31st International Conference on
Automated Planning and Scheduling

August 2 – 13, 2021, virtually from Guangzhou, China

HPlan 2021

Proceedings of the 4th ICAPS Workshop on

Hierarchical Planning

Program Committee

Ron Alford	The MITRE Corporation, McLean, Virginia, USA
Roman Barták	Charles University, Prague, Czech Republic
Gregor Behnke	University of Freiburg, Germany
Pascal Bercher	The Australian National University, Canberra, Australia
Arthur Bit-Monnot	Laboratory for Analysis and Architecture of Systems, LAAS-CNRS, Toulouse, France.
Dillon Chen	The Australian National University, Canberra, Australia
Lavindra de'Silva	University of Cambridge, United Kingdom
Juan Fernández-Olivares	University of Granada, Granada, Spain
Florian Geißer	The Australian National University, Canberra, Australia
Alban Grastien	The Australian National University, Canberra, Australia
Daniel Höller	Saarland University, Saarbrücken, Germany
Jane Jean Kiam	University of the Bundeswehr Munich, Germany
Ugur Kuter	SIFT, LLC, Minneapolis, USA
Songtuan Lin	The Australian National University, Canberra, Australia
Mauricio Cecilio Magnaguagno	Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
Conny Olz	Ulm University, Germany
Simona Ondrčková	Charles University, Prague, Czech Republic
Sunandita Patra	University of Maryland, College Park, Maryland, USA
Felix Richter	Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, Stuttgart, Germany
Dominik Schreiber	Karlsruhe Institute of Technology, Karlsruhe, Germany
Shirin Sohrabi	IBM, Thomas J. Watson Research Center, Yorktown Heights, NY USA
David Speck	University of Freiburg, Germany
Alvaro Torralba	Aalborg University, Denmark
Julia Wichlacz	Saarland University, Saarbrücken, Germany
Zhanhao Xiao	Sun Yat-sen University, Guangzhou, China

Organizing Committee

Pascal Bercher	The Australian National University, Canberra, Australia
Jane Jean Kiam	University of the Bundeswehr Munich, Germany
Zhanhao Xiao	Sun Yat-Sen University, Guangzhou, China
Ron Alford	The MITRE Corporation, McLean, Virginia, USA

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

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

In 2021, the 4th edition of the workshop, we received an astonishing 14 submissions. As in all previous workshops, each paper received at least three reviews, and sometimes four when necessary. Each reviewer had to review at most 2 submissions, which were assigned by interest and expertise. Reviewers crafted over 32 thousand words of commentary, and average of 2,300 per paper, with a quality comparable to reviews in major top-tier conferences. In the end, nine papers were unconditionally accepted. For the other five papers, authors used the reviews to significantly improve their submission, and were accepted after a second round of reviewing.

Like in previous years, a range of topics is addressed in the papers of this workshop.

Two papers are concerned with planning under uncertainty. One of them investigates the computational complexity for fully observable HTN problems with actions having non-deterministic effects. The other describes an HTN planner for solving POMDPs by integrating Monte Carlo Tree Search. Several other works are concerned with solving (hierarchical) planning problems as well. One approach builds upon hybrid planning, a formalism fusing HTN planning with Partial Order Causal Link (POCL) planning. It proposes a preprocessing plus postprocessing technique aimed at reducing the search space by precomputing solutions to subproblems. Two further papers introduce HTN planning systems – one is designed for Goal Task Network (GTN) problems, an extension of HTN problems that allows for the specification of partially ordered goals, and one that is designed for restarting at positions where execution errors occur in the context of integrated planning and acting.

Multiple papers are concerned with or related to plan verification. One paper is concerned with an algorithm based on grammar parsing for verifying that an action sequence is a solution to a totally ordered HTN problem thus exploiting the total order for higher efficiency. Another paper is concerned with HTN plan verification as well, but it proposes a novel way of doing so: It relies on a compilation from such a verification problem into a standard HTN planning problem. Two further papers are devoted to the case when plan verification fails, i.e., in case the provided plan is not a solution to the given HTN planning problem. One of them proposes changing the plan by removing the minimal number of actions to turn it into a solution, whereas the other changes the model instead. The former paper proposes and evaluates a technique based on grammar parsing, whereas the latter does not provide any technique but studies the computational complexity of the underlying decision problem.

Several papers propose applications of hierarchical planning to concrete problems. The applications include multi-agent path finding, the multi-robot task allocation problem (via hierarchical auctions), as well as provisioning a service agent for assistance in carrying out everyday tasks.

As first offered in 2020, this year we again encouraged the submission of challenge papers aimed to highlight important problems in the field of hierarchical planning. Of the two challenge papers accepted, one points out some major challenges in temporal HTN planning to increase its applicability for real-world problems involving nested multi-vehicle routing problems. The other paper argues that by introducing a metric to measure "robustness", plans found using hierarchical planning are more capable of coping with dynamic environments, which is often an issue in real-world problems. We hope that highlighting these challenges will spark interesting discussions at the workshop and lead to potential solutions in the future.

Just like in 2020, both the main conference as well as the workshop were done purely online, executed via gather.town. Last year, it went exceptionally well: Both the invited talk and the poster sessions were well attended with rich discussions, so we are looking forward to this year's virtual HPlan as well!

Due to the much higher number of accepted papers compared to last years (7, 7, and 6 in 2018–2020, with 14 in 2021), we had to increase the workshop length from half-day (4 hours) to 3/4-day (6 hours). All papers will be shortly announced with teaser talks of 5 to 10 minutes, and then discussed in more depth in four poster sessions, each taking 45 minutes and hosting 3 or 4 posters, depending on the session.

As in previous years, we also feature an invited talk, this year by Malik Ghallab, who introduces his work on Hierarchical Online Reasoning for the Integration of Planning and Acting. On the next pages you will find an abstract of the talk, as well as a biography of the speaker.

Pascal, Jean, Zhanhao, and Ron,
HPlan Workshop Organizers,
August 2021

Invited Talk

Each year so far we had one or two invited talks. This year, by *Malik Ghallab*.

Hierarchical Online Reasoning for the Integration of Planning and Acting

Hierarchization in planning has often been viewed mainly as a means for reducing the search complexity, to be paid for with additional domain modeling efforts. HTN planning, for example, has sometime been opposed to generative planning techniques, and referred to as a programming paradigm in planning. We pursue here a quite different motivation for hierarchization than tractable computations in a huge search space. Namely, we are interested in planning for the purpose of acting, and consider hierarchization as a central concept for the integration of reasoning on actions and performing them.

We have argued in [2] that the design of a cognitive actor has to rely on two interconnected principles: (i) *hierarchically organized deliberation*, and (ii) *continual online planning and reasoning*. Many challenging problems for such a design have been underlined in [2], several of which remain pending, while a few have progressed towards acceptable solutions. This talk will report on a line of work that illustrates such a progress.[1] Initiated in [1, Chap. 3], the work was pursued through several algorithmic developments and trials, e.g., in [6, 3, 5]; it reached a comprehensive stage in [4].

The talk will motivate the integrated planning and acting issues and present three technical components:

- A hierarchical task-oriented knowledge representation for expressing *operational models* of actions (*how* to do things), which relies on a collection of refinement methods describing alternative ways to handle tasks and react to events. A refinement method can be any complex algorithm, with subtasks to be refined recursively and *nondeterministic* primitive actions which query and change the world.
- A Refinement Acting Engine (RAE) which interacts with an execution platform and performs online reasoning for the achievement of tasks and reaction to events by following refinement methods adapted to the current dynamic context, and retrying alternative methods when needed. RAE chooses its refinement methods with the help of a online optimizing planner.
- A Monte Carlo Tree Search planner, called UPOM, which assesses the utility of possible methods and finds an approximately optimal one for RAE to pursue an ongoing activity. UPOM is a progressive deepening, receding-horizon anytime planner which relies on domain-dependent heuristics, learned from simulations and/or real-world interactions.

[1] A line of work in collaboration with my co-authors Dana Nau, Paolo Traverso, Sunandita Patra and James Mason.

A few empirical results will be presented.[2] Extensions of the approach for handling temporal issues as well as space and motion planning issues (to address the well known "*Task and Motion Planning*" TAMP problems) will be briefly discussed, together with perspectives for learning refinement methods for operational models.

References

[1] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[2] Malik Ghallab, Dana Nau, and Paolo Traverso. "The Actor's View of Automated Planning and Acting: A Position Paper". In: *Artificial Intelligence* 208 (Mar. 2014), pp. 1–17.

[3] Sunandita Patra, Malik Ghallab, Dana Nau, and Paolo Traverso. "APE: An Acting and Planning Engine". In: *Advances in Cognitive systems* 7 (2019), pp. 175–194.

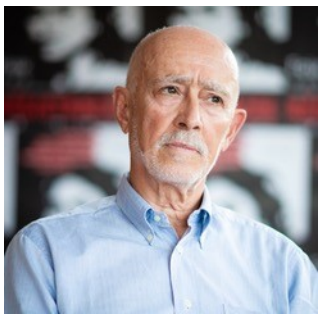
[4] Sunandita Patra, James Mason, Malik Ghallab, Dana Nau, and Paolo Traverso. "Deliberative acting, planning and learning with hierarchical operational models". In: *Artificial Intelligence* 299 (2021), p. 103523. ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103523. URL: https://www.sciencedirect.com/science/article/pii/S0004370221000746.

[5] Sunandita Patra, James Mason, Malik Ghallab, Paolo Traverso, and Dana Nau. "Integrating Acting, Planning, and Learning in Hierarchical Operational Models". In: *International Conference on Automated Planning and Scheduling (ICAPS 2020)*. AAAI Press, 2020, pp. 1–10.

[6] Sunandita Patra, Paolo Traverso, Malik Ghallab, and Dana Nau. "Acting and Planning Using Operational Models". In: *AAAI Conference on Artificial Intelligence (AAAI 2019)*. AAAI Press, 2019, pp. 7691–7698.

Bio

Malik Ghallab, Directeur de recherche emeritus at CNRS
LAAS-CNRS, 7 Av. Colonel Roche, 31077 Toulouse, France



The research activity of Malik Ghallab is focused on AI and Robotics. He contributed to topics such as knowledge representation and reasoning, planning, and learning of skills and models of behaviors. He (co-)authored over 200 technical papers and several books. He taught AI at a few universities in France and abroad; he advised 32 PhDs. He was director of several AI research programs in France, director of LAAS-CNRS and CTO of INRIA. He chairs the Steering Committee of ANITI, the interdisciplinary AI institute of Toulouse. His is involved in initiatives regarding socially responsible research in AI and computational sciences. He is ECCAI Fellow, and Docteur Honoris Causa of Linköping University, Sweden.

[2] A "*Demonstration of Refinement Acting, Planning and Learning System Using Operational Models*" will be presented at the ICAPS 2021 demonstration session (https://icaps21.icaps-conference.org/demos/).

Table of Contents

Scientific Papers

Challenge Papers

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

A Hierarchical Approach to Multi-Agent Path Finding

Han Zhang, Mingze Yao, Ziang Liu, Jiaoyang Li, Lucas Terr, Shao-Hung Chan, T. K. Satish Kumar, Sven Koenig
University of Southern California
{zhan645, mingzeya, ziangliu, jiaoyanl, terr, shaohung}@usc.edu, tkskwork@gmail.com, skoenig@usc.edu

Abstract

The Multi-Agent Path Finding (MAPF) problem arises in many real-world applications, ranging from automated warehousing to multi-drone delivery. Solving the MAPF problem optimally is NP-hard, and existing optimal and bounded-suboptimal MAPF solvers thus usually do not scale to large MAPF instances. Greedy MAPF solvers scale to large MAPF instances, but their solution qualities are often bad. In this paper, we therefore propose a novel MAPF solver, Hierarchical Multi-Agent Path Planner (HMAPP), which creates a spatial hierarchy by partitioning the environment into multiple regions and decomposes a MAPF instance into smaller MAPF sub-instances for each region. For each sub-instance, it uses a bounded-suboptimal MAPF solver to solve it with good solution quality. Our experimental results show that HMAPP solves as large MAPF instances as greedy MAPF solvers while achieving better solution qualities on various maps.

Introduction

The Multi-Agent Path Finding (MAPF) problem arises in many real-world applications, including automated warehousing (Wurman, D'Andrea, and Mountz 2008; Li et al. 2020) and multi-drone delivery (Choudhury et al. 2020). In the MAPF problem, each agent is required to move from a start vertex to a goal vertex on an undirected graph while avoiding conflicts with other agents. A conflict happens when two agents stay at the same vertex or traverse the same edge in opposite directions at the same time.

Two common objectives for the MAPF problem are minimizing the sum of the path costs and minimizing the makespan. Solving the MAPF problem optimally for either objective is NP-hard (Yu and LaValle 2013; Ma et al. 2016). Thus, existing optimal and bounded-suboptimal MAPF solvers (Sharon et al. 2015; Barer et al. 2014) usually do not scale to large MAPF instances. Greedy MAPF solvers (Silver 2005) are able to scale to large MAPF instances, but their solution qualities are often bad.

Although planning can find MAPF solutions of good quality for small MAPF instances, planning in small steps from one vertex to another has the disadvantage that its runtime can dramatically increase with the number of agents and the size of the environment. In this paper, we approach the MAPF problem from a rarely-pursued spatial-hierarchy perspective. We propose a novel MAPF solver, Hierarchical Multi-Agent Path Planner (HMAPP). In HMAPP, a high-level planner generates a high-level plan for each agent that moves the agent from one region to another, and each regional planner subsequently refines the high-level plan to a low-level path for the agent. Therefore, regional planners can use existing MAPF techniques to find solutions with good qualities while the total runtime of HMAPP is still reasonable for large MAPF instances.

Our experimental results show that HMAPP solves as large MAPF instances as greedy MAPF solvers while achieving better solution qualities on various maps. The solutions of HMAPP have makespans for large MAPF instances that are about 50% smaller than the ones of the spatial-hierarchical MAPF solver Ros-dmapf (Pianpak et al. 2019).

Related Work

Spatial hierarchies have been used for path planning (Botea, Müller, and Schaeffer 2004; Pelechano and Fuentes 2016) by partitioning a map into several regions, precomputing and caching the optimal sub-paths that connect adjacent regions and abstracting these sub-paths to edges of a smaller abstract graph, that is then searched. These approaches do not directly apply to MAPF since the cached sub-paths do not take conflicts between agents into account and are thus difficult to reuse for MAPF.

Hierarchies have also been used for multi-agent motion planning (Kapadia et al. 2013; Ma et al. 2017), but these approaches do not use spatial hierarchies but rather planning hierarchies that plan on different abstraction levels, such as path and motion planning. HMAPP can be used for path planning in such approaches.

Spatial hierarchies have not yet been used extensively for MAPF. The Spatially Distributed Multi-Agent Planner (SDP) (Wilt and Botea 2014) partitions a map into high- and low-contention regions and uses different MAPF solvers for regions of different types. Unlike HMAPP, SDP does not partition the map into several regions in the absence of high-contention regions and cannot solve MAPF instances unless all start or goal vertices are in low-contention regions. Ros-dmapf (Pianpak et al. 2019), like HMAPP, partitions a map into several regions. Unlike HMAPP, Ros-dmapf uses answer set programming for the regional planners and has to synchronize the execution of the high-level plans of

all agents, causing agents that reach their next regions earlier than other agents to wait unnecessarily for those other agents, which impacts the solution quality negatively.

Preliminaries

In this section, we provide background material on MAPF, the optimal MAPF solver Conflict-Based Search (CBS) and the bounded-suboptimal MAPF solver Enhanced Conflict-Based Search (ECBS).

MAPF

The MAPF problem is defined by an undirected graph $G = (V, E)$ and a set of m agents $\{a_1 \dots a_m\}$. Each agent has a start vertex $s_i \in V$ and a goal vertex $g_i \in V$. In each timestep, an agent either moves to an adjacent vertex or waits at its current vertex. Both move and wait actions have unit cost unless the agent terminally waits at its goal vertex, which has zero cost. A *path* of an agent is a sequence of move and wait actions from its start vertex to its goal vertex. A *sub-path* of an agent is a sequence of actions from one vertex at a specific timestep to another vertex at a specific timestep. The *path cost* of a path is the accumulated cost of all actions in this path. A *vertex conflict* happens when two agents stay at the same vertex simultaneously, and an *edge conflict* happens when two agents traverse the same edge in opposite directions simultaneously. A *solution* is a set of conflict-free paths of all agents. The *Sum of path Costs* (SoC) is the sum of the path costs of the paths of all agents, and the *makespan* is the maximum path cost of the paths of all agents. In this paper, we consider only graphs that are four-neighbor grids (Stern et al. 2019). However, HMAPP can be applied to any graph as long as a graph-partitioning approach is provided for it.

CBS and ECBS

CBS (Sharon et al. 2015) is an optimal two-level MAPF solver. On the high level, CBS maintains a *Constraint Tree* (CT). Each CT node contains a set of constraints and a set of paths, one for each agent, that satisfies all these constraints. The cost of a CT node is the SoC or makespan of all these paths, depending on the objective of the MAPF problem. On the low level, for each CT node, CBS finds a path for each agent that has the smallest path cost while satisfying all constraints of the CT node (but might conflict with the other paths). When expanding a CT node, CBS returns a solution if its paths are conflict-free. Otherwise, CBS picks a conflict, splits the CT node into two child CT nodes and adds a constraint to each child CT node to prohibit either one or the other of the two conflicting agents from using the conflicting vertex or edge at the conflicting timestep. On the high level, CBS expands nodes in a best-first order. Therefore, the paths of the first expanded CT node with conflict-free paths form an optimal solution.

ECBS(w) (Barer et al. 2014) is a bounded-suboptimal MAPF solver based on CBS. Given suboptimality factor w, ECBS(w) finds a w-suboptimal solution. The high- and low-level search algorithms of ECBS are focal search (Pearl and Kim 1982) instead of best-first search. Unlike best-first search, focal search maintains a FOCAL list, which is a subset of the OPEN list of search nodes, and expands nodes from the FOCAL list based on a user-provided tie-breaking criterion. On the low-level, ECBS uses focal search to find paths that have fewer conflicts with the paths of other agents. On the high-level, ECBS uses focal search to expand CT nodes that more likely lead to a conflict-free bounded-suboptimal solution.

Algorithm 1: HMAPP.

```
   input: A MAPF instance.
1  initialize();
2  find_HL_plan();
3  T ← 0;
4  foreach region r ∈ R do
5  |  P_r.plan_initial_path();
6  end
7  while paths for all agents to their goal vertices have
     not yet been found do
8  |  T ← next timestep when an agent is ready to
   |    enter its next region;
9  |  foreach region r ∈ R do
10 |  |  A' ← agents that are ready to enter r at
   |  |    timestep T;
11 |  |  if A' is not empty then
12 |  |  |  P_r.replan(A');
13 |  |  end
14 |  end
15 |  foreach region r ∈ R do
16 |  |  if an agent is delayed to exit r at timestep T
   |  |    then
17 |  |  |  P_r.replan(∅);
18 |  |  |  if P_r.replan failed to find a solution then
19 |  |  |  |  return failure;
20 |  |  |  end
21 |  |  end
22 |  end
23 end
24 return extract_solution();
```

HMAPP

Algorithm 1 shows the pseudo-code of HMAPP. HMAPP first partitions the vertices into regions. Let R denote the set of all regions. For each pair of adjacent regions, HMAPP finds pairs of adjacent vertices (one from each region), called *boundary pairs*, and uses them to transfer agents between regions. To simplify the interaction between regions, agents are allowed to travel in only one direction through each boundary pair. A *high-level planner* generates a *high-level plan* for each agent (Line 2), which specifies the sequence of regions that the agent should visit to reach its goal vertex. When we describe HMAPP, we assume that the high-level plan of each agent does not include each region more than once so that each agent has at most one sub-path in each region. However, this assumption is only for the ease of pre-

sentation. HMAPP allows the high-level plan of an agent to include a region multiple times and maintains one sub-path for each visit of the agent to the region.

For an agent a_i that moves from region r to its next region r', the *regional planner* $\mathcal{P}_r$ plans a sub-path for a_i to a boundary vertex v that is part of a boundary pair $\langle v, v' \rangle$ to region r'. v (v') is the determined *exit (entry) vertex* of a_i from r (to r'). The *exit (entry) timestep* of a_i from r (to r') is the last timestep that a_i is in r. $\mathcal{P}_r$ initially assumes that a_i immediately exits r once it has followed its sub-path in r. However, the actual exit timestep is determined by the regional planner $\mathcal{P}_{r'}$ when $\mathcal{P}_{r'}$ determines the entry timestep and the sub-path for a_i in r', which must not be smaller than the timestep when a_i has followed its sub-path in r. Once determined, the entry and exit timesteps and the entry and exit vertices of agents can no longer be changed.

In the beginning of Algorithm 1, for each region r, $\mathcal{P}_r$ plans a set of conflict-free sub-paths for all agents in the region (Lines 4-6). We say that agent a_i is *ready to enter* (*exit*) region r' (r) at a timestep t iff (1) a_i has followed its sub-path in r at timestep t and (2) the exit timestep of a_i from r has not been determined yet. Inside the while loop (Lines 7-23), T is updated to the earliest timestep when an agent is ready to enter its next region. HMAPP iterates over each region r and invokes $\mathcal{P}_r$ to determine the entry timesteps for agents that are ready to enter r at timestep T (Lines 9-14). Let a_i be such an agent. We say that a_i is *delayed to exit* its region iff its determined entry timestep is larger than T. $\mathcal{P}_r$ might have to replan the sub-paths of all agents in r if such a delay happens. HMAPP iterates over each region r that has a delayed-to-exit agent and invokes $\mathcal{P}_r$ to replan the sub-paths of all agents (Lines 15-22). Except for the initial planning, each regional planner plans at most twice for each value of T, once to take the ready-to-enter agents into account and once to take the delayed-to-exit agents into account. When replanning, each regional planner is allowed to modify the entire sub-paths of its agents in the region (even the parts before timestep T) as long as they obey the determined entry and exit timesteps and the determined entry and exit vertices. HMAPP repeats this procedure until it has found paths for all agents to their goal vertices. Finally, HMAPP appends the sub-paths in different regions and returns the obtained paths as the solution (Line 24).

The resulting paths are conflict-free because (1) the sub-paths inside each region are conflict-free and (2) no edge conflict happens when an agent exits a region since the movements within each boundary pair are in one direction only. However, HMAPP is not a complete MAPF solver since the sub-instances for the regions can be unsolvable even if a solution for the MAPF instance exists. Limiting the number of agents in each region may make HMAPP complete, which we leave for future work.

HMAPP is a general algorithmic framework that can use different approaches for graph partitioning, high-level planning and regional planning. In the following sections, we describe how each of these components is implemented currently.

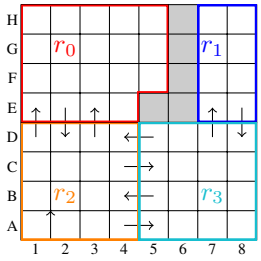


Figure 1: Shows an example of partitioning an 8×8 grid into four regions r_i (for $i = 0, \ldots 3$), each with a different color. Shaded areas are obstacles. Arrows between adjacent vertices indicate boundary pairs and their movement directions.

Graph Partitioning and High-Level Planning

In this paper, we consider only MAPF instances on four-neighbor grids (Stern et al. 2019), and HMAPP partitions the grids into rectangular regions of similar sizes, determined by parameters num_row and num_col, which specify the numbers of regions in the vertical and horizontal directions, respectively. If a region is not connected, then HMAPP partitions it further. HMAPP then iterates over each pair of adjacent regions, collects all pairs of adjacent vertices, one from each each region, that have not yet been used in any boundary pair and adds them to the set of boundary pairs. It assigns alternating directions to boundary pairs so that there are enough boundary pairs for agents to move from one region to another.

A naive partitioning approach can result in a bad partition and poor scalability of HMAPP on grids with obstacles. If HMAPP partitions the grid in Figure 1 into $2 \times 2 = 4$ regions, each of size 4×4, then it further partitions the top-right region into two regions since it is not connected. One of the resulting regions consists of cells $F5, G5$ and $H5$ and is corridor-shaped. For a corridor-shaped region, a solution might not exist for even only two agents. To improve the quality of the resulting partition, (1) if there is a corridor-shaped region, then HMAPP randomly picks one of its adjacent regions (if there is one) and merges these two regions to eliminate regions that contain only narrow corridors, and (2) if there is a pair of adjacent regions that share fewer than two boundary pairs, then HMAPP merges them to ensure that an agent can always reach an adjacent region from its current region. HMAPP repeats this procedure until no such cases exist any longer. Figure 1 shows the region r_0 obtained after merging the top-left region with the corridor-shaped region.

The high-level planner of HMAPP is responsible for finding high-level plans for all agents. For each agent, HMAPP randomly picks one of its shortest paths from its start vertex to its goal vertex that moves from region to region only at boundary pairs in their directions. HMAPP then generates the high-level plan that corresponds to the sequence of regions visited by this path. Due to Partitioning Rule (2) above, there always exists such a path for each agent.

Algorithm 2: *replan*() for regional planners.

input: Regional planner $\mathcal{P}_r$ and a set of agents A', which is the set of agents that are ready to enter r.

1 $\mathcal{P}_r.P \leftarrow ECBS(\mathcal{P}_r.A \cup A', \mathcal{P}_r.C)$;
2 **if** *ECBS failed to find a solution* **then**
3 **return** *failure*;
4 **end**
5 **foreach** *agent* $a_i \in A'$ **do**
6 $t \leftarrow$ entry timestep of a_i according to $\mathcal{P}_r.P$;
7 $\langle v', v \rangle \leftarrow$ the boundary pair a_i uses to enter r;
8 $r' \leftarrow$ the region v' is part of;
9 $\mathcal{P}_r.C.add(entry\langle a_i, v, t \rangle)$;
10 $\mathcal{P}_{r'}.C.add(exit\langle a_i, v', t \rangle)$;
11 **end**
12 $\mathcal{P}_r.A \leftarrow \mathcal{P}_r.A \cup A'$;
13 **return** *success*;

Regional Planning

The regional planners of HMAPP find sub-paths for the agents inside their regions. The regional planner $\mathcal{P}_r$ for region r maintains multiple data structures to keep track of the agents and their sub-paths. $\mathcal{P}_r.A$ is the set of agents that already have determined entry timesteps to r. $\mathcal{P}_r.C$ is the set of constraints of the agents in $\mathcal{P}_r.A$ that keep track of their entry timesteps to r and exit timesteps from r and the associated entry and exit vertices, respectively. Two types of constraints can be added to $\mathcal{P}_r.C$. The first one is an *entry-vertex-timestep constraint* $entry\langle a_i, v, t \rangle$, which enforces that agent a_i enters r from entry vertex v at entry timestep t. The second one is an *exit-vertex-timestep constraint* $exit\langle a_i, v, t \rangle$, which enforces that agent a exits region r from exit vertex v at exit timestep t. $\mathcal{P}_r.P$ is a set of conflict-free sub-paths of the agents in $\mathcal{P}_r.A$ that satisfy the constraints in $\mathcal{P}_r.C$.

HMAPP uses ECBS to solve the regional planning problems. Agent a_i is a *local agent* of region r if a_i does not have a next region in its high-level plan when it is in r; otherwise, agent a_i is a *migrating agent* of region r. For each migrating agent a_i of region r, let p_i denote the sub-path of a_i in r and $\langle v, v' \rangle$ denote the boundary pair to the next region of a_i that p_i leads to. The cost of a_i for $\mathcal{P}_r$ is $cost(p_i) + h(v') + 1$, where $h(v')$ is an admissible heuristic function for the distance from v' to g_i (if all other agents are ignored) and $cost(p_i)$ is the path cost of p_i. For each local agent a_i of r, the cost of a_i for $\mathcal{P}_r$ is the path cost of p_i.

On Line 12 of Algorithm 1, HMAPP invokes Algorithm 2 to plan the entry timestep and sub-path of each agent a_i in A' that is ready to enter region r from region r' via boundary pair $\langle v', v \rangle$ at timestep T and adds both an entry-vertex-timestep constraint to $\mathcal{P}_r.C$ so that a_i must enter region r at v at timestep t (Line 9 of Algorithm 2) and an exit-vertex-timestep constraint to $\mathcal{P}_{r'}.C$ so that a_i must exit from region r' at v' at timestep t (Line 10 of Algorithm 2).

On Line 17 of Algorithm 1, HMAPP invokes Algorithm 2 to replan the sub-paths of the agents in $P_r.A$ to ensure that the newly-added entry-vertex-timestep and exit-vertex-timestep constraints are satisfied. During this procedure, no new constraints are added.

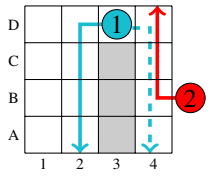


Figure 2: Shows an example where the regional planner replans the sub-path of agent a_1 when agent a_2 is ready to enter the region. Agent a_1 has its start vertex at $D2$, and agent a_2 is ready to enter the region from entry vertex $B4$ at timestep 2.

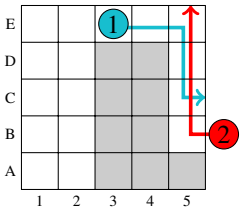


Figure 3: Shows an example where the regional planner is unable to find a solution. Agent a_1 has its start vertex at $E3$, and agent a_2 is ready to enter the region from entry vertex $B5$ at timestep 3.

The regional planning problem is similar to the online MAPF problem (Švancara et al. 2019), where agents move along their paths as T increases. However, agents do not move in the regional planning problem. Therefore, when $\mathcal{P}_r$ replans the sub-paths for the agents in $P_r.A$, it is allowed to modify their entire sub-paths in r.

