal configuration. Each property is a logical predicate, a tuple with a name and terms that relate to objects of the world. A world configuration is a set of such tuples, which is called a state. To modify a state one must apply an operator, which must fulfill certain predicates at the current state, as preconditions, to add and remove predicates, the effects. Each operator applied creates a new intermediate state. The set of predicates and operators represent the domain, while each group of objects, initial and goal states represent a problem within this domain. In order to achieve the goal state the operators are used as rules to determine in which order they can be applied based on their preconditions and effects. To generalize the operators and simplify description one can use free variables to be replaced by objects available, a process called grounding. Once a state that satisfies the goal is reached, the sequence of ground operators is the plan (Nebel 2000). A plan is optimal, iff it achieves the best possible quality in some criteria, such as number of operators, time or effort to execute; or satisficing if it reaches the goal without optimizing any metrics. PDDL (McDermott et al. 1998) is the standard description language to describe domains and problems, with features added through requirements that must be supported by the planner. Among such features are numeric-valued fluents to express numeric assignments and updates to the domain, as well as events and processes to express effects that occur in parallel with the operators in a single instant or during a time interval.

Hierarchical Task Networks

Hierarchical planning shifts the focus from goal states to tasks to exploit human knowledge about problem decomposition using a hierarchy of domain knowledge recipes as part of the domain description (Nau et al. 1999). This hierarchy is composed of primitive tasks that map to operators and non-primitive tasks, which are further refined into sub-tasks using *methods*. The decomposition process is repeated until only primitive-tasks mapping to operators remain, which results in the plan itself. The goal is implicitly achieved by the plan obtained from the decomposition process. If no decomposition is possible, the task fails and a new expansion is considered one level up in the hierarchy, until there are no more possible expansions for the root task, only then a task decomposition is considered unachievable. Unlike classical planning, hierarchical planning only considers tasks obtained from the decomposition process to solve the problem, which both limits the ability to solve problems and improves execution time by evaluating a smaller number of operators. The HTN planning description is more complex than equivalent classical planning descriptions, since it includes domain knowledge with potentially recursive tasks, being able to solve more problems than classical planning.

Symbolic-Geometric Planning

Classical planners with heuristic functions can solve problems mixing symbolic and numeric values efficiently using a process of discretization. A discretization process converts continuous values into sets of discrete symbols at often predefined granularity levels that vary between different domains. However, if the discretization process is not possible, one must use a planner that also supports numeric features, which requires another heuristic function, description language and usually more computing power due to the number of states generated by numeric features. Numeric features are especially important in domains where one cannot discretize the representation, they usually appear in geometric or physics subproblems of a domain and cannot be avoided during planning. Unlike symbolic approaches where literals are compared for equality during precondition evaluation, numeric value comparison is non-trivial. To avoid doing such comparison for every numeric value the user is left responsible for explicitly defining when one must consider rounding errors, which impacts description time and complexity. For complex object instances (in the object-oriented programming sense), such as polygons that are made of point instances, comparison details in the description are error-prone. Details such as the order of polygon points and floating point errors in their coordinates are usually irrelevant for the planner and the domain designer and should not be part of the domain description as they are part of a low-level specification.

Such low-level specifications can be implemented by external function calls to improve what can be expressed and computed by a HTN planner. Such functions come with disadvantages, as they are not expected to keep an external state, returning a single value solely based on the provided parameters. While HTN planners can abstract away the numeric details via external function calls, there are limitations to this approach if a particular function is used in a decomposition tree where it is expected to backtrack and try new values from the function call (i.e. if the function is meant to be used to generate multiple terms as part of the search strategy). An external function must return a list of values to account for all possible decompositions so the planner tries one at a time until one succeeds. Generating a complete list is too costly when compared to computing a single value, as the first value could be enough to find a feasible plan. A semantic attachment, on the other hand, acts as an external predicate that unifies with one possible set of values at a time, rather than storing a complete list of possible sets of values to be stored in the state structure. This implementation saves time and memory during planning, as only backtracking causes the external co-routine to resume generating new unifications until a plan (or a certain amount of plans) is found. Each SA acts as a black box that simulates part of the environment encoding the results in state variables that are often orthogonal to other predicates (Francès et al. 2017).

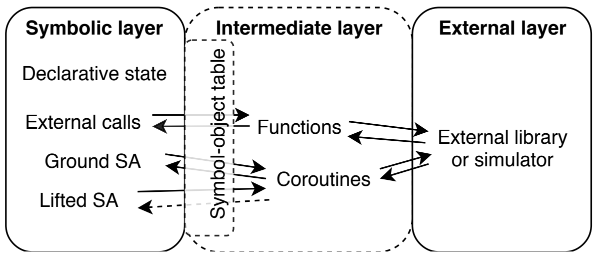


Figure 1: Symbolic and external layers share information through an intermediate layer that maps representations and calls between them.

While common predicates are stored in a state structure, SAs are computed at execution time by co-routines. With a state that is not only declarative, with parts being procedurally computed, it is possible to minimize memory usage and delegate complex state-based operations to external methods otherwise incompatible or too complex for current planning description languages and planners that require grounding.

We abstract away the numeric parts of the planning process encoded through SAs in a layer between the symbolic planner and external libraries. We leverage the abstract architecture of Figure 1 with three layers inspired by the work of de Silva and Meneguzzi (2015). In the symbolic layer we manipulate an anchor symbol as a term, such as *polygon1*, while in the external layer we manipulate a *Polygon instance with N points* as a geometric object based on what the selected external library specifies. With this approach we avoid complex representations in the symbolic layer. Instances created by the external layer that must be exposed to the symbolic layer are compared with stored object instances to reuse a previously defined symbol or create a new one, i.e. always represent position ⟨2,5⟩ with *p1*. This process makes symbol comparison work in the planning layer even for symbols related to complex external objects. The symbol-object table is also used to transform symbols into usable object instances by external function calls and SAs. Such table is global and consistent during the planning process, as each unique symbol will map the same internal object, even if such symbol is discarded in one decomposition branch. Once operations are finished in the external layer the process happens in reverse order, objects are transformed back into symbols that are exposed by free variables. The intermediate layer acts as the foreign function interface between the two layers, and can be modified to accommodate another external library without modifications to the symbolic description.

SAs can work as interpreted predicates (Mohr et al. 2018), evaluating the truth value of a predicate procedurally, and also grounding free variables. SAs are currently limited to be used as method preconditions, which must not contain disjunctions. As only conjunctions and negations are allowed, one can reorder the preconditions during the compilation phase to improve execution time, removing the burden of the domain designer to optimize a mostly declarative description by hand, based on how free variables are used as SA

Listing 1: Abstract method with SAs among preconditions.

```
(:attachments (sa1 ?a ?b) (sa2 ?a ?b))
(:method (m ?t1 ?t2)
  label
  (; preconditions
    (call != ?t1 ?t2) ; no dependencies
    (call != ?fv1 ?fv2) ; ?fv1 and ?fv2 dependencies
    (sa1 ?t1 ?fv1)    ; no dependencies, ground ?fv1
    (pre1 ?t1 ?t2)    ; no dependencies
    (sa2 ?fv1 ?fv2)   ; ?fv1 dependency, ground ?fv2
    (pre2 ?fv3 ?fv1)  ; ?fv1 dependency, ground ?fv3
  )
  (; subtasks
    (subtask ?t1 ?t2 ?fv1 ?fv2)
  )
)
```

terms. Each free variable creates a dependency between the first predicate or SA that contains such variable as a term and the next predicates or SAs that contain the same term. The first predicate or SA is responsible for grounding such variable while the next predicates or SAs only verify if the previously ground value matches with the current state. Predicates have priority over SAs to ground free variables, as the possible values are obtained from the current (finite) state, while SAs may cover a possibly infinite number of values. Consider the abstract method example of Listing 1, with two SAs among preconditions, *sa1* and *sa2*. The compiled output shown in Algorithm 1 has both SAs evaluated after common predicates, while function calls happen before or after each SA, based on which variables are ground at that point. In Line 4 the free variables *fv1* and *fv3* have a ground value that can only be read and not modified by other predicates or SAs. In Line 7 every variable is ground and the second function call can be evaluated.

Algorithm 1 Compilation phase may reorder preconditions to optimize execution time.

```
1: function M($t1, t2$)
2:   if t1 $\neq$ t2
3:     for each fv1, fv3; state $\subset$ {⟨pre1,t1,t2⟩,⟨pre2,fv3,fv1⟩} do
4:       for each sa1(t1, fv1) do
5:         free variable fv2
6:         for each sa2(fv1, fv2) do
7:           if fv1 $\neq$ fv2
8:             decompose([⟨subtask, t1, t2, fv1, fv2⟩])
```

The other limitation of current SA co-routines is that they must unify with a valid value within their internal iterations or have a stop condition, otherwise the HTN process will keep backtracking and evaluating the SA seeking new values and never returning failure. Due to the implementation support of arbitrary-precision arithmetic and accessing data from real-world streams of data/events (which are always new and potentially infinite) a valid value may never be found, and we expect the domain designer to implement mechanisms to limit the maximum number of times a SA might try to evaluate a call (i.e. to have finite stop conditions). This maximum number of tries can be implemented as a counter in the internal state of a SA, which is mostly used to mark values provided to the HTN to avoid repetition, but may achieve side-effects in external structures. The amount of side-effects in both external functions calls and SAs increase the complexity of correctness proofs and the ability to inspect and debug domain descriptions.

Examples

Discrete distance between objects

A common problem when moving in dynamic and continuous environments is to check for object collisions, as agents and objects do not move across tiles in a grid. One solution is to calculate the distance between both objects centroid positions and verify if this value is in a safe margin before considering which action to take. To avoid the many geometric elements involved in this process we can map centroid position symbols to coordinate instances and only check the symbol returned from the symbol-object table, ignoring specific numeric details and comparing a symbol to verify if objects are near enough to collide. This process is illustrated in Figure 2, in which p_0 and p_1 are centroid position symbols that match symbols S_0 and S_1 in the symbol-object table, which maps their value to point objects O_0 and O_1. Such internal objects are used to compute *distance* and return a symbolic distance in situations where the actual numeric value is unnecessary.

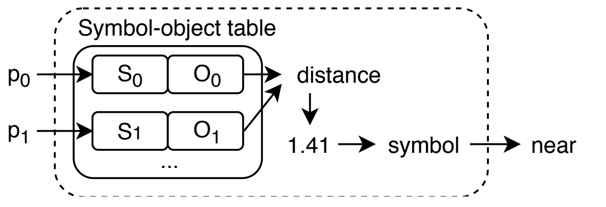


Figure 2: The symbol to object table maps symbols to object-oriented programming instances to hide procedural logic from the symbolic layer.

An iterator for HTN

In order to find a correct number to match a spatial or temporal constraint one may want to describe the relevant interval and precision to limit the amount of possibilities without having to discretely add each value to the state. Planning descriptions usually do not contain information about numeric intervals and precision, and if there is a way to add such information it is through the planner itself, as global definitions applied to all numeric functions, i.e. timestep, mantissa and exponent digits of DiNo (Piotrowski et al. 2016). The STEP SA described in Algorithm 2 addresses this problem, unifying t with one number at time inside the given interval with an ϵ step.

Lazy adjacency evaluation

To avoid having complex effects in the move operators one must not update adjacencies between planning objects during the planning process. Instead one must update only the object position, deleting from the old position and adding the new position. Such positions come from a partitioned space, previously defined by the user. The positions and their adjacencies are either used to generate and store ground operators or stored as part of the state. To avoid both one could implement adjacency as a co-routine while hiding numeric properties of objects, such as position. Algorithm 3 shows the main two cases that appear in planning descriptions. In the first case both symbols are ground, and the co-routine resumes when both objects are adjacent, doing nothing otherwise, failing the precondition. In the second case $s2$, the second symbol, is free to be unified using $s1$, the first symbol, and a set of directions D to yield new positions to replace $s2$ pointer with a valid position, one at a time. In other terms, this co-routine either checks whether $s2$ is adjacent to $s1$ or tries to find a value adjacent to $s1$ binding it to $s2$ if such value exists.

Algorithm 2 The STEP SA replaces the pointer of *t* with a numeric symbol before resuming control to the HTN.

```
1: function STEP($t, min = 0, max = \infty, \epsilon = 1$)
2:   for i ← $min$ to $max$ step $\epsilon$ do
3:     $t$ ← symbol(i)
4:     yield                                    ▷ Resume HTN
```

Algorithm 3 This ADJACENT SA implementation can either check if two symbols map to adjacent positions or generate new positions and their symbols to unify $s2$.

```
 1: D ← {(-1,-1),(0,-1),(1,-1),(-1,0),(1,0),(-1,1),(0,1),(1,1)}
 2: function ADJACENT($s1, s2$)
 3:   $s1$ ← object($s1$)
 4:   if $s2$ is ground
 5:     $s2$ ← object($s2$)
 6:     if |x($s1$) - x($s2$)| ≤ 1 ∧ | y($s1$) - y($s2$)| ≤ 1
 7:       yield
 8:   else if $s2$ is free
 9:     for each (x, y) ∈ D do
10:       nx ← x + x($s1$); ny ← y + y($s1$)
11:       if 0 ≤ nx < WIDTH ∧ 0 ≤ ny < HEIGHT
12:         $s2$ ← symbol(⟨nx, ny⟩)
13:         yield
```

Domains and Experiments

We conducted emprirical tests in a machine with Dual 6-core Xeon CPUs @2GHz / 48GB memory, repeating experiments three times to obtain an average. The results show a substantial speedup over the original classical description from ENHSP (Scala et al. 2016) with more complex descriptions. Our HTN implementation is available at `github.com/Maumagnaguagno/HyperTensioN_U`.

Plant Watering / Gardening

In the Plant Watering domain (Frances and Geffner 2015) one or more agents move in a 2D grid-based scenario to reach taps to obtain certain amounts of water and pour water in plants spread across the grid. Each agent can carry up to a certain amount of water and each plant requires a certain amount of water to be poured. Many state variables can be represented as numeric fluents, such as the

Listing 2: Excerpt of the Plant Watering HTN domain used as input to our implementation, the ADJACENT SA is described separately.

```
(:attachments (adjacent ?x ?y ?nx ?ny ?gx ?gy))
(:method (travel ?a ?gx ?gy)
  base
  (; preconditions
    (call = (call function (x ?a)) ?gx)
    (call = (call function (y ?a)) ?gy)
  )
  () ; empty subtasks
  keep_moving
  (; preconditions
    (adjacent (call function (x ?a))
      (call function (y ?a)) ?nx ?ny ?gx ?gy)
  )
  (; subtasks
    (!move ?a ?nx ?ny)
    (travel ?a ?gx ?gy)
  )
)
```

coordinates of each agent, tap and plant, the amount of water to be poured and being carried by each agent, and the limits of how much water can be carried and the size of the grid. There are two common problems in this scenario, the first is to travel to either a tap or a plant, the second is the top level strategy. To avoid considering multiple paths in the decomposition process one can try to move straight to the goal first, and only to the goal in scenarios without obstacles, which simplifies the travel method. To achieve this straightforward movement we modify the ADJACENT SA to consider the goal position also, using an implementation of Algorithm 4. The top level strategy may consider which plant is closer to a tap or closer to an agent, how much water an agent can carry and so on. The simpler top level strategy is to verify how much water must be poured to a plant, travel to a tap, load water, travel to the previously selected plant and pour all the water loaded. Repeating this process until every plant has enough water poured. The travel method description using our modified JSHOP input language is shown in Listing 2 and compiled to Algorithm 5. We compare with the fastest satisficing configurations of ENHSP (`sat` and `c_sat`) in Figure 3, which shows that our approach is faster with execution times constantly below 0.01s, with both planners obtaining non-step-optimal plans.

Algorithm 4 In this goal-driven ADJACENT SA the positions are coordinate pairs, and two variables must be unified to a closer to the goal position in an obstacle-free scenario.

```
1: function ADJACENT($x, y, nx, ny, gx, gy$)
2:     $x \leftarrow$ numeric($x$); $y \leftarrow$ numeric($y$)
3:     $gx \leftarrow$ numeric($gx$); $gy \leftarrow$ numeric($gy$)
4:     ▷ compare returns -1, 0, 1 for $<, =, >$, respectively
5:     $nx \leftarrow$ symbol($x$ + compare($gx$, $x$))
6:     $ny \leftarrow$ symbol($y$ + compare($gy$, $y$))
7:     yield
```

Car Linear

In the Car Linear domain (Bryce et al. 2015) the goal is to control the acceleration of a car, which has a minimum and maximum speed, without external forces applied, only moving through one axis to reach its destination, and requiring a small speed to safely stop. The idea is to propagate process effects to state functions, in this case acceleration to speed and speed to position, while being constrained to an acceptable speed and acceleration. The planner must decide when and for how long to increase or decrease acceleration, therefore becoming a temporal planning problem. We use a STEP SA to iterate over the time variable and propagate temporal effects and constraints, i.e. speed at time *t*. We compare the execution time of our approach with ENHSP with `aibr`, ENHSP main configuration for planning with autonomous processes, in Table 1. There is no comparison with a native HTN approach, as one would have to add a discrete finite set of time predicates (e.g. ⟨*time 0*⟩) to the initial state description to be selected as time points during planning.

Algorithm 5 Compiled output of the Plant Watering HTN domain excerpt from Listing 2.

```
1: function TRAVEL($a, gx, gy$)
2:     if x($a$) = $gx \land$ y($a$) = $gy$
3:         decompose([])
4:     free variables nx, ny
5:     for each adjacent(x($a$), y($a$), nx, ny, $gx$, $gy$) do
6:         decompose([⟨move, $a$, nx, ny⟩, ⟨travel, $a$, $gx$, $gy$⟩])
```

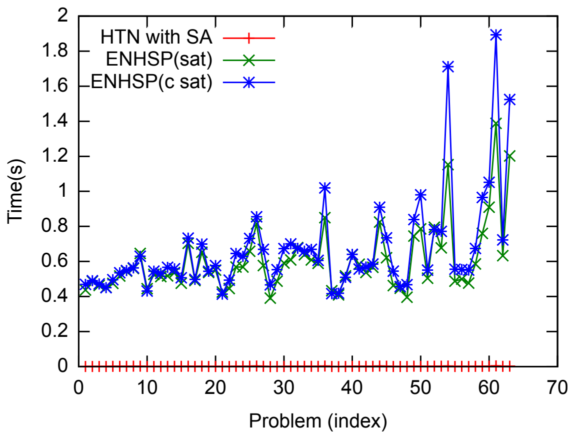


Figure 3: Time in seconds to solve Plant Watering problems.

Problem	1	2	3	4	5	6	7	8	9
ENHSP (aibr)	0.461	0.462	0.427	0.461	0.475	0.474	0.443	0.466	58.256
HTN with SA	0.015	0.015	0.015	0.015	0.015	0.015	0.015	0.015	03.920

Table 1: Time in seconds to solve Car Linear problems.

Bitangent movement

For an agent to move in a continuous space it is common practice to simplify the environment to simpler geometric shapes for faster collision evaluation. One possible simplification is to find a circle or sphere that contains each obstacle and use this new shape to evaluate paths. In this context the best path is the one with the shortest lines between initial position and goal, considering bitangent lines between each simplified obstacle plus the amount of arc traversed on their borders, also know as Dubins path (Dubins 1957). One possible approach for a satisficing plan is to move straight to the goal or to the closest obstacle to the goal and repeat the process. A precondition to such movement is to have a visible target, without any other obstacle between the current and target positions. A second consideration is the entrance direction, as clock or counter-clockwise, to avoid cusped edges. Cusped edges are not part of optimal realistic paths, as the moving agent would have to turn around over a single point instead of changing its direction a few degrees to either side. For the problem defined in Figure 4 the possible paths from point *i* to *g* are ACG, ADH, BEG, BFH.

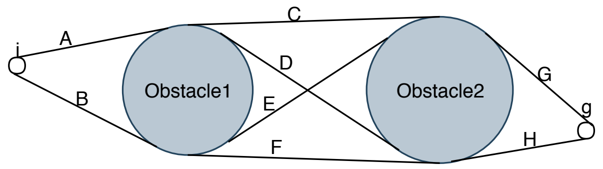


Figure 4: Possible bitangent paths from *i* to *g* with two circular obstacles.

Two possible approaches can be taken to solve the search over circular obstacles using bitangents. One is to rely on an external solver to compute the entire path, a motion planner, which could happen during or after HTN decomposition has taken place. When done during HTN decomposition, as seen in Listing 3, one must call the SEARCH-CIRCULAR function and consume the resulting steps of the plan stored in the intermediate layer, not knowing about how close to the goal it could reach in case of failure. When done after HTN decomposition, one must replace certain dummy operators of the HTN plan and replan in case of failure. The second approach is to rely on parts of the external search, namely the VISIBLE function and CLOSEST SA, to describe continuous search to the HTN planner. The VISIBLE function returns true if from a point on a circle one can see the goal, false otherwise. The CLOSEST SA generates unifications from a circle with an entrance direction to a point in another circle with an exit direction, new points closer to the goal are generated first. Differently from external search, one can deal with failure at any moment, while being able to modify behavior with the same external parts, such as the initial direction the search starts with. Another advantage over the original solution is the ability to ask for N plans, which forces the HTN to backtrack after each plan is found and explore a different path until the amount of plans found equals N or the HTN planner fails to backtrack. A description of such approach is show in Listing 4. The execution time variance between the solutions is not as important as their different approaches to obtain a result, from an external greedy best-first search to a HTN depth-first search. The external search also computes bitangents on demand, as bitangent precomputation takes a significant amount of time for many obstacles.

Conclusion

We developed a notion of semantic attachments for HTN planners that not only allows a domain expert to easily define external numerical functions for real-world domains, but also provides substantial improvements on planning speed over comparable classical planning approaches. The use of semantic attachments improves the planning speed as one can express a potentially infinite state representation with procedures that can be exploited by a strategy described as HTN tasks. As only semantic attachments present in the path decomposed during planning are evaluated, a smaller amount of time is required when compared with approaches that precompute every possible value during operator grounding. Our description language is arguably more readable than the commonly used strategy of developing a domain specific planner with customized heuristics. Specifically, we allow designers to easily define external functions in a way that is readable within the domain knowledge encoded in HTN methods at design time, and also dynamically generate symbolic representations of external values at planning time, which makes generated plans easier to understand.

Listing 3: Search over circular obstacles using bitangents is done entirely by external function and resulting plan steps stored in intermediate layer are consumed by the HTN.

```
(:method (forward ?agent ?goal)
  base
  ((at ?agent ?goal)) ; preconditions
  () ; empty subtasks
  search
  (; preconditions
    (at ?agent ?start)
    (call search-circular ?agent ?start ?goal)
  )
  ; subtasks
  ((apply-plan ?agent ?start 0 (call plan-size)))
)
(:method (apply-plan ?agent ?from ?index ?size)
  index-equals-size
  ((call = ?index ?size)) ; preconditions
  () ; empty subtasks
  get-next-action
  ; preconditions
  ((assign ?to (call plan-position ?index)))
  (; subtasks
    (!move ?agent ?from ?to)
    (apply-plan ?agent ?to (call + ?index 1) ?size)
  )
)
```

Listing 4: Search over circular obstacles using bitangents is done by the HTN using CLOSEST SA to generate each step.

```
(:attachments (closest ?circle ?to ?outcircle
  ?indir ?outdir ?goal))
(:method (forward-attachments ?agent ?goal)
  clockwise
  ((at ?agent ?start)) ; preconditions
  (; subtasks
    (loop ?agent ?start ?start clock ?goal)
  )
  counter-clockwise
  ((at ?agent ?start)) ; preconditions
  (; subtasks
    (loop ?agent ?start ?start counter ?goal)
  )
)
(:method (loop ?agent ?from ?circle ?indir ?goal)
  base
  ((call visible ?from ?circle ?goal)) ; preconditions
  ((!move ?agent ?from ?goal)) ; subtasks
  recursion
  (; preconditions
    (closest ?circle ?to ?outcircle
      ?indir ?outdir ?goal)
    (not (visited ?agent ?to))
  )
  (; subtasks
    (!move ?agent ?from ?to)
    (!!visit ?agent ?from)
    (loop ?agent ?to ?outcircle ?outdir ?goal)
    (!!unvisit ?agent ?from)
  )
)
```

Our work is the first attempt at defining the syntax and operation of semantic attachments for HTNs, allowing further research on search in SA-enabled domains within HTN planners. Future work includes implementing a cache to reuse previous values from external procedures applied to similar previous states (Dornhege, Hertle, and Nebel 2013) and a generic construction to access such values in the symbolic layer, to obtain data from explored branches outside the state structure, i.e. to hold mutually exclusive predicate information. We plan to develop more domains, with varying levels of domain knowledge and SA usage, to obtain better comparison with other planners and their resulting plan quality. The advantage of being able to exploit external implementations conflicts with the ability to incorporate such domain knowledge into heuristic functions, as such knowledge is outside the description. Further work is required to expose possible metrics from a SA to heuristic functions.