Example 1. *Figure 2 shows an example where the regional planner replans the sub-path of agent a_1 when a new agent a_2 is ready to enter the region. Initially, only a_1 is in the region, and the regional planner finds a sub-path for a_1 to exit the region at $A4$ at timestep 4, which is shown by the dashed blue line. At timestep 2, a_1 is at $C4$, and a_2 is ready to enter the region. The regional planner then finds new sub-paths for a_1 and a_2. If a_1 follows its original sub-path, then a_2 must be delayed to exit its region. However, there is an alternative sub-path for a_1, which is shown by the solid blue line and has the same path cost as the current sub-path of a_1. The regional planner therefore replans the sub-paths so that a_1 uses the alternative sub-path and a_2 is not delayed to exit its region. In contrast, an online MAPF solver could not change the movement of the agents before timestep 2.*

Handling Regional Planning Failures

On Lines 9-10 of Algorithm 2, new constraints are added that determine the entry and exit timesteps of agents. Since the exit timestep of an agent from its current region is determined by the regional planner of the next region of the agent, this exit timestep may prevent the regional planner of

the current region of the agent from finding a solution.

Example 2. *Figure 3 shows an example where the regional planner is unable to find a solution. Initially, only agent a_1 is in the region, and the regional planner finds a sub-path for a_1 to exit the region at $C5$ at timestep 4, which is shown by the solid blue line. At timestep 3, agent a_2 is ready to enter the region. The regional planner finds the sub-paths for a_1 and a_2 shown in the figure, where neither agent needs to wait since the regional planner assumes that a_1 exits the region immediately when it is at $C5$ and a_2 exits the region immediately when it is at $E5$. At timestep 4, a_1 is at $C5$ and its exit timestep is determined. Assume that it is 10. The regional planner then finds new sub-paths for a_1 and a_2, for example, where a_1 waits in $E3$ for 6 timesteps and a_2 still does not wait since the regional planner assume that a_2 exits the region immediately when it is at $E5$. At timestep 7, agent a_2 is at $E5$ and its exit timestep is determined. Assume that it is 9. The regional planner then tries to find new sub-paths for a_1 and a_2 but fails since a_2 exiting the region at $E5$ at timestep 9 implies that a_1 cannot exit the region before timestep 12.*

When a regional planner is unable to find a solution (Lines 18-19 of Algorithm 1), HMAPP determines the vertices of all agents at timestep T, deletes all constraints and restarts HMAPP at timestep T. In Example 2, the regional planner fails to find a solution at timestep $T = 7$, and HMAPP restarts at timestep 7 with agent a_1 at $E4$ and agent a_2 at $E5$. Assume that the exit timestep of a_2 is again determined to be 9. The regional planner then finds new sub-paths for a_1 and a_2, for example, where a_1 waits at $E4$ (until a_2 exits the region) and then moves to $C5$ and exits the region immediately. Therefore, HMAPP is now able to find a solution.

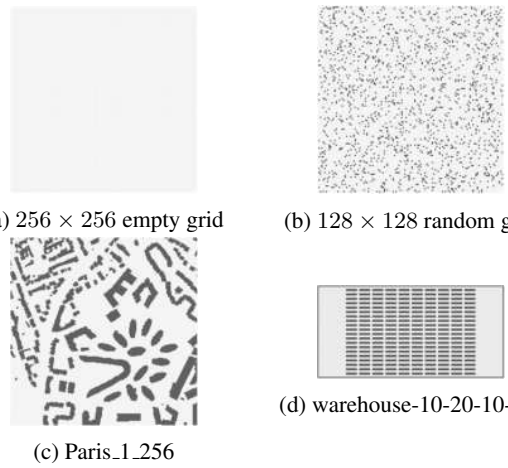
Experimental Evaluation

We compared HMAPP with Ros-dmapf (Pianpak et al. 2019), CA* (Silver 2005), WHCA* (Silver 2005) and ECBS on different grids. CA* is a greedy MAPF solver which plans for one agent at a time. WHCA* is a variant of CA* which interleaves moving agents and planning within a time window of a given length. The objectives for ECBS and the regional planners of HMAPP were all minimizing the SoC, and the suboptimality factors for ECBS and the regional planners of HMAPP were all set to 1.2. The length of the time window of WHCA* was set to 16, which we found to achieve a higher success rate than smaller window sizes while still achieving a small runtime. Except for Ros-dmapf, all MAPF solvers were implemented in C++ and share the same code base as much as possible. We ran all experiments on a laptop with an i7-8850H CPU and 32 GB of memory.

Experiment 1: Comparison with Ros-dmapf. We did not have a working implementation of Ros-dmapf available. Therefore, to compare HMAPP with Ros-dmapf, we ran HMAPP on the 60×60 empty grid MAPF instances used in (Pianpak et al. 2019) and compared the results of HMAPP with the results of Ros-dmapf in the paper. Table 1 shows the average makespans and numbers of moves of HMAPP and Ros-dmapf. The number of moves is the sum of the number of move actions of all agents. HMAPP used both the partition size 10×10 and runtime limit of 100s used in (Pianpak et al. 2019). HMAPP solved all MAPF instances within the time limit. Table 1 shows that the average makespan of the solutions of HMAPP was less than half of the average makespan of the ones of Ros-dmapf for MAPF instances with 576 agents or more.

Agents	Ros-dmapf		HMAPP	
144	149	5,996	99	6,043
288	185	12,934	118	12,487
432	230	21,627	118	19,307
576	264	30,704	116	25,520
720	310	43,740	120	32,951

Table 1: Shows the average makespans and numbers of moves of Ros-dmapf and HMAPP on the 60×60 empty grid for different numbers of agents.



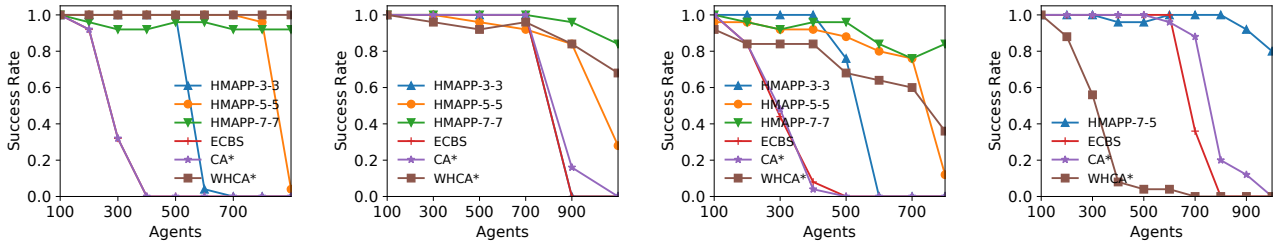
(a) 256×256 empty grid (b) 128×128 random grid

(c) Paris_1_256 (d) warehouse-10-20-10-2-2

Figure 4: Shows the grids of the MAPF instances used in Experiment 2.

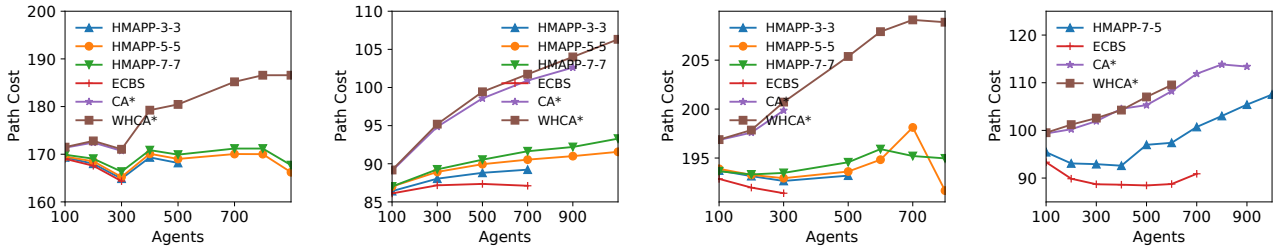
Experiment 2: Comparison with ECBS and greedy MAPF solvers. We evaluated all MAPF solvers on the four grids shown in Figure 4: (a) the 256×256 empty grid, (b) a 128×128 grid with 10% randomly blocked vertices, (c) Paris_1_256 and (d) warehouse-10-20-10-2-2. Grids (c) and (d) are from the MAPF benchmark (Stern et al. 2019). We did not use the empty and random grids from the MAPF benchmark since we were interested in large MAPF instances. For Grids (a)-(c), we used HMAPP with partition sizes $(num\_row, num\_col) = (3, 3)$, $(5, 5)$ and $(7, 7)$. For Grid (d), we used HMAPP with partition size $(7, 5)$ since the runtime of HMAPP turned out to be very sensitive to the size of the regions.

Figure 5 shows that, on most grids, the success rates of ECBS and CA* quickly dropped as the number of agents increased. WHCA* successfully solved all MAPF instances for up to 900 agents on Grid (a). However, on Grids (b)-(d), the success rate of WHCA* was lower than the ones of some versions of HMAPP since WHCA* plans only within a time window of limited length.



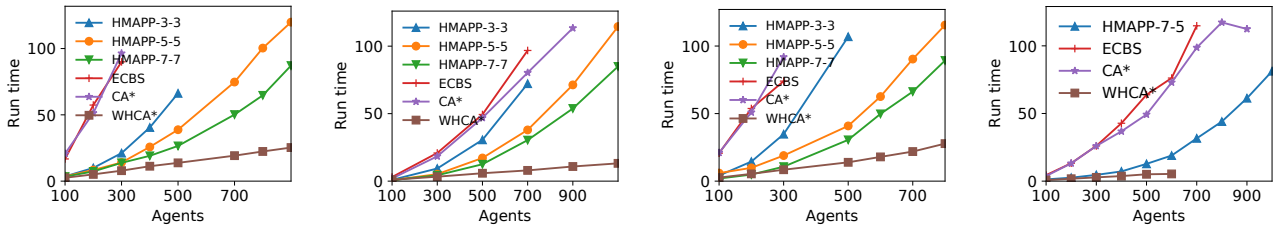
(a) 256 × 256 empty grid (b) 128 × 128 random grid (c) Paris_1_256 (d) warehouse-10-20-10-2-2

Figure 5: Shows the success rates (that is, the percentages of MAPF instances solved within a time limit of two minutes) of various MAPF solvers on each grid for different numbers of agents.



(a) 256 × 256 empty grid (b) 128 × 128 random grid (c) Paris_1_256 (d) warehouse-10-20-10-2-2

Figure 6: Shows the average path costs per agent (averaged over the MAPF instances solved by all MAPF solvers that successfully solved at least one MAPF instance) of various MAPF solvers on each grid for different numbers of agents.



(a) 256 × 256 empty grid (b) 128 × 128 random grid (c) Paris_1_256 (d) warehouse-10-20-10-2-2

Figure 7: Shows the average runtimes (in seconds, averaged over the MAPF instances solved by all MAPF solvers that successfully solved at least one MAPF instance) of various MAPF solvers on each grid for different numbers of agents.

Figure 6 shows that all versions of HMAPP had smaller average path costs on all grids than CA* and WHCA*. Except for Grid (c), the average path costs of HMAPP were more than 10% smaller than those of WHCA* for large numbers of agents. Except for Grid (d), which has many narrow corridors, the average path costs of HMAPP were close to the average path costs of ECBS.

Figure 7 shows that all versions of HMAPP had smaller average runtimes than CA* and ECBS on all grids because HMAPP does not plan paths across the entire grid. However HMAPP had larger average runtimes than WHCA* since WHCA* plans within smaller time windows.

Conclusions and Future Work

In this paper, we have proposed HMAPP which solves the MAPF problem by creating a spatial hierarchy that decomposes a MAPF instance into MAPF sub-instances. Our experimental results show that HMAPP solves as large MAPF instances as greedy MAPF solvers while achieving better solution qualities on various maps.

Our future work includes (1) making HMAPP complete by controlling the number of agents in each region; (2) automatically generating a good partition of the graph of a MAPF instance and (3) developing high-level planning approaches that take potential congestion into account.

Acknowledgments

The research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1817189, 1837779 and 1935712, as well as a gift from Amazon.

References

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *International Symposium on Combinatorial Search (SoCS)*, 19–27.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1(1): 7–28.

Choudhury, S.; Solovey, K.; Kochenderfer, M. J.; and Pavone, M. 2020. Efficient large-scale multi-drone delivery using transit networks. In *IEEE International Conference on Robotics and Automation (ICRA)*, 4543–4550.

Kapadia, M.; Beacco, A.; Garcia, F.; Reddy, V.; Pelechano, N.; and Badler, N. I. 2013. Multi-domain real-time planning in dynamic environments. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 115–124.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. K. S.; and Koenig, S. 2020. Lifelong multi-agent path finding in large-scale warehouses. In *International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 1898–1900.

Ma, H.; Hönig, W.; Cohen, L.; Uras, T.; Xu, H.; Kumar, T. S.; Ayanian, N.; and Koenig, S. 2017. Overview: A hierarchical framework for plan generation and execution in multirobot systems. *IEEE Intelligent Systems* 32(6): 6–12.

Ma, H.; Tovey, C.; Sharon, G.; Kumar, T. K. S.; and Koenig, S. 2016. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *AAAI Conference on Artificial Intelligence (AAAI)*, 3166–3173.

Pearl, J.; and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-4(4): 392–399.

Pelechano, N.; and Fuentes, C. 2016. Hierarchical path-finding for navigation meshes (HNA*). *Computers & Graphics* 59: 68–78.

Pianpak, P.; Son, T. C.; Toups, Z. O.; and Yeoh, W. 2019. A distributed solver for multi-agent path finding problems. In *International Conference on Distributed Artificial Intelligence (DAI)*, 1–7.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219: 40–66.

Silver, D. 2005. Cooperative pathfinding. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 117–122.

Stern, R.; Sturtevant, N. R.; Atzmon, D.; Walker, T.; Li, J.; Cohen, L.; Ma, H.; Kumar, T. K. S.; Felner, A.; and Koenig, S. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *International Symposium on Combinatorial Search (SoCS)*, 151–158.

Švancara, J.; Vlk, M.; Stern, R.; Atzmon, D.; and Barták, R. 2019. Online multi-agent pathfinding. In *AAAI Conference on Artificial Intelligence (AAAI)*, 7732–7739.

Wilt, C. M.; and Botea, A. 2014. Spatially distributed multiagent path planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 332–340.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* 29(1): 9–20.

Yu, J.; and LaValle, S. M. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1443–1449.

Compiling HTN Plan Verification Problems into HTN Planning Problems

Daniel Höller,[1] Julia Wichlacz,[1] Pascal Bercher,[2] Gregor Behnke[3]
[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany,
[2]The Australian National University, Canberra, Australia,
[3]University of Freiburg, Freiburg, Germany,
hoeller@cs.uni-saarland.de, wichlacz@cs.uni-saarland.de, pascal.bercher@anu.edu.au, behnkeg@cs.uni-freiburg.de

Abstract

Plan Verification is the task of deciding whether a sequence of actions is a solution for a given planning problem. In HTN planning, the task is computationally expensive and may be up to NP-hard. However, there are situations where it needs to be solved, e.g. when a solution is post-processed, in systems using approximation, or just to validate whether a planning system works correctly (e.g. for debugging or in a competition). In the literature, there are verification systems based on translations to propositional logic and based on techniques from parsing. Here we present a third approach and translate HTN plan verification problems into HTN planning problems. These can be solved using any HTN planning system. We test our solver on the set of solutions from the 2020 International Planning Competition. Our evaluation is yet preliminary, because it does not include all systems from the literature, but it already shows that our approach performs well compared with the included systems.

1 Introduction

Plan Verification is the task of deciding whether a sequence of actions is a solution for a given planning problem. It is necessary in several situations, e.g. when a plan is post-processed, or to verify whether a planning system works correctly (e.g. for debugging or in a competition).

In classical planning, it can be solved in (lower) polynomial time. In Hierarchical Task Network (HTN) planning (Bercher, Alford, and Höller 2019), the complexity depends on several parameters:

- On whether the decomposition steps (i.e., the chosen methods) leading to a solution are known.
- On the specific problem class (e.g., whether it's partially or totally ordered).

When we look at the formalisms in HTN planning, the decomposition steps are usually not regarded part of a solution. Since only the contained primitive tasks (i.e., actions) need to be executed, there is often no need to do so. However, there are use cases in the literature where they are needed, mainly for communication with a user on different levels of abstraction (Bercher et al. 2021; Behnke et al. 2020a; Köhn et al. 2020; de Silva, Padgham, and Sardina 2019). When they are present (and if full information about task labeling is available so that multiple occurrences of the same task can still be distinguished) plan verification can be solved in polynomial time. Another polytime case is given by Totally Ordered (TO) HTN problems, where all methods and the initial task network are totally ordered. In this case, plan verification is cubic and resembles the problem of parsing in context-free languages. Otherwise, for general partially ordered (PO) HTN problems, it becomes NP-hard (Behnke, Höller, and Biundo 2015).

Both cases, i.e., TO and PO HTN planning problems, were considered in the 2020 International Planning Competition (IPC). Here, the participating systems needed to return the decomposition steps to allow the organizers to verify solutions in polynomial time. However, though it is possible for the solvers to track this information, it causes technical problems – consider e.g. the various compilation steps often performed in preprocessing[1] that need to be undone in post-processing. In other cases it is even not possible, e.g. when postoptimizing solutions or when internally using approximations like e.g. the TOAD system (Höller 2021), which overapproximates the solution set of a problem and needs verification as a regular step of its planning procedure to make sure only to return correct solutions.

In the literature, there are systems to solve the problem via translation to propositional logic (Behnke, Höller, and Biundo 2017) and based on parsing techniques (Barták, Maillard, and Cardoso 2018; Barták et al. 2020).

In this paper, we present an approach to compile HTN verification problems to common HTN planning problems. Our compilation is based on an approach for plan recognition as planning (Höller et al. 2018). It is applicable in both TO and PO HTN planning. We combine our transformation with two planning systems from the PANDA framework (Höller et al. 2021) and evaluate it on a new benchmark set that includes the models from the 2020 IPC and solutions generated by the IPC participants. It yields good results both in PO and in TO HTN planning.

The paper is structured as follows: we first introduce the formal framework used in the paper (Section 2), then we introduce the compilation (Section 3), describe the realization (Section 4) and evaluate the approach (Section 5). We conclude the paper with a short discussion (Section 6).

[1]See e.g. the PANDA grounder (Behnke et al. 2020b) used in the following. Among other compilations, it changes decomposition rules until convergence, which is not a simple task to undo.

2 Formal Framework

In HTN planning there are two types of tasks, *primitive* and *abstract* tasks. Primitive tasks are equivalent to actions in classical planning, i.e., they are directly applicable in the environment and cause state transition. Abstract (or compound) tasks are not directly applicable and need to be decomposed into other tasks in a process similar to the derivation of words from a formal grammar. A solution to an HTN planning problem needs to be derived via this grammar.

Formally, an HTN planning problem is a tuple $p = (F, C, A, M, s_0, tn_I, g, prec, add, del)$. F is a set of propositional state features. A state s is defined by the subset of state features that hold in it, $s \in 2^F$, all other state features are assumed to be *false*. $s_0 \in 2^F$ is the initial state of the problem, and $g \subseteq F$ is the state-based goal description[2]. A state s is called a *goal state* if and only if $g \subseteq s$. A is a set of symbols called *primitive tasks* (also called *actions*). These symbols are mapped to a subset of the state features by the functions $prec$, add, del, which are all defined as $f : A \to 2^F$ and define the actions' preconditions, add-, and delete-effects, respectively. An action a is applicable in a state s if and only if $prec(a) \subseteq s$. When an applicable action a is applied in a state s, the state $s' = \gamma(s, a)$ resulting from the application is defined as $s' = (s \setminus del(a)) \cup add(a)$. A sequence of actions $a_1, a_2, \ldots, a_n$ is applicable in some state s_0 if and only if a_i is applicable in the state s_{i-1}, where s_i for $1 \leq i \leq n$ is defined as $s_i = \gamma(s_{i-1}, a_i)$. We call the state s_n the state *resulting* from the application.

Tasks in HTN planning are maintained in *task networks*. A task network is a partially ordered multiset of tasks. Formally, it is a triple $tn = (T, \prec, \alpha)$. T is a set of identifiers (ids) that are mapped to the actual tasks by the function $\alpha : T \to N$, where $N = A \cup C$ is the union of the primitive tasks A and the abstract (compound) tasks C. $\prec$ is a partial order on the task ids. tn_I is the initial task network, i.e., the task network the decomposition process starts with. Legal decompositions are defined by the set of *(decomposition) methods* M. A method is a pair (c, tn), where $c \in C$ defines the task that can be decomposed using the method, and the task network tn defines into which tasks it is decomposed. When a task t from a task network tn is decomposed using a method (c, tn'), it is replaced by the tasks in tn'. When t has been ordered with respect to other tasks in tn, the new tasks inherit these ordering constraints. Formally, a method $m = (c, tn)$ decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ that contains a task id $t \in T_1$ with $\alpha_1(t) = c$ into a task network tn_2, which is defined as follows. Let $tn' = (T', \prec', \alpha')$ be a copy of tn that uses ids not contained in T_1. Then tn_2 is defined as:

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

[2] In HTN planning, there is usually no state-based goal given, because it can be compiled away. However, it makes our definition in the next sections more natural.

When a task network tn can be decomposed into a task network tn' by applying (a finite sequence of) 0 or more methods, we write $tn \to^* tn'$.

A task network $tn_S = (T_S, \prec_S, \alpha_S)$ is a solution to a given HTN planning problem if and only if the following conditions hold:

1. $tn_I \to^* tn_S$, i.e., it can be derived from the initial task network,
2. $\forall t \in T_S : \alpha_S(t) \in A$, i.e., all tasks are primitive, and
3. there is a sequence $(i_1 i_2 \ldots i_n)$ of the task ids in T_S in line with the ordering constraints $\prec_S$ such that $(\alpha_S(i_1)\alpha_S(i_2)\ldots\alpha_S(i_n))$ is applicable in s_0 and results in a goal state.

We call an HTN method *totally ordered* when the tasks in the contained task network are totally ordered. We call an HTN planning problem totally ordered when all contained methods and the initial task network are totally ordered.

Definition 1 (Plan Verification)**.** *Given an HTN planning problem p and a sequence of actions $(a_1 a_2 \ldots a_n)$, plan verification is the problem to decide whether there is a task network $(T_S, \prec_S, \alpha_S)$ that is a solution for p and for an ordering $(i_1 i_2 \ldots i_n)$ of the task identifiers T_S fulfilling solution criterion 3 as given above, it holds that $(\alpha_S(i_1)\alpha_S(i_2)\ldots\alpha_S(i_n)) = (a_1 a_2 \ldots a_n)$.*

3 Compiling Verification Problems to Planning Problems

Let $p = (F, C, A, M, s_0, tn_I, g, prec, add, del)$ be an HTN planning problem, $\pi = (a_1 a_2 \ldots a_n)$ a sequence of actions out of A, and $v = (p, \pi)$ a plan verification problem. We compile v into a new HTN planning problem $p' = (F', C', A', M', s'_0, tn_I, g', prec', add', del')$ that has a solution if and only if v is solvable.

The encoding is widely identical with the one introduced by Höller et al. (2018) for plan and goal recognition (PGR) as planning. It has also been shown that it can be used for plan repair (Höller et al. 2020b). We first change the state and the actions of the original problem such that the only applicable sequence of actions exactly resembles π. Let $f_0, f_1, f_2, \ldots, f_n$ be new state features. We use them to encode which actions out of π have already been executed. In addition to that, we need a state feature $\bot$, which will never be reachable. The set of state features of p' is defined as $F' = F \cup \{f_0, f_1, f_2, \ldots, f_n\} \cup \{\bot\}$. In the beginning, no action out of π has been executed, i.e., $s'_0 = s_0 \cup \{f_0\}$. We want solutions to exactly equal π, i.e., all actions need to be included. This is enforced by including f_n in the state-based goal definition g, i.e., $g' = g \cup \{f_n\}$.

For $a_i \in \pi$ with $1 \leq i \leq n$, we introduce a new action a'_i. The preconditions of the new actions enforce the correct position in the generated solution

$$prec'(a'_i) = prec(a_i) \cup \{f_{i-1}\},$$

each action deletes its own precondition and adds the one of the next action in the solution

$$add'(a'_i) = add(a_i) \cup \{f_i\},$$
$$del'(a'_i) = del(a_i) \cup \{f_{i-1}\}.$$

Please be aware that an action out of A may appear more than once in the solution. In such cases, there will be multiple copies of the action in A'. The novel actions mimic the state transition of the original ones, but additionally ensure their respective position in the solution. All other actions shall never appear in any solution, so we add the new state feature $\bot$ that is never reachable to their preconditions.

$$\begin{aligned} \forall a \in A : prec'(a) &= prec(a) \cup \{\bot\}, \\ add'(a) &= add(a), \\ del'(a) &= del(a) \end{aligned}$$

The new set of actions is defined as $A' = A \cup \{a'_i \mid a_i \in \pi\}$.

Due to the new state features, preconditions and effects, there is only one sequence of actions that is applicable and leads to a state-based goal. However, none of the new actions can ever be reached by decomposing the initial task network. To make this possible, we need to modify the decomposition hierarchy. It shall be possible for a newly introduced action a' to be placed at exactly those positions where the action a might have been in the original model. We thereby need to keep in mind that there might be multiple copies of some action a, so we cannot just replace them in the methods. We need to introduce a new choice point to choose which copy $a', a'', \dots$ of a shall be at which position in the action sequence. We do this by introducing one novel abstract task c_a for each action a. Let $C^A = \{c_a \mid a \in A\}$. We further introduce new methods to decompose this new task into one of the copies of a.

$$M^A = \{(c_a, (\{i\}, \emptyset, \{i \mapsto a'\})) \mid a' \in \pi\}$$

We define $C' = C \cup C^A$ and $M' = M \cup M^A$ and have fully specified our compiled problem p'.

The resulting encoding is nearly identical with the one used in the fully observable case of plan and goal recognition as planning (Höller et al. 2018). The only difference is the additional precondition of the actions not included in the solution. While the PGR encoding forces these actions to be placed after a given plan prefix of observed actions, the encoding here makes them entirely unreachable.

3.1 Some Technicalities

The models in our benchmark are those from the 2020 IPC, which are modeled in the description language HDDL (Höller et al. 2020a). In HDDL, models may include state-based preconditions for methods. These are preconditions as known from actions, which have to hold for the method to be applicable. The semantics of such preconditions is a bit problematic (see Höller et al. (2020a, p. 5) for a discussion). In HDDL it is defined as follows: a new action is introduced that is inserted in the method and placed before every other task of the method's subtasks. This new action holds the precondition of the method. We will call such actions *technical actions*.

In TO HTN planning, this fully specifies the position where the precondition needs to hold. However, consider the case of PO HTN planning. Here, the task decomposed by the method might be partially ordered with respect to other tasks and the subtasks might be interweaved. As a result, we cannot exactly determine the position the precondition needs to hold. When the state-features contained in the method's precondition are not static (i.e., might change over time), the position where the precondition is checked might change applicability.

Since technical actions are not actually part of the solution, planners will not return them. From a verification perspective, this is problematic, a verifier needs to check whether there *exists* a position where a technical action is applicable. This is a main problem with the SAT-based approach from related work (Behnke, Höller, and Biundo 2017), which needs to get the technical actions as input.

In our approach, handling this issue is not a problem: we leave the preconditions and effects of technical actions unchanged, i.e., they will not be affected by the encoding. As a result, they can be inserted into plans of the compiled problems at arbitrary positions in line with the definition of HTN decomposition. That is, if the plan π to verify has a length of n, then our encoding makes sure that – provided plan π is indeed a valid solution to the original problem – a solution of size at least n will be found (corresponding to π), but additional actions which encode the method preconditions may be included as well at appropriate positions (one for each applied method with a precondition). Note that this means that there is no clear limit on the length of solutions (other than its minimum). Since methods might replace an abstract task by no other task, i.e., delete it, it is not clear how many such empty methods might have been applied in the worst case leading to a plan of a certain length.

3.2 Properties

Let $v = (p, \pi)$ be a Plan Verification Problem and p' the encoding as given above.

Our encoding serves the purpose of deciding whether π is a solution for p. This is being achieved provided that π is a solution for p if and only if p' is solvable, which we capture in the following two theorems. Note that this result (restricted to methods without preconditions) follows as a special case from Thm. 1 by Höller et al. (2018). That theorem from the context of plan recognition states that the encoding – the same one we deploy for plan verification – ensures that the solutions of the encoded problem are *exactly* those of the original problem that start with the enforced actions. In the context of this earlier work, there might have been additional actions after the prefix of enforced actions, namely the remaining plan that should be recognized. In our case, this part remains empty (modulo technical actions encoding method preconditions).

Theorem 1. *When π is a solution for p, then the compiled problem p' is solvable.*

Proof Sketch. Since π is a solution to p, we know that there is a sequence of method applications that transforms the initial task network tn_I into a primitive task network tn, which in turn allows π as executable linearization. Note that we can assume that the solution was achieved by a progression planner, which applies methods and actions in a forward-fashion, since such a progression-based solution exists if and only if

any solution exists at all (Alford et al. 2012, Thm. 3). Thus, we can assume that there is a sequence of method and action applications $\overline{ma}$ that transforms tn_I into π. That sequence can be transformed into a corresponding (and potentially longer) sequence in p'. For each action a_i at position i in π its corresponding encoding a'_i will be executable in the solution π' to p', though the respective sequence of method and action applications will be preceded by the method decomposing c_a thus introducing that encoding of a'_i. Furthermore, every method m in $\overline{ma}$ will be applicable in the corresponding method and action sequence $\overline{ma}'$ leading to π' in p' as well, though immediately preceded by the technical action encoding the precondition of m. □

Theorem 2. *When π is no solution for p, then the compiled problem p' is unsolvable.*

Proof Sketch. This direction is a bit easier to see than the previous one, since the model of p' is an extension of the original one, i.e., it follows the exact same structure, but each action has additional preconditions and thus makes the problem more constrained. So if there is no solution in the original model, there cannot be a solution in the encoded one. □

It was also shown that the transformation maintains most structural properties of the problem (Höller et al. 2020b, Sec. 6.1), i.e., tail-recursive, acyclic, and totally ordered problems remain tail-recursive, acyclic, or totally ordered, respectively. Since we deploy the same encoding we essentially get the same property, though the restriction to a specific solution might lead to even more restrictive cases. E.g., the restriction to the model required to obtain the plan π to verify might turn a problem without any restriction even into a totally ordered acyclic problem. We still can directly conclude the following properties:

Corollary 1. *If p is tail-recursive, p' is tail-recursive. If p is acyclic, p' is acyclic. If p is totally ordered, p' is totally ordered.*

4 Pruning the Model

The encoding makes wide parts of the original actions inapplicable. When realizing it on the lifted definition, this would be detected by the grounding procedure (see e.g. (Ramoul et al. 2017; Behnke et al. 2020b)), which would not generate unreachable parts.

However, we realized our encoding on a fully grounded model. For us, this had two main advantages: First, it simplifies the implementation. Second, since our planning system grounds the model before planning, we have the ground model and do not need to ground twice. However, in order to result in a small model, we need to prune unreachable parts. We use the following two pruning methods, which are similar to what the PANDA grounder described by Behnke et al. (2020b) would do in its grounding process.

4.1 Bottom-up Reachability

By construction, all actions not included in the enforced solution contain the unreachable precondition $\bot$, i.e., the only actions that are (potentially) reachable are those in the enforced solution as well as the technical actions. Let A_T be the set of technical actions. We initialize our reachability analysis with these actions: $N_r = \{a'_i \mid a'_i \in \pi\} \cup A_T$.

We now want to determine the set of reachable methods and abstract tasks. Since we know that certain actions are not reachable, we know that any method which includes such actions will never be part of any solution. Or, the other way around, we know that only methods not including such actions will be part of a solution. For an abstract task c, we know that it can only be part of a solution when there is at least one method that decomposes c that might be part of a solution. Based on these observations, we calculate the sets of methods and abstract tasks that might be part of a solution.

Let TN^N be the set of all task networks over the tasks N. We define the relation $R : TN^N \times 2^N$ with

$$R = \{((T, \prec, \alpha), N') \mid \forall t \in T : \alpha(t) \in N'\}$$

Let $N_r \subseteq N$ be a set of reachable tasks. As discussed above, the set of reachable methods based on this set is defined as

$$M_r = \{(c, tn) \in M \mid (tn, N_r) \in R\}$$

The overall reachability is then defined as follows:

function bottom-up(N_r)
$M_r = \emptyset$
while N_r *changes* **do**
  $M_r = \{(c, tn) \in M \mid (tn, N_r) \in R\}$
  $N_r = N_r \cup \{c \mid (c, tn) \in M_r\}$
return (N_r, M_r)

As given above, this algorithm is started with $N_r = \{a'_i \mid a'_i \in \pi\} \cup A_T$, i.e., the set containing the actions in the enforced solution and all technical actions.

4.2 Top-down Reachability

The sets returned by the analysis given above might include tasks and methods not reachable from the initial task network. We therefore perform a second (top-down) analysis. Let $tn_I = (T_I, \prec_I, \alpha_I)$ be the initial task network and N_r and M_r the sets returned by the bottom-up analysis. We determine the tasks and methods reachable top-down using the following function.

function top-down(T_I, α_I, N_r, M_r)
$M'_r = \emptyset$
$N'_r = \{n \mid \exists i \in T_I : \alpha_I(i) = n\}$
while N'_r *changes* **do**
  $N'_r = N'_r \cup \{n \mid \exists c \in N'_r, (c, (T, \prec, \alpha)) \in M_r), i \in T : \alpha(i) = n\}$
  $M'_r = \{(c, tn) \in M_r \mid c \in N'_r\}$
return (N'_r, M'_r)

We perform a single pass of these two methods and output the resulting (reduced) problem afterwards.

5 Evaluation

We next describe the benchmark set and the systems included in the evaluation. Then we discuss the results.

The experiments ran on Xeon Gold 6242 CPUs using one core, a memory limit of 8 GB, and a time limit of 10 minutes.

5.1 Benchmark Set

We use a new set of benchmark problems that is based on the models from the 2020 IPC. It contains 892 planning problems from 24 domains in TO planning and 224 instances from 9 domains in PO planning. The solutions have been created by 7 different planning systems for TO and by 4 systems for PO; namely by the final versions of the participants of the IPC as well as by planners from the PANDA framework (Höller et al. 2021). In total, this results in 10963 plan verification instances in TO and 1077 in PO HTN planning.

Since plans and domains stem from a recent competition, we consider it an interesting benchmark set with respect to the included plans and the difficulty of the instances. However, there is one weakness that we want to address in future work: Since the current set includes only instances from the *final* planner versions (after the debugging process), it includes only very few instances that are incorrect plans, only 2 instances for TO and 1 for PO.

In related work, this problem was solved by using random walks (Behnke, Höller, and Biundo 2017). However, it is hard to create such instances with appropriate difficulty, and we do not consider the resulting instances as interesting as the positive ones given above. In future work, we want to include instances from early planner versions from the IPC (before debugging) to obtain a more realistic benchmark set. However, this work is still in progress and here we present the results from the benchmark set as described above.

5.2 Systems

We ground the models using the PANDA grounder (Behnke et al. 2020b). Grounding time is included in the runtimes. After transformation and pruning as given in Section 3 and 4, we output the same format as the PANDA grounder.

We use two planner configurations from the PANDA framework (Höller et al. 2021) to solve the resulting HTN planning problems:

- The progression search with the Relaxed Composition (RC) heuristic (Höller et al. 2018, 2020c) and loop detection (Höller and Behnke 2021).
- The SAT-based solver for TO (Behnke, Höller, and Biundo 2018; Behnke 2021) and for PO (Behnke, Höller, and Biundo 2019) HTN planning.

We compare our system against two systems from the literature, a SAT-based and a parsing-based approach.

SAT-based verification. The first system is based on a compilation to propositional logic (Behnke, Höller, and Biundo 2017). It supports both totally ordered and partially ordered problems. However, it relies on an input plan that contains all actions – including the technical actions given above. This makes a comparison difficult. We addressed the issue by removing all method preconditions from the input model provided to this particular system. As a result, no technical actions are introduced and the approach can be applied. Since we remove constraints from the model, all solutions to the original problem are also solutions to the new model. However, the new model allows for more solutions. Thus, the results obtained by this workflow might be incorrect. However, we argue that this does not make the solving process harder and that we can fairly compare our runtimes against this approach.

Parsing-based verification. The second approach from the literature is based on parsing (Barták, Maillard, and Cardoso 2018; Barták et al. 2020). In principle, it supports both TO and PO models. However, while the TO version works fine on our benchmark set, we where not able obtain a stable run with the PO version and thus do not include the results here. We know that this makes our evaluation preliminary and will address the issue in future work.

5.3 Results

In the TO setting, our compilation approach reaches a coverage of 99.4% with the progression search and 89.2% with the SAT-based PANDA. The SAT-based verifier has a coverage of 8.3%, the parsing-based system one of 23.5%. When comparing our two configurations, the SAT-based planner solves 7 instances that the progression search does not solve. For our compilation combined with the progression search planner, only 67 plans of the corpus cannot be verified. In 43 of these cases, we already fail to ground the planning problem (within the given memory/time limits). We currently do grounding without considering the plan to be verified. As such, the grounded model usually contains a significant number of actions, tasks, and methods which cannot be part of the compiled model. From the instances where the grounding was successful, our compilation combined with the progression search solves 99.8% of the instances.

Figure 1 visualizes the runtimes for the TO setting. The curve on the left is the SAT-based approach from related work, which has the worst performance, followed by the parsing-based approach. Our translation combined with the progression search performs best, followed by our translation combined with the SAT-based planner.

For the PO setting, our compilation reaches a coverage of 98.9% with the SAT-based planner and a coverage of 90.3% with the progression search. The SAT-based verifier has a coverage of 72.0%. We assume that the higher coverage is caused by the fact that the plans in the PO setting are significantly shorter. For the PO instances, grounding never fails.

Table 1 shows some characteristic numbers in the TO setting for the different domains. Our approach combined with the progression search has a coverage of 100% in 18 of 24 domains. The parsing-based system has a higher coverage in the Childsnack domain, where we have a coverage of 98.1% and the parsing-based system has 99.6%. Though the parsing-based system also works on a grounded model, it does not use an external grounder and can incorporate reachability information into the grounding process. This seems

Domain	#Plans	Verified				Shortest unverified plan Comp_{pro}	Plan Length			Runtime for Verified			Pearson Correlation
		SAT	Parsing	Comp SAT	Comp pro		Min–Max	Avg	Median	Min–Max	Avg	Median	
AssemblyHierarchical	193	24	102	**193**	**193**	–	4 – 256	31.1	14	0.07 – 0.76	0.2	0.11	0.812
Barman-BDI	423	79	33	396	**423**	–	10 – 1198	128.4	69	0.07 – 6.57	0.3	0.14	0.890
Blocksworld-GTOHP	160	2	5	142	**160**	–	21 – 6661	479.8	209.5	0.07 – 534.39	10.3	0.15	0.906
Blocksworld-HPDDL	172	0	5	143	170	4853	20 – 5732	461.1	163	0.07 – 542.21	15.8	0.19	0.914
Childsnack	529	92	527	516	519	750	50 – 2500	119.8	80	0.12 – 56.20	1.2	0.28	0.864
Depots	455	60	210	436	**455**	–	15 – 971	129.1	92	0.07 – 4.32	0.3	0.13	0.930
Elevator-Learned	2812	2	213	2700	**2812**	–	10 – 2165	225.1	200	0.06 – 6.71	0.3	0.21	0.940
Entertainment	159	111	**159**	**159**	**159**	–	24 – 128	71.7	64	0.08 – 4.33	1.6	0.58	0.199
Factories-simple	123	9	9	96	**123**	–	15 – 2968	623.7	251	0.07 – 17.38	1.8	0.14	0.928
Freecell-Learned	204	0	26	152	**204**	–	57 – 489	162.7	138.5	2.86 – 13.06	4.9	5.2	0.882
Hiking	565	0	156	**565**	**565**	–	26 – 174	70.8	72	0.17 – 45.97	2.4	0.97	0.641
Logistics-Learned	1108	0	9	683	**1108**	–	27 – 2813	413.1	370	0.07 – 14.55	0.6	0.37	0.919
Minecraft-Player	75	0	0	73	74	278	35 – 278	51.9	44	10.64 – 120.90	73.5	93.61	0.923
Minecraft-Regular	766	0	0	616	734	107	35 – 9947	253.8	135	0.12 – 207.75	11.6	1.455	0.326
Monroe-FO	248	0	176	**248**	**248**	–	3 – 96	41.5	39	3.48 – 3.94	3.7	3.66	0.334
Monroe-PO	217	0	63	**217**	**217**	–	6 – 91	45.1	45	3.43 – 3.91	3.7	3.67	0.390
Multiarm-Blocksworld	443	9	22	419	**443**	–	20 – 543	182.1	124	0.07 – 6.25	0.8	0.20	0.903
Robot	117	21	27	85	**117**	–	2 – 1725	272.4	37	0.06 – 59.25	4.2	0.08	0.914
Rover-GTOHP	509	22	172	397	**509**	–	16 – 2640	320.7	212	0.06 – 86.33	5.3	1.49	0.827
Satellite-GTOHP	296	9	84	199	**296**	–	12 – 1584	379.1	270	0.06 – 58.23	6.9	2.67	0.846
Snake	183	153	77	182	**183**	–	2 – 162	20.6	16	0.09 – 8.99	1.1	0.57	0.230
Towers	17	3	5	8	12	8191	1 – 131071	15419.1	511	0.07 – 141.21	14.6	0.195	0.684
Transport	695	65	239	664	678	382	8 – 5077	188.9	76	0.06 – 406.08	2.4	0.1	0.719
Woodworking	494	251	261	**494**	**494**	–	3 – 219	57.5	25	0.08 – 22.49	5.0	1.01	0.994
	10963	912	2580	9783	10896	107	1 – 131071	239.2	119	0.06 – 542.21	3.2	0.29	0.273

Table 1: Characteristic numbers for the TO setting. From left to right: Name of the domain, followed by the number of verification instances and coverage for all systems per domain. Length of the shortest plan that has not been solved by the progression search, statistics regarding plan length and runtime. The last column gives the correlation between plan length and runtime.

Domain	#Plans	Verified			Shortest unverified plan Comp_{SAT}	Plan Length			Runtime for Verified			Pearson Correlation
		SAT	Comp pro	Comp SAT		Min–Max	Avg	Median	Min–Max	Avg	Median	
Barman-BDI	56	37	46	44	90	10 – 1198	108.4	32	0.01 – 495.17	48.1	6.65	0.621
Monroe-Fully-Observable	129	7	**129**	**129**	–	9 – 61	24.9	24	6.85 – 105.65	29.3	14.82	0.656
Monroe-Partially-Observable	104	9	103	**104**	–	9 – 47	23.3	24	3.84 – 88.76	21.9	14.375	0.234
PCP	26	6	**26**	**26**	–	10 – 90	28.0	26	0.01 – 99.15	4.6	0.19	0.772
Rover	144	131	138	**144**	–	8 – 115	31.2	25	0.01 – 113.22	4.3	0.57	0.579
Satellite	246	**246**	**246**	**246**	–	5 – 28	13.5	13	0.00 – 0.11	0.0	0.02	0.677
Transport	183	150	96	**183**	–	8 – 69	27.2	28	0.01 – 139.83	2.1	0.23	0.321
UM-Translog	52	**52**	**52**	**52**	–	7 – 37	16.8	13	0.01 – 0.05	0.0	0.02	0.800
Woodworking	137	**137**	**137**	**137**	–	3 – 20	11.9	12	0.00 – 0.36	0.2	0.14	0.863
	1077	775	973	1065	90	3 – 1198	25.7	18	0.00 – 495.17	8.8	0.19	0.667

Table 2: Characteristic numbers for the PO setting. From left to right: Name of the domain, followed by the number of verification instances and coverage for all systems per domain. Length of the shortest plan that has not been solved by the progression search, statistics regarding plan length and runtime. The last column gives the correlation between plan length and runtime.

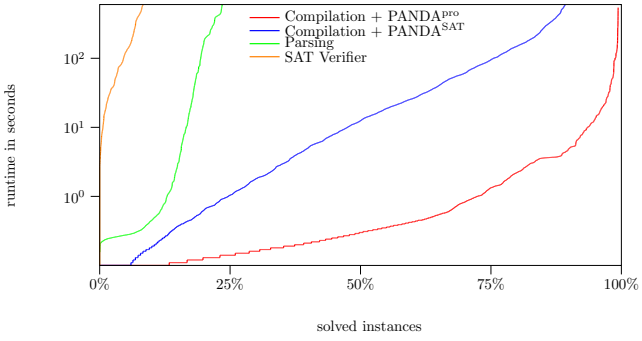


Figure 1: Solved instances in percent (on the y axis) relative to the runtime (on the x axis) for the TO setting.

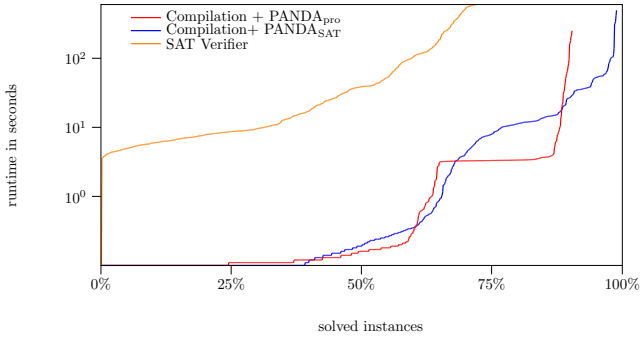


Figure 2: Solved instances in percent (on the y axis) relative to the runtime (on the x axis) for the PO setting.

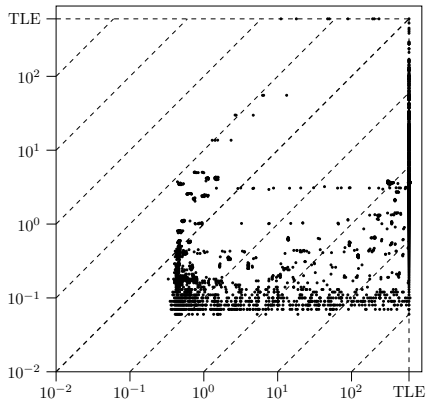


Figure 3: Runtime of our system (y axis) on the TO set compared to the parsing-based approach on the x axis (log scale).

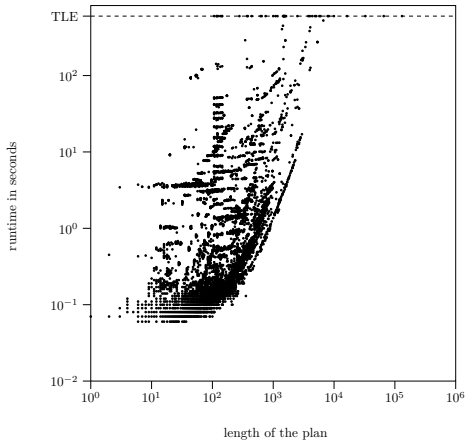


Figure 4: Runtime against length of the verified solution (be aware of the log scale).

to be an advantage in the Childsnack domain. In all other domains, our system has the highest coverage. For those domains where *not* all instances have been solved, we included the length of the shortest plan that could not be verified.

The next table (Table 1) gives statistics on plan length. Regarding the medians, AssemblyHierarchical is the domain with the shortest plans (14), and Towers the one with the longest (511). The median over all domains is 119.

Next, the table gives information about the runtime needed by our approach (combined with the progression search). The longest median runtime is needed for Minecraft-Player. In 16 domains, the median is one second or less. The last column gives the correlation between the plan length and the runtime needed for verification.

Table 2 shows the same characteristics for the PO setting. Notably the plans are much shorter (the average length is smaller by nearly one order of magnitude). Our compilation together with the SAT-based planner can verify all plans for 8 of the 9 domains with only Barman-BDI to have some plans that could not be verified. Notably, these plans are longer than almost all plans.

Figure 3 shows the runtime of our approach with the progression search in the TO setting compared to the parsing-based system. It can be seen that in most instances solved by both systems, our system is faster than the one from the literature. If it is not, the difference is about one second or less. Figure 4 shows a comparison of runtime and plan length. It can be seen that the runtime of equally long plans can be very different, but also that longer plans are harder to verify.

Figure 2 visualizes the runtime in the PO setting. It includes the compilation in combination with SAT-based PANDA and progression search and the SAT-based verifier. Like in the TO setting, our approach outperforms the SAT-based verifier. The plateau in the curves of our approaches are caused by the Monroe domain, where (especially using progression search) nearly the entire time is needed for grounding and not for solving the ground problem.

6 Discussion & Conclusion

We have presented an approach to compile HTN plan verification problems to HTN planning problems and have shown that recent planning systems can solve the resulting problems for plans with reasonable length (i.e., for plan lengths resulting from the recent IPC benchmarks/planners).

A possible criticism of a compilation-based approach might be that one has to rely on the correctness of the applied HTN planning system. So the question is why we rely more on these systems than on the planning system that has generated the plan in the first place. Since HTN planning systems are complex systems, we agree that these systems might also be incorrect (though this might also be the case for verifiers, of course). However the HTN planning systems used in our evaluation return the decomposition steps performed to find a plan as specified for the IPC. Therefore they provide a witness for the validity of their result (at least for cases where they find a solution) that can be checked with the much simpler systems based on these steps. So we can e.g. use the verifier developed for the IPC to check our results.

For cases where the HTN planning system does not find a solution, we cannot provide a meaningful explanation why planning failed. However, please note that what we would like to have here is a certificate of unsolvability of a planning problem, which is at least in classical planning an active field of research (see e.g. (Eriksson, Röger, and Helmert 2017; Eriksson and Helmert 2020)), though we are not aware of similar work in HTN planning.

As most important steps in future work we consider the collection of unsolvable instances from early runs of the IPC planners and the comparison to the parsing-based approach in the setting of PO planning.

The performance of the progression-based system on the PO benchmark set points to other lines of research. One is to help the planners by propagating the implications of the total order of the plan though the partial ordered HTN model. Since this is another compilation, it would preserve the property of needing no specialized solver. A second promising direction is to actually adapt the planner and, e.g., come up with specialized heuristics that take the additional information about the problem into account. Natural candidates would be the landmark heuristic by Höller and

Bercher (2021), which might benefit from the ordering constraints implied by the solutions, or the IP-based heuristic introduced by Höller, Bercher, and Behnke (2020), where it is straightforward to integrate the additional knowledge. However, such systems would, of course, lack the elegance of using standard planning systems.

Acknowledgments

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 232722074 – SFB 1102 / Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102.

References

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *Proc. of the 5th Annual Symposium on Combinatorial Search (SoCS)*, 2–9. AAAI Press.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of Hierarchical Plans via Parsing of Attribute Grammars. In *Proc. of the 28th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 11–19. AAAI Press.

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

Behnke, G. 2021. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning. In *Proc. of the 31st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 25–35. AAAI Press.

Behnke, G.; Bercher, P.; Kraus, M.; Schiller, M.; Mickeleit, K.; Häge, T.; Dorna, M.; Dambier, M.; Minker, W.; Glimm, B.; and Biundo, S. 2020a. New Developments for Robert – Assisting Novice Users Even Better in DIY Projects. In *Proc. of the 30th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 343–347. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the Complexity of HTN Plan Verification and Its Implications for Plan Recognition. In *Proc. of the 25th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 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 *Proc. of the 27th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 20–28. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-Ordered Hierarchical Planning through SAT. In *Proc. of the 32nd AAAI Conf. on Artificial Intelligence (AAAI)*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019. Bringing Order to Chaos – A Compact Representation of Partial Order in SAT-based HTN Planning. In *Proc. of the 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, 7520–7529. AAAI Press.

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

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *Proc. of the 28th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 6267–6275. IJCAI organization.

Bercher, P.; Behnke, G.; Kraus, M.; Schiller, M.; Manstetten, D.; Dambier, M.; Dorna, M.; Minker, W.; Glimm, B.; and Biundo, S. 2021. Do It Yourself, but Not Alone: *Companion*-Technology for Home Improvement – Bringing a Planning-Based Interactive DIY Assistant to Life. *Künstliche Intelligenz – Special Issue on NLP and Semantics* .

de Silva, L.; Padgham, L.; and Sardina, S. 2019. HTN-Like Solutions for Classical Planning Problems: An Application to BDI Agent Systems. *Theoretical Computer Science* 763: 12–37.

Eriksson, S.; and Helmert, M. 2020. Certified Unsolvability for SAT Planning with Property Directed Reachability. In *Proc. of the 30th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 90–100. AAAI Press.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability Certificates for Classical Planning. In *Proc. of the 27th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 88–97. AAAI Press.

Höller, D. 2021. Translating Totally Ordered HTN Planning Problems to Classical Planning Problems Using Regular Approximation of Context-Free Languages. In *Proc. of the 31st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 159–167. AAAI Press.

Höller, D.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System. In *Proc. of the 31st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 168–173. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and Goal Recognition as HTN Planning. In *Proc. of the 30th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, 466–473. IEEE Computer Society.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2021. The PANDA Framework for Hierarchical Planning. *Künstliche Intelligenz* doi:10.1007/s13218-020-00699-y.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020a. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proc. of the 34th AAAI Conf. on Artificial Intelligence (AAAI)*, 9883–9891. AAAI Press.

Höller, D.; and Bercher, P. 2021. Landmark Generation in HTN Planning. In *Proc. of the 35th AAAI Conf. on Artificial Intelligence (AAAI)*, 11826–11834. AAAI Press.

Höller, D.; Bercher, P.; and Behnke, G. 2020. Delete- and Ordering-Relaxation Heuristics for HTN Planning. In *Proc. of the 29th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 4076–4083. IJCAI organization.

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 28th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 114–122. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Plan Repair via Model Transformation. In *Proc. of the 43rd German Conference on AI (KI)*, 88–101. Springer.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020c. HTN Planning as Heuristic Progression Search. *Journal of Artificial Intelligence Research* 67: 835–880.

Köhn, A.; Wichlacz, J.; Torralba, Á.; Höller, D.; Hoffmann, J.; and Koller, A. 2020. Generating Instructions at Different Levels of Abstraction. In *Proc. of the 28th Int. Conf. on Computational Linguistics (COLING)*, 2802–2813. International Committee on Computational Linguistics.

Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN Planning Domain. *International Journal on Artificial Intelligence Tools* 26(5): 1760021:1–1760021:24.

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

Domain Analysis: A Preprocessing Method that Reduces the Size of the Search Tree in Hybrid Planning

Michael Staud
StaudSoft UG (haftungsbeschränkt), Ulm, Germany
michael.staud@staudsoft.com

Abstract

We introduce a new method that can reduce the size of the search tree in hybrid planning. Hybrid planning fuses task insertion HTN planning with POCL planning. As planning is a computationally difficult problem, the size of the search tree can grow exponentially to the size of the problem.

We create so-called plan templates in a preprocessing step of the domain. They can be used to replace an abstract task in a partial plan. This task then no longer needs to be decomposed.

We provide empirical evidence in favor of this approach and show that the use of plan templates can drastically reduce the size of the search tree in hybrid planners. We use a PANDA-like planner as a testbed and publicly available planning domains to verify our claims.

1 Introduction

Planning is an important branch of artificial intelligence and is widely used in practice. We focus on planning algorithms that create action sequences to achieve a given goal in a deterministic and fully observable world. One approach is planning algorithms that create a search tree to find a solution to a problem (Ghallab, Nau, and Traverso 2004). This is used in both classical planning and *Hierarchical Task Network* (HTN) planning. In the latter, the nodes of the tree contain partial plans with plan steps. In classical (non-hierarchical) planning, the nodes store states. Unfortunately, HTN planning is not flexible enough to be used with our approach. We therefore use *Hybrid planning* (Schattenberg and Biundo 2006; Biundo and Schattenberg 2001), which extends HTN planning with partial order causal link (POCL) techniques. It allows the insertion of new plan steps during the planning process (task insertion).

A hybrid planning domain contains primitive and abstract tasks. Primitive tasks correspond to operators known from classical planning. Abstract tasks represent complex courses of actions. They must be decomposed into more concrete tasks using *decomposition methods*. In this process, an instance of an abstract task is replaced by a partial plan, which was stored in the decomposition method. This process is repeated until there are only plan steps with primitive tasks. Each decomposition creates a new choice point in the search tree. This process may introduce new abstract tasks into the partial plan that must be decomposed again. This in turn creates new choice points in the search tree.

Our method is a *domain analysis* technique that preprocesses a planning domain D (see Section 2). The goal is to speed up the planning process. We create new decomposition methods. They decompose an abstract task into a single (newly generated) primitive task. Our innovation is that such a decomposition method does not introduce new plan steps with abstract tasks. The planning algorithm itself is not changed. Before the execution of the planning algorithm, the new domain $D_S \supseteq D$ is created, which contains the new methods and primitive tasks. After the planning algorithm has found a solution P_{Ds}, the plan steps with the newly added primitive tasks are removed from it. They are replaced by partial plans $P_{R_i}(\overline{\tau})$ (called *replacement plans*) that were generated during the preprocessing step. We call this the replacement process (see Section 4). After that, the solution P_{Ds} contains only primitive tasks found in the original domain D. The replacement process uses neither search nor further planning. A *plan template* is defined as the combination of a method, a primitive task and a replacement plan.
Our contribution is an algorithm that generates replacement plans and placeholder tasks in a preprocessing step, which are then used during planning to reduce the size of the search tree. After a plan is found, the placeholder tasks are replaced by the replacement plans.

In the following section, we first introduce hybrid planning. In the next section, we show how to create a plan template with hybrid planning. Then, we describe the post-processing step that is performed after the planning process. At this point, the replacement plans are inserted into the solution. We then present our results, which show the effectiveness of our approach.

2 Hybrid Planning

Hybrid planning combines the concepts of Hierarchical Task Network (HTN) planning and Partial-Order Causal-Link (POCL) planning (Bercher, Alford, and Höller 2019; Biundo and Schattenberg 2001). In our paper, hybrid planning is with task insertion (TIHTN = HTN planning with task insertion). The following definitions are taken in part from Bercher, Keen, and Biundo (2014). Let V be the set of all variables and C be the set of all constants.

A *hybrid planning domain* is a tuple $D = (T_a, T_p, M)$. T_a is the set of abstract tasks, T_p is the set of primitive tasks,

and M is the set of all decomposition methods. All sets are finite.

Both primitive and abstract tasks are tuples $t(\overline{\tau}) = \langle prec_t(\overline{\tau}), eff_t(\overline{\tau})\rangle$ consisting of a precondition and an effect. Each task has parameters $\overline{\tau}$. The preconditions and effects are conjunctions of literals and unequal variable constraints over the task parameters $\overline{\tau} = (\overline{\tau}_1, \ldots, \overline{\tau}_n)$, $\overline{\tau}_i \in C \cup V, i \in [1 \ldots n]$. A literal is an atom or its negation. An atom is a predicate applied to a tuple of terms. A task is grounded if all its variables are bound to constants. A unequal constraint can be between two variables or between a variable and a constant. The preconditions and effects of an abstract task have the semantics of Definition 7 from Bercher et al. (2016). If it is clear from the context, we will omit the parameters of a task.

Partial plans are tuples $P = (PS, \prec, VC, CL)$ consisting of plan steps PS that are uniquely labeled tasks $l : t(\overline{\tau})$. The set $\prec$ contains *ordering constraints* of the form $(l, l') \in PS \times PS$ that induce a partial order on the planning steps in PS. The set VC contains the *unequal variable constraints* and the set CL contains the *causal links*. The CSP in VC must be solvable. A causal link $l : t(\overline{\tau}) \rightarrow_{\phi(\overline{\tau}')} l' : t'(\overline{\tau}')$ (shortened: $l \rightarrow_\phi l'$) links a precondition literal $\phi(\overline{\tau}')$ of the plan step $l' : t'(\overline{\tau}')$ to a unifiable effect of $l : t(\overline{\tau})$. If there is another plan step $t''(\overline{\tau}'')$ in the plan that has an effect $\upsilon(\overline{\tau}'')$ that is unifiable with $\neg\phi(\overline{\tau}')$, and if the ordering constraints allow l'' to be ordered between l and l', we call this a *causal threat*. A precondition without a causal link is called *open*.