Acknowledgements

We acknowledge the support given by CAPES/Pro-Alertas (88887.115590/2015-01) and CNPQ within process number 305969/2016-1 under the PQ fellowship.
This paper was achieved in cooperation with HP Brasil Indústria e Comércio de Equipamentos Eletrônicos LTDA. using incentives of Brazilian Informatics Law (Law n^o 8.2.48 of 1991).

References

Bryce, D.; Gao, S.; Musliner, D. J.; and Goldman, R. P. 2015. SMT-Based Nonlinear PDDL+ Planning. In *AAAI*, 3247–3253.

Dahl, O.-J.; Dijkstra, E. W.; and Hoare, C. A. R. 1972. *Structured programming*. Academic Press Ltd.

de Silva, L., and Meneguzzi, F. 2015. On the design of symbolic-geometric online planning systems. In *2015 Workshop on Hybrid Reasoning (HR 2015)*, 1–8.

Dornhege, C.; Gissler, M.; Teschner, M.; and Nebel, B. 2009. Integrating symbolic and geometric planning for mobile manipulation. In *Safety, Security & Rescue Robotics (SSRR), 2009 IEEE International Workshop on*, 1–6. IEEE.

Dornhege, C.; Hertle, A.; and Nebel, B. 2013. Lazy evaluation and subsumption caching for search-based integrated task and motion planning. In *IROS workshop on AI-based robotics*.

Dubins, L. E. 1957. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of mathematics* 79(3):497–516.

Fox, M., and Long, D. 2003. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*.

Frances, G., and Geffner, H. 2015. Modeling and computation in planning: Better heuristics from more expressive languages. In *ICAPS*, 70–78.

Francès, G.; Ramírez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action descriptions are overrated: Classical planning with simulators. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 4294–4301.

Gerevini, A. E.; Saetti, A.; and Serina, I. 2008. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence* 172(8-9):899–944.

Ilghami, O., and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. Technical report, DTIC Document.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mohr, F.; Lettmann, T.; Hüllermeier, E.; and Wever, M. 2018. Programmatic Task Network Planning. In *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning*, 31–39.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial Intelligence-Volume 2*, 968–973. Morgan Kaufmann Publishers Inc.

Nebel, B. 2000. On the Compilability and Expressive Power of Propositional Planning Formalisms. *Journal of Artificial Intelligence Research* 12:271–315.

Piotrowski, W. M.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic Planning for PDDL+ Domains. In *AAAI Workshop: Planning for Hybrid Systems*.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *ECAI*, 655–663.

Strobel, V., and Kirsch, A. 2014. Planning in the wild: modeling tools for PDDL. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, 273–284. Springer.

Şucan, I. A., and Kavraki, L. E. 2011. Mobile manipulation: Encoding motion planning options using task motion multigraphs. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 5492–5498. IEEE.

Weyhrauch, R. W. 1981. Prolegomena to a theory of mechanized formal reasoning. In *Readings in Artificial Intelligence*. Elsevier. 173–191.

Learning Domain Structure in HGNs for Nondeterministic planning

Morgan Fine-Morris and Héctor Muñoz-Avila
Lehigh University
Bethlehem, PA 18015
{mof217, hem4}@lehigh.edu

Abstract

This paper presents preliminary ideas of our work for automated learning of Hierarchical Goal Networks in nondeterministic domains. We are currently implementing the ideas expressed in this paper.

Introduction

Many domains are amenable to hierarchical problem-solving representations whereby complex problems are represented and solved at different levels of abstraction. Examples include (1) some navigation tasks where hierarchical A* has been shown to be a natural solution solving the navigation problem over different levels of abstraction (Holte et al. 1995; Wang et al. 2014); (2) dividing a reinforcement learning task into subtasks where policy control is learned for subproblems and combined to form a solution for the overall problem (Dayan and Hinton 1993; Dietterich 2000; Diuk et al. 2013); (3) abstraction planning, where concrete problems are transformed into abstract problem formulations, these abstract problems are solved as abstract plans, and in turn these abstract plans are refined into concrete solutions (Knoblock 1994; Bergmann and Wilke 1995); and (4) hierarchical task network (HTN) planning where complex tasks are recursively decomposed into simpler tasks (Currie and Tate 1991; Wilkins 1999; Erol, Hendler, and Nau 1994; Nau et al. 1999). These paradigms have in common a divide-and-conquer method to problem solving that is amenable to stratified representation of the subproblems.

Among the various formalisms, HTN planning has been a recurrent research focus over the years. An HTN planner formulates a plan using actions and HTN methods. The latter describe how and when to reduce complex tasks into simpler subtasks. HTN methods are used to recursively decompose tasks until so-called primitive tasks are reached corresponding to actions that can be performed directly in the world. The HTN planners SHOP and SHOP2 (Nau et al. 1999; 2003) have routinely demonstrated impressive gains in performance (runtime and otherwise) over standard planners. The primary reason for these performance gains is because of the capability of HTN planners to exploit domain-specific knowledge (Wilkins and desJardins 2001). HTNs provide a natural knowledge-modeling representation for many domains (Nau et al. 2005), including military planning (Mitchell 1997; Muñoz-Avila et al. 1999), strategy formulation in computer games (Hoang, Lee-Urban, and Muńoz-Avila 2005; Gorniak and Davis 2007), manufacturing processes (Nau 1994; Tao et al. 2008), project planning (Tate 1976; Ullrich 2005), story-telling (Cavazza, Charles, and Mead 2002), web service composition (Kuter et al. 2005), and UAV planning (Gancet et al. 2005)

Despite these successes, HTN planning suffers from a representational flaw centered around the notion of *task*. A task is informally defined as a description of an activity to be performed (e.g., *find the location of robot r15*) (e.g., the task "dislodge red team from Magan hill" in some adversarial game) and syntactically represented as a logical atom (e.g., *(locate r15)*). (e.g., "(dislodge redteam Magan)"). Beyond this syntax, there is no explicit semantics of what tasks actually mean in HTN representations. HTN planners obviate this issue by requiring that a complete collection of tasks and methods is given, one that decomposes every complex task in every plausible situation. However, the knowledge engineering effort of creating a complete set of tasks and methods can be significant (Estlin, Chien, and Wang 1997). Furthermore, researchers have pointed out that the lack of tasks' semantics make using HTNs problematic for execution monitoring problems (Dvorak, Amador, and Starbird 2008; Dvorak et al. 2009). Unlike goals, which are conditions that can be evaluated against the current state of the world, tasks have no explicit semantics other than decomposing them using methods.

For example, suppose that a team of robots is trying to locate r15 and, using HTN planning, it generates a plan calling for the different robots to ascertain r15's location. While executing the plan generate a complex plan in a gaming task to dislodge red team from Magan hill, the HTN planner might set a complex plan to cutoff access to Magan, surround it, weaken the defenders with artillery fire and then proceed to assault it. If sometime while executing the plan, the opponent abandons the hill, the plan would continue to be executed despite the fact that the task is already achieved. This is due to the lack of task semantics, so their fulfillment cannot be checked against the current state; instead their fulfillment is only guaranteed when the execution of the generated plans is completed.

Hierarchical Goal Networks (HGNs) solve these limitations by representing goals (not tasks) at all echelons of the hierarchy (Shivashankar et al. 2012). Hence, goal fulfillment can be directly checked against the current state. In particular, even when a goal g is decomposed into other goals (i.e., in HGN, HGN methods decompose goals into subgoals), the question if the goal is achieved can be answered directly by checking if it is valid in the current state. So in the previous example, when the opponent abandons the hill, an agent executing the plan knows this goal has been achieved regardless of how far it got into executing the said plan.

Another advantage of HGNs is that it relaxes the complete domain requirement of HTN planning (Shivashankar et al. 2013); in HTN planning a complete set of HTN methods for each task is needed to generate plans. Even if the HGN methods are incomplete, it is still possible to generate solution plans by falling back to standard planning techniques such as heuristic planning (Hoffmann and Nebel 2001) to achieve any open goals. Nevertheless, having a collection of well-crafted HGN methods can lead to significant improvement in performance over standard planning techniques (Shivashankar 2015).

When the HGN domain is complete (i.e., there is no need to revert to standard planning techniques to solve any problem in the domain), its expressiveness is equivalent to Simple Hierarchical Ordered Planning (Shivashankar 2015), which is the particular variant of HTN planning used by the widely used SHOP and SHOP2 (Nau et al. 2003) HTN planners. SHOP requires the user to specify a total order of the tasks; SHOP2 drops this requirement allowing partial-order between the tasks (Nau et al. 2001). Both have the same representation capabilities although SHOP2 is usually preferred since it doesn't force the user to provide a total order for the method's subtasks (Nau et al. 2001).

In this work, we propose the automated learning of HGNs for ND domains extending our previous work on learning HTNs for deterministic domains (Gopalakrishnan, Muñoz-Avila, and Kuter 2018). While work exists on learning goal hierarchies (Reddy and Tadepalli 1997; Könik and Laird 2006; Ontanón et al. 2010), these works are based on formalisms that have more limited representations than HGNs and in fact predate them.

Related Work

Aside from HGNs, researchers have explored other ways to address the limitation associated with the lack of tasks' semantics. For instance, TMKs (Task-Method-Knowledge models) require not only the tasks and methods to be given but also the semantics of the tasks themselves as *(preconditions,effects)* pairs (Murdock and Goel 2001; Murdock 2001). While this solves the issue with the lack of tasks' semantics it may exacerbate the knowledge engineering requirement of HTNs: the knowledge engineer must not only encode the methods and tasks but also must encode their semantics and ensure that the methods are consistent with the given tasks' semantics. To deal with incomplete HTN domains, researchers have proposed translating the methods into a collection of actions so that standard planning techniques can be used (Alford, Kuter, and Nau 2009). There are two limitations with this approach. First, HTN planning is strictly more expressive than standard planning (Erol, Hendler, and Nau 1994), hence the translation will be incomplete in many domains. Second, for domains when translating methods into actions is possible, it may result in exponentially-many actions on the number of methods. HGNs are more in line with efforts combining HTN and standard planning approaches (Kambhampati, Mali, and Srivastava 1998; Estlin, Chien, and Wang 1997); the main difference is that HGNs eliminate the use of tasks all-together while still preserving the expressiveness of Simple Hierarchical Ordered Planning (Shivashankar 2015).

The problem of learning hierarchical planning knowledge has been a frequent research subject over the years. For example, ICARUS (Choi and Langley 2005) learns HTN methods by using skills (i.e., abstract definitions of semantics of complex actions) represented as Horn clauses. The crucial step is a teleoreactive process where planning is used to fill gaps in the HTN planning knowledge. For example, if the learned HTN knowledge is able to get a package from an starting location to a location $L1$ and the HTN knowledge is also able to get the package from a location $L2$ to its destination, but there is no HTN knowledge on how to get the package from $L1$ to $L2$, then an standard planner is used to generate a plan to get the package from $L1$ to $L2$ and skills are used to learn new HTN methods from the plan generated to fill the gap on how to get from $L1$ to $L2$.

Another example is HTN-Maker (Hogg, Muñoz-Avila, and Kuter 2008). HTN-Maker uses task semantics defined as *(preconditions,effects)* pairs, exactly like TMKs mentioned before, to identify sequences of contiguous actions in the input plan trace where the preconditions and effects are met. Task hierarchies are learned when an action sequence is identified as achieving a task and the action sequence is a sub-sequence of another larger action sequence achieving another task. This includes the special case when the sub-sequence and the sequence achieve the same task. In such a situation recursion is learned. HTN-Maker learns incrementally after each training case is given.

HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) transforms the input traces into a constraint satisfaction problem. Like HTN-Maker, it also assumes *(preconditions,effects)* as the task semantics to be given as input. HTNLearn process the input traces converting them into constraints. For example, if a literal p is observed before an action a and a is a candidate first sub-task for a method m, then a constraint c is added indicating that *p is a precondition of m.* These constrains are solved by a MAXSAT solver, which returns the truth value for each constraint. For example, if c is true then p is added as a precondition of m. As a result of the MAXSAT process, HTNLearn is not able to converge to a 100% correct domain (the evaluation of HTNLearn computes the error rates in the learned domain).

Similar to HTN planning, hierarchical decompositions have been used in *hierarchical reinforcement learning* (Parr and Russell 1998; Dietterich 2000). The hierarchical structure of the reinforcement learning problem is analogous to an instance of the decomposition tree that an HTN planner might generate. Given this hierarchical structure, hierarchical reinforcement learners perform value-function composition for a task based on the value functions learned over its subtasks recursively. However, the possible hierarchical decompositions must be provided in advance.

Hierarchical goal networks (HGNs) (Shivashankar et al. 2012) are an alternative representation formalism to HTNs. In HGNs, goals, instead of tasks, are decomposed at every level of the hierarchy. HGN methods have the same fo.rm as HTN methods but instead of decomposing a task, they decompose a goal; analogously instead of subtasks, HGN methods have subgoals. If the domain description is incomplete, HGNs can fall back to STRIPS planners to fill gaps in the domain. On the other hand, total-order HGNs are as expressive as total-order HTNs (Shivashankar 2015) and its partial-order variant (Shivashankar et al. 2016) is as expressive as partial-order HTNs (Alford et al. 2016).

Inductive learning has been used to learn rules indicating goal-subgoal relations in X-learn (Reddy and Tadepalli 1997). This is akin to learning macro-operators (Mooney 1988; Botea, Müller, and Schaeffer 2005); the learned rules and macro-operators provide search control knowledge to reach the goals more rapidly but they don't add expressibility to standard planning. SOAR learns goal-subgoal relations (Könik and Laird 2006). It uses as input annotated behavior trace structures, indicating the decisions that led to the plans; this is used to generate a goal-subgoal relations. Another work on learning goal-subgoal relations is reported in (Ontanón et al. 2010). It uses case-based learning techniques to store goal-subgoal relations, which are then reused by using similarity metrics. These works assume some form of the input traces, unstructured in (Ontanón et al. 2010) and structured in (Könik and Laird 2006), to be annotated with the subgoals as they are accomplished in the traces. In our proposed work, the input traces are not annotated and, more importantly, we are learning HGNs.

Goal regression techniques have been used to generate a plan starting from the goals that must be achieved (Pollock 1998; McDermott 2002). The result of goal regression can be seen as a hierarchy recursively generated by indicating for each goal what subgoals must be achieved. The goal-subgoal relations resulting from goal regression are a direct consequence of the domain's operators: the goals are effects of the operators and the preconditions are the subgoals. In contrast, in a HGN, the hierarchies of goals represent relations between the HGN methods and are not necessarily implied directly from the actions. Making an analogy with HTN methods, HGN methods capture additional domain-specific knowledge (Nau et al. 2003) or generate plans with desirable properties (e.g., taking into account quality considerations) again not explicitly represented in the actions (Hogg, Kuter, and Munoz-Avila 2010).

Work on learning hierarchical plan knowledge is related to learning of context-free grammars (CFGs), which aims at eliciting a finite set of production rules from a finite set of strings (Oates, Desai, and Bhat 2002; Sakakibara 1997). The precise definition of the learning problem varies constraining the resulting CFG by, among others, (1) providing a target function (e.g., obtaining a CFG with the minimum number of production rules) or (2) assuming that negative examples (i.e., strings that must not be generated by the CFG) are given. To learn CFGs, algorithms search for production rules that generate the training set (and none of the negative examples when provided). Grammar learning is exploited by the Greedy Structure Hypothesizer (GSH) (Li, Kambhampati, and Yoon 2009), which uses probabilistic context-free grammars learning techniques to learn a hierarchical structure of the input plan traces. GSH doesnt learn preconditions since its goals are not to generate the grammars for planning but to reflect users preferences. The difference between learning CFG and learning hierarchical planning knowledge is twofold. First, characters that form a string have no meaning. In contrast, actions in a given plan are defined by their preconditions and effects. This means that plausible strings generated by the grammars may be invalid when viewed as plans. Second, learning HGNs requires not only learning the task decomposition but also the preconditions. This is an important difference: HTNs are strictly more expressive than CFGs (Erol, Hendler, and Nau 1996). Intuitively, HTNs are akin to context-sensitive grammars in that they constraint when a decomposition can take place. Context-sensitive grammars are also strictly more expressive then CFGs (Sipser 2006).

Finally, as we will see in the next the next section, our proposed work is related to the notion of planning landmarks (Hoffmann, Porteous, and Sebastia 2004). Given a **planning problem** P, defined as a triple $(s_0, g, \mathcal{A})$, indicating the initial state, the goals and the actions respectively, a planning landmark is either an action $a \in \mathcal{A}$, or state atom $p \in s$ (s is an state, represented as a collection of atoms) that occurs in any solution plan trace solving P. Given the problem description P, planning systems can identify automatically landmarks for P. Planning landmarks have been widely used for automated planning resulting in planners such as LAMA (Richter and Westphal 2010) and the HGN planner GoDel (Shivashankar et al. 2013).

ND learning problem

We want to learn HGNs for fully observable nondeterministic (FOND) planning (Fu et al. 2011; Speck, Ortlieb, and Mattmüller 2015; Winterer, Mattmüller, and Wehrle 2015). In such domains, actions may have multiple outcomes. For example, in the Minecraft simulation, when a character swings a sword to hit a monster, there are two possible outcomes: either the sword hits the monster or the monster parries it and the sword doesn't hit anything.

As discussed before, HTN learners require the tasks semantics to be given either as Horn clauses defining the tasks or as *(preconditions,effects)* pairs. The latter is used, for example, in the nondeterministic HTN learner ND-HTNMaker, a state-of-the-art HTN learner, to pinpoint locations in the traces where the various tasks are fulfilled. ND-HTNMaker enforces a right recursive structure: exactly one primitive task followed by none or exactly one compound task. The main objective of enforcing this right recursive structure is to deal with nondeterminism: if, for example, the character swings the sword (e.g., a primitive task), the follow-up compound task handles the nondeterminism: one method decomposing a compound task t will simply perform the action to swing at the monster followed by t, thereby ensuring that method can be triggered as many times as needed until the monster is hit (and dies). Other methods decomposing t handle the case when the monster has been dealt with (e.g., a method handling the case when "character next to a dead monster"). This ensures that methods learned by HTN-Maker are provable correct (Hogg, Kuter, and Muñoz-Avila 2009). Correctness can be loosely defined as follows: any solution generated by a sound nondeterministic HTN planner such as ND-SHOP (Kuter and Nau 2004) using the learned methods and the nondeterministic actions is also a solution when using the nondeterministic actions (i.e., without the methods).

Like in the deterministic case, the inputs will be (1) a collection of actions $\mathcal{A}$ and (2) a collection of traces $s_0\ a_0\ s_1\ a_1\ \dots\ a_n\ s_{n+1}$, where each $a_i \in \mathcal{A}$. Only this time, any action $a_i \in \mathcal{A}$ may have multiple outcomes; so each occurrence of a_i in the input traces will reflect one such outcome.

Planning in nondeterministic domains requires to account for all possible outcomes. As such, (Cimatti et al. 2003) proposed a categorization of solutions for nondeterministic domains. It distinguishes between weak, strong cyclic and strong solutions for a problem $(s_0, g, \mathcal{A})$. A solution is represented as a policy $\pi : S \to A$, a mapping from the possible states in the world S to actions A, indicating for any given state $s \in S$, what action $\pi(s)$ to take. Given a policy π, an **execution trace** is any sequence $s_0\ \pi(s_0)\ s_1\ \pi(s_1)\ \dots\ \pi(s_n)\ s_{n+1}$, where s_i is an state that can be reached from state s_{i-1} after applying action $\pi(s_{i-1})$.

A solution policy π is *weak* if there exists an execution trace from s_0 to a state satisfying g. Weak solutions guarantee that a goal state can be successfully reached sometimes. For example, in the Minecraft simulation, a policy that assumes a computer-controlled character will always hit any monster it encounters when swinging the sword is considered a weak solution. In particular, this solution would not account for the situation when the monster parries the character's sword attack; e.g., the monster might counter-attack and disable the agent and the agent has not planned what to do in such a situation. Under the fairness assumption, stating that *"every action executed infinitely often will exhibit all its effects infinitely often"* (D'Ippolito, Rodrıguez, and Sardina 2018), a solution π is either strong cyclic or strong if (1) every terminal state entails the goals and (2) for every state s that the agent might finds itself in after executing π from s_0, there exists an execution trace from the state s to a state satisfying g. The difference is that in strong cyclic solutions the same state might be visited more than once whereas in strong solutions this never happens. For example, a strong cyclic solution might have the character swing the sword against the monster and if the monster parries the attack, the character takes an step back to avoid the monster's counter-attack and step towards the monster while taking another swing at it; this can be repeated as many times as needed until the monster dies. Strong solutions are ideal since they never visit the same state but in some domains they might not exists. For instance, there are no strong solutions in the Minecraft simulation mentioned as the monster can repeatedly parry the character's attacks. The same occurs in the robot navigation domain (Cimatti et al. 2003), created to model nondeterminism. In this domain a robot is navigating between offices and when it encounters a closed door for an office it wants to access, the robot will open it. There is an another agent acting in the environment that closes doors at random. So the robot might need to repeatedly execute the action to open the same door.

Solving nondeterministic planning problems is difficult because of what has been dubbed the explosion of states as a result of the nondeterminism (Fu et al. 2011). One demonstrated way to counter this is by adding domain-specific knowledge as described in (Kuter and Nau 2004). While the algorithm described is generic for a variety of ways to encode the domain-specific knowledge, it showcases hierarchical planning techniques outperforming an state-of-the-art nondeterministic planner in some domains including the robot navigation domain. The results show either speedups of several orders of magnitude or the ability to solve problems of sizes, measured by the number of goals to achieve, previously impossible to solve.

Relation to probabilistic domains. In this work we are neither assuming a probability distribution over the possible actions' outcomes to be given nor we aim to learn such a distribution. Once an HGN domain is learned, hierarchical reinforcement learning techniques (Dietterich 2000) can be used to learn a probability distribution over the various possible goal decompositions and exploit the learned distribution during problem solving as done in (Hogg, Kuter, and Munoz-Avila 2010).

We propose to learn bridge atoms and their hierarchical structure with the important constraint that the learned hierarchical structure must encode the domain's nondeterminism in a sound way. For instance, the nondeterministic version of the logistics transportation domain in (Hogg, Kuter, and Muñoz-Avila 2009) extends the deterministic version as follows: when loading a package p into vehicle v in a location l there are two possible outcomes: either p is inside v or p is still at l (i.e., the load action failed). Regardless of possibly repeating the same action multiple times, traces will bring the package to the airport, transport it by air to the destination city, and deliver it. So the kinds of decompositions we are aiming to learn should also work on nondeterministic domains; on the other hand a learned hierarchy would be unsound if, for example, it assumes that the load truck action always succeeds and immediately proceeds to deliver the package to an airport. This will lead to weak solutions.

To correctly handle nondeterminism, we propose forcing a right-recursive structure on lower echelons of the learned HGNs. This takes care of the nondeterminism and combine well with the higher decompositions. For instance, in the transportation domain we identified a goal g_{airp}, for the package p reaching the airport, identified as a bridge atom, and then have all methods achieving g_{airp} be right recursive; e.g., methods of the form $(:\ method\ g_{airp}\ prec\ (g\ g_{airp})\ <)$, where g is some intermediate goal such as loading the package into a vehicle.