A partial plan P may contain abstract tasks. These must be decomposed during the planning process using *decomposition methods*. A method $m = \langle t(\overline{\tau}_m), P_m\rangle$ is a tuple that maps an abstract task $t(\overline{\tau}_m)$ to a partial plan $P_m = (PS_m, \prec_m, VC_m, CL_m)$ that "implements" the task (Bercher et al. 2016, Def. 7). For the following definition, we need the function $unique(P_m, \overline{\tau}) = P^*_m$, that replaces each variable and label with a new name that does not appear in any other partial plan. If a variable is used in the parameters $\overline{\tau}$, it is not changed.

Definition 1 (Task Decomposition)**.** Let $l : t_a(\overline{\tau}_t)$ be a plan step with an abstract task t_a to be decomposed with $m = \langle t_a(\overline{\tau}_m), P_m\rangle$. And let $P'_m = (PS'_m, \prec'_m, VC'_m, CL'_m) = unique(P_m, \overline{\tau}_m)[\overline{\tau}_{m,1}/\overline{\tau}_{t,1}, \ldots, \overline{\tau}_{m,n}/\overline{\tau}_{t,n}]$ be the partial plan to be inserted. The set $\prec_X = \{(l_1, l_2) \in PS \times PS'_m | (l_1, l) \in \prec\} \cup \{(l_1, l_2) \in PS'_m \times PS | (l, l_2) \in \prec\}$ defines the new additional ordering constraints. The set CL_X contains the new causal links. The function $fE(P, \phi)$ returns a task from the partial plan P that has an effect that is unifiable with ϕ. And the function $fP(P, \phi)$ returns a set of tasks, each of which has a precondition that is unifiable with ϕ. The two functions always find a return value, since our domain must follow Definition 7 from Bercher et al. (2016).

$$CL_X = \bigcup_{l' \rightarrow_\phi l'' \in CL} \begin{cases} fE(P'_m, \phi) \rightarrow_\phi l'' & l' = l \\ \bigcup_{l^* \in fP(P'_m, \phi)} l' \rightarrow_\phi l^* & l'' = l \\ l' \rightarrow_\phi l'' & \text{else} \end{cases}$$

The new partial plan has the form $P' = (PS', \prec', VC', CL')$. Where $PS' = (PS \setminus \{l\}) \cup PS'_m$, $\prec' = \prec \cup \prec'_m \cup \prec_X$, $VC' = VC \cup VC'_m$ and $CL' = CL_X \cup CL'_m$ holds.

Task insertion is defined as the insertion of a new plan step with a primitive task $l : t(\overline{\tau}')$.

Definition 2 (Planning Process)**.** A partial plan can be refined by decomposing an abstract task, task insertion, adding a variable constraint, adding an ordering constraint or adding a causal link.

A hybrid planning problem is a combination of a domain D and an initial plan P_{init}. Let L be the set of all conjunctions of grounded literals and L_+ be the set of all conjunctions of positive grounded literals. As in standard POCL planning, we encode the initial state $I_s \in L_+$ and the goal description $G_s \in L$ as two special primitive tasks $t_0(\overline{\tau}) = \langle \varnothing, I_s\rangle$, $t_\infty(\overline{\tau}) = \langle G_s, \varnothing\rangle$ in the initial plan P_{init}.

Definition 3 (Solution)**.** A plan P_{sol} is a solution if there are no open preconditions, when all tasks are primitive and grounded and when there are no causal threats (Bercher, Keen, and Biundo 2014). This implies that all linearization of P_{sol} can be executed.

3 Plan Template Generation

In the preprocessing step, a *plan template* is created for an abstract task $t_a(\overline{\tau}_a) \in T_a$. This is done using a modified hybrid planner. The template is a tuple $P_T = \langle P_R(\overline{\tau}), m_T(\overline{\tau}), p_T(\overline{\tau})\rangle$ over the parameters $\overline{\tau}$ ($\overline{\tau} \supseteq \overline{\tau}_a$). We need to generate:

- a *replacement plan* $P_R(\overline{\tau})$, which is a partial plan
- a primitive task $p_T(\overline{\tau})$
- and a method $m_T(\overline{\tau}) = \langle t_a(\overline{\tau}_a), P'\rangle$ that allows the planner to select the primitive task. It holds $P' = (\{l' : p_T(\overline{\tau})\}, \emptyset, \emptyset, \emptyset)$.

The creation process is then the following:

Task Selection: Select $t_a(\overline{\tau}_a)$ (see Section 3.2).

Problem Creation: The replacement plan $P_R(\overline{\tau})$ is the solution to the following hybrid planning problem in the original domain D: The initial plan P_{init} contains the abstract task $t_a(\overline{\tau}_a)$ and the special tasks $t_0(\overline{\tau}_0)$ and $t_\infty(\overline{\tau}_\infty)$. The initial state I_s is equal to the preconditions of $t_a(\overline{\tau}_a)$ and the goal description G_s is equal to the effects of $t_a(\overline{\tau}_a)$ (if it has effects, otherwise the goal description is empty). The plan steps are then $l_0 : t_0$, $l_a : t_a$, $l_\infty : t_\infty$. Thus, $P_{init} = \{\{l_0, l_a, k_\infty\}, \{(l_0, l_a), (l_a, l_\infty)\}, \varnothing, \varnothing\}$. Note that these tasks need not be grounded. The solution is also not grounded. A replacement plan $P_R(\overline{\tau})$ can and should contain unbounded variables for maximum flexibility. According to our empirical tests, if the abstract task $t_a(\overline{\tau}_a)$ has effects, this improves the quality of the plan template.

Replacement Plan Creation: We use a modified hybrid planner. The *first modification* of the planning process (see Definition 2) is that we introduce a new (additional) refinement type when we encounter an open precondition. Let $r \in prec_t(\overline{\tau})$ be an open precondition of the plan step $l : t(\overline{\tau})$. Then we add a new effect r to the task $t_0(\overline{\tau}_0)$ and create a causal link $l_0 \rightarrow_r l$. Note that the parameters $\overline{\tau}$ from t that occur in r are added as new parameters to $\overline{\tau}_0$.

The *second change* is that the planning problem is now an optimization problem that attempts to minimize an objective function f_{t_a}, described in Section 3.1. A solution P_{sol} must satisfy the solution criteria from Definition 3 and it must be

minimal with respect to f_{t_a}. This is necessary to produce a useful plan template (see Section 5).

The solution $P_{sol} = (PS_{sol}, \prec_{sol}, VC_{sol}, CL_{sol})$ is then the replacement plan $P_R(\overline{\tau})$. The parameters $\overline{\tau}$ contain all variables from P_{sol}.

Primitive Task Creation: From a solution P_{sol}, a primitive task $p_T(\overline{\tau}) = \langle prec_{p_T}(\overline{\tau}), \mathit{eff}_{p_{p_T}}(\overline{\tau})\rangle$ is generated. The precondition $prec_{p_T}(\overline{\tau})$ of this task contains the effects of the primitive task $t_0(\overline{\tau})$ in P_{sol}. The unequal variable constraints VC_{sol} in P_{sol} are also added as a precondition. Thus, $prec_{p_T}(\overline{\tau}) = \mathit{eff}_{t_0}(\overline{\tau}) \cup VC_{sol}$ holds.

The effects $\mathit{eff}_{p_T}(\overline{\tau})$ of the task consist of all effects that would be in the goal state if we were to execute P_{sol}. Thus, we do not add effects that are undone during the execution of the partial plan (since it is a solution, this is true in any linearization). Thus, it holds:

$$\mathit{eff}_T(\overline{\tau}) = \bigcup_{\substack{r \in \mathit{eff}_{p_{T'}}(\overline{\tau'}) \\ l' : p_{T'} \in PS_{sol}}} \begin{cases} r & \text{if no other effect } \neg r' \\ & \text{of a plan step } l'' \text{ exists} \\ & \text{with } (l', l'') \in \prec_{sol} \\ \varnothing & \text{else} \end{cases}$$

Unless all effects and variable constraints are added, it would not be possible to replace the primitive task $p_T(\overline{\tau})$ with the replacement plan $P_R(\overline{\tau})$ without using search (see Section 4).

Method Creation: The decomposition method $m_T(\overline{\tau})$ is created according to its definition. The method $m_T(\overline{\tau})$ and the primitive task $p_T(\overline{\tau})$ are added to the original planning domain D.

3.1 Objective Function

The objective function evaluates a partial plan P. Let $P = (PS, \prec, VC, CL)$ be the current partial plan at template generation. And let $t_0(\overline{\tau}) = \langle prec_{t_0}(\overline{\tau}), \mathit{eff}_{t_0}(\overline{\tau})\rangle \in PS$ be the initial task. The abstract task for which the template is generated is called $t_a(\overline{\tau})$. We define the following auxiliary functions:

- $ch(t_a(\overline{\tau}), a)$: Returns 1 if the literal's predicate occurs as an effect in any decomposition of $t_a(\overline{\tau})$. Otherwise, it returns 0.
- $st_{ct}(A)$: Given a set of atoms A, this function checks whether the state constraints according to Gerevini and Schubert (1998) are violated. It returns ∞ if a violation is found. Otherwise, it returns 0.
- $P(a \in S(t_a))$: Probability that the predicate of a is available as a precondition for t_a in a randomly selected problem in the domain D. This is approximated by solving example domains. We count in all these solutions which predicates occur in causal links. In doing so, we consider only the links that go from a plan step that was not generated by a decomposition of $t_a(\overline{\tau})$ to a plan step that was generated by a decomposition of $t_a(\overline{\tau})$. From these counts, the probability of occurrence of each predicate is calculated.

We tested two different objective functions $f_{t_a}(P_{opt}, t_a)$:

$f_1(P, t_a) = |prec_{t_0}(\overline{\tau})| + |PS|$

$$f_2^x(P, t_a) = |PS| + \sum_{a \in prec_{t_0}(\overline{\tau})} + ch(t_a(\overline{\tau}), a)x + st_{ct}(prec_{t_0}(\overline{\tau}))$$

Regarding f_2^x we used the two variants f_2^1 and $f_2^{P(a \in S(t_a))}$. Often there are multiple solutions where an objective function is minimal or near-minimal. If there are example problems, we evaluate the plan templates with them by solving them. We then select the plan template that maximizes the reduction of the search space. The others are *discarded* (see Table 1, `rover` domain).

3.2 Selection of Abstract Tasks

To determine which abstract task is suitable for generating plan templates, we use the task decomposition graph (Bercher, Keen, and Biundo 2014). For each task, we compute the set of mandatory M_{t_a} and optional tasks O_{t_a}. The mandatory tasks occur in every decomposition of a given abstract task t_a. The optional tasks occur in at least one decomposition. Thus, the amount of mandatory tasks $|M_{t_a}|$ relative to the optional tasks gives us an indication of how much the decompositions of an abstract task t_a will differ from each other. In our experiments, the higher the ratio $|M_{t_a}|/|O_{t_a}|$, the more successful the generation of a plan template. Thus, we select the abstract task with the highest ratio.

4 Replacement Process

The replacement process is a post-processing step performed after the solution $P_{Ds} = (PS_{Ds}, \prec_{Ds}, VC_{Ds}, CL_{Ds})$ has been generated in D_s. The goal is then to generate a solution $P_{Ds'} = (PS_{Ds'}, \prec_{Ds'}, VC_{Ds'}, CL_{Ds'})$ that is valid in the original domain D. For each plan step $l : p_T(\overline{\tau}_t)$, which uses a primitive task from a plan template $P_T(\overline{\tau}) = \langle P_R(\overline{\tau}), m_T(\overline{\tau}), p_T(\overline{\tau})\rangle$, $p_T(\overline{\tau}) = \langle prec_{p_T}(\overline{\tau}), \mathit{eff}_{p_T}(\overline{\tau})\rangle$ we perform the following steps:

Decomposition: The plan step $l : p_T(\overline{\tau})$ is treated as an abstract task and decomposed using the method $m' = \langle p_T(\overline{\tau}), P_R(\overline{\tau})\rangle$. Let $P'_R = (PS'_R, \prec'_R, VC'_R, CL'_R) = \mathit{unique}(P_R, \overline{\tau})[\overline{\tau}_1/\overline{\tau}_{t,1}, \ldots, \overline{\tau}_n/\overline{\tau}_{t,n}]$ (see Definition 1).

Relinking: After the decomposition, the plan steps $l_0, l_\infty \in PS'_R$ are removed and their causal links are relinked. Let $l_0 \to_\phi l'$ be a casual link from l_0. And let $l'' \to_\phi l$ be a causal link to $l : p_T$. Then these two links are replaced by $l'' \to_\phi l'$. Similarly, a causal link of the form $l' \to_\phi l_\infty$ is treated. Let $l \to_\phi l''$ be a causal link with $l : p_T$. Then these two links are replaced again by $l' \to_\phi l''$. Note that because of the way p_T is defined, relinking is always possible. The new partial plan after the decomposition is then $P_{Ds'}$.

Removing Causal Threats: We remove causal threats introduced by the decomposition step. Let $\neg\phi$ be an effect of $l' \in PS_{Ds'}$ that threatens a causal link $l'' \to_\phi l^*$. Let $k_p(l)$ be a function that returns 1 if $l \in PS'_R$ and 0 otherwise.

It cannot be $k_p(l') = k_p(l'') = k_p(l^*)$ because both the replacement plan P_R and the original plan P_{Ds} were solutions (and thus without causal threats). It is also not possible that $k_p(l'') \neq k_p(l^*)$ holds, because then the causal link would have been created during the *relinking* step. And since we have only connected pre-existing causal links together, there can therefore be no causal threat. It follows that

satellite2	1o1s1m	2o1s2m	4o2s3m	3o2s2m
WO/W/I	66/17/3.88	11371/340/33.44	42140/29959/1.40	130791/1873/69.83
	1o2s1m	2o1s1m	2o2s2m	3o1s3
WO/W/I	101/27/3.74	449/109/4.11	4472/119/37.57	> 300000/22312/> 12.44
	3o1s1m	3o2s1m	3o2s3m	3o3s1
WO/W/I	5472/1209/4.52	15679/2773/5.64	> 300000/5483/54.71	29570/12945/2.28
	3o3s2m	3o3s3m	3o1s2m	
WO/W/I	> 300000/2380/> 126.05	> 300000/986/> 304.25	136927/3368/40.65	
woodworking	00	01	02	03
WO/W/I	4370/1738/2.51	67/67/1.00	79/79/1.00	374/224/1.67
	04			
WO/W/I	1930/1797/1.07			
transport	pfile01	pfile02	pfile03	
WO/W/I	497/57/9.20	237244/63242/3.75	252274/127007/1.98	
rover	pfile1	pfile2	pfile3	pfile4
WO/W/I	2194/1006/2.18	130/3347/none	19603/8454/2.31	636/405/1.57
WD/WD2	4330/4786	612/86	21745/22290	1234/552

Table 1: *Satellite Domain:* Problems of the satellite domain solved without and with plan templates. It holds XoYsZm = `Xobs-Ysat-Zmod.hddl` and WO/W/I = Solved without plan template / Solved with plan template / The improvement factor.
Woodworking Domain: Problems of the woodworking domain. In this domain, an abstract task is on average decomposed into two primitive tasks without plan template. Thus, there is not much room for improvement when the plan template is used.
Transport Domain: Problems of the transport domain.
Rover Domain: Problems of the rover domain solved without and with plan templates. The rover domain contains abstract tasks without effects. However, due to the structure of the rover domain, a plan template could still be generated. We also added measurements for 2 plan templates that were discarded because they were not as effective as the selected template. Note that in the `pfile02` example, the selected template performs very poorly and a discarded template reduces the number of search nodes. It holds WD/WD2 = With Discarded Template / With Discarded Template 2.

domain	satellite2	rover	transport	woodworking
States	182	31476	73	66
Generated	1	4	2	4
Used	1	1	2	1

Table 2: Number of search nodes needed to generate a plan template. Note that adding multiple plan templates to a domain can reduce performance because it increases the branching factor during the planning process. In the transport domain, we used 2 templates, which increases the number of states in the first example `pfile01`.

$k_p(l'') = k_p(l^*)$ and $k_p(l') \neq k_p(l'')$.

If $k_p(l') = 0$, then $k_p(l'') = k_p(l^*) = 1$. Then we can add (l', l'') or (l^*, l') to the ordering constraints $\prec_{Ds'}$. This is always possible because $\prec_{Ds}$ cannot contain both (l'', l') and (l', l^*) after the *decomposition* step as defined by Definition 1. All ordering constraints (l_1, l_2) where $k_p(l_1) \neq k_p(l_2)$ holds were added during this step.

If $k_p(l') = 1$, it follows that $k_p(l'') = k_p(l^*) = 0$. If adding (l', l'') or (l^*, l') to the ordering constraints $\prec_{Ds'}$ is not possible, we must relink the causal link $l'' \rightarrow_\phi l^*$. We know that it must hold $\neg\phi \notin \mathit{eff}_{p_T}(\overline{\tau})$ because:

- all effects of a plan step in P_R, except those removed by another task, are added to $\mathit{eff}_{p_T}(\overline{\tau})$.
- if $\neg\phi \in \mathit{eff}_{p_T}(\overline{\tau})$ then there would be no causal threat because P_{Ds} is a solution.

It follows $\neg\phi \notin \mathit{eff}_{p_T}(\overline{\tau})$ and according to Section 3 (Primitive Task Creation) there must be at least one more task $l^{**} \in PS'_R$ with effect ϕ and $(l', l^{**}) \in \prec'_R$. We then add $l^{**} \rightarrow_\phi l^*$ and delete the old causal link.

Suppose the new link is again threatened by another plan step l^{***}, $k_p(l^{***}) = 1$ with an effect $\neg\phi$. Then the link is rewired again. This is repeated until the link is no longer threatened. And there cannot be a plan step l^{***} with $k_p(l^{***}) = 0$ that threatens the link, because in that case it would also threaten the original link, which is a contradiction because P_{Ds} is a solution.

Theorem 1. The replacement process does not introduce new flaws into the plan. The resulting plan is a solution.

Proof Sketch: We will show that the plan satisfies the solution criteria after the replacement process:

- *No Open Preconditions*: In the replacement plan $P_R(\overline{\tau})$ all preconditions are linked. And all other links that are broken during the *relinking* step are reconnected.
- *No Causal Threats*: All causal threats were repaired in the last step of the replacement process.
- *Grounded*: Each plan step is grounded because all parameters of $P_T(\overline{\tau})$ are constants. The parameters also do not violate any variable constraints in $P'_R(\overline{\tau})$. This is the case because we have added all the variable constraints of $P_R(\overline{\tau})$ to the preconditions of the primitive task $p_T(\overline{\tau})$.

□

5 Results

We evaluated four sample domains (University Ulm 2019) with plan templates generated with the $f_2^{P(a \in S(t_a))}$ function (see Table 1). In each test domain, the algorithm selects n abstract tasks for which a plan template is generated according to Section 3.2. The number n of plan templates is determined by the user. We then compared the size of the search tree when using the original domain D and when using the domain D_{new} with the plan template. We counted the search nodes in all our tests. These are shown in the tables.

We used a self-developed partial order planner with support for abstract tasks and TIHTN+POCL planning using the hybrid heuristic $h_{\#F} + h_{TC}$ from Bercher, Keen, and Biundo (2014). The heuristic works with grounded abstract tasks, so we find a random grounding to apply this heuristic to a lifted task. The planner is sound and complete. As a flaw selection heuristic, we select the flaw that has the least number of distinct resolutions. This results in a low branching factor.

The `satellite2` domain showed the largest performance gain. Here, the planner mostly (in 86% of the cases) chose the template method during the planning process, which reduced the number of states by a factor of up to 300. In the `transport` domain, only one plan template was used. In the `rover` example, only one template was also used. Since the rover example contains 9 different abstract tasks, this reduced the usability of our plan template, which only replaced one abstract task.

In all examples, we also tried to use more templates, but this resulted in an increase in the number of search nodes. Without *task insertion* the templates were not used by the planner because in most of the cases the preconditions of the primitive task p_T could not be fulfilled.

We also tested the different evaluation functions from Section 3.1. The f_1 function did not provide any usable templates. Without evaluating the initial conditions added during the planning process, the planning algorithm does not know in which direction to optimize.

The functions f_2^1 and $f_2^{P(a \in S(t_a))}$ provided usable results. But only $f_2^{P(a \in S(t_a))}$ was successful in all sample domains (see Table 2). The f_2^1 function did not produce a usable plan template in the `transport` domain. When we used $f_2^{P(a \in S(t_a))}$, we did not use the same problems for the evaluation of the plan templates and performance measurements.

The performance gain is also supported by research about plan merging. Since we solve a part of the plan independently, the research results of Korf (1987) apply. He showed that if we can solve n subgoals separately, this divides the base and exponent of the complexity function by n. Thus, theoretically, our approach can lead to an exponential reduction in the size of the search tree.

6 Related Work

Macros are also a technique to encapsulate sequences of operators. Like plan templates, they have preconditions and effects like operators. This idea originated in the 1970s (Fikes and Nilsson 1971). They are generated from plans in a preprocessing step. These plans are solutions of example problems. In contrast to our approach, they do not use HTN domains and therefore cannot use the additional information they contain (abstract tasks, methods) (Chrpa, Vallati, and McCluskey 2015).

7 Conclusions

We presented a new domain analysis method that can reduce the size of the search tree. We showed that a drastic reduction in the size of the search tree is empirically possible. This is even more effective when example problems are available to guide the search when generating the plan templates. This is because it improves the quality of the plan templates.

This method can be used wherever many problems need to be solved in the same domain. It can quickly generate plan templates. Further research is needed to determine which domain features make plan template generation successful.

References

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

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 *ECAI 2016*, 225–233. IOS Press.

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

Biundo, S.; and Schattenberg, B. 2001. From Abstract Crisis to Concrete Relief – A Preliminary Report on Combining State Abstraction and HTN Planning. In *ECP 2001*, 157–168. Springer.

Chrpa, L.; Vallati, M.; and McCluskey, T. L. 2015. On the Online Generation of Effective Macro-Operators. IJCAI 15, 1544–1550. AAAI Press.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. IJCAI 71, 608–620. Morgan Kaufmann.

Gerevini, A.; and Schubert, L. 1998. Inferring State Constraints for Domain-Independent Planning. AAAI 98/IAAI 98, 905–912. AAAI Press.

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

Korf, R. E. 1987. Planning as Search: A Quantitative Approach. *AI 87* 33(1): 65–88.

Schattenberg, B.; and Biundo, S. 2006. A Unifying Framework for Hybrid Planning and Scheduling. In *KI 06*, 361–373. Springer.

University Ulm. 2019. PANDA Planning Domains and Problems. Https://www.uni-ulm.de/en/in/ki/research/-software/panda/, Accessed: 2020-10-01.

GTPyhop: A Hierarchical Goal+Task Planner Implemented in Python

Dana Nau,[1,2] **Yash Bansod,**[2] **Sunandita Patra,**[1] **Mark Roberts,**[3] **Ruoxi Li**[1]

[1]Dept. of Computer Science and [2]Institute for Systems Research, U. of Maryland, College Park, MD, USA
[3]The U.S. Naval Research Laboratory, Code 5514, Washington, DC, USA
[1,2]{nau, yashb, patras, rli12314}@umd.edu, [3]{first.last}@nrl.navy.mil

Abstract

The Pyhop planner, released in 2013, was a simple SHOP-style planner written in Python. It was designed to be easily usable as an embedded system in conventional applications such as game programs. Although little effort was made to publicize Pyhop, its simplicity, ease of use, and understandability led to its use in a number of projects beyond its original intent, and to publications by others.

GTPyhop (Goal-and-Task Pyhop) is an extended version of Pyhop that can plan for both goals and tasks, using a combination of SHOP-style task decomposition and GDP-style goal decomposition. It provides a totally-ordered version of Goal-Task-Network (GTN) planning without sharing and task insertion. GTPyhop's ability to represent and reason about both goals and tasks provides a high degree of flexibility for representing objectives in whichever form seems more natural to the domain designer.

1 Introduction

Pyhop[1] is a simple HTN planner written in Python, comprising less than 150 lines of Python code. Its planning algorithm is based on the one in SHOP (Nau et al. 1999), but it avoids the need for a specific "planning" language by having the task network and its methods written directly in Python. Pyhop's development was motivated by an observation that application developers were often writing planning systems themselves, rather than learning specialized AI planning languages (Nau 2013). Pyhop was written in hopes of providing an HTN (Hierarchical Task Network) planner that could be easily understood by non-AI practitioners.

Pyhop's author made little effort to publicize it, but the ease with which it could be understood and used has made it useful for rapid prototyping, leading to its use in a number of projects and publications by others (see Section 2).

This paper describes GTPyhop, which extends Pyhop to plan for goals as well as tasks. It combines aspects of both HTN planning as in Pyhop and SHOP, and HGN planning as in GDP (Goal Decomposition Planner) (Shivashankar et al. 2012). In the terminology of (Alford et al. 2016b), it does a totally-ordered version of GTN planning without sharing and task insertion (there is an example at the end of Section 3).

GTPyhop's source code is about four times as big as Pyhop's. It includes the following features:

- Rather than a task list, GTPyhop has a *to-do* list that contains zero or more actions, tasks, and goals. Like Pyhop, it decomposes tasks using task methods; and like GDP, it decomposes goals using goal methods. However, all methods return to-do lists, rather than Pyhop's task lists or GDP's goal lists (see example at end of Section 3). Thus a planning domain may use any arbitrary combination of task decomposition and goal decomposition.
- Since HGN planning semantics corresponds readily to classical goal semantics (Shivashankar et al. 2012), it can be used to guarantee soundness. To enforce soundness, when GTPyhop decomposes a goal g, it verifies whether the resulting plan actually accomplishes g, and backtracks if g is not accomplished. We anticipate that in future work, this may be useful for purposes such as verification and validation of domain descriptions.
- GTPyhop can load multiple planning domains into memory, and switch among them without having to restart Python each time. GTPyhop also includes more documentation than Pyhop, and additional debugging features.

The GTPyhop software distribution is available for download under an open-source license.[2] In addition to GTPyhop, it includes several example domains, a test harness for running them, and a simple example of planning-and-acting integration: a version of the Run-Lazy-Lookahead actor (Ghallab, Nau, and Traverso 2016) that uses GTPyhop as its planner.

The paper is organized as follows. We provide some context for the original version of Pyhop (Section 2), describe GTPyhop (Section 3), and briefly describe several research projects in which GTPyhop is being used and extended (Section 4). This is followed by discussions of related work (Section 5), some of GTPyhop's limitations (Section 6), and concluding remarks (Section 7).

2 Why does Py Hop?

As we mentioned earlier, Pyhop is basically a simplified version of SHOP that uses Python syntax. For example, actions and methods are written directly as Python functions. Their preconditions are Python *if* tests, and their effects are their returned values.

[1]https://bitbucket.org/dananau/pyhop/src/master/

[2]https://github.com/dananau/GTPyhop

According to (Nau 2013), the original motivation for Pyhop was a workshop on AI in games (Lucas et al. 2012) in which many of the attendees were developing games that incorporated AI planners as subsystems. The approach was like the way AI planning has been used in other systems that operate in dynamically changing environments:

- Approximate a part of the system's objective as a planning problem p, and develop a special-purpose planner for p.
- Use the planner as a subroutine, calling it repeatedly to replan as the world changes. The planner operates online, tightly integrated with the rest of the system.

Such integration is easier if the planner is small, easily understandable, and uses data structures compatible with those used in the larger system in which the planner is embedded, rather than requiring the data to be translated between two different representation schemes. Pyhop was written as an example of how to facilitate such integration.

The only efforts to publicize Pyhop were a brief announcement on the SHOP web site[3] and an invited workshop talk (Nau 2013) with no published paper, just slides. Despite this, a Google Scholar search[4] shows 66 publications that refer to Pyhop. In many of them, Pyhop is used for applications having nothing to do with games. There also have been several forks of Pyhop, e.g., (McGreggor 2014; Cheng et al. 2018), and a re-implementation of Pyhop in C++ (Jacopin 2020).

3 GTPyhop

Figure 1 shows the GTPyhop planning algorithm. Like Pyhop, it does a depth-first search with no loop detection (which wouldn't be useful here, see Alford et al. (2012)). Note that:

1. apply-action-and-continue and refine-task-and-continue handle actions and tasks the same way that Pyhop does.
2. The way refine-goal-and-continue decomposes goals is based on GDP (Shivashankar et al. 2012).
3. A mixture of task decomposition and goal decomposition may be used throughout a planning domain. In the to-do lists T and T_{sub} in lines (*i*), (*ii*), and (*iii*), each element may be a task, goal, or action.
4. In line (*iii*), g is a goal, so GTPyhop needs to ensure that T_{sub} achieves g. To do so, it appends to T_{sub} a dummy action *verify*(g) that has g as a precondition. If the state produced by T_{sub} satisfies g, the action has no effect. Otherwise the action fails, making GTPyhop backtrack.

3.1 Representations and examples

We now describe the basic elements of a GTPyhop planning domain, with examples from the GTN blocks-world domain included with the GTPyhop software distribution.[5]

Domains are Python objects that contain all the elements of a planning domain, e.g., gtpyhop.Domain('blocks_gtn').

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

[4] https://scholar.google.com/scholar?hl=en&q=pyhop

[5] https://github.com/dananau/GTPyhop/tree/main/Examples/blocks_gtn

```
GTPyhop(s_0, T)                                                    (i)
  return seek-plan(s_0, T, [])   # base case for seek-plan

seek-plan(s, T, π)
  # recursive DFS; π is the current partial solution
  if T = [] then return π
  t ← the first element of T
  T' ← the rest of T
  case(t):    # solve t, then plan for T'
    action: return apply-action-and-continue(s, t, T', π)
    task: return refine-task-and-continue(s, t, T', π)
    goal: return refine-goal-and-continue(s, t, T', π)

apply-action-and-continue(s, a, T', π)
  if action a is applicable in state s:
    return seek-plan(γ(s, a), T', π + [a])
  else: return failure

refine-task-and-continue(s, τ, T', π)
  M ← {task-methods that were declared relevant for τ}
  for each m ∈ M that is applicable in s:
    T_sub ← decomp(s, m)                                           (ii)
    π ← seek-plan(s, T_sub + T', π)
    if π ≠ failure then return π
  return failure

refine-goal-and-continue(s, g, T', π)
  M ← {goal-methods that were declared relevant for g}
  for each m ∈ M that is applicable in s:
    T_sub ← decomp(s, m) + [verify(g)]                             (iii)
    π ← seek-plan(s, T_sub + T', π)
    if π ≠ failure then return π
  return failure
```

Figure 1: GTPyhop pseudocode. The arguments are the initial state s_0 and a to-do list T (a list of actions, tasks, and goals). GTPyhop returns a solution plan π, or failure if no solution exists. $\gamma(s, a)$ is the state produced by a, and decomp(s, m) is the to-do list produced by m. "+" is concatenation of lists.