Defining the Learning Problem

Our aim is the automated learning of HGN methods. This includes learning the goals, the goal-subgoal structure of the HGN methods and their applicability conditions. Specifically, the learning problem can be defined as follows: given a set of actions $\mathcal{A}$ and a collection of traces Π generated using actions in $\mathcal{A}$, to obtain a collection of HGN methods. A collection of methods $\mathcal{M}$ is correct if given any *(initial state, goal)* pair (s_0, g), and any solution plan π generated by a sound HGN planner using $\mathcal{M}$ and $\mathcal{A}$, π is a correct plan solving the planning problem $(s_0, g, \mathcal{A})$. An HGN method m is a construct of the form *(:method head(m) preconditions(m) subgoals(m) <(m))* corresponding to the goal decomposed by m (called the head of m), the preconditions for applying m and the subgoals decomposing *head(m)*. Figure 1 shows an example of an HGN method in the logistics transportation domain (Veloso 1994). (the question marks indicate variables. It recursively decomposes the goal of delivering *?pack1* into *?loc2* into three subgoals: (1) delivering *?pack1* to the airport *?airp1* in the same city as its current location *?loc1*, (2) delivering *?pack1* to the airport *?airp2* in the same city as the destination location *?loc2*, and (3) recursively achieve the head goal):

Head: Package-delivery
Preconditions: (at ?pack ?loc1 ?city1) (airport ?airp1 ?city1) (airport ?airp2 ?city2) (location ?loc2 ?city2) ($\neq$?city1 ?city2)
Subgoals: g_1: (package-at ?pack ?airp1) g_2:(package-at ?pack1 ?airp2) g_3:(package-at ?pack ?loc2 ?city2)
Constraints: $g_1 < g_3$, $g_2 < g_3$

Figure 1: Example of an HGN method in the logistics transportation domain. The question marks indicate variables. The goal achieved by the method is the last subgoal, g_3. It recursively decomposes the goal of delivering *?pack* into *?loc2* into three subgoals: (1) delivering *?pack1* to the airport *?airp1* in the same city as its current location *?loc1*, (2) delivering *?pack* to the airport *?airp2* in the same city as the destination location *?loc2*, and (3) g_3 is to be achieved after g_1 and g_2 are achieved.

HGNs planners (Shivashankar et al. 2013; 2012) maintain a list $G = \langle g_1, \ldots, g_n \rangle$ of open goals (i.e., goals to achieve). Planning follows a recursive procedure, starting with $\pi = \langle \rangle$, choosing the first element, g_1, in G, and either (1) applying an HGN method m decomposing g_1 into m's subgoals $\langle g'_1, \ldots, g'_k \rangle$, concatenating m's subgoals into G (i.e, $G = \langle g'_1, \ldots, g'_k, g_1, \ldots, g_n \rangle$ are the new open goals), or (2) applying an action $a \in \mathcal{A}$ achieving g, appending a to π (i.e., $\pi \leftarrow \pi \cdot a$) and removing g from G. In either case it will check if the preconditions of m (respectively, a) are satisfied in the current state. When a is applied, the current state is transformed in the usual way (Fikes and Nilsson 1971). When $G = \emptyset$, π is returned. HGN planners extend this basic procedure to allow the use of standard planning techniques to achieve open goals and to enable a partial ordering between the methods' subgoals. the planner picks the first goal in G without predecessors. For example, in Figure 1, the user may define the constraints: $g_1 < g_3, g_2 < g_3$, and the planner instead of always picking the first subgoal in G, it picks the first subgoal without predecessors. [1]

Learning Hierarchical Goal Structures

We propose transforming the problem of identifying the goals and learning their hierarchical relation into the problem of finding relations between word embeddings extracted from text. Specifically, we propose viewing the collection of input traces Π as text: each plan trace $\pi = s_0\ a_0\ s_1\ a_1\ \ldots\ a_n\ s_{n+1}$ is viewed as a sentence $w_1\ w_2\ \ldots\ w_m$; each action a_i and each atom in s_j is viewed as a word w_k in the sentence. The order of the plan elements in each trace is preserved (we use the term **plan element** to refer to both atoms and actions): the word $w_j = a_i$ appears before the word $w_{j'} = p$, for every $p \in s_{i+1}$. In turn, every $w_{j'}$ appears before $w_{j''} = a_{i+1}$.

Word embeddings are vectors representing words in a multi-dimensional vector space (Bengio et al. 2003; Baroni, Dinu, and Kruszewski 2014). There are a number of algorithms to do this translation (Mikolov et al. 2013; Pennington, Socher, and Manning 2014). They have in common that they represent vector similarity based on the co-occurrence of words in the text. That is, words that tend to occur near one another will have similar vector representations. In our preliminary work we used Word2Vec (Mikolov et al. 2013) (i.e., Word-Neighboring Word), a widely used algorithm for generating word embeddings. Word2Vec uses a shallow neural network, consisting of a single hidden layer, to compute these vector representation; it computes a context window $\mathcal{W}$ consisting of k contiguous words and trains the network using each word $w \in \mathcal{W}$ (i.e., $\mathcal{W}$ is w's context). The window $\mathcal{W}$ is "moved" one word at the time through the text further training the network each time. Training is repeated with windows of size $i = \{1, 2, \ldots k\}$. For this reason, Word2Vec is said to use "dynamic windows". In Word2Vec, similarity is computed with the cosine similarity, sim_C, because it measures how close is the orientation of the resulting vectors, which are distributed in such a way that words frequently co-occurring in the context windows have similar orientation whereas those that co-occur less frequently will have a dissimilar orientation.

There are two particularities of the change of representation from plan elements to word embeddings that is particularly suitable for our purposes: first the procedure is unsupervised. This means in our case that we do not have to annotate the traces with additional information such as where the goals are been achieved in the traces. Second, vector representations are generated based on the context in which they occur (e.g., the dynamic window $\mathcal{W}$ in Word2Vec). In our case, the vector representations of the plan elements will be generated based on their proximity to other plan elements in the traces. These vectors can be clustered together into plan elements that are close to one another.

[1] Actions are *(preconditions,effects)* pairs, where the effects consist of the add- and the delete-lists of atoms. If the preconditions are satisfied in the current state, the state is transformed as indicated by the effects: atoms in the delete-list are removed and atoms in the add-list are added.

Our working hypothesis, supported by previous work (Gopalakrishnan, Muñoz-Avila, and Kuter 2018), is that what we call **bridge atoms**, are ideal candidates for goals. Given two clusters of plan element embeddings, A and B, a *bridge atom*, $bridge_{AB}$, is an atom in either A or B that is most similar to the plan elements in the other set.

Establishing a bridge atom hierarchy is a recursive process that first requires calculating the bridge atom of a corpus, splitting each text around the bridge atom so that each text in the corpus becomes two new texts, before and after the bridge atom, and then repeating the procedure on the resulting sub-corpora.

The procedure for find a bridge atom for a corpora is as follows. We train a Word2Vec model on the corpus to determine the word vectors and cluster them with Hierarchical Agglomerative Clustering. Currently we limit the number of clusters to two, although later research may explore how to determine the number of clusters from the structure of the traces. We determine the cosine distance of each atom in a cluster to each atom in the other and average them together for each atom, selecting the word with the shortest average distance,[2]

$$\begin{aligned} bridge_{AB} = argmin_{a\in A, b\in B}(&\frac{1}{|B|}\Sigma_{b\in B} dist_C(a,b), \\ &\frac{1}{|A|}\Sigma_{b\in B} dist_C(a,b)), \end{aligned} \tag{1}$$

where $dist_C$ is the cosine distance between the vector representations of two atoms. If an action is selected as the bridge atom, we instead use in its place the atom describing one of its goals.

As previously stated, by splitting each trace around the bridge atom, we can form two new sub-corpora, one from the section of each trace before the bridge atom and one from the section after the bridge atom. Then we recursively perform the procedure for bridge atom selection on each new corpora, keeping track of the hierarchical relationship of each sub-corpora to the other corpora. If during the division process, a section of a trace becomes shorter than some threshold, we discard it from the sub-corpora. Progress along any branch of recursion halts once there are insufficient traces in a sub-corpus for training.

We use the hierarchy of bridge atoms as a guide for building a set of hierarchical methods. At the lowest level of division are single-action or short multi-action sections of the traces. Each of these sections will become a method with a single goal (an effect of an action) or a method with multiple goals (one for each of the actions). Each of these methods have two subgoals: one for the subsection of trace before a bridge atom and another one for the trace after that bridge atom.

Each action is annotated with its preconditions. The preconditions of a method can be extrapolated from the preconditions of the actions into which it decomposes by regressing over the actions of that section of the plan trace in reverse, collecting the action preconditions and removing from the preconditions any atom which is in the effects of chronologically-preceding action.

[2]This is different from the formula we used in (Gopalakrishnan, Muñoz-Avila, and Kuter 2018), which computed the maximum similarity.

Current Status

We are using a variant of the Pyhop HTN planner (`https://bitbucket.org/dananau/pyhop`). Our variant introduces nondeterminism in the actions and generates solution policies as described in (Kuter and Nau 2004).

Our experiments use a nondeterministic variant of the logistics domain (Veloso 1992). In the domain, packages must be relocated from one location to another. Trucks transport packages within cities, and airplanes transport packages between cities via airports. Nondeterminism is introduced via the load and unload operators, which have two outcomes, success (the package is loaded onto/unload from the specified vehicle) or failure (the package does not move from its original location). We have also added rockets that transport packages between cities on different planets via launchpads. All traces demonstrate a plan for achieving the same goal, the relocation of a package from a location in one city on the starting planet to a location in a city on the destination planet.

To ensure that Word2Vec can identify common bridge atoms across the corpus, the package and each location must have the same name in all traces. Although Word2Vec typically works best on a corpus of thousands of texts or more, we are able to learn reasonable bridge atoms from hundreds of texts by increasing the number of epochs and lowering learning rate. For our problem design, a reasonable first bridge atom is one that involves the package and a rocket or launchpad, as transporting the package from the start planet to the destination planet marks the halfway point in the traces. From a corpus of 700 traces, with 1000 epochs and a learning rate of 0.00025, our first bridge atom is the action unload(package, rocket).

Because word embeddings are sensitive to word context, the trace structure influences the bridge atom hierarchy. Which atoms are included in the trace and where they are included is important. We are experimenting with two different variants of state expression within traces. In one variant, we list each action preceded by its deletelist and followed by its addlist. If an atom occurs in the addlist of one action and the deletelist of the subsequent action, that atom will only appear in the addlist of the first action. In another variant, we list actions preceded by their preconditions and followed by their effects. In both variants, atoms are listed alphabetically.

Acknowledgements. This research was supported by ONR under grants N00014-18-1-2009 and N68335-18-C-4027.

References

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 3022–3029.

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *International Joint Conference on Artificial Intelligence (IJCAI).*

Baroni, M.; Dinu, G.; and Kruszewski, G. 2014. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *ACL (1)*, 238–247.

Bengio, Y.; Ducharme, R.; Vincent, P.; and Jauvin, C. 2003. A neural probabilistic language model. *Journal of machine learning research* 3(Feb):1137–1155.

Bergmann, R., and Wilke, W. 1995. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research (JAIR)* 3:53–118.

Botea, A.; Müller, M.; and Schaeffer, J. 2005. Learning partial-order macros from solutions. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 231–240.

Cavazza, M.; Charles, F.; and Mead, S. J. 2002. Interacting with virtual characters in interactive storytelling. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, 318–325. ACM.

Choi, D., and Langley, P. 2005. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, 51–68.

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

Currie, K., and Tate, A. 1991. O-Plan: The open planning architecture. *Artificial Intelligence* 52(1):49–86.

Dayan, P., and Hinton, G. E. 1993. Feudal reinforcement learning. In *Advances in neural information processing systems*, 271–278.

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research (JAIR)* 13:227–303.

D'Ippolito, N.; Rodrıguez, N.; and Sardina, S. 2018. Fully observable non-deterministic planning as assumption-based reactive synthesis. *Journal of Artificial Intelligence Research* 61:593–621.

Diuk, C.; Schapiro, A.; Córdova, N.; Ribas-Fernandes, J.; Niv, Y.; and Botvinick, M. 2013. Divide and conquer: hierarchical reinforcement learning and task decomposition in humans. In *Computational and robotic models of the hierarchical organization of behavior*. Springer. 271–291.

Dvorak, D. L.; Amador, A. V.; and Starbird, T. W. 2008. Comparison of goal-based operations and command sequencing. In *Proceedings of the 10th International Conference on Space Operations*.

Dvorak, D. D.; Ingham, M. D.; Morris, J. R.; and Gersh, J. 2009. Goal-based operations: An overview. *JACIC* 6(3):123–141.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *National Conference on Artificial Intelligence (AAAI)*, 1123–1128.

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

Estlin, T. A.; Chien, S.; and Wang, X. 1997. An argument for a hybrid HTN/operator-based approach to planning. In *European Conference on Planning (ECP)*, 184–196.

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

Fu, J.; Ng, V.; Bastani, F. B.; Yen, I.-L.; et al. 2011. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, 1949.

Gancet, J.; Hattenberger, G.; Alami, R.; and Lacroix, S. 2005. Task planning and control for a multi-uav system: architecture and algorithms. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, 1017–1022. IEEE.

Gopalakrishnan, S.; Muñoz-Avila, H.; and Kuter, U. 2018. Learning task hierarchies using statistical semantics and goal reasoning. *AI Communications* 31(2):167–180.

Gorniak, P., and Davis, I. 2007. Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters. In *AIIDE*, 14–19.

Hoang, H.; Lee-Urban, S.; and Muńoz-Avila, H. 2005. Hierarchical plan representations for encoding strategic game AI. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

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

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2010. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *Conference on Artificial Intelligence (AAAI)*, 950–956. AAAI Press.

Holte, R. C.; Perez, M.; Zimmer, R.; and MacDonald, A. 1995. Hierarchical a*. In *Symposium on Abstraction, Reformulation, and Approximation*.

Kambhampati, S.; Mali, A.; and Srivastava, B. 1998. Hybrid planning for partially hierarchical domains. In *National Conference on Artificial Intelligence (AAAI)*, 882–888.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial intelligence* 68(2):243–302.

Könik, T., and Laird, J. E. 2006. Learning goal hierarchies from structured observations and expert annotations. *Machine Learning* 64(1-3):263–287.

Kuter, U., and Nau, D. S. 2004. Forward-chaining planning in nondeterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, 513–518.

Kuter, U.; Sirin, E.; Nau, D. S.; Parsia, B.; and Hendler, J. 2005. Information gathering during planning for web service composition. *Journal of Web Semantics (JWS)* 3(2-3):183–205.

Li, N.; Kambhampati, S.; and Yoon, S. W. 2009. Learning probabilistic hierarchical task networks to capture user preferences. In *IJCAI*, 1754–1759.

McDermott, D. V. 2002. Estimated-regression planning for interactions with web services. In *AIPS*, volume 2, 204–211.

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*, 3111–3119.

Mitchell, S. 1997. A hybrid architecture for real-time mixed-initiative planning and control. In *Innovative Applications of Artificial Intelligence Conference (IAAI)*, 1032–1037.

Mooney, R. J. 1988. Generalizing the order of operators in macro-operators. In *Machine Learning*, 270–283.

Muñoz-Avila, H.; McFarlane, D.; Aha, D. W.; Ballas, J.; Breslow, L.; and Nau, D. S. 1999. Using guidelines to constrain interactive case-based HTN planning. In *International Conference on Case-Based Reasoning (ICCBR)*, 288–302.

Murdock, J. W., and Goel, A. K. 2001. Meta-case-based reasoning: Using functional models to adapt case-based agents. In Aha, D. W.; Watson, I.; and Yang, Q., eds., *Fourth International Conference on Case-Based Reasoning*.

Murdock, J. W. 2001. *Self-improvement through self-understanding: Model-based reflection for agent adaptation*. Ph.D. Dissertation, Georgia Institute of Technology.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In Dean, T., ed., *International Joint Conference on Artificial Intelligence (IJCAI)*, 968–973. Morgan Kaufmann.

Nau, D. S.; Muñoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-order planning with partially ordered subtasks. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

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

Nau, D. S.; Au, T.-C.; 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.

Nau, D. S. 1994. Manufacturing-operation planning vs AI planning. In *Third International Conference on Information and Knowledge Management*.

Oates, T.; Desai, D.; and Bhat, V. 2002. Learning k-reversible context-free grammars from positive structural examples. In *International Conference on Machine Learning (ICML)*, 459–465.

Ontanón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2010. On-line case-based planning. *Computational Intelligence* 26(1):84–119.

Parr, R. E., and Russell, S. 1998. *Hierarchical control and learning for Markov decision processes*. University of California, Berkeley Berkeley, CA.

Pennington, J.; Socher, R.; and Manning, C. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1532–1543.

Pollock, J. L. 1998. The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence* 106(2):267–334.

Reddy, C., and Tadepalli, P. 1997. Learning goal-decomposition rules using exercises. In *International Conference on Machine Learning (ICML)*, 843–851.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Sakakibara, Y. 1997. Recent advances of grammatical inference. *Theoretical Computer Science* 185(1):15–45.

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-Volume 2*, 981–988. International Foundation for Autonomous Agents and Multiagent Systems.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The godel planning system: a more perfect union of domain-independent and hierarchical planning. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 2380–2386. AAAI Press.

Shivashankar, V.; Alford, R.; Roberts, M.; and Aha, D. W. 2016. Cost-optimal algorithms for hierarchical goal network planning: A preliminary report. In *ICAPS Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP)*.

Shivashankar, V. 2015. *Hierarchical Goal Network Planning: Formalisms and Algorithms for Planning and Acting*. Ph.D. Dissertation, Dept. of Computer Science, University of Maryland.

Sipser, M. 2006. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston.

Speck, D.; Ortlieb, M.; and Mattmüller, R. 2015. Necessary observations in nondeterministic planning. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, 181–193. Springer.

Tao, F.; Zhao, D.; Hu, Y.; and Zhou, Z. 2008. Resource service composition and its optimal-selection based on particle swarm optimization in manufacturing grid system. *IEEE Transactions on industrial informatics* 4(4):315–327.

Tate, A. 1976. Project planning using a hierarchic non-linear planner. Technical Report 25, Department of Artificial Intelligence, University of Edinburgh.

Ullrich, C. 2005. Course generation based on htn planning. In *LWA*, 74–79.

Veloso, M. M. 1992. Learning by analogical reasoning in general problem solving. PhD thesis CMU-CS-92-174, School of Computer Science, Carnegie Mellon University.

Veloso, M. M. 1994. *Planning and learning by analogical reasoning*. Springer-Verlag.

Wang, H.; Zhou, J.; Zheng, G.; and Liang, Y. 2014. Has: Hierarchical a-star algorithm for big map navigation in special areas. In *Digital Home (ICDH), 2014 5th International Conference on*, 222–225. IEEE.

Wilkins, D., and desJardins, M. 2001. A call for knowledge-based planning. *AI Magazine* 22(1):99–115.

Wilkins, D. E. 1999. Using the sipe-2 planning system. *Artificial Intelligence Center, SRI International, Menlo Park, CA*.

Winterer, D.; Mattmüller, R.; and Wehrle, M. 2015. Stubborn sets for fully observable nondeterministic planning. In *ICAPS*. AAAI Press.

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 HTN Methods with Preference from HTN Planning Instances

Zhanhao Xiao[a], Hai Wan *[a], Hankui Hankz Zhuo[a], Andreas Herzig[b], Laurent Perrussel[c] and Peilin Chen[a]

[a]School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
[b]IRIT, CNRS, Toulouse, France
[c]University of Toulouse, Toulouse, France

Abstract

The hierarchical task network (HTN) planning technique is used in a growing number of real-world applications. However in many domains, such as the logistics domain, as there exist thousands of cases, it is difficult and time-consuming for humans to specify all HTN methods to cover all desirable plans. This suggests that it is important to learn HTN methods to accomplish the tasks via decomposition. The traditional HTN-method learning approaches require complete executable plans and annotated tasks, which are often difficult to acquire in real-world applications. In this paper, we propose a novel framework to learn HTN methods from HTN instances with incomplete method sets and without annotated tasks. Besides, previous approaches demand total orders on the subtasks in the methods while our approach is capable of learning methods with partial orders. To reduce the number of methods learned, we consider priorities on methods and compute the minimal set of methods based on prioritized preferences. By taking experiments on three well-known planning domains, we demonstrate that our approach is effective, especially on solving new HTN problems.

Introduction

The hierarchical task network (HTN) planning technique (Erol *et al.* 1994) is increasingly used in a number of real-world applications (Lin *et al.* 2008; Behnke *et al.* 2019). In the real-world logistics domain, such as Amazon and DHL Global Logistics, the shipment of packages is arranged via decomposition into a more detailed shipment arrangement in a top-down way according to the predefined decomposition methods. In practice, there exist a vast number of cases occurring, such as the delay caused by the weather, leading that it is difficult and time-consuming for humans to find all complete methods for all actions. This suggests that it is important to learn methods to help humans to improve the HTN domain.

Normally the domain experts have partially hierarchical domain knowledge, which possibly is not sufficient to cover all desirable solutions (Kambhampati *et al.* 1998). Different from classical planning which pursues an executable plan to achieve the declarative goal, the solution to the HTN planning problem requires to consider the hierarchical procedural goals, which are given by HTN methods. With partially hierarchical domain knowledge, a solution cannot be found via decomposition according to the given methods. One main reason lies in that the given method set is incomplete, which includes at least an incomplete method lacking subtasks. Keeping the hierarchical procedural knowledge, Geier and Bercher (2011) proposed a hybrid planning formalization, *HTN planning with task insertion (TIHTN planning)*, to allow generating plans via decomposing tasks according to the methods but also inserting tasks from outside the given methods. The following example shows an incomplete method set.

Example 1. *Consider an example in the logistics domain, suppose every task has only one method and a decomposition tree is shown in Figure 1. The initial task* ship(pkg1, whA, shopB) *is to ship a package from city A to city B and it has a method: to ship the package from the warehouse to the airport by truck, from city A to city B by plane and from the airport to the shop by truck. But in case the plane is not in the airport of city A, then the air transportation task cannot be accomplished. When arranging the plane to airport A,* fly(plane1, airpA)*, is done before loading to the plane, it generates an executable plan. If it is not allowed to insert actions, there is no plan to achieve the task* ship.

Actually, the plan with the inserted tasks offers a reference to accomplish the compound tasks, we refine the method by adding the inserted tasks. For example, the method of airShip is refined by adding fly as its subtask. By refining methods, we obtain new methods which generate the missing tasks, resulting in a new decomposition tree.

In practice, the missing of subtasks happens more likely on the methods of some compound tasks than on the methods of some other compound tasks. For example, the task ship is decomposed into the inter-city shipment and the intra-city shipment, while decomposing the task airShip varies with the place where the plane stays. It leads to a priority on the methods: some methods have a high priority to be refined. An excess of methods learned will slow down problem-solving, so we hope to learn as few refined methods

*Corresponding author.

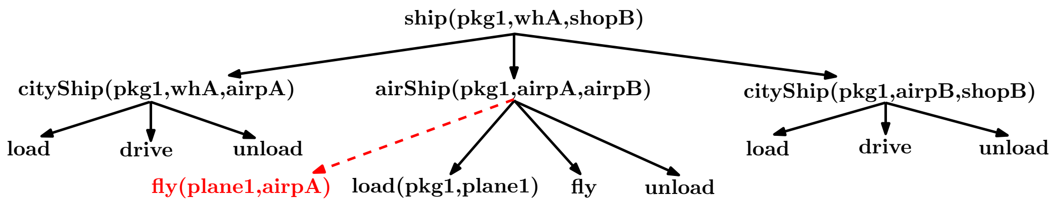


Figure 1: This is an Example of a decomposition tree from an incomplete HTN method set. The initial task ship(pkg1, whA, shopB) is decomposed into a sequence of primitive tasks (the black leaves) according to the original methods. But when the plane1 is not in airport A, the sequence is not executable. It becomes executable if arranging plane 1 to airport A before loading the package, which implies that fly(plane1, airpA) should be considered as a subtask of airShip.

as possible. The task is however challenging, as tasks can be inserted in various methods and an exponential number of method sets need to be considered.

The traditional approaches to learning HTN methods, such as (Hogg *et al.* 2008; Zhuo *et al.* 2014; Lotinac and Jonsson 2016), only concentrate on declarative goals and omit procedural knowledge obtained from the domain designer, which cannot be replaced simply by declarative goals. For example, every package needs a security check before being uploaded into the plane. If the action model is not complete, such as the 'check' action has not the effect 'checked', the declarative goal may not capture it. Besides, the approaches (Hogg *et al.* 2008; Zhuo *et al.* 2014) require the annotated preconditions and effects of tasks, which omit the hierarchical procedural goals and only consider the declarative goals like classical planning, so they require a complete executable plan as input. Whereas it is not a simple task to obtain complete plans, particularly when it involves thousands of situations. Furthermore, in many domains, it is difficult to verify the correctness of the annotations of tasks when they are taken as input. In this paper, we propose a novel framework to learn HTN methods from HTN instances with an incomplete method set, which always cannot generate executable plans only via decomposition. Besides, previous approaches restrict the tasks in the methods to be totally ordered, while we allow them to be partially ordered. Last but not least, we consider a prioritized preference on the methods learned.