States are Python objects that contain collections of state-variable bindings. When one first defines a state s, it is easiest to write the variable bindings in dictionary form, as in Figure 2. Afterward, one can use a Python version of state-variable notation, e.g., s.pos[x] = 'hand' in Figure 3.

Actions and methods are Python functions, with the current state as the first argument. There isn't a special reasoning system (e.g., SHOP's Horn-clause inference) to evaluate an action's or method's preconditions. Instead, preconditions are Python *if* tests. Similarly, an action's effects and a method's to-do list are produced by ordinary Python computations.

For example, Figure 3 shows the blocks-world pickup action. Its arguments are the current state s and the block x to pick up. If the precondition (the first *if* test) is satisfied, the action modifies the state to say that x is in the robot hand, and returns the modified state. Otherwise the action returns no value, which tells GTPyhop the action is inapplicable.

Tasks and task methods: Tasks are written as Python tuples. For example, let ('take',x) be the task of picking up a block

```
sus_s0 = gtpyhop.State('Sussman initial state')
# Python dictionary notation for sus_s0.pos['a'] = 'table', etc.
sus_s0.pos = {'a':'table', 'b':'table', 'c':'a'}
sus_s0.clear = {'a':False, 'b':True, 'c':True}
sus_s0.holding = {'hand':False}

sus_sg = gtpyhop.Multigoal('Sussman goal')
sus_sg.pos = {'a':'b', 'b':'c'}
```

Figure 2: GTPyhop version of the Sussman anomaly (Ghallab, Nau, and Traverso 2004, Section 4.4). In the initial state sus_s0, blocks a and b are on the table, and block c is on a. The goal sus_sg specifies that a is on b, and b is on c.

```
def pickup(s,x):
   if s.pos[x] == 'table' and s.clear[x] == True  \
         and s.holding['hand'] == False:
      s.pos[x] = 'hand'
      s.clear[x] = False
      s.holding['hand'] = x
      return s
gtpyhop.declare_actions(pickup)
```

Figure 3: The blocks-world pickup action. The arguments are the current state s and a block x. If the precondition (the *if* test) is satisfied, the action modifies s and returns it. The last line declares pickup to be an action.

```
def m_take(s,x):
   if s.clear[x] == True:          # precondition
      if s.pos[x] == 'table':      # decide what to do
         return [('pickup', x)]
      else: return [('unstack', x, s.pos[x])]
gtpyhop.declare_task_methods('take',m_take)
```

Figure 4: A task method. Its arguments are the current state s and a block x. If the precondition is satisfied, it returns a to-do list containing a pickup action if x is on the table, or an unstack action if x is on a block. The last line declares m_take to be relevant for all tasks of the form (take, ...).

x that may be either on the table or a block. Figure 4 shows a method for this task. If there are several methods for the same task, GTPyhop (like Pyhop and SHOP) will try them in the order that they occur in the source file.

Goals can be represented in two ways. A *unigoal* is a triple that represents a desired value for a state variable, e.g., ('pos', 'a', 'b') is the goal of reaching any state s such that s.pos['a']=='b'. A *multigoal* is a state-like object that represents a conjunction of unigoals, e.g., sus_sg in Figure 2 represents the conjunction of ('pos', 'a', 'b') and ('pos', 'b', 'c').

Goal methods include *unigoal methods* for decomposing unigoals, and *multigoal methods* for decomposing multigoals. Figure 5 shows a multigoal method that implements a near-optimal block-stacking algorithm. For example, find_plan(sus_s0,sus_sg) returns the following plan:

```
def m_moveblocks(s, mgoal):
   for x in all_clear_blocks(s):   # find a block to move
      stat = status(x, s, mgoal)
      if stat == 'move-to-block':
         where = mgoal.pos[x]   # where to move it
         return [('take',x), ('put',x,where), mgoal]     (iv)
      elif stat == 'move-to-table':
         return [('take',x), (put,x,'table'), mgoal]     (v)
   for x in all_clear_blocks(s):   # resolve deadlock
      if status(x, s, mgoal) == 'waiting'  \
            and s.pos[x] != 'table':
         return [('take',x), ('put',x,'table'), mgoal]     (vi)
   return [ ]    # no blocks need to be moved
gtpyhop.declare_multigoal_methods(m_moveblocks)
```

Figure 5: A multigoal method that implements the block-stacking algorithm in (Gupta and Nau 1992): if a block needs moving and is ready to move to its final location, then do so and continue planning for mgoal; else if there's a deadlock then resolve it and continue planning for mgoal; else we're done. There are two helper functions: all_clear_blocks returns a list of clear blocks, and status tells whether a block x needs to be moved and whether it is ready to be moved.

```
[('unstack', 'c', 'a'), ('putdown', 'c'), ('pickup', 'b'),
 ('stack', 'b', 'c'), ('pickup', 'a'), ('stack', 'a', 'b')]
```

A GTN planning example. In lines (*iv*), (*v*), and (*vi*) of Figure 5, the to-do lists contain two tasks and a multigoal. GTPyhop (Figure 1) uses the right kind of method for each.

4 Example Usages

There are several research projects in which GTPyhop is being used and extended. We briefly describe them below.

Bansod et al. (2021) describes an integrated system for hierarchical planning and acting in dynamically changing environments. An important component of this system is a re-entrant planning algorithm based on GTPyhop.

A paper in preparation integrates GTPyhop with reinforcement learning. The work uses the goal network provided by GTPyhop to guide a curricula for multi-task learning. During acting, it executes the goal network provided by GTPyhop.

GTPyhop is also being used in a research project to develop temporal planning algorithms in multi-agent environments. For this purpose, modifications are being made to support communication among multiple agents, and representation and reasoning about temporal constraints.

In our future work, we anticipate the possibility of using goals for verification and validation of domain descriptions.

5 Related Work

The closest related theoretical work is (Alford et al. 2016b), which related task networks and goal networks under various semantics, including HTN, HGN, and GTN planning, task (or goal) insertion, and sharing.

Several HTN planners introduced in the 1970s through 1990s are no longer available for comparison. One of the first

was NOAH (Sacerdoti 1975), which was followed by Nonlin (Tate 1977), the SIPE family (Wilkins 1990), O-Plan (Currie and Tate 1991; Tate, Drabble, and Kirby 1994), PRS (Ingrand et al. 1996; Meyers 2016), and UMCP (Erol 1996).

Many HTN planners provide a planner-specific language in which to write the HTN methods. The SHOP planners (Nau et al. 1999, 2003; Goldman and Kuter 2019) make use of Lisp's extensibility to define a Lisp-like language for this purpose. Sohrabi et al. (2009) extended PDDL3 with HTNs to support preferences and converted this extended PDDL3 format for a variant of SHOP2.

In JSHOP2 (Ilghami and Nau 2003), methods are written in the same Lisp-like language, but JSHOP2 compiles them to Java to perform the search. Similarly, the HyperTensioN planner converts a planning model into the Ruby language and was recently extended to support semantic attachments for HTN (Magnaguagno and Meneguzzi 2020).

The totSAT planner (Behnke, Höller, and Biundo 2018) converts totally-ordered HTN planning problems into a SAT formula. PANDA (Höller et al. 2021) is a planner that integrates various approaches to hierarchical planning. Both planners emphasize the importance of a common language for problem definition and propose a Hierarchical Domain Definition Language (HDDL) for it (Höller et al. 2020).

HDDL was also the input language for the recent Hierarchical Track of the International Planning Competition.[6] A full discussion of all planners in that competition is out of scope for a short paper, but we highlight the winners SIADEX (de la Asunción et al. 2005; Castillo et al. 2005) and HyperTensioN (Magnaguagno and Meneguzzi 2020), which both translated HDDL to their specific format.

IxTeT (Ghallab and Laruelle 1994) was a temporal HTN planner which used a specialized language to encode its methods. OpenPRS[7] is a C implementation of PRS. ASPEN (Fukunaga et al. 1997; Chien et al. 2000) is both a planner and framework for planning in space applications; it uses a planner-specific language for encoding plans. FAPE (Dvorák et al. 2014; Bit-Monnot et al. 2020) is a recent planner that supports a subset of the ANML language (Smith, Frank, and Cushing 2008). A more recent HTN planner, SHPE (Menif, Jacopin, and Cazenave 2014), has been specifically developed for AI planning in video games. It uses a simplified variant of ANML (Smith, Frank, and Cushing 2008) to encode problems that are compiled into C++ to perform search. The Adversarial HTN Planner (Ontañón and Buro 2015) allows for HTNs to be used in iterated environments such as Real Time Strategy games; problems are encoded in a language provided by the system. A variety of research has extended this planner (e.g., (Lin et al. 2020; Sun et al. 2017)).

Like Pyhop and GTPyhop, there are several HTN planners in which domains and problems are written in a conventional programming language. The planner by Neufeld et al. (2018) uses C++ for this purpose; its HTN primitives link with Behavior Trees, a common representation for computer game agents. The planner by Soemers and Winands (2016) also uses C++ to represent HTN problems; this planner introduced a mechanism to reuse the existing solution for faster replanning. The UPOM planner introduced in (Patra et al. 2020) uses Python to represent hierarchical operational models.

[6] http://gki.informatik.uni-freiburg.de/competition/

[7] https://git.openrobots.org/projects/openprs

6 Limitations

One limitation involves the goal representation's expressivity. A GTPyhop goal, like a state (see Figure 2), is a set of state-variable bindings. The goal is the conjunction of those bindings, without a way to represent more complicated logical expressions. There probably are some workarounds, but we have not yet considered this. In our work so far, this limitation has not been a major problem.

In many HTN planners, a method or action may contain free variables for which there are several possible instantiations. When the planner creates instances of the method or action, it may backtrack over these instantiations. In contrast, in GTPyhop (like Pyhop) there is no notion of instantiating a method. The method is a piece of Python code that GTPyhop calls directly. The method may contain a variety of local variables, but it is up to the method's author to specify how these variables will acquire their values.

In HTN planners that use planner-specific languages, the methods' preconditions and subtasks are data structures that the planner can reason about before deciding which methods to use in a planning problem. This has enabled several recent advances in HTN-planning search heuristics (Alford et al. 2016a) and other speedup techniques (Behnke, Höller, and Biundo 2018). In contrast, GTPyhop does not know its methods' preconditions and subtasks in advance, because each method is a Python program that *computes* a list of subtasks and subgoals that may depend on the current state (e.g., Figures 4 and 5). A potential way to circumvent this limitation might be to evaluate the method, see what tasks, goals, and actions it returns, and *then* use this information to provide input to a search heuristic—but we have not tried to implement this to see how well it would work. For now, GTPyhop (like Pyhop and SHOP) just does a depth-first search, trying methods in the order that the domain author defined them.

7 Conclusions

Pyhop implemented a version of SHOP-style HTN planning in which methods and actions were written directly in Python. Despite a minimal amount of publicity and no publication, it has been used in several systems that went beyond its original intent of a simple planner for game systems.

GTPyhop extends Pyhop to provide a version of totally-ordered Goal-Task-Network planning without sharing and task insertion. GTPyhop also includes several other features, as described in the introduction. We are working now to extend GTPyhop to incorporate temporal and multi-agent concerns. Section 4 has briefly described the directions that this work is taking.

Acknowledgments. Many thanks to the anonymous reviewers for their insightful remarks. This work has been supported in part by ONR grant N000142012257 and NRL grants N0017320P0399 and N00173191G001. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

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 *ICAPS*.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN problem spaces: Structure, algorithms, termination. In *SOCS*.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016b. Hierarchical planning: relating task and goal decomposition with task sharing. In *IJCAI*, 3022–3028.

Bansod, Y.; Nau, D. S.; Patra, S.; and Roberts, M. 2021. Refinement Acting vs. Simple Execution Guided by Hierarchical Planning. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – totally-ordered hierarchical planning through SAT. In *AAAI*.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: a constraint-based planner for generative and hierarchical temporal planning. In *arXiv:2010.13121*.

Castillo, L.; Fdez-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2005. Temporal enhancements of an HTN planner. In *Conf. Spanish Assoc. for Artificial Intelligence*, 429–438.

Cheng, K.; Wu, L.; Yu, X.; Yin, C.; and Kang, R. 2018. Improving HTN planning performance by the use of domain-independent heuristic search. *Knowledge-Based Systems* 142: 117–126.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - automating space mission operations using automated planning and scheduling. In *SpaceOps 2000*.

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

de la Asunción, M.; Castillo, L.; Fdez-Olivares, J.; García-Pérez, O.; González, A.; and Palao, F. 2005. SIADEX: An interactive knowledge-based planner for decision support in forest fire fighting. *AI Communications* 18(4): 257–268.

Dvorák, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. planning and acting with temporal and hierarchical decomposition models. In *ICTAI*, 115–121.

Erol, K. 1996. *Hierarchical task network planning: formalization, analysis, and implementation*. Ph.D. thesis, University of Maryland.

Fukunaga, A. S.; Rabideau, G.; Chien, S.; and Yan, D. 1997. ASPEN: A Framework for Automated Planning and Scheduling of Spacecraft Control and Operations. In *ISAIRAS*.

Ghallab, M.; and Laruelle, H. 1994. Representation and Control in IxTeT, a Temporal Planner. In *AIPS*, 61–67.

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

Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical task network planning in Common Lisp: the case of SHOP3. In *Proc. European Lisp Symposium*, 73–80.

Gupta, N.; and Nau, D. S. 1992. On the Complexity of Blocks-World Planning. *Artificial Intelligence* 56(2-3): 223–254.

Höller; Behnke; Bercher; and Biundo. 2021. The PANDA framework for hierarchical planning. *KI-Künstliche Intelligenz* 1–6.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: an extension to PDDL for expressing hierarchical planning problems. In *AAAI*.

Ilghami, O.; and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. Technical Report CS-TR-4597, UMIACS-TR-2004-40, University of Maryland.

Ingrand, F. F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*, 43–49. IEEE.

Jacopin, E. 2020. Simple-HTN-Planner. GitHub. URL https://github.com/PCfVW/Simple-HTN-Planner.

Lin, S.; Anshi, Z.; Bo, L.; and Xiaoshi, F. 2020. HTN Guided Adversarial Planning for RTS Games. In *ICMA*, 1326–1331.

Lucas, S. M.; Mateas, M.; Preuss, M.; Spronck, P.; and Togelius, J., eds. 2012. *Artificial and Computational Intelligence in Games*, volume 6. Schloss Dagstuhl.

Magnaguagno, M. C.; and Meneguzzi, F. 2020. Semantic Attachments for HTN Planning. In *AAAI*, 9933–9940.

McGreggor, D. 2014. PyHOP, version 2.0. GitHub. URL https://github.com/oubiwann/pyhop.

Menif, A.; Jacopin, É.; and Cazenave, T. 2014. SHPE: HTN planning for video games. In *Workshop on Computer Games*, 119–132.

Meyers, K. L. 2016. A Procedural Knowledge Approach to Task-level Control. In *AIPS*, 158–165.

Nau, D. S. 2013. Game applications of HTN planning with state variables. In *ICAPS 2013 Workshop on Planning in Games*. URL https://cs.umd.edu/~nau/papers/nau2013game.pdf. Invited talk.

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.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: simple hierarchical ordered planner. In *IJCAI*, 968–973.

Neufeld, X.; Mostaghim, S.; and Brand, S. 2018. A hybrid approach to planning and execution in dynamic environments through HTNs and behavior trees. In *AIIDE*.

Ontañón, S.; and Buro, M. 2015. Adversarial HTN Planning for Complex Real-Time Games. In *IJCAI*, 1652–1658.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. S. 2020. Integrating acting, planning, and learning in hierarchical operational models. In *ICAPS*, 478–487.

Sacerdoti, E. 1975. The Nonlinear Nature of Plans. In *IJCAI*.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*, 981–988.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *ICAPS KEPS Workshop*.

Soemers, D. J. N. J.; and Winands, M. H. M. 2016. HTN Plan Reuse for video games. In *CIG*, 1–8.

Sohrabi, S.; Baier, J. A.; and McIlraith, S. A. 2009. HTN Planning with Preferences. In *IJCAI*, 1790–1797.

Sun, L.; Jiao, P.; Xu, K.; Yin, Q.; and Zha, Y. 2017. Modified Adversarial HTN Planning in RTS Games. *Applied Sciences* 7(9).

Tate, A. 1977. Generating project networks. In *IJCAI*, 888–893.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: an open architecture for command, planning and control. In Zweben, M.; and Fox, M. S., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational intelligence* 6(4): 232–246.

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

Integrating Planning and Acting With a Re-Entrant HTN Planner

Yash Bansod[1], Dana Nau[1,2], Sunandita Patra[2], Mak Roberts[3]
[1]Institute for Systems Research and [2]Dept. of Computer Science, Univ. of Maryland, College Park, MD, USA
[3]The U.S. Naval Research Laboratory, Code 5514, Washington, DC, USA
{yashb, nau, patras}@umd.edu, {mak.roberts}@nrl.navy.mil

Abstract

A major problem with integrating HTN planning and acting is that, unless the HTN methods are very carefully written, unexpected problems can occur when attempting to replan if execution errors or other unexpected conditions occur during acting. To overcome this problem, we present a re-entrant HTN planning algorithm that can be restarted for replanning purposes at the point where an execution error occurred, and an HTN acting algorithm that can restart the HTN planner at this point. We show through experiments that our algorithm is an improvement over a widely used approach to planning and control.

1 Introduction

HTN planners use *descriptive models* of actions tailored to compute the next states in a state transition system efficiently. In most cases[1] they assume a world that is closed, static, and deterministic. However, executing the plan in open, dynamic, and nondeterministic environments, characteristic of many practical problems, generally leads to failure. The planning domain will rarely be an entirely accurate model of the actor's environment, and execution of the plan may fail due to (i) failure in execution of actions, (ii) occurrence of unexpected events, (iii) because the planning was solved with incorrect or partial information.

Plans are needed for deliberative acting, but are not sufficient for it (Pollack and Horty 1999). Many deliberative acting approaches seek to combine the descriptive models used by the planner with the *operational models* used by the actor (Ingrand and Ghallab 2017). In contrast, others seek to directly integrate planning and acting using operational representations (Patra et al. 2019, 2020).

An early version of HTN planning was the Simple Hierarchical Ordered Planner (SHOP) (Nau et al. 1999), and its successors SHOP2 and SHOP3 (Nau et al. 2003; Goldman and Kuter 2019). SHOP and its successors are written in the LISP programming language, which limits its adoption.

Python is a much more widely adopted programming language used by roboticists, game developers, machine learning engineers, and AI engineers. The Pyhop planner (Nau 2013a,b) adapts the SHOP planning algorithm so that methods and actions are written directly in Python. GTPyhop (Nau et al. 2021), a recent extension to Pyhop, combines both HTN planning and hierarchical goal-network (HGN) planning (Shivashankar et al. 2012).

One difficulty with integrating acting with HTN planning is responding to action failures at execution time. If one tries to replan by calling the HTN planner with the new current state but the same task as before, unfortunate results can occur (Section 5.1). In this paper we describe a way to overcome that problem. Our primary contributions are:

1. We describe IPyHOP, a planner that can respond to plan-execution failures by resuming the planning process at the point where the failure occurred. IPyHOP's planning algorithm is based on GTPyhop (Nau et al. 2021), but with the following key changes: it uses iteration rather than recursion, and it preserves the hierarchy in the planning solution and returns a solution task network rather than a simple plan. Thus, if an unexpected problem occurs during plan execution, the actor can call IPyHOP with a pointer to the point in the task network where the execution failure occurred, and IPyHOP can resume planning from that point onward.
2. Inspired by the RAE actor in (Ghallab, Nau, and Traverso 2016, Chapter 3), we provide a new acting algorithm, Run-Lazy-Refineahead, that integrates efficiently with IPyHOP. Run-Lazy-Refineahead calls IPyHOP to get a solution task network and executes the actions in the task network by sending them to its execution platform. If an execution failure occurs, it gives IPyHOP a pointer to where the failure occurred and requests replanning.

After discussing related work in Section 2, Section 3 explains our notation and briefly provides background on Pyhop and GTPyhop. Section 4 explains the HTN planning in IPyHOP. Section 5 explains the Run-Lazy-Lookahead and Run-Lazy-Refineahead algorithms and how IPyHOP can be used in integrated HTN planning and acting or deliberative HTN acting. Sections 6 and 7 describe an experimental domain for HTN planning and experiments that compare Run-Lazy-Lookahead and Run-Lazy-Refineahead algorithms for deliberative HTN acting. Finally, Section 8 summarizes our work and discusses limitations and some avenues for future research.

[1]There are some exceptions, e.g., (Kuter and Nau 2005; Hogg, Kuter, and Muñoz-Avila 2009; Chen and Bercher 2021).

2 Related Work

AI planning. HTN planning is a widely adopted approach to AI planning in the gaming industry (Neufeld et al. 2017). One of the first HTN planners was Nets of Action Hierarchies (NOAH) (Sacerdoti 1975). Since then, numerous HTN planners have been developed. Some of the best-known ones are Nonlin (Tate 1977), System for Interactive Planning and Execution (SIPE) and SIPE-2 (Wilkins 1990), Open Planning Architecture (O-Plan) (Currie and Tate 1991) and its successor O-Plan2 (Tate, Drabble, and Kirby 1994), Universal Method Composition Planner (UMCP) (Erol 1996), SHOP (Nau et al. 1999) and its successor SHOP2, and SHOP3 (Nau et al. 2003; Goldman and Kuter 2019), and SIADEX (Castillo et al. 2005). Additionally, there are various HTN planners like Simple Hierarchical Planning Engine (SHPE) (Menif, Jacopin, and Cazenave 2014) that are specifically developed for AI planning in video games.

A wide body of literature also exists on Monte Carlo tree search based planning in games. Monte Carlo tree search refers to simulated execution (Feldman and Domshlak 2013, 2014), sampling outcomes of action models (Yoon, Fern, and Givan 2007; Teichteil-Koenigsbuch, Infantes, and Kuter 2008), and hindsight optimization (Yoon et al. 2008).

Planning and acting. Musliner et al (2008) propose a way to do online planning and acting. The old plan is executed repeatedly in a loop while the planner synthesizes a new plan (which the authors say can take a significant amount of time), and the new plan is not installed until planning has been finished. This way of repeated planning and acting is similar to the Run-Concurrent-Lookahead algorithm defined in (Ghallab, Nau, and Traverso 2016, Chapter 2).

Other similar algorithms, e.g., Run-Lookahead and Run-Lazy-Lookahead, are also defined in (Ghallab, Nau, and Traverso 2016, Chapter 2). Here Lookahead is any *online* planning algorithm. Each time Run-Lookahead calls the Lookahead planner, it performs only the first action of the plan that Lookahead returned. This way of execution is effective, for example, in unpredictable or dynamic environments in which some of the states are likely to be different from what the planner predicted. In contrast, Run-Lazy-Lookahead executes each plan as far as possible, calling Lookahead again only when the plan ends, or a plan simulator says that the plan will no longer work properly.

BDI Architectures. BDI (Belief-Desire-Intention) architectures (De Silva and Padgham 2005; Bauters et al. 2014; Yao et al. 2021; Sardina, De Silva, and Padgham 2006) have some similarity to our work, but BDI systems are mostly reactive. They differ from us with respect to their primitives as well as their methods or plan-rules. In general, BDI systems will not replan, but they will select and execute an untried method when failure occurs. Some BDI approaches, e.g., (Yao et al. 2021) can also replan, but their agent model is non-hierarchical. (Clement, Durfee, and Barrett 2007) integrates BDI architecture with hierarchical agent models for temporal planning and coordination in multi-agent systems.

3 Background: Pyhop and GTPyhop

In this paper an HTN *planning domain* is defined as a pair $\Sigma = (O, M)$, and an HTN *planning problem* is defined as a 4-tuple $\mathcal{P} = (s_0, w, O, M)$, where s_0 is the initial state, w is the inital task network, O is a set of operators, and M is a set of HTN methods. For details, see (Bansod 2021). We assume that primitive tasks can be directly executed by the execution engine but non-primitive tasks need refinement before execution. We also assume that the planning domain is deterministic and fully observable, but the execution environment is nondeterministic—hence a solution plan returned by the planner might not always work correctly at execution time.

GTPyhop (Nau et al. 2021) is a domain-independent Goal Task Network (GTN) planning system written in Python. GTPyhop is a progressive totally-ordered GTN planner i.e. it plans for a sequence of tasks and goals in the same order that they will later be executed. This behavior helps avoid some of the goal-interaction issues that arise in other HTN planners, making the planning algorithm relatively simple. The planning algorithm is sound and complete over a large class of problems. Since GTPyhop knows the complete world-state at each step of the planning process, it can use highly expressive domain representations.

GTPyhop uses recursion for task and goal refinement. Writing the algorithm as a recursive algorithm is intuitive, and it follows the description of HTN planning algorithm in many texts. The algorithm is also simple to implement, and the recursion stack efficiently handles the refinement and backtracking during task planning.

Like most HTN planners, GTPyhop has two limitations that limit its ability to do effective replanning. First, the use of recursion prevents the code from being re-entrant. If it is necessary to replan because of an action failure, the only alternative is to call GTPyhop again, which can lead to incorrect results (see Section 5.1). Second, GTPyhop returns a plan π, but not the refinement tree that produced π. In order to replan if an action failure occurs, it is necessary for a planner to know not only the action that failed but what tasks it was trying to achieve at that point in the planning process. That requires a copy of the refinement tree.

4 HTN Planning in IPyHOP

IPyHOP overcomes the limitations of GTPyhop in two ways. First, it uses an iterative tree traversal procedure for task refinement, with the refinement and backtracking done by tree traversal algorithms. This supports considerably more control over how the algorithm refines tasks. Second, it accepts a (partial) task tree and returns the entire solution task network. This change supports adding hierarchical knowledge for the replanning process. IPyHOP implements GTN planning like GTPyhop. However, in this paper, we discuss only the HTN planning functionality of IPyHOP.

Let u represent a grounded task node. Then, *task*(u) defines the grounded task $t = t(r_1, ..., r_k)$ corresponding to u. *refined*$(u) \in$ {*true, false*} represents if the node has been refined. *operator*(u) represents the operator $o \in O$ that is relevant to task t if the task was primitive. *visited*$(u) \in$ {*true, false*} represents if the node has been visited. *state*(u)

Algorithm 1: HTN Planning in IPyHOP.

```
1  IPyHOP(s, w, O, M):
2  p ← root(w)
3  while true do
4      u ← first_unrefined_bfs_successor(w, p)
5      if u = ∅ then
6          if p = root(w) then
7              break
8          else
9              p ← parent(p)
10             continue
11     t ← task(u)
12     if t is primitive then
13         o ← operator(u)        \\ here o ∈ O
14         s' ← o(s, r_1, ..., r_k)
15         if s' is valid then
16             s ← s'
17             refined(u) ← true
18         else
19             w, u ← backtrack(w, u)
20             p ← parent(u)
21     if t is non-primitive then
22         if visited(u) then
23             s ← state(u)
24         else
25             visited(u) ← true
26             state(u) ← s
27         foreach m ∈ methods(u) where m ∈ M do
28             t' ← m(s, r_1, ..., r_k)
29             methods(u) ← methods(u)\m
30             if t' is valid then
31                 refined(u) ← true
32                 add_nodes(u, t')
33                 p ← u
34                 break;
35         if not refined(u) then
36             w, u ← backtrack(w, u)
37             p ← parent(u)
38 return w
```

represents the state when the node was first visited. And *methods*(u) represents the methods applicable to the task t that haven't been used for refinement of u, given that the task is non-primitive.

Algorithm 1 is IPyHOP's HTN planning algorithm. The first_unrefined_bfs_successor(w, p) returns the first unrefined node found during a breadth first search in w, starting from node p. In IPyHOP, backtrack(w, u) is a subroutine (see Algorithm 2) that modifies the task network given that refinement of node u failed. W_p is a list of nodes in a depth first search pre-ordering in w, starting from node p. After backtracking, the non-primitive task node u', the node refined before the current task node u, is again marked for refinement. The add_nodes(u, t') subroutine adds the sub-tasks t' as nodes to the refined node u.

Algorithm 2: IPyHOP Backtracking.

```
1  backtrack(w, u):
2  p ← parent(u)
3  W_p ← dfs_preorder_nodes(w, p)
4  foreach v ∈ reversed(W_p) do
5      refined(v) ← false
6      if v is non-primitive then
7          W_v ← descendants(v)
8          w ← w\W_v
9          return w, v
10 w ← {root(w)}
11 return w, root(w)
```

5 Integrating IPyHOP with an Actor

As explained in Section 2, a popular way of integrating a planner and an actor is by using algorithms like Run-Lazy-Lookahead. In Section 5.1 we describe the Run-Lazy-Lookahead algorithm and some of its features. We explain its use in deliberative HTN acting and point to some of its limitations. In Section 5.2 we describe the Run-Lazy-Refineahead algorithm for deliberative HTN acting.