Our contributions are listed as follows. First, we propose an approach to learning new methods by refining the original methods based on decomposition trees with task insertion. Second, we give a framework METHODLEARN to learn HTN methods from HTN instances with an incomplete method set. To reduce the number of methods learned, the method set learned by METHODLEARN is minimal w.r.t. a given prioritization. Third, we take experiments on three well-known domains and compare the percentage of solving new problems on our approach with method sets of different incompleteness and a classical learning approach, HTN-MAKER (Hogg *et al.* 2008). The experiment result shows that our approach is effective, especially on solving new HTN problems.

Related Work

Besides those we mentioned above, there have been action model learning approaches related with our work. Garland *et al.* (2001) proposed an approach to construct and maintain hierarchical task models from a set of annotated examples provided by domain experts. Similar to the annotated tasks, obtaining these annotated examples is difficult and needs a lot of human effort. Our work also is related to the works on learning the precondition of HTN methods (Ilghami *et al.* 2005; Xu and Muñoz-Avila 2005), which take the hierarchical relationships between tasks, the action models, and a complete description of the intermediate states as input. The similar work also includes (Nejati *et al.* 2006) and (Reddy and Tadepalli 1997), which used means-end analysis to learn structures and preconditions of the input plans. The precondition and effect of primitive actions can also be learned in (Zhuo *et al.* 2009). All these approaches to learning the precondition of methods require a complete method set as input.

The work on hybrid planning which combines classical planning and HTN planning is also related with our work. By relaxing the restriction of generating plans only via decomposition, Geier and Bercher (2011) proposed propositional TIHTN planning which allows to inserting primitive tasks to obtain executable plans. Later Alford *et al.* (2015) generalized it into lifted TIHTN planning by allowing variables in actions and predicates. In this paper, we focus on HTN planning and aim to learn HTN methods with the help of TIHTN planning.

Problem Definition

We adapt the definitions of propositional HTN planning (Geier and Bercher 2011). For a propositional language $\mathcal{L}$, a state is a subset of the propositions in $\mathcal{L}$. In HTN planning, actions[1], noted $\mathcal{A}$, are classified into two categories: the actions the agent can execute directly are called *primitive actions* or *operators*, noted $\mathcal{O}$, while the rest are called *compound actions*, noted $\mathcal{C}$. Every primitive action o is a tuple $(\mathsf{pre}(o), \mathsf{add}(o), \mathsf{del}(o))$ where $\mathsf{pre}(o)$ is a conjunction of literals called its precondition; $\mathsf{add}(o)$ and $\mathsf{del}(o)$ are sets of propositional symbols called its positive and negative effect. A primitive action o is *applicable* in a state s if $s \models \mathsf{pre}(o)$, which results in a state $\gamma(s,o)=(s \setminus \mathsf{del}(o)) \cup \mathsf{add}(o)$. A sequence of primitive actions $o_1,\ldots,o_n$ is *executable* in a state s_0 iff there is a state sequence $s_1,\ldots,s_n$ such that $\forall_{1 \le i \le n}, \gamma(s_{i-1}, o_i)=s_i$ and o_i is applicable in s_{i-1}.

Given a set R, we use $\overline{R}$ to denote the set of all sequences over R and use $|R|$ to denote the cardinality of R. For its

[1]"Action" is also called "task name".

subset X and a function $f : R \longrightarrow S$, its restriction to X is $f|_X = \{(r,s) \in f \mid r \in X\}$. For a binary relation $Q \subseteq R \times R$, we define its restriction to X by $Q|_X = Q \cap (X \times X)$.
Task networks. A task network is a tuple $\mathsf{tn}=(T, \prec, \alpha)$ where T is a set of tasks, $\prec \subseteq T \times T$ is a set of ordering constraints over T and $\alpha : T \longrightarrow \mathcal{A}$ labels every task with an action.

Every task is associated to an action and the ordering constraints restrict the execution order of tasks. A task t is called *primitive* if $\alpha(t)$ is primitive, otherwise called *compound*. A task network is called *primitive* iff it contains only primitive tasks.

We say two task networks $\mathsf{tn} = (T, \prec, \alpha)$ and $\mathsf{tn}' = (T', \prec', \alpha')$ are *isomorphic*, denoted by $\mathsf{tn} \cong \mathsf{tn}'$, if and only if there exists a bijection $f : T \longrightarrow T'$ such that for all $t_1, t_2 \in T$, $t_1 \prec t_2$ iff $f(t_1) \prec' f(t_2)$ and $\alpha(t_1) = \alpha'(f(t_1))$, $\alpha(t_2) = \alpha'(f(t_2))$.
HTN methods. Compound actions cannot be directly executed and need to be decomposed into a task network according to HTN methods. Each *HTN method* $m=(c, \mathsf{tn}_m)$ consists of a compound action c and a task network tn_m whose inner tasks are called *subtasks*. Note that a compound action c may have more than one decomposition method.

In a task network, the decomposition is done by selecting a compound task, adding its subtask network and replacing it. The constraints about the decomposed task t are propagated to its subtasks: the tasks before t are before all its subtasks and the tasks after t are after all its subtasks.
HTN problems. An HTN planning *domain* is a tuple $\mathfrak{D} = (\mathcal{L}, \mathcal{O}, \mathcal{C}, \mathcal{M})$ where $\mathcal{M}$ is a set of decomposition methods and $\mathcal{O} \cap \mathcal{C} = \emptyset$. We call a pair (s_0, t_0) an *instance* where s_0 is the initial state and t_0 is the initial task. An HTN *problem* is a tuple $\mathcal{P} = (\mathfrak{D}, s_0, t_0)$.

In different literature, the solution to the HTN problem has different forms: mostly a plan (such as (Erol *et al.* 1994)), a primitive task network (such as (Behnke *et al.* 2017)) and a list of decomposition trees (such as (Zhuo *et al.* 2014)). In this paper, we consider a solution to the HTN problem as a decomposition tree rooted in the initial task t_0.

A decomposition tree is a tuple $\mathcal{T} = (T, E, \prec, \alpha, \beta)$ where (T, E) is a tree, with nodes T and with directed edges $E : T \longrightarrow \overline{T}$ mapping each node to an ordered list of its children; $\prec$ is a set of constraints over T; function $\alpha : T \longrightarrow \mathcal{A}$ links tasks and actions; function $\beta : T \longrightarrow \mathcal{M}$ labels every inner node with a decomposition method.

We use $\ll$ to denote the transitive closure of $\prec$ and the order defined by E. We say t_1 is a predecessor of t_2 if $t_1 \ll t_2$. Dually, we also say t_2 is a successor of t_1. According to $\ll$, we say the sequence constituted by the leaf nodes of $\mathcal{T}$ is its plan, denoted by $\vartheta(\mathcal{T})$.

Definition 1 (**Valid decomposition trees**). *A decomposition tree $\mathcal{T}$ is valid w.r.t. an HTN problem $\mathcal{P} = (\mathfrak{D}, s_0, t_0)$ iff its plan $\vartheta(\mathcal{T})$ is executable in s_0 and its root is t_0 and for every inner node t where $\beta(t) = (c, \mathsf{tn}_m)$, it satisfies:*

1. *$\alpha(t) = c$;*
2. *$(E(t), \prec|_{E(t)}, \alpha|_{E(t)}) \cong \mathsf{tn}_m$;*
3. *if $(t, t') \in \prec$ then for every $st \in E(t)$, $(st, t') \in \prec$;*
4. *if $(t', t) \in \prec$ then for every $st \in E(t)$, $(t', st) \in \prec$;*
5. *there are no t_1, t_2 such that $t_1 \ll t_2$ and $t_2 \ll t_1$.*

Solutions. A solution to an HTN problem $\mathcal{P} = (\mathfrak{D}, s_0, t_0)$ is a valid decomposition tree $\mathcal{T}$ w.r.t. $\mathcal{P}$ and we say (s_0, t_0) is solved under $\mathfrak{D}$ and is satisfied by $\mathcal{T}$.

Example 2 (Example 1 cont.). *If plane1 is already at airport A in s_0, the decomposition tree drawn with **black** arrows shown in Figure 1 is a solution to the HTN problem. $\sigma_1 = \langle$load;drive;unload;load;fly;unload;load;drive;unload$\rangle$ is its plan.*

Method Learning. In this paper, we assume that the original methods are kept as they come from the expert knowledge and they are sound in some situations. So, we only consider adding methods into the original domain. For an HTN domain $\mathfrak{D} = (\mathcal{L}, \mathcal{O}, \mathcal{C}, \mathcal{M})$ and a method set $\mathcal{M}'$, we use $\mathfrak{D}+\mathcal{M}' = (\mathcal{L}, \mathcal{O}, \mathcal{C}, \mathcal{M} \cup \mathcal{M}')$ to denote the resulting domain by adding $\mathcal{M}'$ into $\mathfrak{D}$.

An HTN method learning problem is defined as a tuple $(\mathfrak{D}, \mathcal{I})$ where $\mathfrak{D}$ is an HTN domain and $\mathcal{I}$ is a set of instances. A solution of the HTN method learning problem is a set of methods $\mathcal{M}'$ which should satisfy:

- all instances in the set $\mathcal{I}$ are solved under $\mathfrak{D}+\mathcal{M}'$;
- the learned method set $\mathcal{M}'$ is as minimal as possible;
- the learned methods in $\mathcal{M}'$ have as little inserted subtasks as possible.

Refining Methods via Task Insertion

In this paper, we focus on the HTN problem with an incomplete method set, where there is no valid decomposition tree w.r.t. the problem. In other words, there is no executable plan obtained only by applying methods. By allowing inserting tasks, (Geier and Bercher 2011) proposes a hybrid planning formalization, TIHTN planning. A solution to the TIHTN problem is a TIHTN plan which is a primitive action sequence executable in the initial state and includes all primitive tasks obtained by applying methods and inserted primitive tasks. (Alford *et al.* 2015) gives a progression policy for TIHTN planning and it is not difficult to design a progression-based algorithm to find a TIHTN plan and a decomposition tree which excludes inserted primitive tasks.

Actually, the inserted tasks in the TIHTN plan are subtask candidates: they provide clues for refining the original methods by adding them as subtasks. Then, based on a TIHTN plan, we propose the completion profile to refine methods and complete decomposition trees.

Inspired by (Alford *et al.* 2015), we propose a progression-based algorithm to search TIHTN plans in Algorithm 1. First, we say a task is unconstrained in the current state if all its predecessors have been done and use uncons to denote the set of unconstrained tasks in the current state. In every step, we choose non-determinitiscally an unconstrained task to perform or decompose (line 4), where performance updates the state (line 9) and decomposition updates the tree (line 14-15). Once a task is performed or decomposed, it is labelled as ‘done’ (line 16). If the precondition of the primitive task chosen is not satisfied in the current

```
Algorithm 1: HPLAN($\mathfrak{D}, s_0, t_0$)
   input : An HTN domain $\mathfrak{D}$ and an instance $(s_0, t_0)$
   output: A decomposition tree $\mathcal{T}$ and a plan $\sigma$
1  $s \leftarrow s_0$;     $\sigma \leftarrow \emptyset$;
2  $\mathsf{uncons} \leftarrow T \leftarrow t_0$;     $E \leftarrow \emptyset$;
3  while $\mathsf{uncons} \neq \emptyset$ do
4      choose non-deterministically some $t \in \mathsf{uncons}$;
5      if $t$ is primitive then
6          if $s \not\models \mathsf{pre}(\alpha(t))$ then
7              find a plan $\sigma'$ to $s'$ where $s' \models \mathsf{pre}(\alpha(t))$;
8              $\sigma \leftarrow \sigma \circ \sigma'$;
9              $s \leftarrow s'$;
10         $\sigma \leftarrow \sigma \circ \alpha(t)$;
11         $s \leftarrow \gamma(s, \alpha(t))$;
12     else
13         choose non-deterministically
             $m = (c, \mathsf{tn}_m) \in \mathcal{M}$ s.t. $\alpha(t) = c$;
14         $T \leftarrow T \cup T_m$ s.t. $\mathsf{tn}_m = (T_m, \prec, \alpha_m)$;
15         $E \leftarrow E \cup \{t \times T_m\}$;
16     label $t$ is done;
17     update $\prec$ and $\mathsf{uncons}$ in $T$;
18     if all $t \in T$ are done then
19         Return $\sigma$ and $\mathcal{T}$
20 Return fail
```

state, it searches a plan to satisfy it (line 7) via an off-the-shelf planner, FF planner, which actually is a classical planning problem. When all tasks are labelled as done, it returns a TIHTN plan and a decomposition tree excluding inserted tasks.

Example 3 (Example 2 cont.). *If plane1 is not at airport A in s_0, the decomposition tree in Example 2 is not valid as its plan σ_1 is not executable in s_0. While $\sigma_2 = \langle$load;drive;unload;fly;load;fly;unload;load;drive;unload$\rangle$ is a TIHTN plan to the problem.*

Refining Methods and Completing Decomposition Trees

Suppose the TIHTN planner outputs a plan σ and a decomposition tree $\mathcal{T}$, we use I_σ to denote all the inserted tasks in σ. The TIHTN plan actually is an ordering of primitive tasks and we extend the $\ll$ relation of $\mathcal{T}$ by considering the execution order of primitive actions in σ. To get the compound tasks, we use $N_\mathcal{T}$ to denote the inner nodes of the decomposition tree $\mathcal{T}$. Next, we show how to link these inserted tasks with the inner nodes $N_\mathcal{T}$ of the decomposition tree $\mathcal{T}$ to generate a new decomposition tree.

Definition 2. *We define a completion profile as a function $\rho : I_\sigma \longrightarrow N_\mathcal{T}$, such that for every inserted task $t' \in I_\sigma$ there is not a primitive task $t_p \in \sigma$ where either both $t_p \ll \rho(t')$ and $t' \ll t_p$, or $\rho(t') \ll t_p$ and $t_p \ll t'$.*

Intuitively, every inserted task is associated with a compound task as its subtask. Every inserted task is restricted to be performed before the predecessors and after the successors of its corresponding compound task.

Next, we define how to refine a method by inserting tasks. A completion profile leads to a set of refined methods by adding the relevant inserted tasks into the original methods. Formally, for a completion profile ρ, let t be an inner node in the decomposition tree, we use $T^t_\rho = \{t' \mid \rho(t') = t\}$ to denote all inserted tasks associated with t. Then we use $T(\rho)$ to denote the range of function ρ, i.e., the inner nodes which have a non-empty set T^t_ρ. The inserted subtasks with the original subtasks of t compose a new subtask network, written by $\mathsf{tn}^t_\rho = (T^t_\rho, \ll|_{T^t_\rho}, \alpha_\sigma)$, where α_σ is the function α from the plan σ. Every non-empty set T^t_ρ leads to a refined method $m^t_\rho = (c, (T_m \cup T^t_\rho, \prec_m \cup \ll|_{T^t_\rho}, \alpha_m \cup \alpha_\sigma))$ w.r.t. the original method $\beta(t) = m = (c, (T_m, \prec_m, \alpha_m))$. We use $\mathcal{M}_\rho$ to denote the set of refined methods from the completion profile ρ.

Example 4 (Example 3 cont.). *For the TIHTN plan σ_2 and the decomposition tree in Example 1, we have a completion profile ρ where $\rho(t_1) = \mathsf{airShip}$ and $\alpha(t_1) = \mathsf{fly(plane1, airpA)}$. The refined method is $(\mathsf{airShip}, (T'_m, \prec'_m, \alpha'_m))$ where $T'_m = \{\mathsf{fly, load, fly, unload}\}$.*

The completion profile actually completes the decomposition tree: the inserted tasks are connected with their corresponding inner nodes as their children. When we add new nodes into the decomposition tree, the integrity of ordering constraints will be destroyed. To avoid that, we define an operator closure to complete the ordering constraints. Formally, for a tree $\mathcal{T} = (T, E)$, we define its closure on the ordering constraint $\prec$ as $\mathsf{closure}(T, E, \prec)$, which is given by:

$$\prec \cup \bigcup_{t \in T} \{(t', ch), (ch, t'') \mid ch \in E(t), t' \prec t, t \prec t''\}.$$

Intuitively, the closure operation completes the ordering constraints about the children which should be inherited from their parent.

Next we show how to complete the decomposition tree according to the completion profile.

We define the completion of the decomposition tree $\mathcal{T}$ by completion profile ρ w.r.t. TIHTN plan σ as $\mathcal{T}_\rho = (T', E', \prec', \alpha', \beta')$, which is given by:

$$T' := T \cup \textstyle\bigcup_{t \in T(\rho)} T^t_\rho$$

$$E' := E \cup \{(t, st) \mid t \in T, st \in T(\mathsf{tn}^t_\rho)\}$$

$$\prec' := \mathsf{closure}(T', E', \prec) \cup \textstyle\bigcup_{t \in T(\rho)} \ll|_{T^t_\rho}$$

$$\alpha' := \alpha \cup \alpha_\sigma$$

$$\beta' := (\beta \setminus \{(t, m) \mid t \in T(\rho)\}) \cup \{(t, m^t_\rho) \mid t \in T(\rho)\}$$

The procedure of completing a decomposition tree consists of first connecting the inserted tasks with the inner nodes, then completing the ordering constraints and finally updating the method applied as the refined method. The decomposition tree being completed will satisfy the instance:

Proposition 1. *Given an HTN problem $\mathcal{P} = (\mathfrak{D}, s_0, t_0)$, let σ be one of its TIHTN plans and $\mathcal{T}$ be its corresponding decomposition tree and ρ be one of their completion profiles. Then the completed decomposition tree $\mathcal{T}_\rho$ satisfies the instance (s_0, t_0) under the new domain $\mathfrak{D}$+$\mathcal{M}_\rho$.*

Proof. First, we show that $\mathcal{T}_\rho$ is a valid decomposition tree w.r.t. $\mathfrak{D}$+$\mathcal{M}_\rho$. For every node t in $\mathcal{T}_\rho$ with $\beta'(t) = (c, \mathsf{tn}_\rho)$, i) the function α is not reduced, so $\alpha'(t) = c$; ii) the edges between the task t and its inserted tasks T^t_ρ are added, so the task network induced by its children is isomorphic with m^t_ρ; iii) $\mathsf{closure}(T', E', \prec)$ guarantees that all ordering constraints of t are propagated to the inserted tasks and $\ll|_{T^t_\rho}$ only introduces the ordering constraints among the inserted subtasks in the same method, so conditions 3. and 4. are satisfied; iv) as the completion profile guarantees that no contradict pair about $\ll$ is introduced, condition 5. is satisfied.

Without removing nodes, the root of $\mathcal{T}_\rho$ is still t_0. As the plan $\vartheta(\mathcal{T}_\rho)$ is the TIHTN plan σ executable in s_0, $\mathcal{T}_\rho$ satisfies the instance (s_0, t_0). □

When an HTN problem has incomplete methods, the completion profile offers a way to improve the HTN domain:

Theorem 1. *If an HTN problem $\mathcal{P}$=$(\mathfrak{D}, s_0, t_0)$ has a TIHTN plan but no solution, then there is a completion profile ρ where the HTN problem $\mathcal{P}' = (\mathfrak{D}$+$\mathcal{M}_\rho, s_0, t_0)$ is solvable.*

Proof. Let σ be a TIHTN plan of $\mathcal{P}$ with its decomposition tree $\mathcal{T}$. Suppose ρ is a completion profile w.r.t. σ and $\mathcal{T}$. By Proposition 1, the decomposition tree $\mathcal{T}_\rho$ satisfies the instance (s_0, t_0) under the new domain $\mathfrak{D}$+$\mathcal{M}_\rho$. So, σ is a solution of the HTN problem $\mathcal{P}'$. □

When the completion profile only add decomposition methods, we have a corollary:

Corollary 2. *Every plan of the HTN problem $\mathcal{P} = (\mathfrak{D}, I)$ is also a plan of the HTN problem $\mathcal{P}' = (\mathfrak{D}$+$\mathcal{M}', I)$.*

Proof. As the original methods are still in the domain, the valid decomposition trees of the original problem $\mathcal{P}$ are also valid decomposition trees of the new HTN problem $\mathcal{P}'$. So, plans of $\mathcal{P}$ are also plans of $\mathcal{P}'$. □

Prioritized Preferences

To formalize the experience that the missing of subtasks happens more likely on some methods than other methods, we consider a priority on the methods. Generally, the priority comes from the confidences of domain experts on methods: the method believed to lack subtasks more likely to have a higher priority.

Given a method set $\mathcal{M}$, we define a prioritization as a partition on it: P=$\langle P_1, ..., P_n\rangle$ where $\bigcup_{1\leq j\leq n} P_j = \mathcal{M}$. Intuitively, the decomposition methods in P_i have a higher priority to be refined than those in P_j if $i > j$. We further consider the prioritized preference in terms of cardinality.

Given a prioritization P=$\langle P_1, ..., P_n\rangle$ of $\mathcal{M}$, we consider the prioritized preference $\leq_P$ as follows: for $\mathcal{M}_1$,$\mathcal{M}_2 \subseteq \mathcal{M}$, if there is some $1 \leq i \leq n$ such that

- $|\mathcal{M}_1 \cap P_i| \leq |\mathcal{M}_2 \cap P_i|$ and
- for all $1 \leq j < i$, $|\mathcal{M}_1 \cap P_j| = |\mathcal{M}_2 \cap P_j|$,

then we write $\mathcal{M}_1 \leq_P \mathcal{M}_2$. We say $\mathcal{M}_1$ is strictly preferred over $\mathcal{M}_2$ w.r.t. P, written by $\mathcal{M}_1 <_P \mathcal{M}_2$, if $\mathcal{M}_1 \leq_P \mathcal{M}_2$ and $\mathcal{M}_2 \not\leq_P \mathcal{M}_1$.

Preferred Completion Profiles

Generally, we hope to find a completion profile changing the original methods minimally under the prioritized preference.

We first define some notations: for a refined method m^t_ρ, we use $\tau(m^t_\rho)$ to denote its original method m. For a refined method set $\mathcal{M}'$, we use $\tau(\mathcal{M}')$ to denote all the original methods of the refined methods in $\mathcal{M}'$, i.e., $\tau(\mathcal{M}') = \{m \in \mathcal{M} |\ m = \tau(m'), m' \in \mathcal{M}'\}$. Note that several completions may be associated with the same decomposition method. For two decomposition methods m'_1 and m'_2, if $\tau(m'_1) = \tau(m'_2)$, we say m'_1 and m'_2 are *homologous*.

Definition 3. *Given a TIHTN plan and its decomposition tree, a completion profile ρ is preferred w.r.t. preference P if there is not a completion profile ρ', such that $\tau(\mathcal{M}_{\rho'}) <_P \tau(\mathcal{M}_\rho)$.*

Intuitively, the preferred completion profile refines methods minimally under the prioritized preference.

Next, we will show how to find the preferred completion profile, as shown in Algorithm 2. First, we consider all inserted tasks in the plan as unlabelled (line 1). Then we scan all inner nodes from the nodes with a method of higher priority to the nodes with a method of lower priority (line 2-3). Next, for an inner node, we find the set of candidate subtasks Δ_t from the inserted tasks, which do not violate the ordering constraints if they were inserted as its subtasks (line 5). More specially, for the inner node t, the inserted tasks which are executed between the last task required to be executed ahead of t and the first task required to be after t, are allowed to be added as subtasks of t. According to the total order + in the decomposition tree, we define the subtasks candidate set Δ_t of t as the set of the unlabelled inserted tasks between the last predecessor of t and the first successor of t. Finally, we associate all tasks in the subtask candidate set to t (line 5) and label them as subtasks (line 6). When all inserted tasks are labelled, it returns a preferred completion profile. It must terminate and the worst case is that the inserted tasks are associated with the root task.

Algorithm 2 only scan the nodes of the decomposition tree once and searching the subtask candidate set can be done in linear time, so the algorithm terminates in polynomial time.

Algorithm 2: COMPLETE$(\sigma, \mathcal{T}, P)$

```
  input : A TIHTN plan $\sigma$, its decomposition tree $\mathcal{T}$
          and a prioritization $P = (P_1, ..., P_n)$ on $\mathcal{M}$
  output: A completion profile $\rho$
1 $I \leftarrow I_\sigma$;
2 for $j \leftarrow n$ to $1$ do
3     for each $t \in N_\mathcal{T}$ s.t. $\beta(t) \in P_j$ do
4         if $I \neq \emptyset$ then
5             for every $t' \in \Delta_t \cap I$, set $\rho(t') = t$;
6             $I \leftarrow I \setminus \Delta_t$;
7 return $\rho$
```

Actually, to find a preferred completion profile, we only need to scan the inner nodes in the decomposition tree according to the preference and link appropriate inserted tasks with inner nodes, which can be done in polynomial time.

Observe that the more detailed tasks are more sensitive to these situations and more easily to be thoughtless. There exists a class of HTN domains where actions can be stratified according to the decomposition hierarchy (Erol *et al.* 1996; Alford *et al.*). In this case, we assume that an action is more abstract than its subtasks and we consider a preference in terms of a stratum-based priority: the more abstract actions have a lower priority to be refined.

Learning Methods from Instances

As stated above, we only consider to introduce new methods into the HTN domain. However, an excess of methods introduced may slow down problem-solving significantly, as there are excessive choices to decompose tasks. To reduce the number of methods learned, we consider the minimal set of methods learned under the prioritized preference. We propagate the prioritized preference to the refined methods: if $\tau(m') \in P_j$ then $m' \in P_j$.

Definition 4. *Given a method set $\mathcal{M}'$ and its prioritization P, a subset $\mathcal{M}'_0$ of $\mathcal{M}'$ is the minimal set w.r.t. P if there is not a subset $\mathcal{M}'_1$ of $\mathcal{M}'$ such that $\mathcal{M}'_1 <_P \mathcal{M}'_0$.*

To learn as few methods as possible, we first compute a preferred completion profile w.r.t. the stratum-based prioritized preference and then compute the minimal method set.

Suppose $\mathcal{M}'$ is a set of methods learned, we use $\mathcal{I}(\mathcal{M}')$ to denote the solvable subset of $\mathcal{I}$ w.r.t. $\mathfrak{D}$+$\mathcal{M}'$. Furthermore, we use $\overline{\mathcal{T}}(\mathcal{M}')$ to denote the set of decomposition trees w.r.t. $\mathfrak{D}$+$\mathcal{M}'$. The decomposition trees and the instances solved are monotonic w.r.t. the methods learned:

Proposition 2. *If $\mathcal{M}_1 \subseteq \mathcal{M}_2$, then $\overline{\mathcal{T}}(\mathcal{M}_1) \subseteq \overline{\mathcal{T}}(\mathcal{M}_2)$ and $\mathcal{I}(\mathcal{M}_1) \subseteq \mathcal{I}(\mathcal{M}_2)$.*

Proof. When $\mathcal{M}_1 \subseteq \mathcal{M}_2$, it means that there are more methods to be chosen to decompose compound tasks, in consequence there will be more decomposition trees generated.

As every HTN plan comes from the decomposition tree, if an instance $i \in \mathcal{I}(\mathcal{M}_1)$ has an HTN plan, then it has a decomposition tree $\mathcal{T}^i$ satisfying it, entailing that $\mathcal{T}^i \in \overline{\mathcal{T}}(\mathcal{M}_2)$. Thus, the instance i also in $\mathcal{I}(\mathcal{M}_2)$. □

Method substitution. The completion profiles from various instances may induce many refined methods which decompose the same compound action and generate similar executable plans. The vast increase in the number of methods will slow down the problem-solving significantly and we need to reduce the redundant refined methods which can be replaced by other methods. To compute the minimal set, we need to remove the redundant refined methods and define a method substitution operator.

Definition 5. *Given a decomposition tree $\mathcal{T}$=$(T, E, \prec, \alpha, \beta)$, let T_{m_1} be the inner nodes with method m_1 and $m_2 = (c, (T_2, \prec_2, \alpha_2))$ be a homologous method with m_1. We define the decomposition tree that substitutes the method m_1 in $\mathcal{T}$ with m_2 as $\mathsf{sub}(\mathcal{T}, t, m') = (T', E', \prec', \alpha', \beta')$, given by:*

$$\begin{aligned}
T' &:= (T \setminus \textstyle\bigcup_{t\in T_{m_1}} T_t^{\mathsf{add}}(m_1)) \cup \bigcup_{t\in T_{m_1}} T_t^{\mathsf{add}}(m_2)\\
E' &:= E|_{T'} \cup \textstyle\bigcup_{t\in T_{m_1}} (\{t\} \times T_t^{\mathsf{add}}(m_2))\\
\prec' &:= \mathsf{closure}(T', E', \prec \cup \prec_2)\\
\alpha' &:= \alpha'|_{T'} \cup \alpha_2\\
\beta' &:= (\beta \setminus \{(t, m_1) | t \in T_{m_1}\}) \cup \{(t, m_2) | t \in T_{m_1}\}
\end{aligned}$$

where $T_t^{\mathsf{add}}(m)$ denotes the inserted subtasks w.r.t. t for the refined method m.

After substituting a method m_1 with another homologous method m_2, if the resulting decomposition tree still satisfies the instance, it means that for this instance, the replaced method m_1 is redundant and can be replaced by m_2.

Proposition 3. *For two homologous methods m_1, m_2, let $\mathcal{T}' = \mathsf{sub}(\mathcal{T}, m_1, m_2)$. If $\mathcal{T}$ satisfies an instance (s_0, t_0) and $\vartheta(\mathcal{T}')$ is executable in s_0, then $\mathcal{T}'$ satisfies (s_0, t_0).*

Proof Sektch. It is not difficult to prove $\mathcal{T}'$ is a valid decomposition tree, which entails that it satisfies the instance. □

Next, we generalize the notion of substitution into the decomposition tree set: given a decomposition tree set $\overline{\mathcal{T}}$ and two method sets $\mathcal{M}'_1, \mathcal{M}'_2$, we define the set of the decomposition trees that substitutes every occurrence of every method m'_1 in $\mathcal{M}'_1$ with some method m'_2 in $\mathcal{M}'_2$ which is homologous with m'_1, written by $\mathsf{sub}(\overline{\mathcal{T}}, \mathcal{M}'_1, \mathcal{M}'_2)$. With the substitution operator, we can reduce the refined methods:

Proposition 4. *Given an HTN domain $\mathfrak{D}$ and an instance set $\mathcal{I}$, let $\overline{\mathcal{T}}$ be its decomposition tree set, each tree of which satisfies its corresponding instance. For a refined method set $\mathcal{M}'$ and its subset $\mathcal{M}'_j$, if the plan of every decomposition tree in $\mathsf{sub}(\overline{\mathcal{T}}, \mathcal{M}', \mathcal{M}'_j)$ is executable in the corresponding initial state, then $\mathcal{I}(\mathcal{M}') = \mathcal{I}((\mathcal{M}' \setminus \mathcal{H}(\mathcal{M}'_j)) \cup \mathcal{M}'_j)$ where $\mathcal{H}(\mathcal{M}'_j)$ is the set of methods homologous with the methods in $\mathcal{M}'_j$.*

Proof. By Proposition 3, for every instance $i = (s_0^i, t_I^i)$ in $\mathcal{I}$ and its decomposition tree $\mathcal{T}^i \in \overline{\mathcal{T}}$, if $\vartheta(\mathsf{sub}(\mathcal{T}^i, \mathcal{M}', \mathcal{M}'_j))$ is executable in s_0^i, then it satisfies i. When the methods in $(\mathcal{H}(\mathcal{M}'_j) \setminus \mathcal{M}'_j)$ are substituted, every instance is satisfied w.r.t. the remaining methods. □

Theorem 3. *Given a method set $\mathcal{M}'$ and its prioritization P, there exists a minimal subset $\mathcal{M}'_0$ of $\mathcal{M}'$ w.r.t. P such that $\mathcal{I}(\mathcal{M}'_0) = \mathcal{I}(\mathcal{M}')$.*

Proof. As $\mathcal{M}'$ is finite, the minimal subset exists and in the worst case, $\mathcal{M}'$ itself is minimal. □

Next we give an algorithm for the HTN method learning problem, as shown in Algorithm 3. The framework consists of two main components: the first iteration for learning methods by refining methods (line 2-8) and the second iteration for reducing refined methods (line 9-11). The first iteration first finds a TIHTN plan and its decomposition tree for every instance (line 3) by HPLAN and then computes

Algorithm 3: METHODLEARN($\mathfrak{D}, \mathcal{I}, P$)

```
input  : An HTN domain $\mathfrak{D}$, an instance set $\mathcal{I}$ and a
         prioritization $P = (P_1, ..., P_n)$ on $\mathcal{M}$
output: A new method set $\mathcal{M}'$
1  $\mathcal{M}' \leftarrow \emptyset$;   $\overline{\mathcal{T}} \leftarrow \emptyset$;
2  for each $i$ in $\mathcal{I}$ do
3      compute a plan and decomposition tree
         $(\sigma^i, \mathcal{T}^i) = $ HPLAN$(\mathfrak{D}, i)$;
4      $\rho^i = $ COMPLETE$(\sigma^i, \mathcal{T}^i, P)$;
5      complete the decomposition tree $\mathcal{T}^i$ to $\mathcal{T}^i_\rho$ by $\rho^i$;
6      $\overline{\mathcal{T}} \leftarrow \overline{\mathcal{T}} \cup \mathcal{T}^i_\rho$;
7      construct a new method set $\mathcal{M}^i_\rho$ from $\rho^i$;
8      $\mathcal{M}' \leftarrow \mathcal{M}' \cup \mathcal{M}^i_\rho$;
9  for $j \leftarrow 1$ to $n$ do
10     compute the minimal subset $\mathcal{M}'_j$ of $P_j[\mathcal{M}']$ s.t.
         every tree in sub$(\overline{\mathcal{T}}, \mathcal{M}', \mathcal{M}'_j)$ satisfies their
         corresponding instance;
11 $\mathcal{M}' \leftarrow \bigcup_{1 \le j \le n} \mathcal{M}'_j$;
12 return $\mathcal{M}'$
```

each preferred completion profile (line 4) by COMPLETE. Next, these decomposition trees are completed and a set of refined methods are constructed according to the completion profiles (line 5-8).

To reduce the refined methods, if the constants in the inserted subtasks are identical with the arguments in the subtasks and compound task, we use the corresponding variables to replace the constants in the inserted subtasks (line 8).

In the second iteration, we use a greedy strategy to find the minimal set: the refined methods with lower priority are reduced first, which is the opposite against the procedure of searching the preferred completion profile. Here we use $P_j[\mathcal{M}']$ to denote the refined methods in $\mathcal{M}'$ whose original methods are in the priority P_j. By Proposition 4, the refine methods in $P_j[\mathcal{M}']$ can be replaced by $\mathcal{M}'_j$ for every instance and $\mathcal{M}'_j$ is minimal in the priority P_j.

To pursue the refined methods with as few subtasks as possible, we take a breadth-first strategy to find the inserted tasks when computing TIHTN plans (line 7 in Algorithm 1).

In fact, our approach outputs a method set which sometimes may be a second-best solution for criterion 2 and 3 of the solutions, while it satisfies criterion 1:

Theorem 4. *Suppose $\mathcal{M}'$ is the method set learned by* METHODLEARN$(\mathfrak{D}, \mathcal{I}, P)$*, if every instance in $\mathcal{I}$ has a TIHTN plan under the domain $\mathfrak{D}$, then it also has a solution under the domain $\mathfrak{D}$+$\mathcal{M}'$.*

Proof. As every instance has a TIHTN plan, by Proposition 1, there exists a set of decomposition trees $\overline{\mathcal{T}}$, each of which satisfies each instance w.r.t. the domain $\mathfrak{D}$+$\mathcal{M}''$ where $\mathcal{M}''$ is a method set obtained via completion profiles (line 2-8 in Algorithm 3). Then $\mathcal{M}' \subseteq \mathcal{M}''$. By Proposition 4, each decomposition tree in sub$(\overline{\mathcal{T}}, \mathcal{M}'', \mathcal{M}')$ satisfies its corresponding instance in $\mathcal{I}$. Thus, every instance is solvable under the new domain $\mathfrak{D}$+$\mathcal{M}'$. □

Experimental Analysis

We have implemented Algorithm 3 based on Python 3.0 and developed an HTN method learner METHODLEARN. In this section, we evaluate METHODLEARN in three well-known planning domains comparing with HTN-MAKER (Hogg *et al.* 2008) on the learning performances.

We consider three domains which are evaluated on in HTN-MAKER (Hogg *et al.* 2008), *i.e.,* Logistics, Satellite, and Blocks-World, to evaluate our approach. We first get the problem generators from International Planning Competition website[2] and randomly generate 100 instances for each domain and take 75 instances as the training set and 25 instances as the testing set. We run METHODLEARN and HTN-MAKER with 75 instances growingly as input and obtain different learned method sets from these two approaches. The planning instance in the testing set is considered as solved, if its goal is achieved by a plan computed under the learned HTN method set via an HTN planner. For HTN-MAKER we use the well-known HTN planner SHOP2 (Nau *et al.* 2003) and for our approach METHODLEARN we still use Algorithm 1 without task insertion. The time bound of the HTN planner is set to 3600 seconds. In order to check if an instance is solved, we add a verifying action whose precondition is the goal and whose effect is empty in the last subtask of the initial task. The learning performance is measured via the percentage of the number of the solved instances on that of the testing instances, which is called the percentage of instance solved.

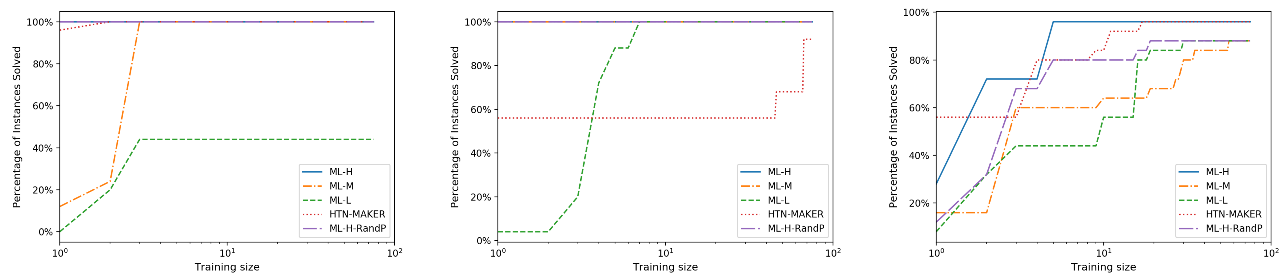
To simulate the incomplete method set as the input of METHODLEARN, we take the HTN domain description in the website[3] of SHOP2, and remove different sets of subtasks. Furthermore, to evaluate the influence of the different incompleteness of the given method sets on the learning performance, we consider three removal cases: 1) remove one primitive task from each method (if exists), with meaning the high completeness, noted by ML-H; 2) remove two primitive tasks from each method (if exists), noted by ML-M, with meaning the middle completeness; 3) remove one more compound task in some method of ML-M, noted by ML-L, with meaning the low completeness.

Consider the method set shown in Figure 1, For ML-H, we remove the first drive and the first fly in the methods cityShip and airShip, respectively, while for ML-M, we remove all drive and fly in the methods. For ML-L, the first cityShip is additionally removed from the method of ship based on the ML-M setting.

Figure 2 shows the learning performances of our approach and HTN-MAKER in the three domains. Generally, with the training set growing, the percentage of the problems solved increases, which does not violate Proposition 2. For the Logistics and Satellite domains, in the settings of ML-H and ML-M, METHODLEARN learns the necessary methods to solve all testing problems from a few instances. It is because the structure of these two domains is relatively straightforward and the decomposition trees still can be constructed by the incomplete method sets. In the ML-L setting, the com-

[2]http://ipc02.icaps-conference.org/

[3]https://www.cs.umd.edu/projects/shop/



(a) The Logistics Domain (b) The Satellite Domain (c) The Blocks-World Domain

Figure 2: The Percentage of Solving Instances on Our Approach with Different Incomplete Method Sets and HTN-MAKER

pound action removed in the Logistics Domain, cityShip, contains more arguments, making the methods learned become more case-specific, which cannot contribute to other instances. For the Blocks-World domain, it cannot achieve the full convergence in each setting. The reason is that there are a few special instances which are significantly different from the training instances, resulting in that the methods learned hardly suit these special testing instances.

To evaluate our assumption on the stratum-based prioritized preference, we also compare it against a random prioritized preference, denoted by ML-H-RandP in Figure 2. When a completion profile associates the inserted tasks to a more abstract tasks, it generates a more case-specific method which may not suit other instances. It is shown that considering the stratum-based prioritized preference leads to a better learning performance.

Discussion and Conclusion

We suppose that in the original method set, every compound action at least has a method to decompose. Our approach also can accept classical planning instances which only have a goal formula: we can trivially introduce a compound action of achieving the goal which is decomposed into a verification action whose precondition is the goal and whose effect is empty. Note that we only invoke a TIHTN planner to obtain plans for refining methods and focus on HTN problems. Also, the TIHTN planner needs to search actions to insert from the vast number of action candidates, a refined method including the missing subtasks helps to find the plan.

To sum up, we present a framework to learn HTN methods from HTN instances by refining methods. We also show that the methods learned by our framework are likely to solve new instances in the same classical planning domain. The experiment results demonstrate that our approach outperforms the traditional method learning approach, HTN-MAKER, given an appropriately incomplete method set as input. It is also illustrated that the stratum-based prioritized preference is effective.

Acknowledgements

This paper was supported by the National Natural Science Foundation of China (No. 61573386), National Key R&D Program of China (No. 2018YFC0830600), Guangdong Province Natural Science Foundation (No. 2016A030313292), Guangdong Province Science and Technology Plan projects (No. 2016B030305007 and No. 2017B010110011), Guangzhou Science and Technology Project (No. 201804010435).

References

[Alford *et al.*] Ronald Alford, Vikas Shivashankar, Ugur Kuter, and Dana S. Nau. HTN problem spaces: Structure, algorithms, termination. In *Proceedings of the 5th Annual Symposium on Combinatorial Search, (SOCS-12)*, pages 2–9.

[Alford *et al.* 2015] Ron Alford, Pascal Bercher, and David W Aha. Tight bounds for HTN planning with task insertion. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1502–1508, 2015.

[Behnke *et al.* 2017] Gregor Behnke, Daniel Höller, and Susanne Biundo. 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)*, pages 20–28. AAAI Press, 2017.

[Behnke *et al.* 2019] Gregor Behnke, Marvin R. G. Schiller, Matthias Kraus, Pascal Bercher, Mario Schmautz, Michael Dorna, Michael Dambier, Wolfgang Minker, Birte Glimm, and Susanne Biundo. Alice in DIY wonderland or: Instructing novice users on how to use tools in DIY projects. *AI Commun.*, 32(1):31–57, 2019.

[Erol *et al.* 1994] Kutluhan Erol, James Hendler, and Dana S Nau. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128, 1994.

[Erol *et al.* 1996] Kutluhan Erol, James Hendler, and Dana S Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.

[Garland *et al.* 2001] Andrew Garland, Kathy Ryall, and Charles Rich. Learning hierarchical task models by defining and refining examples. In *Proceedings of the First International Conference on Knowledge Capture (K-CAP 2001),* *October 21-23, 2001, Victoria, BC, Canada*, pages 44–51. ACM, 2001.

[Geier and Bercher 2011] Thomas Geier and Pascal Bercher. On the decidability of HTN planning with task insertion. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, volume 22, pages 1955–1961, 2011.

[Hogg *et al.* 2008] Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter. HTN-MAKER: learning HTNs with minimal additional knowledge engineering required. In *Proceedings of the 23rd National Conference on Artificial Intelligence, (AAAI-08)*, pages 950–956, 2008.

[Ilghami *et al.* 2005] Okhtay Ilghami, Héctor Muñoz-Avila, Dana S. Nau, and David W. Aha. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the 22nd International Conference on Machine Learning (ICML-05)*, pages 337–344, 2005.

[Kambhampati *et al.* 1998] Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 882–888, 1998.

[Lin *et al.* 2008] Naiwen Lin, Ugur Kuter, and Evren Sirin. Web service composition with user preferences. In *Proceedings of European Semantic Web Conference (EWSC)*, pages 629–643. Springer, 2008.

[Lotinac and Jonsson 2016] Damir Lotinac and Anders Jonsson. Constructing hierarchical task models using invariance analysis. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-16)*, pages 1274–1282, 2016.

[Nau *et al.* 2003] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.

[Nejati *et al.* 2006] Negin Nejati, Pat Langley, and Tolga Könik. Learning hierarchical task networks by observation. In *Proceedings of the 23rd International Conference on Machine Learning (ICML-06)*, pages 665–672, 2006.

[Reddy and Tadepalli 1997] Chandra Reddy and Prasad Tadepalli. Learning goal-decomposition rules using exercises. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML-97)*, pages 278–286, 1997.

[Xu and Muñoz-Avila 2005] Ke Xu and Héctor Muñoz-Avila. A domain-independent system for case-based task decomposition without domain theories. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 234–240, 2005.

[Zhuo *et al.* 2009] 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 *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 1804–1810, 2009.

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

More Succinct Grounding of HTN Planning Problems – Preliminary Results

Gregor Behnke and **Daniel Höller** and **Pascal Bercher** and **Susanne Biundo**
Institute of Artificial Intelligence, Ulm University, D-89069 Ulm, Germany
{gregor.behnke, daniel.hoeller, pascal.bercher, susanne.biundo}@uni-ulm.de

Abstract

Planning systems usually operate on grounded representations of the planning problems during search. Further, planners that use translations into other combinatorial problems also often perform their translations based on a grounded model. Planning models, however, are commonly defined in a lifted formalism. As such, one of the first preprocessing steps a planner performs is to generate a grounded representation. In this paper we present a new approach for grounding HTN planning problems that produces smaller groundings than the previously published method. We expect this decrease in size to lead to more efficient planners.

1 Introduction

Most modelling languages for planning problems (such as PDDL (McDermott 2000)) allow for specifying planning problems in a *lifted* fashion, e.g. by allowing the modeller to specify actions with parameters whose preconditions and effects are specified using literals referring to these parameters. Using a lifted representation, a modeller can easily write models with a large number of instantiated actions without the need to enumerate them explicitly. More importantly, a lifted representation of the planning problem enables the modeller to specify a single planning domain that can be used in multiple planning problems without any change to the model. In a grounded formalism, the domain (e.g. the set of actions) changes depending on the planning problem at hand, while it does not in a lifted representation.

Unfortunately, to plan directly using only the lifted model is rather difficult, which is witnessed by the absence of a large body of work e.g. on lifted heuristics. Most planners transform the lifted representation of the planning problem they receive as an input into a grounded representation before planning. Planning is then performed on the grounded representation, for which heuristics are readily available. Naively grounding the lifted representation by simply instantiating all its elements is seldom feasible due to the huge size of the naively grounded model. Instead, the grounding procedure aims to remove as many unnecessary instantiations as possible. Smaller groundings are generally advantageous to planners, as their per-search-node effort decreases and the quality of heuristics can improve. Even small decreases in the size of the grounding can have a huge impact on the efficiency of the planner. As such, grounding is a critical step in the process of planning.

For Hierarchical Task Network (HTN) planning (Bercher, Alford, and Höller 2019), there is – as far as we know – only a single paper explicitly concerned with grounding HTN planning domains (Ramoul et al. 2017). Several other HTN planners plan in a grounded way (e.g. FAPE (Dvorak et al. 2014) and PANDA (Bercher, Keen, and Biundo 2014; Bercher et al. 2017)), but there is no published work about their grounding procedures.

In this paper we report on the grounding procedure used in a variety of systems based on the PANDA framework, e.g. the plan-space-based system (Bercher, Keen, and Biundo 2014; Bercher et al. 2017), the progression-based system (Höller et al. 2018; Höller et al. 2019b), the SAT-based system for totally and partially ordered HTN planning (Behnke, Höller, and Biundo 2018a; 2018b; 2019a; 2019b), and the SAT-based HTN plan verifier (Behnke, Höller, and Biundo 2017). PANDA and its grounder have also already been used in practical applications where a fast grounding procedure is necessary. Notably, we have used it to create plans instructing novice users on how to use electronic tools for DIY home-improvement projects (Behnke et al. 2018; 2019). We start by describing the lifted HTN planning formalism, then give an overview of grounding in planning, describe the grounding procedure used by PANDA, and lastly compare our grounding against the grounding found by GTOHP (Ramoul et al. 2017).

2 Lifted HTN Planning Formalism

Before explaining our HTN grounding procedure, we start by briefly describing the formalism of lifted HTN planning. We have based our formalism on the lifted one by Alford, Bercher, and Aha (2015), which in turn is based on the formalism by Geier and Bercher (2011).