5.1 Where Run-Lazy-Lookahead Fails

The Run-Lazy-Lookahead algorithm, previously introduced in Ghallab, Nau, & Traverso (Chapter 2 2016) is a deliberative acting algorithm. It executes each plan π as far as possible, calling Lookahead again only when π ends or a plan simulator says that π will no longer work properly. This way of execution can help in environments where it is computationally expensive to call Lookahead, and the actions in π are likely to produce the predicted outcomes. It can also use a plan simulator, which may use the planner's prediction function γ or may do a more detailed computation (e.g., a physics-based simulation, a Monte-Carlo simulation, et cetera.) that would be too time-consuming for the planner to use. The simulator should return failure if its simulation indicates that π will not work correctly. For example, if it finds that an action in π will have an unsatisfied precondition, or if the simulation indicates that the π will not achieve the goal g when it is supposed to.

We can use IPyHOP as the Lookahead planner in Run-Lazy-Lookahead to integrate HTN planning and acting. However, this repeated planning and acting procedure does not work well with HTN planners. The problem can be visualized with the following abstract example.

Example 1. Suppose we want to plan for a task network consisting of two tasks $t1$ and $t2$. Let there be two methods $m1\_t1$ and $m2\_t1$ that are applicable to $t1$. And two methods $m1\_t2$ and $m2\_t2$ that are applicable to $t2$. Let primitive tasks be represented in syntax $o\langle i \rangle$, ex. $o1, o2$ et cetera. Let $m1\_t1$ refine $t1$ into $o1$ and $o2$. Let $m2\_t1$ refine $t1$ into $o3$, $o4$, and $o5$. Let $m1\_t2$ refine $t2$ into $o4$, $o5$ and $o6$. And let $m2\_t2$ refine $t2$ into $o7$ and $o8$. Also, for the sake of this example assume that all tasks, methods, and the operators defined here are grounded. These individual refinements can be visualized in Figure 1. We will assume that all the methods and operators have no pre-conditions and all are applicable

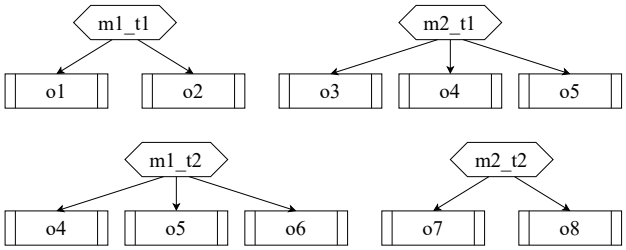


Figure 1: Refinement of task $t1$ using $m1\_t1$ and $m2\_t1$. And refinement of task $t2$ using $m1\_t2$ and $m2\_t2$. (Example 1).

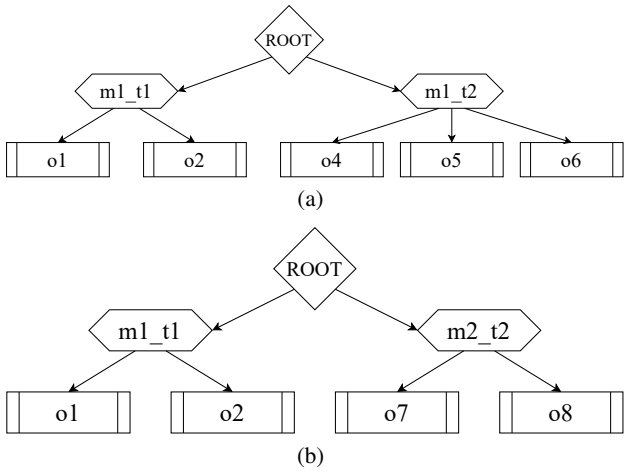


Figure 2: Task network visualizations: (a) After first planning attempt. (b) Re-planning after failure in execution of o6.

anytime in the planning process. Also, assume that the HTN planner always prioritizes refinement of tasks using the first method over second.

The solution tree that IPyHOP will return is visualized by Figure 2(a). This solution implies that the plan represented in the form of a primitive task sequence will be $\pi = \langle o1, o2, o4, o5, o6 \rangle$. The primitive task sequence is found by performing a Depth First Search (DFS) tree traversal on the solution tree. Let us assume that while executing this plan, $o6$ nondeterministically fails. We update our model of $o6 \in O$ (if required) used by the planner and perform re-planning again. The new solution tree that IPyHOP will return is visualized by Figure 2(b). The actor will now execute the plan $\pi = \langle o1, o2, o7, o8 \rangle$. This means that the action sequence executed by our actor is $\alpha = \langle o1, o2, o4, o5, o6, o1, o2, o7, o8 \rangle$, when in fact it should have been $\alpha = \langle o1, o2, o4, o5, o6, o7, o8 \rangle$ for the given scenario. This action sequence was executed because we did re-planning for the completed task $t1$ along with the failed task $t2$. ■

Technically, it is possible to prevent degenerate executions like in Example 1 from happening by cleverly designing methods that consider failures or having some flags in the state that gets modified. However, as the complexity of the task network increases, this approach quickly becomes infeasable. One of the most significant limitations of HTN planning is the substantial domain engineering effort required in writing HTN methods. Domain authoring is especially hard because the HTN formalism requires users to provide methods to cover every possible scenario that the agent could encounter. If the HTN planner finds itself in a situation the user had not anticipated, it will behave unexpectedly or fail without returning a solution. Moreover, there are many scenarios where it is impossible to account for such occurrences while authoring the domain.

5.2 Run-Lazy-Refineahead

The problems with Run-Lazy-Lookahead occur due to incompatibilities between the definition of the Lookahead planner and the definition of an HTN planner. The signature of a Lookahead planner is (Σ, s, g), whereas the signature of HTN planners is (s, w, O, M). However, the goal g and task network w are notably different. The goal for a planner might stay unchanged as the plan is executed. However, the task network is constantly modified. Replacing the Lookahead planner with IPyHOP leads to repeated planning for some of the completed tasks from the original task network w in a new state s'.

By visualizing the planning problem as a graph, however, the solution seems apparent. We compute the modified task network based on the location of the failure in the task network. Then modify the task network again using the backtrack feature of the planner. And then resume the planning process. During re-planning, the planner marks the nodes that were refined because of this re-planning process.

The task network described in Example 1 is simplistic, and finding the modified task network is trivial. We compute the parent task node of the failed primitive task node and only re-plan for the computed task node. However, for a more complicated task network, this will not work. We will have to come up with a more sophisticated algorithm. Let us understand this with another example.

Example 2. We want to plan for a task network with tasks $t1$, $t2$, and $t3$. Let us assume that the planner generated the solution task network represented in Figure 3(a). We start implementing the primitive tasks in this solution tree as encountered in a DFS tree traversal from the root node. The primitive task sequence or the plan is $\pi = \langle o1, o2, ..., o11, o12 \rangle$. However, while executing this plan, assume that $o7$ nondeterministically fails. We need to find the new task network our planner should use for re-planning. Unlike the previous example, replanning just for the parent task node $t4$ of the failed primitive task node $o7$ is incorrect because the failure in executing $o7$ means that $o11$'s preconditions will not be satisfied later in the plan. Thus, additional replanning will be needed in order to prevent the entire plan from failing.

In the above explained scenario, we should modify the solution task network by removing refinements of all the tasks that come after the failed node $o7$ in the Pre-ordered DFS traversal. Alternatively, this could be done more efficiently

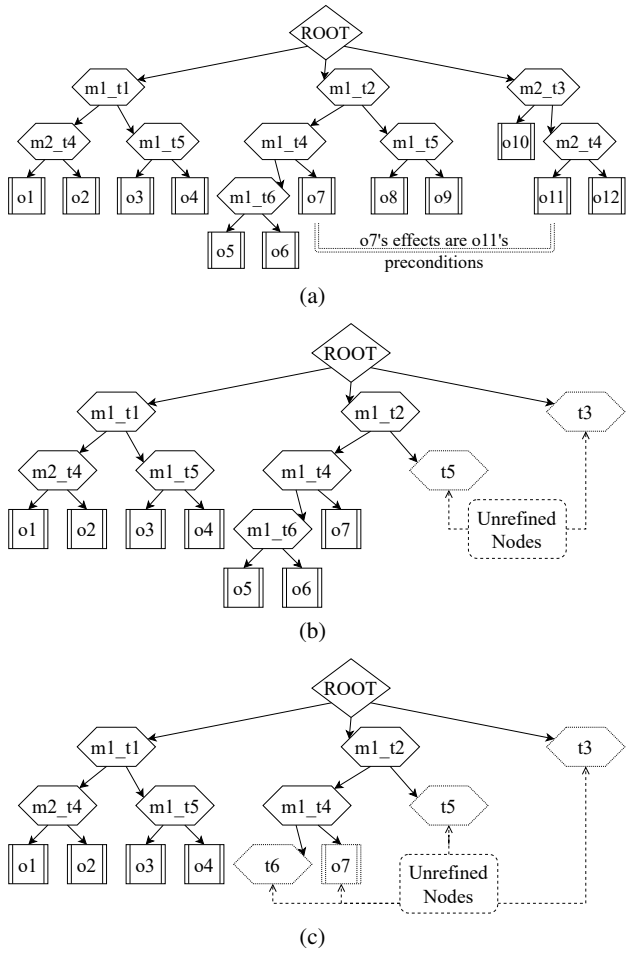


Figure 3: (a) Solution task network after initial planning. (b) Modified task network after failure in execution of $o7$. (c) Modified task network after backtracking from $o7$ on the modified task network in (b). (Example2).

using the Un-Refine-Post algorithm (Algorithm 3). At this point, the modified task network should look like Figure 3(b). Now, we again modify this task network by backtracking on the failed node $o7$. At this point, the modified task network should look like Figure 3(c). We update our model of $o7 \in O$ (if required) used by the planner and perform re-planning again. The planner marks the nodes it refines in this re-planning problem and returns another solution task network for us to execute.

Note that during execution, we only execute the primitive tasks that the planner marked during re-planning. We compute the marked primitive tasks in this solution tree by performing a DFS tree traversal from the root node. ■

This way of repeated planning and acting leads to the formulation of Algorithm 4, Run-Lazy-Refineahead.

Run-Lazy-Refineahead is a repeated planning and acting algorithm for integrating HTN planning and acting. Here, Refineahead is any online HTN planner that provides the solution as a refined task network and provides control over its backtracking feature.

Algorithm 3: Un-Refine-Post. Algorithm used to modify a task network w after failure at u.

```
Un-Refine-Post(w, u):
while true do
    p ← parent(u)
    foreach v ∈ BFS_Successors(p) s.t. v after u do
        refined(v) ← false
        if v is non-primitive then
            W_v ← descendants(v)
            w ← w\W_v
    u ← p
    if u = root(w) then
        break
return w
```

Algorithm 4: Run-Lazy-Refineahead.

```
Run-Lazy-Refineahead(Σ, w):
s ← abstraction of observed state ξ
while true do
    w ← Refineahead(Σ, s, w)
    if w = failure then
        return failure
    π ← marked primitive tasks in DFS(w)
    a ← first action in π
    while π ≠ ⟨⟩ and Simulate(Σ, s, π) ≠ failure do
        a ← pop-first-action(π)
        perform(a)
        s ← abstration of observed state ξ
    if π ≠ ⟨⟩ then
        w ← Un-Refine-Post(w, a)
        w, a ← Backtrack(w, a)
    else
        break
```

Run-Lazy-Refineahead executes each plan π as far as possible, calling Refineahead again only when π ends or a plan simulator says that π will no longer work properly. This way of execution can help in environments where it is computationally expensive to call Refineahead, and the actions in π are likely to produce the predicted outcomes. Simulate is the plan simulator, which may use the planner's prediction function γ or may do a more detailed computation (e.g., a physics-based simulation, a Monte-Carlo simulation, et cetera.) that would be too time-consuming for the planner to use. Simulate should return failure if its simulation indicates that π will not work correctly. For example, if it finds that an action in π will have an unsatisfied precondition.

On failure in executing the plan, the tasks refined after the failed task a in the task network w are un-refined using the Un-Refine-Post (Algorithm 3), and backtracking is performed using the Backtrack algorithm of an HTN planner, e.g., Algorithm 2. The resulting task network obtained

after these modifications is re-used for the next re-planning process.

Intuitively, deliberative HTN acting implemented in Run-Lazy-Refineahead is more efficient than in Run-Lazy-Lookahead. Since for every re-planning, the Refineahead needs to re-plan only for a subset of the task network, compared to the entire task network for Lookahead, the planning time on average will be lower. Also, since the actions corresponding to the tasks that have already been executed are no longer planned for during re-planning, repetition of already executed tasks will be minimized. Thus, Run-Lazy-Refineahead will lead to executing action sequences with an overall cost less than that by Run-Lazy-Lookahead.

6 Experimental Setup

We used the Robosub Domain (citation removed for blind reviewing) for our experimental evaluation. The Robosub Domain was derived from the RoboSub 2019 competition,[2] where an autonomous underwater vehicle performs various compulsory and optional tasks autonomously to score points in the competition. A planning domain was written for the refinement of these tasks. The planning domain consisted of seventeen primitive task operators and twenty-one task refinement methods for refining ten non-primitive tasks.

We statistically analyzed the performance of Run-Lazy-Lookahead and Run-Lazy-Refineahead approaches for deliberative HTN acting for the above-defined tasks. For the RoboSub competition, we fixed the initial location of the robot and a few other constraints. However, we varied the location of various objects in the planning problem. The initial task network always contains a single task named *competition-task* that needs to be refined to complete all the required tasks based on the competition deliverable.

The planning problem is solved using the IPyHOP planner, and the resulting plan is executed by a simple actor communicating with an execution platform. The execution environment is nondeterministic, which leads to occasional failures in the execution of actions. The repeated planning and acting is done using Run-Lazy-Lookahead and Run-Lazy-Refineahead algorithms. The complete refinement and execution for one such planning problem is termed as a *test case* x, where x_i corresponds to the i^{th} planning problem with the initial state s_i, where s_i is the i^{th} state in I. We repeat this deliberative HTN acting process for all initial states. The execution of all the test cases x_i is known as an *experiment*, e, where $e = \langle x_1, x_2, ..., x_j \rangle$, where $j = \|I\|$. We repeat the experiment 11 times, i.e. $E = \langle e_1, e_2, ..., e_{11} \rangle$.

We evaluate performance with three metrics:

- **Total iterations taken**: This metric calculates the total number of iterations taken by the planner for a given test case. Calculating iterations provides a good estimate of the planner's total planning time for a test case.
- **Total action cost**: This measures the total cost of an action sequence for a given test case. Execution of smaller action sequences will generally lead to lower total action costs.
- **Final state reward**: This measures the reward obtained based on the final state of the robot in a test case. This is a good indicator of how well the competition task was completed.

The raw data collected d_{raw} in each experiment $e \in E$ described earlier was accumulated into a single dataset D_{raw}, where $D_{raw} = \langle d_1, d_2, ..., d_{11} \rangle$. D_{raw} was post-processed to calculate the required metrics and the results were stored in a single numpy array representing the results dataset $D_{results}$. Let the size of the dataset $D_{results}$ be $[\|e\| \times \|a\| \times \|x\| \times \|m\|]$. Here $\|e\| = 11$ is the number of experiments performed, $\|a\| = 2$ is the number of deliberative HTN acting algorithms being compared, $\|x\| = 10000$ is the number of test cases solved, and $\|m\| = 3$ is the number of metrics evaluated.

The $D_{results}$ dataset was processed further by doing a reduce mean operation across the $zero^{th}$ axis of the dataset. Thus the dataset D_{exp_mean} of size $[\|a\| \times \|x\| \times \|m\|]$ was generated. Each element in the dataset D_{exp_mean} represents the mean value of a metric for a given test case across experiments. Since the value of a metric for a given test case varies across experiments due to the non-determinism of the execution environment, taking the mean across experiments gives us a more reliable estimate of that metric for a given test case. The metrics calculated in the dataset D_{exp_mean} are illustrated in Figures 4(a), 4(b), and 4(c).

It is possible that different numbers of failures occur with each test case and a more fair assessment would compare only situations with the same number of failures. Another form of post-processing was done on D_{raw} to generate the D_{eqv} dataset to balance this concern and these results are presented in Figures 4(d), 4(e), and 4(f). The values represented by the D_{eqv} dataset are less accurate since they only use a single data point for a metric of a given test case. Comparatively, the metric measurements from D_{exp_mean} are computed by performing a mean operation across 11 values for each metric in a test case. To improve the accuracy of the metric measurements by D_{eqv} dataset, we will need to perform more experiments such that multiple data points are available for each metric in the dataset.

7 Results and Discussion

Our results suggest that Run-Lazy-Refineahead is a better algorithm for deliberative HTN acting compared to Run-Lazy-Lookahead. In Figure 4(a), we show the values of the metric - the total number of iterations taken by the planner, for Run-Lazy-Lookahead (blue histogram) and Run-Lazy-Refineahead (purple histogram). The relation of this metric for the two deliberative acting algorithms is visualized in Figure 4(d) as a scatter plot. Based on our results, we can state that the Run-Lazy-Refineahead leads to the generation of shorter and easily solvable re-planning problems. Also, we can see in Table 2 that the average time spent in planning during Run-Lazy-Refineahead is $\approx 80\%$ of the average time spent in planning during Run-Lazy-Lookahead.

Figure 4(b) shows the values of the total-action-cost metric, for Run-Lazy-Lookahead and Run-Lazy-Refineahead. The relation of this metric for the two deliberative acting al-

[2] https://robosub.org/programs/2019/

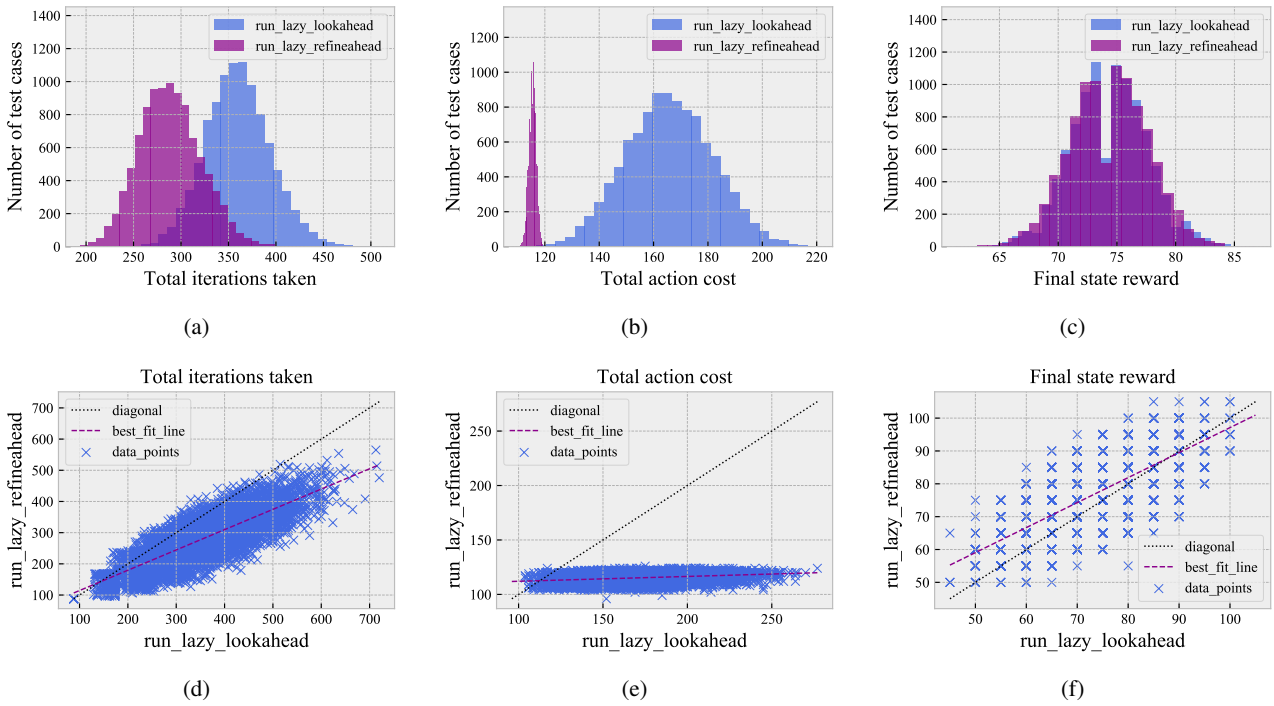


Figure 4: Results of three metrics: total iterations (left), action cost (middle), and final state reward (right). Each pair shows the distribution visualized using histograms (top, using D_{exp_mean}) and the relation visualized by fitting a line on the scatter plot (bottom, using D_{eqv}).

Metric	Run-Lazy-Lookahead		Run-Lazy-Refineahead	
	Mean	SD	Mean	SD
Total iterations taken	364.470	34.178	290.592	32.639
Total action cost	165.928	16.002	115.478	1.339
Final State Reward	74.368	3.183	74.326	3.242

Table 1: Overview of results obtained using D_{mean_exp}

Metric	Refineahead / Lookahead Mean	Best-fit line	
		Slope	Y-intercept
Total iterations taken	0.796	0.639	58.296
Total action cost	0.682	0.044	108.282
Final State Reward	1.049	0.805	14.540

Table 2: Overview of results obtained using D_{eqv}

gorithms is visualized in Figure 4(e). Thus, we can state that the Run-Lazy-Refineahead leads to the execution of smaller action sequences. In Table 2 we can see that the average cost of executing action sequences generated from Run-Lazy-Refineahead is $\approx 70\%$ of the average cost of executing action sequences generated from Run-Lazy-Lookahead.

Figure 4(c) shows show the values of the final-state-reward metric, for Run-Lazy-Lookahead and Run-Lazy-Refineahead. The relation of this metric for the two deliberative acting algorithms is shown in Figure 4(f). The results show that the improvements mentioned earlier were realized without sacrificing the average final state reward.

There is also a hidden burden associated with using the Run-Lazy-Lookahead algorithm not portrayed by our experiments. Authoring the domain for use in the Run-Lazy-Lookahead algorithm requires accounting for numerous scenarios where failures would lead to repeated tasks, getting stuck in infinite task loops, getting stuck in non-recoverable states, et cetera. These problems can be addressed by clever definitions of task methods and flags in the state. However, it might not be possible to eliminate these undesirable behaviors. In more modest domain model definitions like ours, this problem is not as pronounced. However, as the domain models get more and more comprehensive, this problem quickly worsens. In Run-Lazy-Refineahead, however, the planner always resumes after backtracking on the node that caused the failure. Thus, repetition of tasks and other unexpected behaviors are minimized.

For our experiments, every effort was made to make deliberative HTN acting using Run-Lazy-Lookahead as efficient as possible. Optimizing the performance of the Run-Lazy-Lookahead algorithm was our prime focus. The task methods, operators, and state definition were designed primarily for use in the Run-Lazy-Lookahead algorithm. Then the same domain model definition and state definition were used for the Run-Lazy-Refineahead algorithm. This reuse of domain definition leads to the planner performing many unnecessary constraint checks during task refinement required for Run-Lazy-Lookahead but are not required for Run-Lazy-Refineahead. The domain authoring for use in

Run-Lazy-Refineahead is much more straightforward and concise. If the domain model definition was primarily designed for Run-Lazy-Refineahead, the results would considerably shift in its favor. The metrics would remain the same for Run-Lazy-Refineahead but significantly worsen for the Run-Lazy-Lookahead. However, even though the calculated metrics would remain the same, the second execution would be computationally faster than the first since simpler domain model definitions are being used for task refinement process.

Hence we can comfortably state that Run-Lazy-Refineahead is a better alternative to Run-Lazy-Lookahead for deliberative HTN acting.

8 Summary and Future work

In this paper we have presented new algorithms for integrated HTN planning and acting.

The first main contribution is an HTN planner, IPyHOP. IPyHOP is an iterative tree traversal-based HTN planning algorithm written in Python that provides extensive control over its task network refinement. Since the algorithm is iteration-based, the task network refinement can be paused, modified, and resumed at the user's discretion. This level of control makes it a great choice for planning in scenarios where re-planning is required. Since IPyHOP uses the Python programming language, authoring domain model definitions does not require developers to learn specialized programming languages. Instead, developers can write the task methods as Python functions. Also, since it follows an object-oriented design, it is effortless to integrate and debug it with other computer programs. IPyHOP is envisioned to make HTN planning accessible to a much broader audience who were earlier reluctant to adopt it for their planning problems due to a lack of HTN planners in Python.

The second main contribution is a deliberative HTN actor, Run-Lazy-Refineahead. Run-Lazy-Refineahead is a repeated planning and acting algorithm specially designed for deliberative HTN acting. We showed experimentally that it performs better for deliberative HTN acting than Run-Lazy-Lookahead, a popular acting-and-planning algorithm. Run-Lazy-Refineahead uses the hierarchical nature of the refined task network generated by HTN planners like IPyHOP to develop smaller and smaller task refinement problems as the execution proceeds. The improvement can be beneficial in deliberative HTN acting in fast-moving dynamic worlds like in games or in robotics scenarios.

We hope that the large community of roboticists and game developers who program their systems in Python adopt IPyHOP, and Run-Lazy-Refineahead for HTN planning, and integrated planning and acting.

8.1 Limitations and Future Work

In some aspects, HTN planning is quite controversial. The controversy lies in its requirement for well-conceived and well-structured domain knowledge. Such knowledge is likely to contain rich information and guidance on how to solve a planning problem, thus encoding more of the solution than was envisioned for classical planning systems. This structured and rich knowledge gives a primary advantage to HTN planners in terms of speed and scalability when applied to real-world problems compared to their counterparts in the classical planning world. However, this also makes their performance depend on the users' definition of suitable domain-specific task methods.

IPyHOP faces many of the same challenges as other HTN planners, namely:

- Domain engineering effort in writing methods: The HTN formalism requires implementing methods to cover every possible scenario that the agent could encounter. An HTN planner trying to plan for an unanticipated state may fail without returning a solution.
- Brittleness in open and dynamic environments: The previous problem is intensified in open, dynamic environments. Nondeterministic events or outcomes can result in unanticipated situations, and HTN planners are not well suited to work in open and dynamic environments.
- Effective domain-independent HTN planning heuristics: Heuristics are crucial in guiding an algorithm toward high-quality solutions. HTN planners often rely heavily on the user-provided knowledge through the definition of methods in providing the necessary guidance.

These limitations are important areas for future research on improving IPyHOP.

In some aspects, the integration of HTN planning and acting using Run-Lazy-Refineahead that we proposed here can be interpreted as a simple HTN planner *guided* acting. Some algorithms directly integrate a planner's descriptive model into a hierarchical actor to select refinement methods, while others directly integrate planners that plan using operational representations with the actor RAE e.g., (Patra et al. 2019, 2020). Combining a hierarchical planner and an actor using this strategy leads to much more efficient and tighter integration. We believe a similar form of integration is also possible for HTN planners and HTN actors. An HTN planner like IPyHOP could be directly integrated with an HTN actor like RAE-lite, where the HTN actor would decide on the method it uses for task refinement based on recommendation of the HTN planner.

For hierarchical acting and planning, there are two main ways to represent an objective: tasks and goals. A task is an activity to be accomplished by an actor, while a goal is a final state that should be reached. Depending on a domain's properties and requirements, users can choose between task-based and goal-based approaches. Since IPyHOP is based on GTPyhop (Nau et al. 2021), it supports both HTN and HGN planning. However, we have not made any use of HGN planning in this paper. For future work, we intend to do experimental evaluations of Run-Lazy-Refineahead versus Run-Lazy-Lookahead on HGN versions of our test domains.

9 Acknowledgments

This work has been supported in part by ONR grant N000142012257 and NRL grants N0017320P0399 and N00173191G001. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

Bansod, Y. 2021. *Refinement Acting vs. Simple Execution Guided by Hierarchical Planning*. Master's thesis, University of Maryland. URL https://www.cs.umd.edu/users/nau/others-papers/bansod2021refinement.pdf.

Bauters, K.; Liu, W.; Hong, J.; Sierra, C.; and Godo, L. 2014. Can(Plan)+: Extending the operational semantics of the BDI architecture to deal with uncertain information. In *UAI*.

Castillo, L.; Fdez-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2005. Temporal enhancements of an HTN planner. In *Conf. Spanish Assoc. for Artificial Intelligence*, 429–438.

Chen, D.; and Bercher, P. 2021. Fully observable nondeterministic HTN planning – formalisation and complexity results. In *ICAPS*, volume 31, 74–84.

Clement, B. J.; Durfee, E. H.; and Barrett, A. C. 2007. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research* 28: 453–515.

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

De Silva, L.; and Padgham, L. 2005. Planning on demand in BDI systems. In *ICAPS (Poster)*.

Erol, K. 1996. *Hierarchical task network planning: formalization, analysis, and implementation*. Ph.D. thesis, University of Maryland.

Feldman, Z.; and Domshlak, C. 2013. Monte-Carlo planning: Theoretically fast convergence meets practical efficiency. *arXiv preprint arXiv:1309.6828* .

Feldman, Z.; and Domshlak, C. 2014. Monte-Carlo tree search: To MC or to DP? In *ECAI*, 321–326.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical task network planning in Common Lisp: the case of SHOP3. In *Proc. European Lisp Symposium*, 73–80.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *IJCAI*, 1708–1714.

Ingrand, F.; and Ghallab, M. 2017. Deliberation for autonomous robots: a survey. *Artificial Intelligence* 247: 10–44.

Kuter, U.; and Nau, D. S. 2005. Using domain-configurable search control for probabilistic planning. In *AAAI*, 1169–1174.

Menif, A.; Jacopin, É.; and Cazenave, T. 2014. SHPE: HTN planning for video games. In *Workshop on Computer Games*, 119–132. Springer.

Musliner, D.; Pelican, M. J.; Goldman, R. P.; Kresbach, K. D.; and Durfee, E. H. 2008. The evolution of CIRCA, a theory-based AI architecture with real-time performance guarantees. In *AAAI Spring Symp.: Emotion, Personality, and Social Behavior*.

Nau, D. 2013a. Game applications of HTN planning with state variables. In *ICAPS Workshop on Planning in Games*.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proc. 16th IJCAI*, 968–973.

Nau, D.; Patra, S.; Roberts, M.; Bansod, Y.; and Li, R. 2021. GTPyhop: A hierarchical goal+task planner implemented in Python. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Nau, D. S. 2013b. Pyhop, version 1.2.2: A simple HTN planning system written in Python. Software release. URL https://bitbucket.org/dananau/pyhop/src/master/.

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

Neufeld, X.; Mostaghim, S.; Sancho-Pradel, D. L.; and Brand, S. 2017. Building a planner: A survey of planning systems used in commercial video games. *IEEE Transactions on Games* 11(2): 91–108.

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019. Acting and planning using operational models. In *AAAI*, 7691–7698.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating acting, planning, and learning in hierarchical operational models. In *ICAPS*, 478–487.

Pollack, M. E.; and Horty, J. F. 1999. There's more to life than making plans: plan management in dynamic, multiagent environments. *AI Magazine* 20(4): 71–71.

Sacerdoti, E. 1975. The Nonlinear Nature of Plans. In *IJCAI*, 206–214.

Sardina, S.; De Silva, L.; and Padgham, L. 2006. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. 5th AAMAS*, 1001–1008.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*, 981–988.

Tate, A. 1977. Generating project networks. In *Proc. 5th IJCAI*, 888–893.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: an open architecture for command, planning and control. In Zweben, M.; and Fox, M. S., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Teichteil-Koenigsbuch, F.; Infantes, G.; and Kuter, U. 2008. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *Sixth International Planning Competition at ICAPS*.

Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational intelligence* 6(4): 232–246.

Yao, Y.; Alechina, N.; Logan, B.; and Thangarajah, J. 2021. Intention progression using quantitative summary information. In *Proc. 20th AAMAS*, 1416–1424.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: a baseline for probabilistic planning. In *ICAPS*, 352–359.

Yoon, S. W.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *AAAI*, 1010–1016.

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

On the Computational Complexity of Correcting HTN Domain Models

Songtuan Lin, Pascal Bercher
School of Computing, College of Engineering and Computer Science
The Australian National University
{songtuan.lin, pascal.bercher}@anu.edu.au

Abstract

Incorporating user requests into planning processes is a key concept in developing flexible planning technologies. Such systems may be required to change its planning model to adapt to certain user requests. In this paper, we assume a user provides a non-solution plan to a system and asks it to change the planning model so that the plan becomes a solution. We study the computational complexity of deciding whether such changes exist in the context of Hierarchical Task Network (HTN) planning. We prove that the problem is NP-complete in general independent of what or how many changes are allowed. We also identify several conditions which make the problem tractable when they are satisfied.

1 Introduction

Incorporating humans into planning processes has emerged as the frontier of the research in automatic planning for its potential to accomplish highly complicated tasks, e.g., see the works by Ferguson, Allen, and Miller (1996), Ferguson and Allen (1998), Ai-Chang et al. (2004), Bresina et al. (2005), and Behnke et al. (2016). One major challenge faced by the community in this direction is how to deal with the situation where a planning agent acts different from what a user expects. For instance, an agent may find a planning problem being unsolvable under its model whereas a user thinks it is not the case, or an agent offers a plan which differs from the one produced by a user himself/herself. The treatment for this problem varies in the role a user plays in the planning process. An end user may be curious about why the system's behavior is not in line with his/her expectation, namely looking for the explanations about the questions like "*why the problem is unsolvable?*" and "*why my plan is not a solution?*". Such explanations might be formulated either via transforming the planning model accordingly, e.g., changing the initial state (Göbelbecker et al. 2010) and abstracting the planning model to a certain level (Sreedharan, Srivastava, and Kambhampati, 2018; 2019), or via adjusting the user's expectation, e.g., correcting the plan the user has in mind (Barták et al. 2021a) and model reconciliation (Chakraborti et al., 2017; 2020). On the other hand, if the human involved is a domain writer, he/she may want to modify the planning model so that the agent's behavior can align with his/her anticipation. To this end, providing *modeling assistance* to help the domain writer comprehend the planning domain (Olz et al. 2021) or identify possible modeling errors via model transformations (Keren et al. 2017; Sreedharan et al. 2020) is vital especially when the planning domain is rather complicated.

In this paper, we re-visit a scenario we previously studied (Lin and Bercher 2021) where a user provides a plan and claims that it is supposed to be a solution to some planning problem, though it is actually not, and transformations on the planning model are required so that it will be. In our earlier work, we investigated the computational complexity of deciding whether such transformations can be found in the framework of totally ordered HTN (TOHTN) planning, which is a hierarchical approach of planning. Here we will extend those results. Our contributions are twofold. 1) We generalize our study to cover partially ordered HTN (PO-HTN) planning. 2) We consider the scenario with regard to different forms of the user input. For instance, a user could provide a partially ordered or sequential potential solution plan. The main results are summarized in Tab. 1

2 HTN Planning

We start with an introduction to the HTN formalism, which is based on the one by Bercher, Alford, and Höller (2019) and by Geier and Bercher (2011). We first give the definition of task networks.

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

Definition 2. Two task networks $tn = (T, \prec, \alpha)$ and $tn' = (T', \prec', \alpha')$ are said to be isomorphic, written $tn \cong tn'$, if and only if there exists a one-to-one mapping $\phi : T \to T'$ such that for all $t \in T$, $\alpha(t) = \alpha'(\phi(t))$, and for all $t_1, t_2 \in T$, if $(t_1, t_2) \in \prec$, $(\phi(t_1), \phi(t_2)) \in \prec'$.

The task names in a task network are further categorized as being primitive or compound. Primitive task names are mapped to respective actions by a function δ. The action of a primitive task name p, $\delta(p) = (prec, add, del)$, consists of p's precondition, add, and delete list, respectively. We also write $(prec(p), add(p), del(p))$ for short. On the other hand, a compound task name c can be refined (decomposed) into a task network tn by some method $m = (c, tn)$.

Complexity	Changes	Theorems	
		Any Changes	*k Changes*
NP-complete	Action	Cor. 2	Cor. 4
	Order	Thm. 2	

(a) The complexity of changing planning models provided with a PO task network that is supposed to be a solution.

Complexity	Changes	Theorems	
		Any Changes	*k Changes*
NP-complete	Action	Cor. 6	Cor. 8
	Order	Thm. 6	
P (Conditioned)	Action	Thm. 3 & 7	?

(b) The complexity of changing planning models provided with a PO/TO task network and a method sequence that is supposed to generate it. Special cases with changing actions being allowed that cover both totally ordered and partially ordered HTN planning are in P. Whether similar cases exist for the bounded version remains open (marked with '?').

Complexity	Changes	Theorems	
		Any Changes	*k Changes*
NP-complete	Action	Cor. 10	Cor. 13
	Order	Cor. 11	

(c) The complexity of changing planning models provided with an action sequence that is supposed to be a linearisation of a non-given solution task network.

Table 1: The computational complexity of the problems studied in this paper and the respective theorems (corollaries). The column 'Changes' specifies the target that changes are imposed to, i.e., changing actions or ordering constraints. The column 'Any Changes' refers to the case where an arbitrary number of changes can be applied, and 'k Changes' refers to the case where at most k changes can be applied.

Given a task network tn, the notations $T(tn)$, $\prec(tn)$, and $\alpha(tn)$ refer to the task identifier set, the partial order, and the identifier-name mapping function of tn, respectively. For a method m, we use $tn(m)$ to refer to its task network.

For convenience, we also define a restriction operation.

Definition 3. Let D and V be two arbitrary sets, $R \subseteq D \times D$ be a relation, $f : D \to V$ be a function and tn be a task network. The restrictions of R and f to some set X are defined by

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

A planning problem is then defined as follows.

Definition 4. An HTN planning problem P is a tuple (D, tn_I, s_I) where D is called the domain of P. It is a tuple (F, N_p, N_c, δ, M) in which F is a finite set of facts, N_p is a finite set of primitive task names, N_c is a finite set of compound task names with $N_c \cap N_p = \emptyset$, $\delta : N_p \to 2^F \times 2^F \times 2^F$ is a function that maps primitive task names to their actions, and M is a set of (decomposition) methods. tn_I is the initial task network, and $s_I \in 2^F$ is the initial state.

Definition 5. Let $tn = (T, \prec, \alpha)$ be a task network, $t \in T$ be a task identifier, c be a compound task name with $(t, c) \in \alpha$, and $m = (c, tn_m)$ be a method. We say m decomposes tn into another task network $tn' = (T', \prec', \alpha')$, written $tn \to_m tn'$, if and only if there exists a task network $tn'_m = (T_m, \prec_m, \alpha_m)$ with $tn'_m \cong tn_m$ such that

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

Additionally, a task network tn is decomposed into another task network tn' by a sequence of methods $\overline{m} = m_1 \cdots m_n$ ($n \in \mathbb{N}^0$ with $\mathbb{N}^0 = \mathbb{N} \cup \{0\}$), written $tn \to^*_{\overline{m}} tn'$, if and only if there exists a sequence of task networks $tn_0 \cdots tn_n$ such that $tn_0 = tn$, $tn_n = tn'$, and for each $1 \leq i \leq n$, $tn_{i-1} \to_{m_i} tn_i$. Particularly, $tn \to^*_{\overline{m}} tn$ if $\overline{m}$ is empty.

The solution criteria of a planning problem are then defined as follows.

Definition 6. Let $P = (D, tn_I, s_I)$ be an HTN planning problem. A solution to P is a task network tn such that all tasks in it are primitive, there exists a method sequence $\overline{m}$ that decomposes tn_I into it, i.e., $tn_I \to^*_{\overline{m}} tn$, and it possesses a linearisation of the tasks that is executable in s_I.

A linearisation $t_1 \cdots t_n$ of a (primitive) task network is executable in a state s if there exists a sequence of states $s_0 \cdots s_n$ such that $s_0 = s$, and for each $1 \leq i \leq n$, $s_{i-1} \subseteq prec(\alpha(t_i))$ and $s_i = (s_{i-1} \backslash del(\alpha(t_i))) \cup add(\alpha(t_i))$.)

The presented definition is standard in HTN planning as proposed by Erol, Hendler, and Nau (1996) and used in subsequent publications as well (Bercher, Alford, and Höller 2019). Other formalizations of hierarchical planning such as hybrid planning (Bercher et al. 2016) which fuses HTN planning with Partial Order Causal Link (POCL) where in solution plans *every* linearization is executable. We will also provide this alternative solution criterion.

Definition 7. Let $P = (D, tn_I, s_I)$ be an HTN planning problem. A solution to P is a task network tn such that all tasks in it are primitive, there exists a method sequence that decomposes tn_I into tn, and every linearisation of tn is executable in s_I.

The reason for including the more restricted solution criterion is to be able to identify the cause of computational hardness when model changes are required, though it is somehow unrealistic. A more practical one would be 'a task network tn is a solution *iff* it can be obtained via decompositions, and by adding some ordering constraints, every linearisation of it is executable'. However, the requirement of asking for additional ordering constraints has the same algorithmic lower bound as deciding whether tn has an executable linearisation, which itself is NP-hard already (Nebel and Bäckström 1994; Erol, Hendler, and Nau 1996)[1], because if such extra ordering constraints can be found,

[1]See Bercher (2021) for a discussion and further related work.

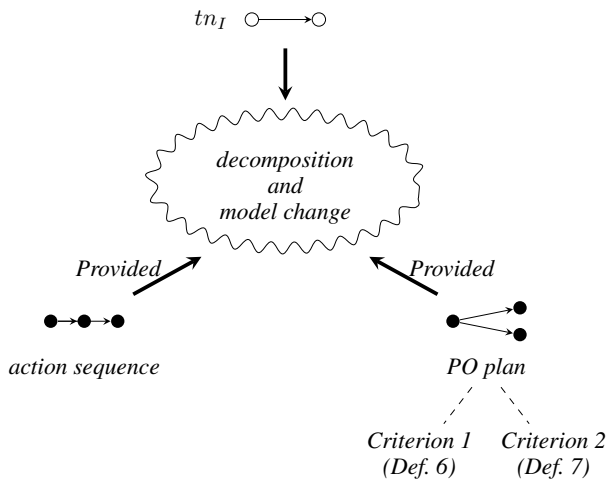


Figure 1: The scenarios where model change is involved. Criterion 1 states that a task network is a solution *iff* it is a refinement of tn_I and there exists a linearisation of it which is executable, whereas Criterion 2 requires that every linearisation is executable.

tn must have an executable linearisation. Thus, if we demand the solution criterion given by Def. 6 or the one requiring extra ordering constraints, it would not be clear where NP-hardness comes from. On the other hand, verifying whether all linearizations of a task network are executable was shown to be tractable (Nebel and Bäckström 1994; Chapman 1987).[2] Consequently, we list Def. 6 only for the sake of completeness, and we will adhere to Def. 7 throughout the paper in order to eliminate the ambiguous hardness source, unless otherwise indicated.

Fig. 1 previews what scenarios will be considered next. In the right branch we assume that a partially ordered plan is provided that is supposed to be a solution. Although we provide two solution criteria with regard to this case, we will primary focus on the one given by Def. 7. In the left branch we consider the case where an action sequence is provided rather than a partially ordered plan.

3 Changing the Model

For the purpose of changing planning models, we shall first define the allowed changes. We have introduced several model-change operations in the context of totally ordered HTN planning in our earlier work (Lin and Bercher 2021), which is a restricted version of HTN planning where the tasks in each task network in a planning model are totally ordered. For such a task network tn, its definition can be simplified by regarding it as a sequence of task names, i.e., $tn \in (N_p \cup N_c)^*$. We first reproduce the definitions of those operations since we will require them later on.

Definition 8. Let p be a primitive task name, $m = (c, tn)$ with $tn = t_1 \cdots t_n$ be a method, and $1 \leq i \leq n+1$ be an integer. The operation $\text{ACT}^{+}_{\text{TO}}$ is a function that takes as inputs p, m, and i and outputs a new method $m' = (c, tn')$ such that $tn' = tn_1\, p\, tn_2$ where $tn_1 = t_1 \cdots t_{i-1}$ and $tn_2 = t_i \cdots t_n$.

Definition 9. Let $m = (c, tn)$ be a method where $tn = tn_1\, p\, tn_2$ with $tn_1 = t_1 \cdots t_{i-1}$ and $tn_2 = t_{i+1} \cdots t_n$ be two sequences of task names, and p be a primitive task name. The operation $\text{ACT}^{-}_{\text{TO}}$ is a function that takes as inputs m and i and outputs a new method $m' = (c, tn')$ such that $tn' = tn_1\, tn_2$.

We use C_{TO} to refer to the set of changes allowed in totally ordered HTN planning. On top of those operations, we define several new operations that are targeted at partially ordered HTN planning problems. We first consider the operations that change the ordering constraints in a method.

Definition 10. Let $m = (c, tn)$ with $tn = (T, \prec, \alpha)$ be a method, and $t_1, t_2 \in T$ be two task identifiers. The operation ORD^{+} is a function that takes as inputs m and (t_1, t_2) and outputs a new method $m' = (c, tn')$ with $tn' = (T', \prec', \alpha')$ such that $T' = T$, $\prec' = (\prec \cup \{(t_1, t_2)\})^{+}$[3], and $\alpha' = \alpha$.

Definition 11. Let $m = (c, tn)$ with $tn = (T, \prec, \alpha)$ be a method, and $t_1, t_2 \in T$ be two task identifiers with $(t_1, t_2) \in \prec$. The operation ORD^{-} is a function that takes as inputs m and (t_1, t_2) and outputs a new method $m' = (c, tn')$ with $tn' = (T', \prec', \alpha')$ such that $T' = T$, $\prec' = \prec \backslash \{(t_1, t_2)\}$, and $\alpha' = \alpha$.

We then consider the operations that change the actions (primitive tasks) in a method. We start with the operation which adds an action to a method's task network.

Definition 12. Let $m = (c, tn)$ with $tn = (T, \prec, \alpha)$ be a method, $T_A = \{t_1, \cdots, t_n\}$ and $T_B = \{t'_1, \cdots t'_m\}$ with $n, m \in \mathbb{N}$ and $T_A \cap T_B = \emptyset$ be two subsets of T, and $p \in N_p$ be a primitive task name. The operation $\text{ACT}^{+}_{\text{PO}}$ is a function that takes as inputs m, T_A, T_B, and p and outputs a new method $m' = (c, tn')$ with $tn' = (T', \prec', \alpha')$ such that $T' = T \cup \{t\}$ with $t \notin T$ be a new task identifier, $\prec' = (\prec \cup \prec_A \cup \prec_B)^{+}$ with $\prec_A = \bigcup_{i=1}^{n} \{(t_i, t)\}$ and $\prec_B = \bigcup_{i=1}^{m} \{(t, t'_i)\}$, and $\alpha' = \alpha \cup \{(t, p)\}$.

Informally, the above operation inserts a primitive task to a position in tn that is after the tasks listed in T_A and before those in T_B. For instance, a new task is placed before all tasks in tn if $T_A = \emptyset$ and $T_B = T$. On the other hand, when removing an action from a method, we should delete all ordering constraints associated with this action.

Definition 13. Let $m = (c, tn)$ with $tn = (T, \prec, \alpha)$ be a method, and $t \in T$ be a task identifier. The operation $\text{ACT}^{-}_{\text{PO}}$ is a function that takes as inputs m and t and outputs a new method $m' = (c, tn')$ with $tn' = tn|_{T \backslash \{t\}}$.

Similarly, we use C_{PO} to refer to the set of change operations allowed in a partial order setting. Given two methods m, m' and a sequence of model-change operations

[2] Nebel and Baeckstroem did not show this in the context of HTN planning, but for unconditional *event systems*. These, however, perfectly coincide with a partially ordered set of actions such as in primitive task networks. A more detailed discussion can be found in the work by Bercher and Olz (2020).

[3] The superscript $^{+}$ refers to the transitive closure.

$\mathcal{X} = x_1(m_1, *) \cdots x_n(m_n, *)$ where for each $1 \leq i \leq n$, $x_i \in C_{\text{TO}}$ if a total order setting is given, otherwise $x_i \in C_{\text{PO}}$, m_i is a method, and $*$ refers to the remaining parameters in the operation. We write $m \rightarrow^*_{\mathcal{X}} m'$ if $m = m_1$, $m' = x_n(m_n, *)$, and for each $1 \leq i \leq n-1$, $m_{i+1} = x_i(m_i, *)$.

Definition 14. Let $P = (D, tn_I, s_I)$ with $D = (F, N_p, N_c, \delta, M)$ and $M = \{m_1, \cdots, m_n\}$ be a planning problem, and $\mathcal{X}$ be a sequence of method-changes. A problem $P' = (D', tn_I, s_I)$ with $D' = (F, N_p, N_c, \delta, M')$ and $M' = \{m'_1, \cdots, m'_n\}$ is obtained from P by applying $\mathcal{X}$, written $P \rightarrow^*_{\mathcal{X}} P'$ if and only if for each $1 \leq i \leq n$, either $m'_i = m_i$ or there exists a sub-sequence X_i of $\mathcal{X}$ such that $m_i \rightarrow^*_{X_i} m'_i$.

The definition is applied to both partially ordered and totally ordered HTN planning, and it implies that the method set in P maintains a one-to-one mapping to that in P'. We use $\beta_{\mathcal{X}} : M \rightarrow M'$ to denote this mapping, where for each method m_i with $1 \leq i \leq n$, $\beta_{\mathcal{X}}(m_i) = m'_i$.

Now we have defined all necessary model changes, we can move on to investigate the computational complexity of checking whether a change sequence exists that turns the given task network into a solution.

4 Complexity of Correcting the Model – Given Just A Task Network

We start by considering the question asking whether there exists a sequence of model-change operations with arbitrary length that turns a given partially ordered task network into a solution. We formulate the decision problem as follows, which generalizes the old one we gave for totally ordered HTN planning.

Definition 15. Let $X \subseteq \{\text{ACT}^+_{\text{SET}}, \text{ACT}^-_{\text{SET}}, \text{ORD}^+, \text{ORD}^-\}$ and $|X| \geq 1$, $\text{SET} \in \{\text{TO}, \text{PO}\}$, P be a planning problem, and tn be a task network. The problem $\text{FIXMETHS}^{\text{X}}_{\text{SET}}$ with SET specifying whether it is in a TO or a PO setting is to decide whether there is a sequence of change operations $\mathcal{X}$ consisting of the operations restricted by X such that $P \rightarrow^*_{\mathcal{X}} P'$, and tn is a solution to P'.

The hardness of the problem in a PO setting can be immediately obtained under the solution criterion given by Def. 6 (because deciding whether a partially ordered task network has an executable linearisation is already NP-hard). Thus, the question of interest is whether NP-hardness (henceforth NP-completeness) holds when we employ the solution criterion given by Def. 7. For this, we first consult our old result (Lin and Bercher 2021) that the problem is NP-complete in totally ordered HTN planning.

Proposition 1 (Lin and Bercher (2021, Thm. 1–4))**.** *Given an $X \subseteq \{\text{ACT}^+_{\text{TO}}, \text{ACT}^-_{\text{TO}}\}$ and $|X| \geq 1$, $\text{FIXMETHS}^{X}_{\text{TO}}$ is NP-complete.*

This proposition holds for both solution criteria given by Def. 6 and 7 because every task network in totally ordered HTN planning has only one linearisation. Since totally ordered HTN planning is a restricted version of partially ordered HTN planning, the hardness of the variants in the context of partially ordered HTN planning where only changing actions is allowed follows directly.

Corollary 1. $\text{FIXMETHS}^{X}_{\text{PO}}$ *with $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}\}$ and $|X| \geq 1$ is NP-hard.*

Next we show that these variants are in NP as well. To this end, we first prove that there always exists a polynomial upper bound of the length of the *shortest* change sequence that turns the given task network into a solution independent of what changes are allowed.

Lemma 1. *Let P and tn be a planning problem and a task network given by an instance of the $\text{FIXMETHS}^{X}_{\text{PO}}$ problem with $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ and $X \geq 1$. There must exist a change sequence $\mathcal{X}$ consisting of changes restricted by X such that $P \rightarrow^*_{\mathcal{X}} P'$, tn is a solution to P', and $|\mathcal{X}| \leq (\sum_{(c,tn_m) \in M} |T(tn_m)| + |\prec(tn_m)|) + |T(tn)| + |\prec(tn)|$ provided that any change sequence exists that meets the restriction of X and turns tn into a solution.*

Proof. We first consider the variant where all changes are allowed. We need to show that the upper bound presented is sufficient for the *shortest* change sequence. In such a change sequence, the number of action deletions must *not* exceed the total number of tasks in all methods, which is $\sum_{(c,tn_m) \in M} |T(tn_m)|$, otherwise, there must exist some action that is added first and removed afterward, and thus a shorter change sequence exists. For the same reason, the number of ordering constraint deletions is smaller or equal to $\sum_{(c,tn_m) \in M} |\prec(tn_m)|$, which is the total number of ordering constraints in all methods. On the other hand, the number of action insertions in the shortest change sequence cannot exceed the total number of tasks in tn (i.e., $|T(tn)|$), otherwise, some inserted actions must be deleted, and thus a shorter change sequence exists. The same argument holds for the number of ordering constraint insertions, which cannot exceed $|\prec(tn)|$. Thus, the presented upper bound holds.

For the remaining variants, the length of the shortest change sequence is strictly smaller than the presented upper bound because some changes are forbidden, e.g., if only adding actions is allowed, the length of the shortest change sequence must not exceed $|T(tn)|$. Thereby, the upper bound holds for all variants. □

The presented lemma not only reveals the NP-membership of the variants where only changing actions is allowed, but the fact that all classes are in NP.

Theorem 1. *Let $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ and $X \geq 1$. $\text{FIXMETHS}^{X}_{\text{PO}}$ is in NP.*

Proof. For *each* $X \subseteq \{\text{ACT}^+_{PO}, \text{ACT}^-_{PO}, \text{ORD}^+, \text{ORD}^-\}$ and $X \geq 1$, we can guess a change sequence of length smaller or equal to the upper bound stated in Lem. 1 which turns P into P' and consists of operations restricted by X. This step can be done in poly-time because the change sequence is bounded in length by a polynomial. Afterward, we verify whether *every* linearisation of tn is executable, which can be accomplished in polynomial time as well (Nebel and Bäckström 1994; Chapman 1987). Lastly, we employ the non-deterministic VERIFYTN algorithm (Behnke, Höller,

and Biundo 2015) to check whether tn_I can be decomposed into tn under the modified domain. Although the VERIFYTN algorithm is developed under the solution criterion given by Def. 6, it can be employed here because it is exploited in the sense that we do not need to consider the executability of tn (which has been verified previously). Thus, $\text{FIXMETHS}^{\text{X}}_{PO}$ is in NP. □

The NP-completeness of the variants where only changing actions in methods is allowed is thus a direct corollary of the previous results.

Corollary 2. $\text{FIXMETHS}^{\text{X}}_{\text{PO}}$ *with* $X \subseteq \{\text{ACT}^{+}_{\text{PO}}, \text{ACT}^{-}_{\text{PO}}\}$ *and* $|X| \geq 1$ *is NP-complete.*

What is new compared to totally ordered HTN planning are the operations that change ordering constraints in methods. It turns out that deciding whether we can transform a plan into a solution via changing ordering constraints in methods is NP-complete as well.

Theorem 2. $\text{FIXMETHS}^{\text{ORD}^{+}}_{\text{PO}}$ *is NP-complete.*

Proof. Membership has been given by Thm. 1. For hardness, we reduce from the *independent set* problem. The independent set problem is that given a graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, we want to decide whether there is a subset $V' \subseteq V$ such that $|V'| = k$, and there are no two vertices in V' which are connected to each other by an edge in E. Suppose $k \in \mathbb{N}$ and $G = (V, E)$ with $V = \{v_1, \cdots v_n\}$ and $E = \{e_1, \cdots, e_m\}$ are the integer and the graph given by an instance of the independent set problem. The key idea of the reduction is constructing a planning problem P whose initial task network tn_I encodes the structure of G. To this end, we construct one compound task v^c_i $(1 \leq i \leq n)$ for each vertex v_i and one primitive task e^p_i $(1 \leq i \leq m)$ for each edge e_i. The initial task network tn_I consists of two parts as shown by Fig. 2. The first part contains the *unordered* tasks $v^c_1, \cdots, v^c_n$. The second part is m continuous blocks $E_1 \cdots E_m$[4]. A block E_i $(1 \leq i \leq m)$ consists of the primitive task e^p_i, two compound tasks $v^c_{i_1}$ and $v^c_{i_2}$ $(1 \leq i_1, i_2 \leq n)$ whose respective vertices v_{i_1} and v_{i_2} in G are connected by the edge e_i, and one additional compound task h^c_i. Further, the block also has the ordering constraints $(e^p_i, v^c_{i_1})$, $(e^p_i, v^c_{i_2})$, and (e^p_i, h^c_i) which are drawn by thin arrows. Each thick arrow in the figure represents a set of ordering constraints specifying that all tasks in the left-hand side are ordered before those in the right-hand side. Afterward, we construct one method $m_{v_i} = (v^c_i, tn_{v_i})$ with $tn_{v_i} = (\{t_1, t_2\}, \emptyset, \{(t_1, s), (t_2, s)\})$ for each v^c_i in which s is an action. Additionally, for each h^c_i $(1 \leq i \leq m)$, we construct a method $m_{h_i} = (h^c_i, tn_{h_i})$ such that $tn_{h_i} = (\{t_1, t_2\}, \emptyset, \{(t_1, s), (t_2, s)\})$ as well. Finally, we construct the target task network tn as shown by Fig. 2. By construction, each compound task in tn_I has only one method that can decompose it. Adding an ordering constraint to some method m_{v_i} with $1 \leq i \leq n$ is now equivalent to selecting the respective vertex into the independent set. Next we show that an independent set of size k exists if and only if tn can be turned into a solution by adding ordering constraints to methods.

($\Longrightarrow$): Suppose V' is an independent set of size k. The change sequence that turns tn into a solution can be found as follows. For each $v_i \in V'$, we add the ordering constraint (t_1, t_2) to the method m_{v_i}. Afterward, we examine whether there exists some edge e_j of which two endpoints are not in V', and if it is the case, we add the ordering constraint (t_1, t_2) to the method m_{h_j}. By accomplishing this procedure, tn can now be obtained from tn_I.

($\Longleftarrow$): Suppose $\mathcal{X}$ is a change sequence that turns tn into a solution. An independent set of size k can be found by examining each operation in $\mathcal{X}$ iteratively and checking whether it adds the ordering constraint (t_1, t_2) to some method m_{v_i} $(1 \leq i \leq n)$. If so, the respective vertex v_i is in the set. The remaining operations that adds (t_1, t_2) to some m_{h_i} $(1 \leq i \leq m)$ can be simply ignored. □

Note that the only difference between the solution (which is uniquely defined) to the (unmodified) planning problem P and the task network tn in the presented proof is their ordering constraints. Thus, the proof still holds when the operations that change actions in methods are allowed. Moreover, since each method constructed in the proof does not have any ordering constraint at the beginning, allowing ordering constraint deletions is redundant as well. The following result is then a direct corollary.

Corollary 3. *Let* $X \subseteq \{\text{ACT}^{+}_{\text{PO}}, \text{ACT}^{-}_{\text{PO}}, \text{ORD}^{+}, \text{ORD}^{-}\}$ *and* $X \geq 1$. $\text{FIXMETHS}^{X}_{\text{PO}}$ *is NP-complete.*

Instead of asking whether there exists a change sequence of arbitrary length that transforms a task network into a solution, we are also interested in finding an optimal one. The decision problem asking for that is formulated in terms of an additional integer k.

Definition 16. Let $X \subseteq \{\text{ACT}^{+}_{\text{SET}}, \text{ACT}^{-}_{\text{SET}}, \text{ORD}^{+}, \text{ORD}^{-}\}$ and $X \geq 1$, $\text{SET} \in \{\text{TO}, \text{PO}\}$, and $k \in \mathbb{N}$, the problem $\text{FIXMETHS}^{\text{X},k}_{\text{SET}}$ is identical to $\text{FIXMETHS}^{\text{X}}_{\text{SET}}$ except that any change sequence should be limited in length by k.