Assume that $\mathcal{L} = (P, T, V, C)$ is a quantifier- and function-free first-order predicate logic with the following elements. P is a finite set of *predicate symbols*. A predicate's arity defines its number of parameter variables (taken from V), each having a certain type (defined in T). T is a finite set of *type symbols*. V is a finite set of typed variable symbols to be used by the parameters of the predicates in P. C is a finite set of typed constants. Based on the predicate logic $\mathcal{L}$, we denote with S the power set of all ground facts

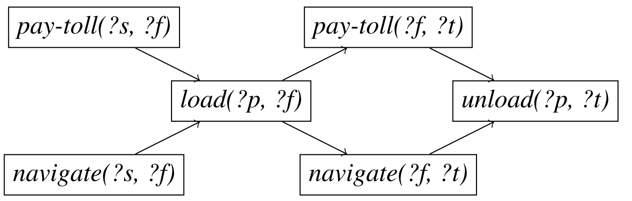


Figure 1: A task network in a simple transportation domain. If performed, it will transport a package from its initial location to its target location. We assume that there is just one transporter. The variables *?s* (start location), *?f* (initial package location), and *?t* (target package location) are of type location. The variable *?p* is of type package. Parallelism between the pay-toll and navigate tasks models that the toll can be paid at any time while the transporter is on its way from the one location to another.

over $\mathcal{L}$.

The most basic data structure in HTN planning is a *task network*. It represents a partially ordered multi-set of tasks. HTN planning distinguishes two types of tasks: primitive and abstract ones. Task networks can contain both primitive and abstract tasks. Each task is identified by its task name and a parameter sequence. For instance, a (primitive) task for driving from a source location $?ls$[1] to a destination location $?ld$ is denoted by the first-order atom *drive(?ls, ?ld)*. We do not differentiate between the expressions *task* and *task names* – both are used synonymously.

Definition 1 (Task Network)**.** *A task network tn over a set of* task names *X (first-order atoms) is a tuple $(I, \prec, \alpha, VC)$ with the following elements:*

1. *I is a finite (possibly empty) set of* task identifiers.
2. *$\prec$ is a strict partial order over I.*
3. *$\alpha : I \to X \times \overline{V}$ maps task symbols to task names and their parameter variables*
4. *VC is a set of variable constraints. Each constraint can bind two task parameters to be (non-)equal or it can constrain a task parameter to be (non-)equal to a constant.*

For simplicity, we only allow variables as arguments for tasks. If it is desired that an argument of a task should be a constant, one can simply introduce a new variable and bind it to the value of the constant in VC. As an example for a task network consider the one shown in Fig. 1.

Task networks can contain primitive and abstract tasks. *Primitive tasks* are identical to actions in classical planning. They are identified via first-order atoms like *drive(?f, ?t)* and are specified via their preconditions *pre* and effects *eff*. For the purposes of this paper, we assume that *pre* is a conjunction of positive first-order literals over $\mathcal{L}$'s predicates and *eff* is a conjunction of (positive and negative) first-order literals. The variables occurring in *pre* and *eff* must be parameters of *name*. Note that for more complex preconditions and effects, this normal form can be achieved via compilation into (potentially) multiple new actions. However a native approach for handling them is usually more efficient.

Abstract tasks are identified by their name and arguments, e.g. *navigate(?f, ?t)*. Their semantics is given in terms of pre-defined means for performing them, which are described by *decomposition methods M*. A decomposition method $m \in M$ is a tuple (c, tn, VC) consisting of an abstract task name c, a task network tn, and a set of variable constraints VC. The variable constraints VC allow to specify (co)designations between the parameters of c and either the variables in the task network tn or constants.

An HTN planning problem consists of the problem's primitive and abstract tasks, all available decomposition methods, the initial state and the initial task network.

Definition 2 (Planning Problem)**.** *A* lifted HTN planning problem *$\mathcal{P}$ is a tuple $(\mathcal{L}, T_P, T_C, M, s_I, tn_I)$, where:*

- *$\mathcal{L}$ is a quantifier- and function-free first-order predicate logic.*
- *T_P and T_C are finite sets of primitive and abstract tasks.*
- *M is a finite set of decomposition methods with abstract tasks from T_C and task networks over the names $T_P \cup T_C$.*
- *$s_I \in S$ is the initial state, i.e., a ground conjunction of positive literals over the predicates assuming closed world assumption.*
- *tn_I is the initial task network, not necessarily ground.*

An HTN planning problem is called ground if all predicates of its predicate logic have arity zero (i.e. they have no parameters).

The aim in an HTN planning problem is to refine a given initial abstract task s_I into an executable, ground, primitive task network. A task network is primitive if all tasks in it are primitive. It is ground if all variables are assigned to constants via variable constraints. It is further executable if there is a linearisation of its tasks that is executable in the initial state. The refinement of the initial task network is performed via repeatedly applying decomposition methods to the abstract tasks contained in it and the resulting task networks. Applying a decomposition method (c, tn_c, VC) to a task network tn means to replace an occurrence of the task c in tn by the contents of the task network tn_c and to add the variable constraints VC to the resulting task network. In addition we have to add variable constraints that co-designate the parameter variables of the abstract task in the method with the actual parameters of the task c that is decomposed inside tn.

Definition 3 (Decomposition)**.** *Let $m = (c(?x_1, \ldots, ?x_n), tn_m)$ with $tn_m = (I_m, \prec_m, \alpha_m, VC_m))$ be a decomposition method, $tn_1 = (I_1, \prec_1, \alpha_1, VC_1)$ a task network. We assume that $I_m \cap I_1 = \emptyset$ and that the sets of variables occurring in tn_1 and tn_m are disjunct, which can be achieved by renaming. Then, m decomposes a task identifier $i \in I_1$ into a task network $tn_2 = (I_2, \prec_2, \alpha_2, VC_2)$ if and only if $\alpha_1(i) = c(?y_1, \ldots, ?y_n)$ and*

$$\begin{aligned} I_2 &= (I_1 \setminus \{i\}) \cup I_m \\ \prec_2 &= (\prec_1 \cup \prec_m \cup \\ &\qquad \{(i_1, i_2) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \cup \\ &\qquad \{(i_1, i_2) \in I_m \times I_1 \mid (i, i_2) \in \prec_1\}) \end{aligned}$$

[1]We adopt the convention of PDDL (McDermott 2000) to denote variables with a prefixed question mark. I.e. $?a$ is a variable, while a denotes a constant.

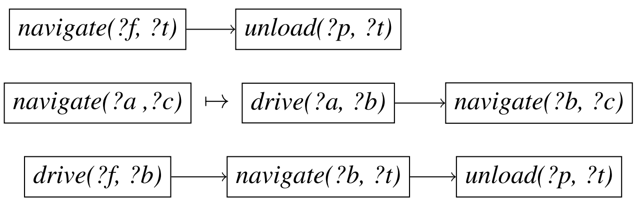


Figure 2: The first row shows a task network tn_1. The second row shows a method for the navigate task. The result of applying this method to the navigate task in tn_1 results in the task network shown in the third row.

$$\begin{aligned} &\setminus \{(i', i'') \in I_1 \times I_1 \mid i' = i \text{ or } i'' = i\} \\ \alpha_2 &= (\alpha_1 \cup \alpha_m) \setminus \{(i, c(?y_1, \ldots, ?y_n))\} \\ VC_2 &= VC_1 \cup VC_2 \cup \{?x_i = ?y_i \mid 1 \leq i \leq n\} \end{aligned}$$

Instead of introducing additional equality constraints over the variables, we could also replace all occurrences of $?x_i$ in tn_1 with $?y_1$, which is more useful in practice as it introduces variables only if necessary. Further, variable constraints are simply added and not propagated. This eliminates the necessity for handling constraints between propagated variables. Of course, an implementation would always propagate variable constraints as far as possible.

In Def. 1, we allowed a task network to contain no tasks at all, i.e. we allowed for I to be the empty set. Thus methods may decompose abstract tasks into such task networks. This is sensible and occurs (somewhat) frequently in practice. Consider the abstract task $navigate(?f, ?t)$ where $?f = ?t$. Such a task can be achieved without doing anything, i.e. by the empty task network. All ordering constraints relating to it are not lost, as their transitive implications are kept during decomposition. Similarly, the parameter variables of the decomposed abstract task remain variables of the decomposed task network, i.e. no constraints can be lost.

As an example for applying a decomposition method, consider the task networks and the method shown in Fig. 2.

3 Grounding Planning Problems

As in classical planning, both theoretical reserach (see e.g. (Höller et al. 2014; Behnke, Höller, and Biundo 2015; Höller et al. 2016; Behnke et al. 2016; Bercher et al. 2016; Alford et al. 2016)) and practical research (Behnke et al. 2018; 2019) on hierarchical planning is usually done on *grounded*, i.e. variable-free models instead of lifted models. Especially newer search-based HTN planners like FAPE (Dvorak et al. 2014), GTOHP (Ramoul et al. 2017), or PANDA (Bercher, Keen, and Biundo 2014; Bercher et al. 2017) ground a given lifted planning problem prior to search. A grounded model allows for both a more efficient implementation of the search itself and for easier to compute and more concise heuristics. In contrast, the translation technique by Alford et al. (2016) is executed on the lifted model – grounding is only performed on the resulting classical model.

In theory, computing a grounded model based on a given lifted model is easy. One has to compute all possible instantiations of lifted predicates, primitive and abstract tasks, and methods and replace their lifted versions appropriately by them. For details regarding the full grounding process we refer to the work of Alford, Bercher, and Aha (2015). Naturally such a grounding will be exponential in size with respect to the original domain. As such, a fully grounded model is not useful in many practical cases, as handling it within given memory and time limits is hard or even impossible.

In many planning problems, computing all instantiations of all predicates, tasks, and methods is not necessary. For example, it is not necessary to create a grounding $drive(l_1, l_2)$ of the drive action if there is no road between the locations l_1 and l_2. For such an instantiation $drive(l_1, l_2)$, we know a priori that its precondition can never be fulfilled[2]. Thus this action cannot be part of any plan. Ideally, we would like to compute only those groundings of predicates, tasks, and methods that occur in some solution to the planning problem. Determining (exactly) whether this is the case is unfortunately undecidable. This is caused by the fact that deciding whether a given HTN planning problem has a solution or not is undecidable (Erol, Hendler, and Nau 1996). If determining that a task occurs in no solution would (in general) be decidable, we would have a finite-time procedure for testing whether a given HTN planning problem is solvable: simply run the test on all its primitive tasks. A solution exists if and only if at least one of them is contained in any solution (excluding the detectable case of a possible empty solution, which can be tested in advance in polynomial time).

Instead, we aim at computing an approximation of this property. I.e. we are looking for a subset of all ground instances of predicates, tasks, and methods such that all ground instances not included in that set are not contained in any solution. As such, we do not include a grounding if we can prove that it cannot be contained in a solution.

This technique of approximate grounding is widely used in classical planning. In general, an action is not included in the grounding if it cannot be part of any executable plan in the *delete-relaxation* of the problem. The delete-relaxation of a planning problem is a copy of the problem in which all negative effect literals are removed. For a given action one can determine in polynomial time whether it is part of any delete-relaxed plan (Bylander 1994). The set of these actions is usually computed via a *planning graph* (Blum and Furst 1997). Often, this reduction leads to a significant decrease in the size of the grounded problem. Some planning systems, like FF (Hoffmann and Nebel 2001) first compute the full grounding and subsequently prune actions[3]. This, however, does not eliminate the bottle-neck of grounding, but makes the grounding smaller for the planning process itself. An efficient implementation based on DATALOG was proposed by Helmert (2009), which does not have this bottle-neck of a full instantiation.

To the best of our knowledge there is currently only one publication in the field of HTN planning devoted

[2]Assuming that there is no means to build new roads.

[3]Note that FF uses the concept of inertia (Koehler and Hoffmann 2000) to simplify the preconditions and effects before full grounding.

to grounding in more detail, which is the grounder of GTOHP (Ramoul et al. 2017). It uses a grounding procedure similar to that of FF (Hoffmann and Nebel 2001) and similarly uses the concept of inertia (Koehler and Hoffmann 2000). Inertia of a predicate describes the ways its truth value can change while a plan is executed – not at all, only from negative to positive (or vice versa), or in both directions. In inertia-based simplification, a primitive task whose precondition evaluates to false under the computed inertia values is removed from the planning problem, as it can never become executable. Subsequently, all methods it is contained in are removed as well. If an abstract task has no applicable method remaining it is likewise removed.

Note that the procedure used by GTOHP removes effectless actions from the methods they are contained in (Ramoul et al. 2017). The respective methods are not pruned afterwards (which would be incorrect), but considered part of the correct grounding without the removed effectless actions. According to the formalisation of HTN planning, these actions can however be contained in plans – and pose constraints in them. As such as it makes any found solution (potentially) invalid as it may not adhere to the solution criteria of HTN planning. As a notable example, a state-based goal description (like used in classical planning) can be encoded in an HTN planning problem as an additional effectless action, which would be pruned by GTOHP. As such, the planner would not be obliged to reach a goal state. Also "moving" the preconditions of these effectless actions to other actions within the same method is not correct (i.e. equivalent transformation). Moving the precondition would require it to hold in conjunction with the precondition of another action, which is not required in the original problem, as another action could be ordered in between. Secondly, the implementation of GTOHP does not allow for two parameters of one action or method to be instantiated with the same constant. Consider as an example a method that paints two wooden boards $?b_1$ and $?b_2$ in colours $?c_1$ and $?c_2$. GTOHP enforces that $?c_1$ and $?c_2$ are different without this constraint being a part of the domain. This leads to an invalid grounding, this time when the (only) solution uses the method where both colours are, e.g. red, as we only have red paint. For our evaluation (Sec. 5), we have fixed both issues in the code of GTOHP.

4 Grounding HTNs

Our grounding procedure includes three steps: a lifted domain simplification, computing delete-relaxed reachability, and a hierarchical reachability analysis based on a graph called the Task Decomposition Graph (TDG) (Elkawkagy et al. 2012; Bercher et al. 2017).

4.1 Parameter Splitting

As a first step, we perform simplification operations on the lifted model. For example, we compile disjunctions in preconditions into additional actions, and compile away negative preconditions. Similarly, we compile away variables occurring in preconditions and effects (i.e. those that are contained in quantified expressions) into additional parameters.

Beside these common simplifications known from classical planning, our grounder performs an HTN-specific simplification operation on the lifted model with the aim of reducing the size of the grounding. In some HTN planning domains, lifted decomposition methods contain variables that are (1) used only as parameters of a single or very few subtasks and (2) which are not parameters of the abstract task. As an example, consider an abstract task $A(?x)$ with a method decomposing it into the tasks $B(?x, ?y)$ and $C(?x, ?z)$. Further assume that all variables have the same type t which contains the constants $C = \{c_1, \ldots, c_n\}$. If we ground this method, it has n^3 ground instances. Notably, we have to ground every possible combination of the otherwise independent parameters $?y$ and $?z$.

We can equivalently represent this method by three new methods while introducing two new abstract tasks. Let these abstract tasks be $B^*(?x)$ and $C^*(?x)$. The three decomposition methods are $A(?x) \mapsto B^*(?x), C^*(?x)$[4], $B^*(?x) \mapsto B(?x, ?y)$, and $C^*(?x) \mapsto C(?x, ?z)$. For these three methods, there are $2n^2 + n$ groundings plus an additional $2n$ new groundings of abstract tasks (B^* and C^*), which is a significant improvement over the original model.

In general, we can perform this operation whenever there is a variable $?x$ in a method that is only a parameter of one of the subtasks A and its variable constraints connect it only to the other parameters of A. We can sometimes also perform this splitting of parameters into additional methods if the variable occurs in multiple subtasks $A_1, \ldots, A_k$. Since we have to equivalently transform the model, we have to assure that by applying multiple decompositions, the original method is still correctly represented in the model. As such, we only split away a group of actions $A_1, \ldots, A_k$ if all of them have the same relative ordering against the other tasks in the method. We then replace them by a new single abstract task A^* with this relative order and a method for A^* decomposing it into $A_1, \ldots, A_k$ with their internal order. By applying the method for A^* to the instance of A^* in the changed main decomposition method, we obtain the original method. Note that A^* can have more parameter variables than the individual actions, i.e. it can increase the number of abstract tasks significantly. Their number is however limited by the number of ground methods. Thus we assume that this compilation is not an issue in practice.

4.2 Delete-Relaxed Reachability

After the initial simplification of the domain, we perform a delete-relaxed reachability analysis to determine which groundings of primitive tasks can possibly occur in any executable task network. Our implementation is succinct in the sense that it never considers groundings that are not delete-relaxed reachable, similar to the DATALOG-based implementation by Helmert (2009). We have opted for a native implementation of the planning graph algorithm.

4.3 TDG-based Hierarchical Reachability

Our hierarchical reachability analysis is based on a datastructure called the Task Decomposition Graph (TDG).

[4]To remain correct, B^* and C^* have the order of B and C.

We first introduce the definition as given by Bercher et al. (2017). After introducing the declarative definition, we describe how it is built algorithmically (in the next section).

Definition 4 (Task Decomposition Graph (TDG)). *Let $\mathcal{P} = \langle \mathcal{L}, T_P, T_C, M, s_I, tn_I \rangle$ be an HTN planning problem. Without loss of generality, we assume that tn_I contains just a single ground abstract task* TOP *for which there is exactly one method in M.*[5]

The bipartite graph $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$, consisting of a set of task vertices V_T, method vertices V_M, and edges $E_{T\to M}$ and $E_{M\to T}$ is called the TDG of $\mathcal{P}$ if it holds:

1. ***Base Case** (task vertex for the given task)*
   TOP $\in V_T$*, the TDG's root.*
2. ***Method Vertices** (derived from task vertices)*
   Let $c \in V_T$ and there is a method $(c, tn, VC) \in M$. Then, for all groundings v_m that satisfy the variable constraints in VC it holds that:
   - $v_m \in V_M$
   - $(v_t, v_m) \in E_{T\to M}$.
3. ***Task Vertices** (derived from method vertices)*
   Let $v_m \in V_M$ with $v_m = (c, tn, VC)$ and $tn = (I, \prec, \alpha, VC)$. Then, for all tasks $i \in I$ with $\alpha(i) = v_t$ the following holds:
   - $v_t \in V_T$
   - $(v_m, v_t) \in E_{M\to T}$.
4. ***Tightness***
   $\mathcal{G}$ is minimal, such that 1. to 3. hold.

A TDG is a directed graph. Nodes represent either ground tasks or ground methods. A task node has outgoing edges to each applicable ground method, and each method has outgoing edges to its ground subtasks. This means the graph is a representation of hierarchical reachability, i.e. which ground tasks and methods can possibly be reached via decomposition. As can be seen from the definition, it is bound linearly in the number of ground methods. It can be constructed in linear time in case the planning problem $\mathcal{P}$ is ground and in exponential time in case $\mathcal{P}$ is lifted.

Note that the TDG can represent HTN planning problems that contain cyclic methods. A cyclic decomposition is a sequence of decompositions of a grounded task c that results in a task network containing c again. If the planning problem contains such a cycle, the edge representing the method that produces the recursive occurrence of c simply points back to the vertex created for the first occurrence of c.

TDGs constructed based on the definition contain only those groundings reachable from the initial task by decomposition. As proposed by Elkawkagy, Schattenberg, and Biundo (2010) one can delete those method nodes that contain a primitive task not reachable in a state-based reachability analysis like the planning graph. As a consequence of removing those methods[6], there may be abstract tasks in the TDG that cannot be decomposed into a task network containing only primitive actions any more. For example, removing a method containing a not delete-relaxed reachable action might remove the only option to exit a recursive method structure. If such an abstract task occurs in a task network during decomposition, we know that it is impossible to refine that task network into a solution. We can thus prune the abstract task – and consequently all methods it is contained it. Removing these methods may again allow us to remove other abstract tasks, thus one can repeat this process until convergence (Def. 2b). These tasks can be identified in polynomial time by relying on a bottom-up reachability analysis (Alford et al. 2014, proof of Thm. 3.1).

We parametrize the previous definition of a TDG by specifying an additional set of primitive ground tasks: these are the actions that are (supposed to be) reachable (like, e.g. the actions reachable in the planning graph).

Definition 5 (Pruned TDG). *Let $\mathcal{P} = \langle \mathcal{L}, T_P, T_C, M, s_I, tn_I\rangle$ be an HTN planning problem and $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ the respective TDG according to Def. 4. Let X be the set of actions as given above.*

Then, the pruned TDG $\mathcal{G}_X = \langle V'_T, V'_M, E'_{T\to M}, E'_{M\to T}\rangle$ *that exploits the reachability information for the actions in X is given as the minimal connected subgraph containing* TOP *such that:*

1. ***Remove Useless Method Vertices***
   A method vertex $v_m = (c, tn, VC) \in V_M$ with $tn = (I, \prec, \alpha, VC)$ is in V'_M if and only if I does not contain a task i with $\alpha(i) \notin X$ in case $\alpha(i)$ is primitive or with $\alpha(i)$ being useless, in case it is abstract (see below).
2. ***Identify Useless Abstract Task Vertices***
   An abstract task vertex $v_t \in V'_T$ is called useless if one of the following holds:
   (a) the pruned TDG $\mathcal{G}_X$ does not contain children for v_t (i.e., all successors of v_t were pruned)
   (b) there is no acyclic connected subgraph of the pruned TDG $\mathcal{G}_X$ with root v_t, in which every abstract method vertex has exactly one outgoing edge and no vertex is useless (i.e., the task v_t cannot be decomposed into a set of primitive tasks)

Whereas the parameter splitting described before is proposed in this paper the very first time, the TDG-based grounding procedure in contrast has a rather long history. Initial ideas were first proposed by Elkawkagy, Schattenberg, and Biundo (2010) showing how to compute a pruned decomposition *tree* (TDT), which was exploited during search. They described the key ideas of deleting actions that cannot be reached by a delete-relaxed reachability analysis, triggering further deletions of methods and possibly abstract tasks. The TDT was subsequently extended to a graph (Elkawkagy et al. 2012), but without altering the deployed reachability analysis. Later, we extended the reachability analysis to also prune those abstract tasks from the TDG that cannot be refined into a primitive task network (Bercher et al. 2017). However, we did not yet provide a formal, declarative definition of the resulting pruned TDG there. Furthermore, similar to the work by Elkawkagy, Schattenberg, and Biundo (2010; 2012) we only explained that this pruned TDG may be used

[5]If the problem specifies an initial partial plan tn_I we can obtain the required form by adding a new artificial (parameter-free) abstract task TOP that decomposes exactly into tn_I.

[6]Or in rare cases by a mistake of the modeller.

as a basis for heuristics. Here we explain how it also serves the purpose of obtaining a ground model. The following section is another yet unpublished contribution that is essential for the efficiency of the grounding/TDG construction procedure.

4.4 Avoiding the Bottleneck

Beside the final size of the grounding, it is – in practice – crucial to avoid large sets of intermediate groundings during the computation process. A naive idea would be to compute the full TDG and prune it afterwards. This corresponds to computing a full instantiation of all actions in classical planning and performing a reachability analysis on them. Both the full TDG and the full instantiation of actions usually contain unnecessary groundings that will be pruned afterwards. In this section we describe how the pruned TDG can be computed (somewhat) efficiently, without the need to compute the full TDG first.

When building the TDG in a top-down manner, it will initially include the initial task and is iteratively extended by adding nodes for each applicable method and its subtasks. To handle cyclic decompositions, we keep a set of created task groundings. Whenever we add a new decomposition method, we check for all its subtasks whether they are contained in the set of already created task groundings. If so, we use that already existing node to add the respective edge implied by the method to the graph and don't recurse through that grounding – as it has already been (or is in the process of being) fully expanded. This way, the procedure terminates also on cyclic HTN planning problems and it is ensured that tasks that are not reachable via the hierarchy are never included. However, when primitive tasks are added to the graph, it has to be checked whether these are reachable via state transition, and given that they are not, the graph has to be pruned as given in Def. 5.

Alternatively, one could construct a superset of the pruned TDG in a bottom-up manner. We start with nodes for the primitive tasks that are reachable under delete-relaxation. Then, for each method where all subtasks are included in the graph, nodes for the method and for the task the method decomposes are included. Methods and tasks are added until convergence. We again use a set of created task groundings to handle cyclic decompositions in the planning problem. That way, the graph never includes tasks that would need to be pruned based on state-based reachability information. However, it might include tasks and methods that are not reachable from the initial task. These can be removed by a depth-first search afterwards.

When using both the top-down and the bottom-up computation, state-based and hierarchical reachability analysis influence each other. The hierarchy might e.g. exclude actions that are necessary to fulfil other actions' preconditions. An action a_1 pruned due to state-based reachability may exclude a method that has been the only source of reachability of another action a_2. This means that the two analyses should be iterated until the grounding converged. PANDA's grounding does so.

The question is now if a system should rely on the top-down or the bottom-up building process. There is no (domain-independent) answer to this question. It is easy to construct planning problems that result in large intermediate graphs for both ways of construction. Which one works better depends on the specific structure of the problem at hand. We do not yet know how to determine which algorithm will perform better a priori. Therefore we developed a way of construction that combines the benefits of both procedures.

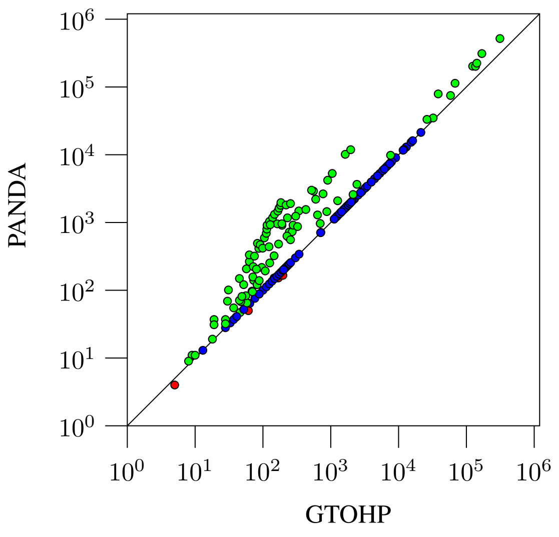


Figure 3: Number of primitive tasks in groundings computed by GTOHP and our grounder. Green indicates that PANDA finds the smaller grounding, red that GTOHP does and blue indicates that both groundings have equal size.

We start with a top-down construction, but instead of creating the grounded nodes directly, it maintains for each parameter of each task and method a list of all constants that might be assigned to the parameter. This avoids the creation of all combinations of constants, to the cost of losing the information which parameter *combinations* are valid. When primitive tasks are reached, the constant set is further reduced via state-based reachability, and this reduction is propagated through the graph. In a second step, top-down grounding is performed using the reduced constant sets.

5 Evaluation

The implementation of the described grounding procedure is included in the PANDA planner. It is primarily implemented in Scala. The grounder accepts HTN planning problems formulated in HDDL as its input (Höller et al. 2019a).

In this preliminary evaluation we do not compare the runtime of the different approaches (top-down, bottom-up, two-way), but compare the size of the resulting grounding with a system from related work. We are currently re-implementing the grounder and will present runtime results in a follow-up paper.

We have compared our grounding procedure against GTOHP (Ramoul et al. 2017), which is the so-far only published grounding procedure for HTN planning. In the evaluation, we have included all 202 instances used in the recent

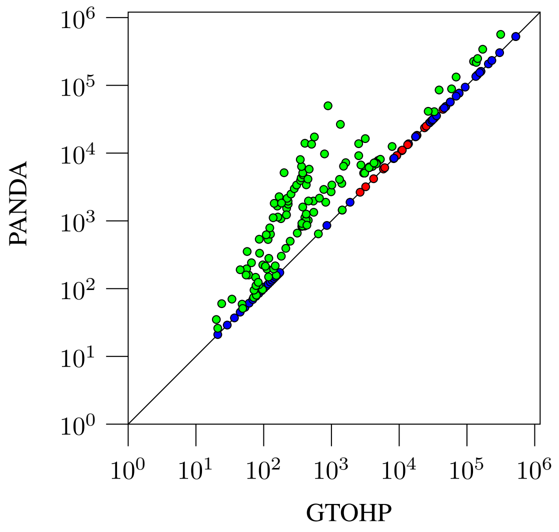


Figure 4: Number of decomposition methods in groundings computed by GTOHP and our grounder. Green indicates that PANDA finds the smaller grounding, red that GTOHP does and blue indicates that both groundings have equal size.

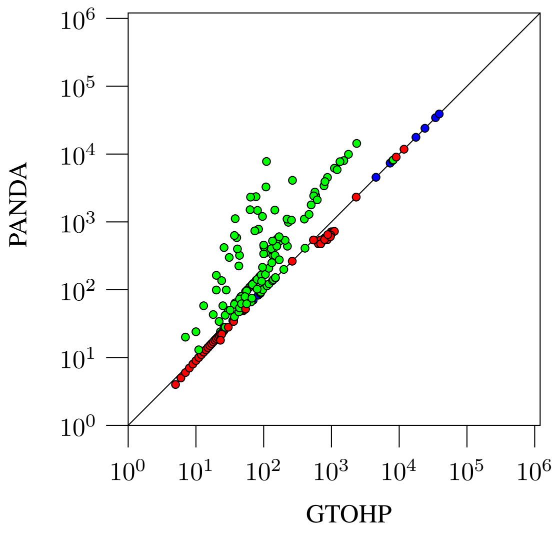


Figure 5: Number of abstract tasks in groundings computed by GTOHP and our grounder. Green indicates that PANDA finds the smaller grounding, red that GTOHP does and blue indicates that both groundings have equal size.

evaluation of Tree-Rex (Schreiber et al. 2019), which uses the GTOHP grounder.

Except for one domain (TRANSPORT), we always found at most as many primitive tasks in the grounding as GTOHP. In TRANSPORT, we usually needed a few more ground instances, as its actions contain disjunctive preconditions which we compile away while GTOHP handles them natively. A scatter plot of the results is shown in Fig. 3. We find smaller groundings in 96 instances, larger ones in 5, and 101 groundings of equal size. On average our groundings are 26.19% smaller with a maximum of 90.64% or 205.913 tasks reduction.

For the number of methods, results are shown in Fig. 4. We find smaller groundings in 132 instances, larger ones in 18, and 52 groundings of equal size. On average our groundings are 37.38% smaller with a maximum of 98.22% or 249.828 methods reduction.

For the number of abstract tasks, results are shown in Fig. 5. We find smaller groundings in 113 instances, larger ones in 80, and 8 groundings of equal size. On average our groundings are 24.02% smaller with a maximum of 98.59% or 11.980 abstract tasks reduction. As we can see from the scatter plot, if we produce a larger grounding, it is usually not significantly larger. At the maximum, our grounding contains 381 more abstract tasks due to our parameter splitting, which produces significantly fewer methods in several instances.

6 Conclusion

Most recent systems in HTN planning realise the planning process in a fully grounded way. A smaller grounding usually improves the performance of the planner. For example, a smaller grounding allows for heuristics to be computed faster – and for them to be more precise. Further, the search mechanics of the planner is faster the smaller the grounding is, as fewer actions and methods have to be considered. Lastly, a smaller grounding also reduces the size of encodings, e.g. into propositional logic, which makes the translated problem (potentially) easier to solve. Despite these advantages, little work has been published on grounding techniques especially for HTN planning. We present our grounding procedure and discuss how to compute it efficiently. Our empirical evaluation shows that it leads to smaller groundings than related work.

Acknowledgments

This work was done within the technology transfer project "Do it yourself, but not alone: Companion-Technology for DIY support" of the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems" funded by the German Research Foundation (DFG). The industrial project partner is the Corporate Research Sector of the Robert Bosch GmbH.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight bounds for HTN planning. In *Proceedings of the 25th International*

Conference on Automated Planning and Scheduling (ICAPS 2015), 7–15. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2014. On the feasibility of planning graph style heuristics for HTN planning. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 2–10. AAAI Press.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: relating task and goal decomposition with task sharing. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*. AAAI Press.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan – How hard can that be? In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 38–46. AAAI Press.

Behnke, G.; Schiller, M.; Kraus, M.; Bercher, P.; Schmautz, M.; Dorna, M.; Minker, W.; Glimm, B.; and Biundo, S. 2018. Instructing novice users on how to use tools in DIY projects. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence and the 23rd European Conference on Artificial Intelligence (IJCAI-ECAI 2018)*, 5805–5807. IJCAI.

Behnke, G.; Schiller, M.; Kraus, M.; Bercher, P.; Schmautz, M.; Dorna, M.; Dambier, M.; Minker, W.; Glimm, B.; and Biundo, S. 2019. Alice in DIY-wonderland or: Instructing novice users on how to use tools in DIY projects. *AI Communications* 32(1):31–57.

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 Press.

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 Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018a. totSAT – Totally-ordered hierarchical planning through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018b. Tracking branches in trees – A propositional encoding for solving partially-ordered HTN planning problems. In *Proceedings of the 30th International Conference on Tools with Artificial Intelligence (ICTAI 2018)*, 73–80. IEEE Computer Society.

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)*. AAAI Press.

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)*. IJCAI.

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)*. 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)*, 225–233. IOS Press.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 480–488. IJCAI.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of the 7th Annual Symposium on Combinatorial Search (SoCS 2014)*, 35–43. AAAI Press.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1-2):165–204.

Dvorak, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. A flexible ANML actor and planner in robotics. In *Proceedings of the 4th Workshop on Planning and Robotics (PlanRob)*, 12–19.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving hierarchical planning performance by the use of landmarks. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI 2012)*, 1763–1769. AAAI Press.

Elkawkagy, M.; Schattenberg, B.; and Biundo, S. 2010. Landmarks in hierarchical planning. In *Proceedings of the 20nd European Conference on Artificial Intelligence (ECAI 2010)*, 229–234. IOS Press.

Erol, K.; Hendler, J.; and Nau, D. 1996. Complexity results for HTN planning. *Annals of Mathematics and AI* 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. AAAI Press.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14:2531–302.

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)*, volume 263, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling, (ICAPS 2016)*, 158–165. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, B. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 114–122. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2019a. HDDL – A language to describe hierarchical planning problems. In *Proceedings of the Second ICAPS Workshop on Hierarchical Planning*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019b. On guiding search in HTN planning with classical planning heuristics. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. IJCAI.

Koehler, J., and Hoffmann, J. 2000. On the instantiation of ADL operators involving arbitrary first-order formulas. In *Proceedings of the 14th Workshop on New Results in Planning, Scheduling and Design (PuK 2000)*, 74–82.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN planning domain. *International Journal on Artificial Intelligence Tools* 26(5):1–24.

Schreiber, D.; Balyo, T.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*. AAAI Press.

Parsing-based Approaches for Verification and Recognition of Hierarchical Plans

Roman Barták
Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic

Adrien Maillard
GMV
Flight Dynamics and Operations
Toulouse, France

Rafael C. Cardoso
University of Liverpool
Department of Computer Science
Liverpool, United Kingdom

Abstract

Hierarchical planning, in particular, Hierarchical Task Networks, was proposed as a method to describe plans by decomposition of tasks to sub-tasks until primitive tasks, actions, are obtained. Plan verification assumes a complete plan as input, and the objective is finding a task that decomposes to this plan. In plan recognition, a prefix of the plan is given and the objective is finding a task that decomposes to the (shortest) plan with the given prefix. This paper describes how to verify and recognize plans using a common method known from formal grammars, by parsing.

Introduction

Hierarchical planning is a practically important approach to automated planning based on encoding abstract plans as *hierarchical task networks* (HTNs) (Erol, Hendler, and Nau 1996). The network describes how compound tasks are decomposed, via decomposition methods, to sub-tasks and eventually to actions forming a plan. The decomposition methods may specify additional constraints among the sub-tasks such as partial ordering and causal links.

As of this writing, there exist only two systems for *verifying* if a given plan complies with the HTN model, that is, if a given sequence of actions can be obtained by decomposing some task. One system is based on transforming the verification problem to SAT (Behnke, Höller, and Biundo 2017) and the other system is using parsing of attribute grammars (Barták, Maillard, and Cardoso 2018). Only the parsing-based system supports HTN fully (the SAT-based system does not support the decomposition constraints).

Parsing became popular in solving the *plan recognition problem* (Vilain 1990) as researchers realized soon the similarity between hierarchical plans and formal grammars, specifically context-free grammars with parsing trees close to decomposition trees of HTNs. The plan recognition problem can be formulated as the problem of adding a sequence of actions after some observed partial plan such that the joint sequence of actions forms a complete plan generated from some task (more general formulations also exist). Hence plan recognition can be seen as a generalization of plan verification. There exist numerous approaches to plan recognition using parsing or string rewriting (Avrahami-Zilberbrand and Kaminka 2005; Geib, Maraist, and Goldman 2008; Geib and Goldman 2009; Kabanza et al. 2013), but they use hierarchical models that are weaker than HTNs. The languages defined by HTN planning problems (with partial-order, preconditions and effects) lie somewhere between context-free (CF) and context-sensitive (CS) languages (Höller et al. 2014) so to model HTNs, one needs to go beyond the CF grammars. Currently, the only grammar-based model that fully covers HTNs uses attribute grammars (Barták and Maillard 2017). Moreover, the expressivity of HTNs makes the plan recognition problem undecidable (Behnke, Höller, and Biundo 2015). At the moment, there is only one approach for HTN plan recognition. This approach relies on translating the plan recognition problem to a planning problem (Höller et al. 2018), which is a technique that was first introduced in (Ramírez and Geffner 2003).

In this paper, we focus on verification and recognition of HTN plans using parsing. The uniqueness of the proposed methods is that they cover full HTNs including task interleaving, partial order of sub-tasks, and other decomposition constraints (prevailing constraints, specifically). The methods are derived from the plan verification technique proposed in (Barták, Maillard, and Cardoso 2018).

There are two novel contributions of this paper. First, we will simplify the above mentioned verification technique by exploiting information about actions and states to improve practical efficiency of plan verification. Second, we will extend that technique to solve the plan (task) recognition problem. We will show that the verification algorithm can be much simpler and, hence, it is expected to be more efficient. For plan recognition, the method proposed in (Höller et al. 2018) can in principle support HTN fully, if a full HTN planner is used (which is not the case yet as prevailing conditions are not supported). However, like other plan recognition techniques it requires the top task (the goal) and the initial state to be specified as input. A practical difference of our methods is that they do not require information about possible top (root) tasks and an initial state as their input. This is particularly interesting for plan/task recognition, where existing methods require a set of candidate tasks (goals) to select from (in principle, they may use all tasks as candidates, but this makes them inefficient).

Background on Planning

In this paper, we work with classical STRIPS planning (Fikes and Nilsson 1971) that deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal conditions. World states are modelled as sets of propositions that are true in those states, and actions are are modelled to change the validity of certain propositions.

Classical Planning

Formally, let P be a set of all propositions modelling properties of world states. Then a state $S \subseteq P$ is a set of propositions that are true in that state (every other proposition is false). Later, we will use the notation $S^+ = S$ to describe explicitly the valid propositions in the state S and $S^- = P \setminus S$ to describe explicitly the propositions not valid in the state S.

Each action a is described by three sets of propositions (B_a^+, A_a^+, A_a^-), where $B_a^+, A_a^+, A_a^- \subseteq P, A_a^+ \cap A_a^- = \emptyset$. Set B_a^+ describes positive preconditions of action a, that is, propositions that must be true right before the action a. Some modeling approaches allow also negative preconditions, but these preconditions can be compiled away. For simplicity reasons we assume positive preconditions only (the techniques presented in this paper can also be extended to cover negative preconditions directly). Action a is applicable to state S iff $B_a^+ \subseteq S$. Sets $A_a^+$and A_a^- describe positive and negative effects of action a, that is, propositions that will become true and false in the state right after executing the action a. If an action a is applicable to state S then the state right after the action a is:

$$\gamma(S,a) = (S \setminus A_a^-) \cup A_a^+. \quad (1)$$

$\gamma(S,a)$ is undefined if an action a is not applicable to state S.

The classical planning problem, also called a STRIPS problem, consists of a set of actions A, a set of propositions S_0 called an initial state, and a set of goal propositions G^+ describing the propositions required to be true in the goal state (again, negative goal is not assumed as it can be compiled away). A solution to the planning problem is a sequence of actions $a_1, a_2, \ldots, a_n$ such that $S = \gamma(...\gamma(\gamma(S_0,a_1),a_2),...,a_n)$ and $G^+ \subseteq S$. This sequence of actions is called a *plan*.

The *plan verification problem* is formulated as follows: given a sequence of actions $a_1, a_2, \ldots, a_n$, and goal propositions G^+, is there an initial state S_0 such that the sequence of actions forms a valid plan leading from S_0 to a goal state? In some formulations, the initial state might also be given as an input to the verification problem.

Hierarchical Task Networks

To simplify and speeed up the planning process, several extensions of the basic STRIPS model were proposed to include some control knowledge. Hierarchical Task Networks (Erol, Hendler, and Nau 1996) were proposed as a planning domain modeling framework that includes control knowledge in the form of recipes on how to solve specific tasks. The recipe is represented as a task network, which is a set of sub-tasks to solve a given task together with the set of constraints between the sub-tasks. Let T be a compound task and $(\{T_1, ..., T_k\}, C)$ be a task network, where C are its constraints (see later). We can describe the decomposition method as a derivation (rewriting) rule:

$$T \rightarrow T_1, ..., T_k \ [C]$$

The planning problem in HTN is specified by an initial state (the set of propositions that hold at the beginning) and by an initial task representing the goal. The compound tasks need to be decomposed via decomposition methods until a set of primitive tasks – actions – is obtained. Moreover, these actions need to be linearly ordered to satisfy all the constraints obtained during decompositions and the obtained plan – a linear sequence of actions – must be applicable to the initial state in the same sense as in classical planning. We denote an action as a_i, where the index i means the order number of the action in the plan (a_i is the i-th action in the plan). The state right after the action a_i is denoted S_i, S_0 is the initial state. We denote the set of actions to which a task T decomposes as $act(T)$. If U is a set of tasks, we define $act(U) = \cup_{T \in U} act(T)$. The index of the first action in the decomposition of T is denoted $start(T)$, that is, $start(T) = min\{i | a_i \in act(T)\}$. Similarly, $end(T)$ means the index of the last action in the decomposition of T, that is, $end(T) = max\{i | a_i \in act(T)\}$.

We can now define formally the constraints C used in the decomposition methods. The constraints can be of the following three types:

- $t_1 \prec t_2$: a precedence constraint meaning that in every plan the last action obtained from task t_1 is before the first action obtained from task t_2, $end(t_1) < start(t_2)$,
- $before(U,p)$: a precondition constraint meaning that in every plan the proposition p holds in the state right before the first action obtained from tasks U, $p \in S_{start(U)-1}$,
- $between(U,V,p)$: a prevailing condition meaning that in every plan the proposition p holds in all the states between the last action obtained from tasks U and the first action obtained from tasks V,
$\forall i \in \{end(U), \ldots, start(V) - 1\}, p \in S_i$.

The *HTN plan verification problem* is formulated as follows: given a sequence of actions $a_1, a_2, \ldots, a_n$, is there an initial state S_0 such that the sequence of actions is a valid plan applicable to S_0 and obtained from some compound task? Again, the initial state might also be given as an input in some formulations.

The *HTN plan recognition problem* is formulated as follows: given a sequence of actions $a_1, a_2, \ldots, a_n$, is there an initial state S_0 and actions $a_{n+1}, \ldots, a_{n+m}$ for some $m \geq 0$ such that the sequence of actions $a_1, a_2, \ldots, a_{n+m}$ is a valid plan applicable to S_0 and obtained from some compound task? In other words, if the given actions form a prefix of some plan obtained from some compound task T. We will be looking for such a task T minimizing the value m (the number of added actions to complete the plan). If only the task T is of interest (not the actions $a_{n+1}, \ldots, a_{n+m}$) then it can be referred to as the *task (goal) recognition problem*.

The Plan Verification Algorithm

The existing parsing-based HTN verification algorithm (Barták, Maillard, and Cardoso 2018) uses a complex structure of a timeline. This structure maintains the decomposition constraints so that they can be checked when composing sub-tasks to a compound task. We propose a simplified verification method that does not require this complex structure, as it checks all the constraints directly in the input plan. This makes the algorithm easier to implement and also potentially faster. Another difference is that we do not assume that the initial state is passed as input, instead we set the initial state as the preconditions of the first action in the plan. However, adding support for it is trivial as we would only have to add the initial state that was given as input to the preconditions of the first action in the plan.

The novel hierarchical plan verification algorithm is shown in Algorithm 1. It first calculates all intermediate states (lines 2-8) by propagating information about propositions in action preconditions and effects. At this stage, we actually solve the classical plan validation problem as the algorithm verifies that the given plan is causally consistent (action precondition is provided by previous actions or by the initial state). The original verification algorithm did this calculation repeatedly each time it composed a compound task. It is easy to show that every action is applicable, that is, $B^+_{a_i} \subseteq S_{i-1}$ (lines 2 and 4). Next, we will show that $\gamma(S_i, a_{i+1}) = S_{i+1} = (S_i \setminus A^-_{a_{i+1}}) \cup A^+_{a_{i+1}}$. Left-to-right propagation (line 4) ensures that $(S_i \setminus A^-_{a_{i+1}}) \cup A^+_{a_{i+1}} \subseteq S_{i+1}$. Right-to-left propagation (line 6) ensures that preconditions are propagated to earlier states if not provided by the action at a given position. In other words, if there is a proposition $p \in S_{i+1} \setminus A^+_{a_{i+1}}$ then this proposition should be at S_i. Line 6 adds such propositions to S_i so it holds $(S_i \setminus A^-_{a_{i+1}}) \cup A^+_{a_{i+1}} = S_{i+1}$. However, if $p \in A^-_{a_{i+1}}$ then p would be deleted by the action a_{i+1}, which means that the plan is not valid. The algorithm detects this at lines 7-8.

When the states are calculated, we apply a parsing algorithm to compose tasks. Parsing starts with the set of primitive tasks (line 9), each corresponding to an action from the input plan. For each task T, we keep a data structure describing the set $act(T)$, that is, the set of actions to which the task decomposes. We use a Boolean vector I of the same size as the plan to describe this set; $a_i \in act(T) \Leftrightarrow I(i) = 1$. To simplify checks of decomposition constraints, we also keep information about the index of first and last actions from $act(T)$. Together, the task is represented using a quadruplet (T, b, e, I) in which T is a task, b is the index in the plan of the first action generated by T, e is the index in the plan of the last action generated by T (we say that $[i, j]$ represents the interval of actions over which T *spans*), and I is a Boolean vector as described above.

The algorithm applies each decomposition rule to compose a new task from already known sub-tasks (line 12). The composition consists of merging the sub-tasks, when we check that every action in the decomposition is obtained from a single sub-task (line 20), that is, $act(T_0) = \bigcup_{j=1}^{k} act(T_j)$ and $\forall i \neq j : act(T_i) \cap act(T_j) = \emptyset$. We also check all the decomposition constraints; the pseudo-code is

```
   Data: a plan $\mathbf{P} = (a_1, ..., a_n)$ and a set of decomp. methods
   Result: a Boolean equal to true if the plan can be derived from some compound task, false otherwise
1  Function VERIFYPLAN
2      $S_0 \leftarrow B^+_{a_1}$
3      for $i$ = 1 to $n$ do
4          $S_i \leftarrow B^+_{a_{i+1}} \cup (S_{i-1} \setminus A^-_{a_i}) \cup A^+_{a_i}$
5      for $i$ = $n$-1 down to 0 do
6          $S_i \leftarrow S_i \cup (S_{i+1} \setminus A^+_{a_{i+1}})$
7          if $S_i \cap A^-_{a_i} \neq \emptyset$ then
8              return false
9      sp $\leftarrow \emptyset$; new $\leftarrow \{(A_i, i, i, I_i) \mid i \in 1..n\}$
       Data: $A_i$ is a primitive task corresponding to action $a_i$, $I_i$ is a Boolean vector of size $n$, such that $\forall i \in 1..n, I_i(i) = 1, \forall j \neq i, I_i(j) = 0$
10     while new $\neq \emptyset$ do
11         sp $\leftarrow$ sp $\cup$ new; new $\leftarrow \emptyset$
12         foreach decomposition method $R$ of the form $T_0 \to T_1, ..., T_k$ $[\prec, \text{pre}, \text{btw}]$ such that $\{(T_j, b_j, e_j, I_j) | j \in 1..k\} \subseteq$ sp do
13             if $\exists (i, j) \in \prec : \neg(e_i < b_j)$ then
14                 break
15             $b_0 \leftarrow \min\{b_j | j \in 1..k\}$
16             $e_0 \leftarrow \max\{e_j | j \in 1..k\}$
17             for $i$ = 1 to $n$ do
18                 $I_0(i) \leftarrow \sum_{j=1}^{k} I_j(i)$;
19                 if $I_0(i) > 1$ then
20                     break
21             if $\exists (U, p) \in \text{pre} : p \notin S_{\min\{b_j | j \in U\}-1}$ then
22                 break
23             if $\exists (U, V, p) \in \text{btw}\ \exists i \in \max\{e_j | j \in U\}, \ldots, \min\{b_j | j \in V\} - 1 : p \notin S_i$ then
24                 break
25             new $\leftarrow$ new $\cup \{(T_0, b_0, e_0, I_0)\}$
26             if $\forall k, I_0(k) = 1$ then
27                 return true
28     return false
```

Algorithm 1: Parsing-based HTN plan verification

a direct rewriting of constraint definitions. If all tests pass, the new task is added to a set of tasks (line 25). Then we know that the task decomposes to actions, which form a subsequence (not necessarily continuous) of the plan to be verified. The process is repeated until a task that decomposes to all actions is obtained (line 27) or no new task can be composed (line 10). The algorithm is *sound* as the returned task decomposes to all actions in the input plan. If the algorithm finishes with the value **false** then no other task can be derived. As there is a finite number of possible tasks, the algorithm has to finish, so it is *complete*.

The Plan Recognition Algorithm

Any plan verification algorithm, for example, the one from the previous section, can be extended to plan recognition by feeding the verification algorithm with actions $a_1, \ldots, a_{n+k}$, where we progressively increase k. The actions $a_1, \ldots, a_n$ are given as an input, while the actions $a_{n+1}, \ldots, a_{n+k}$ need to be generated (planned). However, this generate-and-verify approach would be inefficient for larger k as it requires exploration of all valid sequences of actions with the prefix $a_1, \ldots, a_n$. Assume that there could be 5 actions at the position $n+1$ and 6 actions at the position $n+2$. Then the generate-and-verify approach explores up to 30 plans (not every action at the position $n+2$ could follow every action at the position $n+1$) and for each plan the verification part starts from scratch as the plans are different.

This is where the verification algorithm from (Barták, Maillard, and Cardoso 2018) can be used as it does not require exactly one action at each position. The algorithm stores actions (sub-tasks) independently and only when it combines them to form a new task, it generates the states between the actions and checks the constraints for them. This resembles the idea of the Graphplan algorithm (Blum and Furst 1997). There are also sets of candidate actions for each position in the plan and the plan-extraction stage of the algorithm selects some of them to form a causally valid plan. We use compound tasks together with their decomposition constraints to select and combine the actions (we do not use parallel actions in the plan).

The algorithm from (Barták, Maillard, and Cardoso 2018) extended to the plan recognition problem is shown in Algorithm 2. It starts with actions $a_1, \ldots, a_n$ (line 2) and it finds all compound tasks that decompose to subsets of these actions (lines 4-30). This inner while-loop is taken from (Barták, Maillard, and Cardoso 2018), we only syntactically modified it to highlight the similarity with the verification algorithm from the previous section. If a task that decomposes to all current actions is found (line 30) then we are done. This is the goal task that we looked for and its timeline describes the recognized plan. Otherwise, we add all primitive tasks corresponding to possible actions at position $n+1$ (line 33). Note that these are not parallel actions, the algorithm selects exactly one of them for the plan.

Now, the parsing algorithm continues as it may compose new tasks that include one of those recently added primitive tasks. Notice that the algorithm uses all composed tasks from previous iterations in succeeding iterations so it does not start from scratch when new actions are added. This process is repeated until the goal task is found. The algorithm is clearly *sound* as the task found is the task that decomposes to the shortest plan with a given prefix. This goes from the soundness and completeness of the verification algorithm (in particular, no task that decomposes to a shorter plan exists). The algorithm is *semi-complete* as if there exists a plan with the length $n+k$ and with a given prefix, the algorithm will eventually find it at the $(k+1)$-th iteration. If no plan with a given prefix exists then the algorithm will not stop. However, recall that the plan recognition problem is undecidable (Behnke, Höller, and Biundo 2015) so any plan recognition approach suffers from this deficiency.

```
Data: a plan $\mathbf{P} = (a_1, ..., a_n)$, $A_i$ is a primitive task
      corresponding to action $a_i$, and a set of
      decomposition methods
Result: a Task that decomposes to a plan with prefix $\mathbf{P}$
1  Function RecognizePlan
2      new ← $\{(A_i, i, i, \{(B^+_{a_i}, \emptyset, a_i, A^+_{a_i}, A^-_{a_i})_i\}) | i \in 1..n\}$ ;
3      sp ← $\emptyset$; $l \leftarrow n$;
4      while new $\neq \emptyset$ do
5          sp ← sp ∪ new; new ← $\emptyset$;
6          foreach decomposition method $R$ of the form
             $T_0 \rightarrow T_1, ..., T_k[\prec, \mathrm{pre}, \mathrm{btw}]$ such that
             $\{(T_j, b_j, e_j, tl_j) | j \in 1..k\} \subseteq$ sp do
7              if $\exists (i,j) \in \prec : \neg(e_i < b_j)$ then
8                  break
9              $b_0 \leftarrow \min\{b_j | j \in 1..k\}$
10             $e_0 \leftarrow \max\{e_j | j \in 1..k\}$
11             $tl \leftarrow \{(\emptyset, \emptyset, empty, \emptyset, \emptyset)_i | i \in b_0..e_0\}$
12             for $j$ = 1 to k; $i = b_j$ to $e_j$ do
13                 $(\mathrm{Pre}^+_1, \mathrm{Pre}^-_1, a_1, \mathrm{Post}^+_1, \mathrm{Post}^-_1)_i \in tl$
14                 $(\mathrm{Pre}^+_2, \mathrm{Pre}^-_2, a_2, \mathrm{Post}^+_2, \mathrm{Post}^-_2)_i \in tl_j$
15                 if $a_1 \neq empty$, $a_2 \neq empty$ then
16                     break
17                 $\mathrm{Pre}^+_1 \leftarrow \mathrm{Pre}^+_1 \cup \mathrm{Pre}^+_2$
18                 $\mathrm{Pre}^-_1 \leftarrow \mathrm{Pre}^-_1 \cup \mathrm{Pre}^-_2$
19                 $\mathrm{Post}^+_1 \leftarrow \mathrm{Post}^+_1 \cup \mathrm{Post}^+_2$
20                 $\mathrm{Post}^-_1 \leftarrow \mathrm{Post}^-_1 \cup \mathrm{Post}^-_2$
21                 if $a_1 = empty$ then
22                     $a_1 \leftarrow a_2$
23             ApplyPre($tl, pre$);
24             ApplyBetween($tl, btw$);
25             Propagate($tl, b_0, e_0 - 1$);
26             if $\exists (\mathrm{Pre}^+, \mathrm{Pre}^-, a, \mathrm{Post}^+, \mathrm{Post}^-) \in tl$ :
                 $\mathrm{Pre}^+ \cap \mathrm{Pre}^- \neq \emptyset$ then
27                 break
28             new ← new ∪ $\{(T_0, b_0, e_0, tl)\}$
29             if $b_0 = 1, e_0 = l, \forall(\_,\_, a_j, \_,\_)_j \in tl$ :
                 $a_j \neq empty$ then
30                 return $(T_0, tl)$
31     $l \leftarrow l + 1$;
32     new ← $\{(A, l, l, \{(B^+_a, \emptyset, a, A^+_a, A^-_a)_l\}) |$
33         action $a$ can be at position $l$; $A$ is a primitive task for $a\}$
34     goto 4
```

Algorithm 2: Parsing-based HTN plan recognition

The algorithm maintains a timeline for each compound task to verify all the constraints. This is the major difference from the above verification algorithm that points to the original plan. This timeline has been introduced in (Barták, Maillard, and Cardoso 2018), where all technical details can be found. We include a short description to make the paper self-contained. A *timeline* is an ordered sequence of slots, where each slot describes an action, its effects, and the state right before the action. For task T, the actions in slots are exactly the actions from $act(T)$. Both effects and states are modelled using two sets of propositions, Post^+ and Post^- modeling positive and negative effects of the action and Pre^+ and Pre^- modeling propositions that must and must not be the true in the state right before the action. Two sets are used as the state is specified only partially and propositions are added to it during propagation so it is necessary to keep information about propositions that must not be true in the state.

The timeline always spans from the first to the last action of the task. Due to interleaving of tasks (actions from one task might be located between the actions of another task in the plan), some slots of the task might be *empty*. These empty slots describe "space" for actions of other tasks. When we are merging sub-tasks (lines 12-22), we merge their timelines, slot by slot. This is how the actions from sub-tasks are put together in a compound task. Notice, specifically, that it is not allowed for two merged sub-tasks to have actions in the same slot (line 15). This ensures that each action is generated by exactly one task.

```
Data: a set of $slots$, a set of $before$ constraints
Result: an updated set of slots
1 Function ApplyPre($slots, pre$)
2     foreach $(U, l) \in pre$ do
3         $id = \min\{b_j | j \in U\}$;
4         $\text{Pre}^+_{id} \leftarrow \text{Pre}^+_{id} \cup \{p | l = p\}$;
5         $\text{Pre}^-_{id} \leftarrow \text{Pre}^-_{id} \cup \{p | l = \neg p\}$
```

Algorithm 3: Apply before constraints

```
Data: a set of $slots$, a set of $between$ constraints
Result: an updated set of slots
1 Function ApplyBetween($slots, between$)
2     foreach $(U, V, l) \in between$ do
3         $s = \max\{e_i | i \in U\} + 1$;
4         $e = \min\{b_i | i \in V\}$;
5         for $id = s$ to $e$ do
6             $\text{Pre}^+_{id} \leftarrow \text{Pre}^+_{id} \cup \{p | l = p\}$;
7             $\text{Pre}^-_{id} \leftarrow \text{Pre}^-_{id} \cup \{p | l = \neg p\}$
```

Algorithm 4: Apply between constraints

Propositions from $before$ and $between$ constraints are "stored" in the corresponding slots (Algorithms 3 and 4) and their consistency is checked each time the slots are modified (line 26 of Algorithm 2). Consistency means that no proposition is true and false at the same state. Information between subsequent slots is propagated similarly to the verification algorithm (see Algorithm 5). Positive and negative propositions are now propagated separately taking in account empty slots. If there is no action in the slot then effects are unknown and hence propositions cannot be propagated.

```
Data: a set of slots $slots$
Result: an updated set of slots
1 Function Propagate($slots, lb, ub$)
      /* Propagation to the right        */
2     for $i = lb$ to $ub$ do
3         if $a_i \neq empty$ then
4             $\text{Pre}^+_{i+1} \leftarrow$
                  $\text{Pre}^+_{i+1} \cup (\text{Pre}^+_i \setminus \text{Post}^-_i) \cup \text{Post}^+_i$;
5             $\text{Pre}^-_{i+1} \leftarrow$
                  $\text{Pre}^-_{i+1} \cup (\text{Pre}^-_i \setminus \text{Post}^+_i) \cup \text{Post}^-_i$
      /* Propagation to the left         */
6     for $i = ub$ down to $lb$ do
7         if $a_i \neq empty$ then
8             $\text{Pre}^+_i \leftarrow \text{Pre}^+_i \cup (\text{Pre}^+_{i+1} \setminus \text{Post}^+_i)$;
9             $\text{Pre}^-_i \leftarrow \text{Pre}^-_i \cup (\text{Pre}^-_{i+1} \setminus \text{Post}^-_i)$
```

Algorithm 5: Propagate

Example

A unique property of the proposed techniques is handling task interleaving – actions generated from different tasks may interleave to form a plan. This is the property that parsing techniques based on CF grammars cannot handle.

The example in Figure 1 demonstrates how the timelines are filled by actions as the tasks are being derived/composed from the plan. Assume, first, that a complete plan consisting of actions $a_1, a_2, \ldots, a_7$ is given. The plan recognition algorithm can also handle such situations, when a complete plan is given, so it can serve for plan verification too (the verification variant of Algorithm 2 should stop with a failure at line 33 as no action can be added during plan verification). In the first iteration, the algorithm will compose tasks T_2, T_3, T_4 as these tasks decompose to actions directly. Notice, how the timelines with empty slots are constructed. We know where the empty slots are located as we know the exact location of actions in the plan. In the second iteration, only the task T_1 is composed from already known tasks T_3 and T_4. Notice how the slots from these tasks are copied to the slots of a new timeline for T_1. On the contrary, the slots in original tasks remain untouched as these tasks may merge with other tasks to form alternative decomposition trees (see the discussion below). Finally, in the third iteration, tasks T_1 and T_2 are merged to a new task T_0 and the algorithm stops there as a complete timeline that fully spans the plan is obtained (condition at line 30 of Algorithm 2 is satisfied).

Let us assume that there is a constraint *between*$(\{a_1\}, \{a_3\}, p)$ in the decomposition method for T_3. For example, this constraint may model a causal link between a_1 and a_3. When composing the task T_3, the second slot of its timeline remains empty, but the proposition p is placed there (see Algorithm 4). This proposition is then copied to the timeline of task T_1, when merging the timelines (line 17 of Algorithm 2), and finally also to the timeline of task T_0. During each merge operation, the algorithm checks that p can still be in the slot, in particular, that p is not required to be false at the same slot

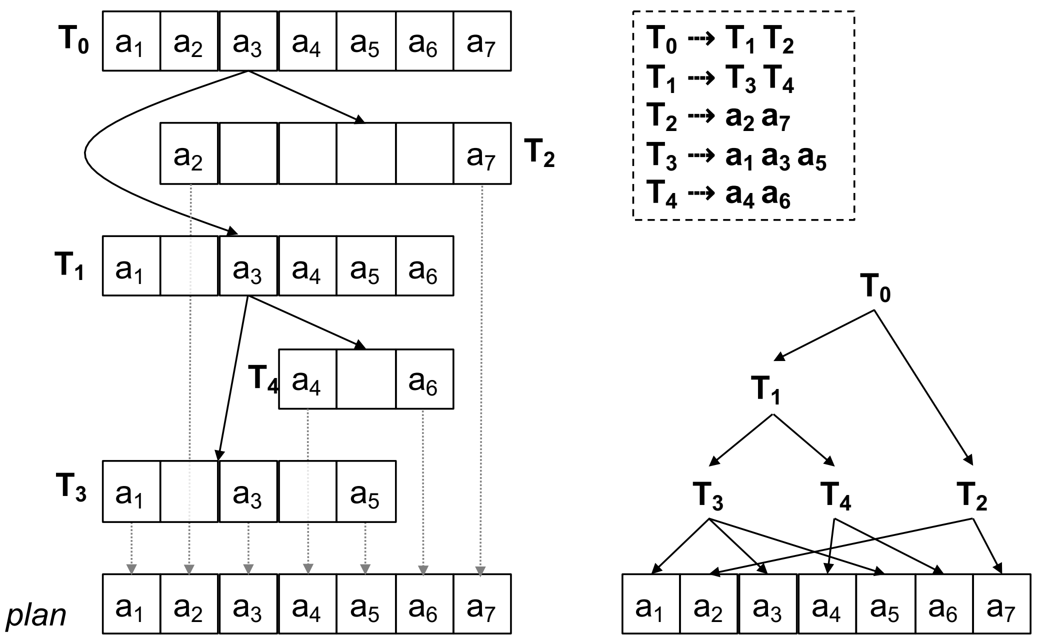


Figure 1: Example of parsing-based plan verification/recognition (the right side shows the decomposition tree with the decomposition rules above it; the left side shows the tasks with timelines and filled slots)

(line 26). Algorithm 2 repeatedly checks the constraints from methods.

The new plan verification algorithm (Algorithm 1) handles the method constraints more efficiently as it uses the complete plan with states to check them. Moreover, the propagation of states is run just once in Algorithm 1 (lines 2-8), while Algorithm 2 runs it repeatedly each time the task is composed from subtasks. Hence, each constraint is verified just once in Algorithm 1, when a new task is composed. In particular, the constraint *between*$(\{a_1\}, \{a_3\}, p)$ is verified with respect to the states when task T_3 is introduced. Otherwise, both Algorithm 1 and Algorithm 2 derive the tasks in the same order (if the decomposition methods are explored in the same order). Instead of timelines, Algorithm 1 uses the Boolean vector I to identify actions belonging to each task. For example, for task T_3 the vector is $[1, 0, 1, 0, 1, 0, 0]$ and for task T_4 it is $[0, 0, 0, 1, 0, 1, 0]$. When composing task T_1 from T_3 and T_4 the vectors are merged to get $[1, 0, 1, 1, 1, 1, 0]$ (see the loop at line 17). Notice that the vector always spans the whole plan, while the timelines start at the first action and finish with the last action of the task (and hence the same timeline can be used for different plan lengths).

Assume now that only plan prefix consisting of $a_1, a_2, \ldots, a_6$ is given. The plan recognition algorithm (Algorithm 2) will first derive tasks T_3 and T_4 only. Specifically, task T_2 cannot be derived yet as action a_7 is not in the plan. In the second iteration, the algorithm will derive task T_1 by merging tasks T_3 and T_4, exactly as we described above. As no more tasks can be derived, the inner loop finishes and the algorithm attempts to add actions that can follow the prefix $a_1, a_2, \ldots, a_6$ (line 33). Let action a_7 be added at the 7-th position in the plan; actually all actions, that can follow the prefix, will be added as separate primitive tasks at position 7. Now the inner loop is restarted and task T_2 will be added in its first iteration. In the next iteration, task T_0 will be added and this will be the final task as it satisfies the condition at line 30.

Assume, hypothetically, that the verification Algorithm 1 is used there. When it is applied to plan $a_1, a_2, \ldots, a_6$, the algorithm derives tasks T_1, T_3, T_4 and fails as no task spans the whole plan and no more tasks can be derived. After adding action a_7, the algorithm will start from scratch as the states might be different due to propagating some propositions from the precondition of a_7. Hence, the algorithm needs to derive the tasks T_1, T_3, T_4 again and it will also add tasks T_0, T_2 and then it will finish with success.

It may happen, that action a_5 can also be consistently placed to position 7. Then, we can derive two versions of task T_3, one with the vector $[1, 0, 1, 0, 1, 0, 0]$ and the other one with vector $[1, 0, 1, 0, 0, 0, 1]$. Let us denote the second version as T_3'. Both versions can then be merged with task T_4 to get two versions of task T_1, one with the vector $[1, 0, 1, 1, 1, 1, 0]$ and one with the vector $[1, 0, 1, 1, 0, 1, 1]$. Let us denote the second version as T_1'. The Algorithm 1 will stop there as no more tasks can be derived. Notice that tasks T_1, T_3, T_4 were derived repeatedly. If we try a_5 earlier than a_7 at position 7 then tasks T_1, T_3, T_4 will actually be generated three times before the algorithm finds a complete plan. On the contrary, Algorithm 2 will add actions a_5 and a_7 together as two possible primitive tasks at position 7. It will use tasks T_1, T_3, T_4 from the previous iteration, it will add tasks T'_1, T'_3 as they can be composed from the primitive tasks (using the last a_5), it will also add tasks T_0, T_2 (using the last a_7), and will finish with success. Notice that T'_1 cannot be merged with T_2 to get a new T'_0 as T'_1 has action a_5 at the 7-th slot while T_2 has a_7 there so the timelines cannot be merged (line 15 of Algorithm 2).

Possible Extensions

To describe the verification and recognition algorithms, we used a "pure" model of HTN. Specifically, each task decomposes to a non-empty set of sub-tasks, meaning that the right-hand side of each derivation rule is non-empty. In some practical applications, it might be useful to also use decomposition methods with empty task networks. Imagine a task describing that some agent moves to a specific location. This task can be full-filled by action *move* so there will be a method, where the task decomposes to this action. However, if the agent is already at the specific location then no action is necessary and the task is already full-filled. This can be modeled by an alternative method that decomposes the task to an empty task network with the precondition (*before*) constraint specifying that the agent is at the required location. Such empty methods can be compiled away, for example, using the techniques for converting grammars to a normal form. Nevertheless, the presented verification and recognition algorithms can also be extended to handle derivation rules with empty right-hand side. We will demonstrate this extension for the verification Algorithm 1. Note, that tasks that decompose to an empty task network are treated in a similar way as tasks that decompose directly to actions, that is, they are added in the initialization stage (line 9). We only need to identify the proper location indexes of these tasks and this is where the $before$ constraint can be used. Assume the following method with empty right-hand side:

$$T \rightarrow \emptyset \quad [\mathit{before}(\emptyset, p)].$$

First, the constraint $\mathit{before}(U, p)$ has originally been defined for a non-empty subset U of tasks in the task network, but the task network is now empty so, in this special case, we allow $U = \emptyset$. Second, the verification algorithm already calculated all the states S_i between the actions. The precondition constraint tells us, where the task T can be inserted. Specifically, if $p \in S_i$, that is, the precondition constraint holds at state S_i, then we add a primitive task $(T, i+0.1, i+0.1, I)$ to the initial set of tasks *new* (line 9 of Algorithm 1), where the Boolean vector I consists of zeros only (the task T does not decompose to any action). We use the $(i+0.1)$ index as the task T is sitting between actions A_i and A_{i+1} and we need to ensure that possible precedence constraints involving T work fine. The rest of the verification algorithm remains without further modification, we only need to properly round the indexes when checking the state constraints.

The second extension, that we are going to discuss, is about the top task to be recognized/verified. Recall, that neither of the proposed techniques requires a top task to be given at input. In some applications, a task network with constraints is given as input and the plan should correspond to this network. This can be trivially handled by the proposed techniques by introducing, to the HTN model, a dummy root task that decomposes to this task network and modifying the terminal conditions of the algorithms to look for this specific root task rather than for any task (line 27 of Algorithm 1 and line 30 of Algorithm 2). However, what if the plan consists of interleaved sub-plans obtained from several tasks that are not known a priori? This situation can also be handled by modifying the termination condition. Instead of looking for a single task that spans the whole plan, we need to look for a set of already recognized tasks such that they do not share any action and, together, they span the whole plan. Note, however, that such a test can be computationally expensive if implemented in a naive way by checking all subsets of tasks.

Conclusions

In the paper, we proposed two versions of a parsing technique for verification of HTN plans and for recognition of HTN plans. To the best of our knowledge, these are the only approaches that fully cover HTN, including all decomposition constraints. These approaches can be applied to solve both verification and recognition problems, but as we demonstrated using an example, each of them has some deficiencies when applied to the other problem.

The next obvious step is implementation and empirical evaluation of both techniques. There is no doubt that the novel verification algorithm is faster than the previous approaches (Behnke, Höller, and Biundo 2017) and (Barták, Maillard, and Cardoso 2018). The open question is how much faster it will be, in particular for large plans. The efficiency of the novel plan recognition technique in comparison to existing compilation technique (Höller et al. 2018) is less clear as both techniques use different approaches, bottom-up vs. top-down. The disadvantage of the compilation technique is that it needs to re-generate the known plan prefix, but it can exploit heuristics to remove some overhead there. On the contrary, the parsing techniques looks more like generate-and-test, but controlled by the hierarchical structure. It also guarantees finding the shortest extension of plan prefix.

Acknowledgements Research is supported by the Czech Science Foundation under the project P103-18-07252S.

References

[Avrahami-Zilberbrand and Kaminka 2005] Avrahami-Zilberbrand, D., and Kaminka, G. A. 2005. Fast and complete symbolic plan recognition. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, 653–658. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Barták and Maillard 2017] Barták, R., and Maillard, A. 2017. Attribute grammars with set attributes and global constraints as a unifying framework for planning domain models. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, 39–48. New York, NY, USA: ACM.

[Barták, Maillard, and Cardoso 2018] Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of hierarchical plans via parsing of attribute grammars. In *ICAPS*, 11–19. AAAI Press.

[Behnke, Höller, and Biundo 2015] Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In Brafman, R. I.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, 25–33. AAAI Press.

[Behnke, Höller, and Biundo 2017] Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) - verifying solutions of hierarchical planning problems. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, 20–28. AAAI Press.

[Blum and Furst 1997] Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1):281 – 300.

[Erol, Hendler, and Nau 1996] Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Ann. Math. Artif. Intell.* 18(1):69–93.

[Fikes and Nilsson 1971] Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd international joint conference on Artificial intelligence*, IJCAI'71, 608–620.

[Geib and Goldman 2009] Geib, C. W., and Goldman, R. P. 2009. A probabilistic plan recognition algorithm based on plan tree grammars. *Artif. Intell.* 173(11):1101–1132.

[Geib, Maraist, and Goldman 2008] Geib, C. W.; Maraist, J.; and Goldman, R. P. 2008. A new probabilistic plan recognition algorithm based on string rewriting. In *ICAPS*, 91–98. AAAI.

[Höller et al. 2014] Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 447–452. IOS Press.

[Höller et al. 2018] Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and goal recognition as HTN planning. In *ICTAI*, 466–473. IEEE.

[Kabanza et al. 2013] Kabanza, F.; Filion, J.; Benaskeur, A. R.; and Irandoust, H. 2013. Controlling the hypothesis space in probabilistic plan recognition. In *IJCAI*, 2306–2312. IJCAI/AAAI.

[Ramírez and Geffner 2003] Ramírez, M., and Geffner, H. 2003. Plan recognition as planning. In *IJCAI*.

[Vilain 1990] Vilain, M. 1990. Getting serious about parsing plans: A grammatical analysis of plan recognition. In *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1*, AAAI'90, 190–197. AAAI Press.