We have shown in our previous work that the problem is NP-complete in a total order setting (Lin and Bercher, Cor. 1). In a partial order setting, any given $\text{FIXMETHS}^{\text{X}}_{\text{PO}}$ instance can be reduced to a $\text{FIXMETHS}^{\text{X},k}_{\text{PO}}$ instance by replicating the planning problem and the target task network given and setting k to the upper bound given by Lem. 1. Hardness thus follows immediately. For membership, although the given k can be exponentially large via logarithmic encoding, we can always guess a change sequence of length smaller than the minimum of k and the polynomial bound given by Lem. 1. Thereby, the problem is in NP as well.

Corollary 4. *Let* $X \subseteq \{\text{ACT}^{+}_{\text{PO}}, \text{ACT}^{-}_{\text{PO}}, \text{ORD}^{+}, \text{ORD}^{-}\}$ *and* $X \geq 1$. $\text{FIXMETHS}^{X,k}_{\text{PO}}$ *is NP-complete.*

5 Complexity of Fixing the Model – Given A Task Network and A Method Sequence

So far our investigation only consider a given planning problem and a task network which is supposed to be a solution.

[4]Note that each E_i $(1 \leq i \leq m)$ is *not* a compound task but an abbreviation of a component in tn_I.

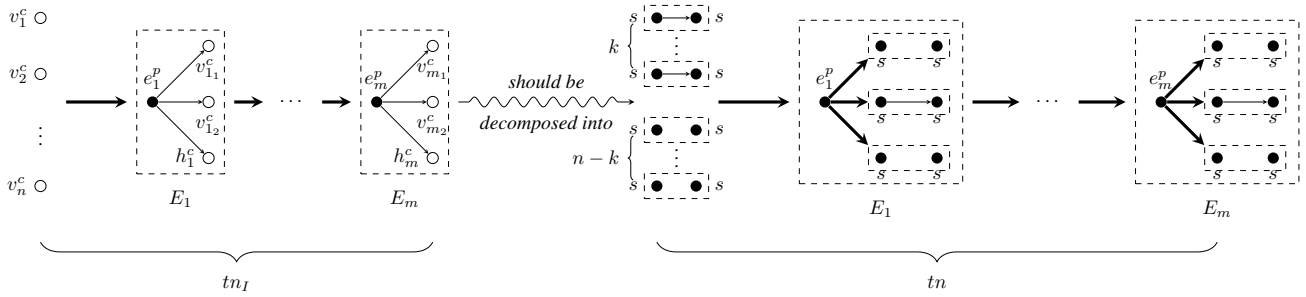


Figure 2: The constructions of tn_I and tn in the proof of Thm. 2. Each thick arrow represents a set of ordering constraints specifying that the tasks in the *lhs* are ordered before those in the *rhs*. Each thin arrow denotes a single ordering constraints. Each dashed rectangle represents nothing but a group of tasks that is part of tn_I or tn.

One can identify that one possible source of hardness is that we do not know which methods should be applied to generate the task network in question. To eliminate this source, we consider another scenario where we are given not only a task network and a planning problem, but a decomposition method sequence that is supposed to decompose the initial task network into the given one. For the practical motivation for this scenario, consider, e.g., a scenario in the context of modeling assistance where a user provides a plan as well as a method sequence to a planning system and argues that the plan must be generated by the given method sequence, whereas a plan verification system (Behnke, Höller, and Biundo 2017; Barták, Maillard, and Cardoso 2018; Barták et al. 2020, 2021b) rejects the plan. Thus, there must be some methods in the planning model that are incorrectly implemented. Correcting the model and identifying which methods are flawed can not only satisfy user requests but serve as counter-factual explanations (Ginsberg 1986; Chakraborti et al. 2017; Chakraborti, Sreedharan, and Kambhampati 2020) telling users what are the implementation errors in the case that plan verification fails, e.g., in a hierarchical planning competition.

Definition 17. Let $X \subseteq \{\text{ACT}^{\ddagger}_{\text{SET}}, \text{ACT}^{-}_{\text{SET}}, \text{ORD}^{+}, \text{ORD}^{-}\}$ and $|X| \geq 1$, $\text{SET} \in \{\text{TO}, \text{PO}\}$, P be a planning problem, $\overline{m} = m_1 \cdots m_n$ ($n \in \mathbb{N}^0$) be a sequence of methods, and tn be a task network. The problem $\text{FIXMSEQ}^{\text{X}}_{\text{SET}}$ with SET specifying whether it is in a TO or a PO setting is to decide whether there is a sequence of change operations $\mathcal{X}$ consisting of the operations restricted by X such that $P \to^*_{\mathcal{X}} P'$, and $tn_I \to^*_{\overline{m}'} tn$ with $\overline{m}' = \beta_{\mathcal{X}}(m_1) \cdots \beta_{\mathcal{X}}(m_n)$.

In our early study (Lin and Bercher 2021) we have shown that the problem is NP-complete in general in the context of totally ordered HTN planning. Here we will extend this result by showing that the presence of the method sequence does make the problem become easier when certain conditions are satisfied.

Proposition 2 (Lin and Bercher (2021, Thm. 5))**.** $\text{FIXMSEQ}^{\text{X}}_{\text{TO}}$ *with* $X \subseteq \{\text{ACT}^{\ddagger}_{\text{TO}}, \text{ACT}^{-}_{\text{TO}}\}$ *is NP-complete.*

Theorem 3. *Let* $X = \{\text{ACT}^{\ddagger}_{\text{TO}}, \text{ACT}^{-}_{\text{TO}}\}$. *The problem* $\text{FIXMSEQ}^{\text{X}}_{\text{TO}}$ *can be decided in constant time if* tn_I *contains no primitive tasks and there exists at least one method* $m_i = (c_i, tn_i)$ *(*$1 \leq i \leq n$*) in* $\overline{m}$ *such that for all* $m_j = (c_j, tn_j)$ *with* $1 \leq j \leq n$ *and* $j \neq i$, $c_i \neq c_j$.

Proof. Suppose m_i is the method in $\overline{m}$ that satisfies those conditions. A change sequence that turns tn into a solution can always be found by first removing every action from each method in $\overline{m}$ and then inserting the tasks in tn in turn into m_i. Thus, the problem is constant time decidable. □

Unfortunately, Thm. 3 does not hold in the case where we are only allowed to add or remove actions.

Theorem 4. $\text{FIXMSEQ}^{\text{ACT}^{-}_{\text{TO}}}_{\text{TO}}$ *is NP-complete even if* tn_I *and* $\overline{m}$ *satisfy the conditions presented in Thm. 3.*

Proof. Let $X = \text{ACT}^{-}_{\text{TO}}$. Membership follows from Prop. 2. For hardness, we reduce from the general $\text{FIXMSEQ}^{\text{X}}_{\text{TO}}$ problem. Let $P = (D, tn_I, s_I)$ with $D = (F, N_p, N_c, \delta, M)$, $\overline{m}$, and tn be the planning problem, the method sequence, and the task network given by an instance of the general $\text{FIXMSEQ}^{\text{X}}_{\text{TO}}$ problem, respectively. We construct an equivalent instance as follows. We first construct a planning problem $P' = (D', tn'_I, s_I)$ with $D' = (F, N_p, N_c \cup \{c'\}, \delta, M \cup \{m'\})$ where c' is an additional compound task name, $m' = (c', \varepsilon)$ decomposes c' into an empty task network, and $tn'_I = tn_I\, c'$. Afterwards, we construct the method sequence $\overline{m}' = \overline{m}\, m'$ and keep tn unchanged. Since m' results in an empty task network, we implicitly forbid $\text{ACT}^{-}_{\text{TO}}$ being applied to it. Thus, the general problem has a 'yes' answer if and only if the problem we constructed has one. □

Theorem 5. $\text{FIXMSEQ}^{\text{ACT}^{\ddagger}_{\text{TO}}}_{\text{TO}}$ *is NP-complete even if* tn_I *and* $\overline{m}$ *satisfy the conditions presented in Thm. 3.*

Proof. Let $X = \text{ACT}^{\ddagger}_{\text{TO}}$. Membership follows from Prop. 2. For hardness, we reduce from the general $\text{FIXMSEQ}^{\text{X}}_{\text{TO}}$ problem. Let $P = (D, tn_I, s_I)$ with $D = (F, N_p, N_c, \delta, M)$, $\overline{m}$, and tn be the planning problem, the method sequence, and the task network given by an instance of the general $\text{FIXMSEQ}^{\text{X}}_{\text{TO}}$ problem, respectively. To complete the reduction, we first construct the planning problem $P' = (D', tn'_I, s_I)$ with $D' = (F, N_p \cup \{p'_1, p'_2\}, N_c \cup \{c'_1, c'_2\}, \delta, M \cup \{m'_1, m'_2\})$ where p'_1 and p'_2 are two additional primitive tasks, c'_1 and c'_2 are additional compound

tasks, $m'_1 = (c'_1, p'_1)$ and $m'_2 = (c'_2, p'_2)$ decompose c'_1 and c'_2 to p'_1 and p'_2, respectively, and $tn'_I = tn_I\, c'_1\, c'_1\, c'_2$. Next we construct the method sequence $\overline{m}' = \overline{m}\, m'_1\, m'_1\, m'_2$ and the task network $tn' = tn\, p'_1\, p'_1\, p'_2$ that should be a solution to the modified planning problem. The existence of p'_1 and p'_2 ensures that actions cannot be added to m'_1 and m'_2. Thus, the general $\text{FIXMSEQ}^{\text{X}}_{\text{TO}}$ instance has a *yes* answer if and only if the problem we construct has one. □

Next we extend our investigation to partially ordered HTN planning. We again consider the problem under the solution criterion given by Def. 7. Note that the polynomial upper bound presented in Lem.1 still holds because the methods in $\overline{m}$ is a subset of M, i.e., the number of methods that need to be changed is smaller than the size of M, and thus the minimal number of changes required must not exceed that upper bound. NP-membership thus follows immediately.

Corollary 5. *Let* $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ *and* $X \geq 1$. $\text{FIXMSEQ}^{X}_{\text{PO}}$ *is in NP.*

On the other hand, the NP-hardness of the variants in a PO setting where only changing actions is allowed is a direct corollary of Prop. 2. Taken together, we immediately have the following result.

Corollary 6. $\text{FIXMSEQ}^{X}_{\text{PO}}$ *with* $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}\}$ *and* $|X| \geq 1$ *is NP-complete.*

Next we consider the complexity of the problem when changing ordering constraints is allowed.

Theorem 6. $\text{FIXMSEQ}^{\text{ORD}^+}_{\text{PO}}$ *is NP-complete.*

Proof. Membership has been given by Cor. 5. For hardness, we again reduce from the independent set problem. Given any instance of the independent set problem, we first construct a planning problem P and a target task network tn that are identical to those presented in the proof of Thm. 2. We have argued there that any compound task in the constructed initial task network has only one method which can decompose it. Thus, we can choose any method sequence that results in a solution to P as $\overline{m}$, and the proof still holds. □

The presented proofs also imply that allowing any combination of the defined changes will not make the problem easier because of the same argument made for Cor. 3.

Corollary 7. *Let* $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ *and* $X \geq 1$. $\text{FIXMSEQ}^{X}_{\text{PO}}$ *is NP-complete.*

Additionally, Thm. 3 can be further generalized in the framework of partially ordered HTN planning since a change sequence can always be constructed by following the same procedure if the conditions described there hold.

Theorem 7. $\text{FIXMSEQ}^{X}_{\text{PO}}$ *can be decided in constant time if* $\{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}\} \subseteq X$, tn_I *does not contain any primitive task and there exists at least one unique method in* $\overline{m}$.

We now proceed to study the complexity of finding the minimum number of changes required. We again define the problem by introducing an extra integer k.

Definition 18. Let $k \in \mathbb{N}$. The problem $\text{FIXMSEQ}^{\text{X},k}_{\text{SET}}$ with $X \subseteq \{\text{ACT}^+_{\text{SET}}, \text{ACT}^-_{\text{SET}}, \text{ORD}^+, \text{ORD}^-\}$ and $X \geq 1$ and $\text{SET} \in \{\text{TO}, \text{PO}\}$ is identical to $\text{FIXMSEQ}^{\text{X}}_{\text{SET}}$ except that we demand that *any* change sequence is limited in size by k.

The NP-completeness of the problem in the context of totally ordered HTN planning has been given by our previous work (Lin and Bercher 2021). For partially ordered HTN planning, since the polynomial upper bound given by Lem. 1 still holds, the arguments made for Cor. 4 is still valid, which implies NP-completeness.

Corollary 8. *Let* $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ *and* $X \geq 1$. $\text{FIXMSEQ}^{X,k}_{\text{PO}}$ *is NP-complete.*

One may ask whether there exist some conditions that make the problem easier once they are satisfied. For example, in totally ordered HTN planning, the problem can be decided in polynomial time if each method in the given method sequence decomposes a unique compound task (Lin and Bercher 2021). However, we cannot guarantee that the same argument holds in a partial order setting due to the existence of isomorphic task networks. Thus, whether such conditions exist in the context of partially ordered HTN planning is still an open question and will be studied in the future.

6 Complexity of Fixing the Methods – Given An Action Sequence

Our previous discussion over partially ordered HTN planning is based on the solution criterion given by Def. 7 because deciding whether a partially ordered task network has an executable linearisation is intractable. The remaining question is whether changing planning models becomes easier under the solution criterion given by Def. 6 if an executable linearisation of a task network is already provided. We formally define this problem as follows.

Definition 19. Let $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ and $X \geq 1$, P be a partially ordered HTN planning problem, and π be an action sequence. We define the problem $\text{FIXTSEQ}^{\text{X}}_{\text{PO}}$ as: Is there a sequence of method-change operations $\mathcal{X}$ such that $P \rightarrow^*_{\mathcal{X}} P'$, P' has a solution that possesses a linearisation which is identical to π, and $\mathcal{X}$ consists of the operations with respect to the value of X.

Clearly, Lem. 1 still holds for this problem because an action sequence π is actually a totally ordered task network which itself is a special partially ordered task network. It then follows that all variants are in NP.

Corollary 9. *Let* $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ *and* $X \geq 1$. $\text{FIXTSEQ}^{X}_{\text{PO}}$ *is in NP.*

If we restrict ourselves to totally ordered HTN planning and only consider the operations that change actions in methods, then the problem is identical to the one we studied before (Lin and Bercher 2021), which implies the NP-completeness of these variants.

Corollary 10. *Let* $X \subseteq \{\text{ACT}^+_{\text{PO}}, \text{ACT}^-_{\text{PO}}\}$ *and* $|X| \geq 1$. $\text{FIXTSEQ}^{X}_{\text{PO}}$ *is NP-complete.*

For the variants where only changing ordering constraints is allowed, it turns out that they are NP-complete as well.

Corollary 11. *Let* $X \subseteq \{\text{ORD}^+, \text{ORD}^-\}$ *and* $|X| \geq 1$. $\text{FIXTSEQ}^{X}_{\text{PO}}$ *is NP-complete.*

Proof. Membership has been given by Cor. 9. Hardness follows from that fact that VERIFYSEQ is NP-hard for the

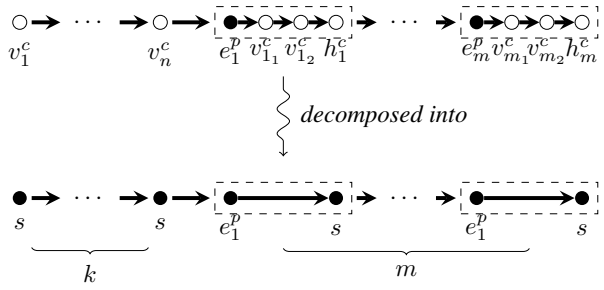


Figure 3: The construction for the proof of Prop. 1 in our earlier work (Lin and Bercher 2021). The constructed domain contains only one primitive task (action) s.

class $\text{HTN}_{\text{unordered}}$ (Behnke, Höller, and Biundo 2015, Cor. 5) where $\text{HTN}_{\text{unordered}}$ refers to the class of totally unordered HTN planning problems. If a planning problem is in $\text{HTN}_{\text{unordered}}$, deleting ordering constraints is clearly redundant. Adding ordering constrains is also pointless because those operations will only increase the possibility that a given action sequence is not a valid linearisation of a task network into which can be decomposed from the initial task network of a planning problem. Therefore, any VERIFY-SEQ instance with the input planning problem belonging $\text{HTN}_{\text{unordered}}$ can be reduced to a FIXTSEQ^{X}_{PO} instance with an arbitrary $X \subseteq \{\text{ORD}^+, \text{ORD}^-\}$ and $|X| \geq 1$. □

When it comes to the combination of changing actions and changing ordering constrains, we shall first consult the proof of Prop. 1 presented in our earlier work (Lin and Bercher 2021). The reduction we constructed is similar to the one shown in Thm. 2 except that tn_I is totally ordered, and each compound task in tn_I is now decomposed into an empty task network by the respective method, see Fig. 3. By construction, the only way to reach the target action sequence is by adding s to some methods. Thus, the operation that deletes an action immediately becomes pointless. Although our original proof is not concerned with changing ordering constraints, those are pointless as well because we can neither change the existed ordering constraints in tn_I nor add new ones to methods. The following result thus follows immediately.

Corollary 12. *Let* $X \subseteq \{\text{ACT}^{+}_{\text{PO}}, \text{ACT}^{-}_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ *and* $X \geq 1$. $\text{FIXTSEQ}^{X}_{\text{PO}}$ *is NP-complete.*

The decision problem aiming at finding the minimal number of changes required is formulated as follows.

Definition 20. Let $k \in \mathbb{N}$, P be a partially ordered HTN planning problem, and π be an action sequence. For each $X \subseteq \{\text{ACT}^{+}_{\text{PO}}, \text{ACT}^{-}_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ and $X \geq 1$, the problem $\text{FIXTSEQ}^{X,k}_{\text{PO}}$ is identical to $\text{FIXTSEQ}^{X}_{\text{PO}}$ except that we demand that *any* change sequence is bounded by k.

Both membership and hardness are implied by the presence of the polynomial upper bound of the minimal number of changes required.

Corollary 13. *Let* $X \subseteq \{\text{ACT}^{+}_{\text{PO}}, \text{ACT}^{-}_{\text{PO}}, \text{ORD}^+, \text{ORD}^-\}$ *and* $X \geq 1$. $\text{FIXTSEQ}^{X,k}_{\text{PO}}$ *is NP-complete.*

7 Conclusion

We investigated the computational complexity of deciding whether there exists a sequence of model change operations (could be of limited length) that transforms a planning problem into another one that has a given task network as a solution in the context of partially ordered HTN planning. Our results indicate that the problem is NP-complete unless additional constraints are specified, e.g., having no primitive task in the initial task network of a planning problem and having no duplicate methods in a decomposition method sequence that is supposed to generate a solution. Our results can be exploited in the future by transforming the decision problems into some well-studied NP-complete problems which can be solved by efficient solvers, e.g., SAT, and fully integrating model-change into MIP systems.

References

Ai-Chang, M.; Yglesias, J.; Chafin, B.; Dias, W.; Maldague, P.; Bresina, J.; Charest, L.; Chase, A.; Hsu, J.-J.; Jonsson, A.; Kanefsky, B.; Morris, P.; and Rajan, K. 2004. MAPGEN: mixed-initiative planning and scheduling for the Mars Exploration Rover mission. *IEEE Intelligent Systems* 19: 8–12.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of Hierarchical Plans via Parsing of Attribute Grammars. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling, ICAPS 2018*, 593–600. AAAI.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021a. Correcting Hierarchical Plans by Action Deletion. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021*. IJCAI.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021b. On the Verification of Totally-Ordered HTN Plans. In *Proceedings of the 4th ICAPS Workshop on Hierarchical Planning, HPlan 2021*.

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

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.

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

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) - Verifying solutions of hierarchical planning problems. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling, ICAPS 2017*, 20–28. AAAI.

Bercher, P. 2021. A Closer Look at Causal Links: Complexity Results for Delete-Relaxation in Partial Order Causal Link (POCL) Planning. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021*, 36–45. AAAI.

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

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

Bercher, P.; and Olz, C. 2020. POP $\equiv$ POCL, Right? Complexity Results for Partial Order (Causal Link) Makespan Minimization. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence, AAAI 2020*, 9785–9793. AAAI.

Bresina, J. L.; Jónsson, A. K.; Morris, P. H.; and Rajan, K. 2005. Activity Planning for the Mars Exploration Rovers. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling, ICAPS 2005*, 40–49. AAAI.

Chakraborti, T.; Sreedharan, S.; and Kambhampati, S. 2020. The Emerging Landscape of Explainable Automated Planning & Decision Making. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI 2020*, 4803–4811. IJCAI.

Chakraborti, T.; Sreedharan, S.; Zhang, Y.; and Kambhampati, S. 2017. Plan Explanations as Model Reconciliation: Moving Beyond Explanation as Soliloquy. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017*, 156–163. IJCAI.

Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32(3): 333–377.

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

Ferguson, G.; and Allen, J. F. 1998. TRIPS: An Integrated Intelligent Problem-Solving Assistant. In *Proceedings of the 15th AAAI Conference on Artificial Intelligence, AAAI 1998*, 567–572. AAAI.

Ferguson, G.; Allen, J. F.; and Miller, B. W. 1996. TRAINS-95: Towards a Mixed-Initiative Planning Assistant. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems, ICAPS 1996*, 70–77. AAAI.

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.

Ginsberg, M. L. 1986. Counterfactuals. *Artificial Intelligence* 30: 35–79.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; and Nebel, B. 2010. Coming Up With Good Excuses: What to do When no Plan Can be Found. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, 81–88. AAAI.

Keren, S.; Pineda, L. E.; Gal, A.; Karpas, E.; and Zilberstein, S. 2017. Equi-Reward Utility Maximizing Design in Stochastic Environments. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017*, 4353–4360. IJCAI.

Lin, S.; and Bercher, P. 2021. Change the World – How Hard Can that Be? On the Computational Complexity of Fixing Planning Models. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021*.

Nebel, B.; and Bäckström, C. 1994. On the Computational Complexity of Temporal Projection, Planning, and Plan Validation. *Artificial Intelligence* 66(1): 125–160.

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

Sreedharan, S.; Chakraborti, T.; Muise, C.; Khazaeni, Y.; and Kambhampati, S. 2020. – D3WA+ – A Case Study of XAIP in a Model Acquisition Task for Dialogue Planning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling, ICAPS 2020*, 488–497. AAAI.

Sreedharan, S.; Srivastava, S.; and Kambhampati, S. 2018. Hierarchical Expertise Level Modeling for User Specific Contrastive Explanations. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI 2018*, 4829–4836. IJCAI.

Sreedharan, S.; Srivastava, S.; Smith, D.; and Kambhampati, S. 2019. Why Can't You Do That HAL? Explaining Unsolvability of Planning Tasks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, 1422–1430. IJCAI.

On the Verification of Totally-Ordered HTN Plans

Roman Barták,[1] **Simona Ondrčková,** [1] **Gregor Behnke,** [2] **Pascal Bercher** [3]
[1] Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
[2] University of Freiburg, Faculty of Engineering, Freiburg, Germany
[3] The Australian National University, College of Engineering & Computer Science, Canberra, Australia
{bartak,ondrckova}@ktiml.mff.cuni.cz, behnkeg@informatik.uni-freiburg.de, pascal.bercher@anu.edu.au

Abstract

Verifying HTN plans is an intractable problem with two existing approaches to solve the problem. One technique is based on compilation to SAT. Another method is using parsing, and it is currently the fastest technique for verifying HTN plans. In this paper, we propose an extension of the parsing-based approach to verify totally-ordered HTN plans more efficiently. This problem is known to be tractable if no state constraints are included, and we show theoretically and empirically that the modified parsing approach achieves better performance than the currently fastest HTN plan verifier when applied to totally-ordered HTN plans.

Introduction

Plan verification is about finding if a given action sequence forms a correct plan according to a given planning domain model. For classical plans, the verification problem consists of checking if the action sequence is executable starting with the initial state and checking if the goal condition is satisfied in the final state (Howey and Long 2003). For hierarchical plans, plan verification additionally requires that the action sequence can be obtained by decomposition of some task. A specific root task, which decomposes to the action sequence, might also be given to describe the goal task.

There exist two approaches to hierarchical plan verification. One uses a translation of the verification problem into a Boolean satisfiability problem (Behnke, Höller, and Biundo 2017). The second uses parsing and it supports state constraints (Barták, Maillard, and Cardoso 2018; Barták et al. 2020). The hierarchical planning domain model can be seen as a formal grammar (Höller et al. 2014; Höller et al. 2016; Barták and Maillard 2017) and the plan verification problem is then similar to checking if a word (action sequence) belongs to the language generated by the grammar, which can be done by parsing. Parsing does not require information about the goal task – the method finds any task that decomposes to the action sequence, which makes it appropriate also for plan and task recognition (Vilain 1990).

The parsing-based approach seems to be significantly faster than the SAT-based approach (Barták et al. 2020). Nevertheless, both approaches struggle from the combinatorial explosion and, depending on the domain; they can verify plans of lengths up to a few dozens of actions. This is not surprising as the problem of verifying hierarchical plans is NP-hard (Behnke, Höller, and Biundo 2015; Bercher et al. 2016) and hence computationally expensive. This holds for general hierarchical plans with task interleaving and the partial order of tasks. However, as can be seen from the International Planning Competition 2020 on HTN planning, many domain models contain totally-ordered tasks. Further, there is a significant body of research dedicated to totally-ordered HTN planning in particular (Olz, Biundo, and Bercher 2021; Behnke and Speck 2021; Behnke 2021; Lin and Bercher 2021; Schreiber et al. 2019; Behnke, Höller, and Biundo 2018; Alford, Kuter, and Nau 2009; Marthi, Russell, and Wolfe 2007; Nau et al. 1999). Plan verification for a totally-ordered problem without state constraints is known to be tractable (Behnke, Höller, and Biundo 2015). Nevertheless, no hierarchical plan verifier exploits this theoretical result, no verifier specializes in exploiting the total order, and no generalisation to problems with state constraints exists.

We propose an extension of the parsing-based verification algorithm (Barták et al. 2020) to work faster for totally-ordered domain models. While the CYK algorithm (Sakai 1962) for parsing context-free grammars appears to be applicable here at first glance, this is not the case. Totally-ordered models can contain state constraints, which cannot, to our current knowledge, be compiled or handled by the CYK algorithm. Our primary modification is in handling precedence relations in the totally-ordered setting. The extended algorithm still works for arbitrary partially ordered hierarchical plans. It detects if the model uses totally-ordered tasks, and then uses a more strict formulation of precedence constraints, which decreases the number of generated tasks significantly. The algorithm works in a bottom-up fashion starting with a given action sequence $\bar{a}$. It terminates once a compound task is found that can be decomposed into $\bar{a}$. Apart from other work on plan verification, our approach is loosely related to another that aims at computing abstract plans that are maximally abstract while still allowing to generate a non-redundant plan (de Silva, Padgham, and Sardina 2019). The proposed algorithm also performs a bottom-up approach, though it requires a specific decomposition rather than the entire model.

HTN Plan Verification by Parsing

We use a standard STRIPS formalization (Fikes and Nilsson 1971). Let P be a set of propositions describing prop-

erties of world states. Then, a world state is modeled as a set $S \subseteq P$ of propositions that are true in that state (every other proposition is false). Each action a is modeled by three sets of propositions $(\texttt{pre}(a), \texttt{eff}^+(a), \texttt{eff}^-(a))$, where $\texttt{pre}(a), \texttt{eff}^+(a), \texttt{eff}^-(a) \subseteq P$ and $\texttt{eff}^+(a) \cap \texttt{eff}^-(a) = \emptyset$. The set $\texttt{pre}(a)$ describes positive preconditions of action a. These propositions must be true right before the action a. Action a is applicable to state S iff $\texttt{pre}(a) \subseteq S$. Sets $\texttt{eff}^+(a)$ and $\texttt{eff}^-(a)$ describe the positive and negative effects of action a. These propositions will become true or 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:

$$\gamma(S, a) = (S \setminus \texttt{eff}^-(a)) \cup \texttt{eff}^+(a).$$

$\gamma(S, a)$ is undefined if action a is not applicable to state S. We say that an action sequence $(a_1, \ldots, a_n)$ is *executable* with respect to a given initial state S_0 if the precondition of each action is satisfied in the state right before it:

$$\texttt{pre}(a_i) \subseteq \gamma(\gamma(\ldots\gamma(S_0, a_1), \ldots), a_{i-1}).$$

Hierarchical Task Network Planning (Erol, Hendler, and Nau 1996) was proposed as a planning framework that includes control knowledge as recipes for solving specific tasks. The recipe is modeled using a task network – a set of sub-tasks to solve the task and a set (a conjunction) 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 rewriting rule saying that T decomposes to sub-tasks $T_1, ..., T_k$ under the constraints C:

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

The order of sub-tasks in the rule does not matter (opposite to rewriting rules in grammars) as the precedence constraints in C explicitly describe the order. If the tasks $T_1, ..., T_k$ in each method are totally ordered, then we speak about a *totally-ordered HTN model*.

HTN planning problems are specified by an initial state S_0 and an initial task representing the goal. This goal task needs to be decomposed via decomposition methods until a set of primitive tasks – actions – is obtained. These actions must be totally ordered and satisfy all the constraints obtained during decompositions. The obtained plan $(a_1, \ldots, a_n)$ must be executable with respect to S_0. The state right after the action a_i is denoted S_i. 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, $end(T) = max\{i | a_i \in act(T)\}$.

The decomposition constraints for a method $T \rightarrow T_1, ..., T_k$ can be of the following three types, where the first is also known as an ordering constraint and the latter two are essentially state constraints ($U, V, \{t_1, t_2\} \subseteq \{T_1, ..., T_k\}$):

- $t_1 \prec t_2$: 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*(p, U): 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, p, V): a prevailing constraint 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 $(a_1, a_2, \ldots, a_n)$ and an initial state S_0, is the sequence of actions executable with respect to S_0 and obtained from some compound task?

Algorithm 1 presents the recent parsing-based approach to HTN plan verification (Barták et al. 2020) extended with the check of total-order constraints at line 13 (see the next section). The set $\prec$ represents the precedence constraints of the method, bef is the set of $before$ constraints, and btw is the set of $between$ constraints. Executability of the action sequence is verified at lines 2-5. The `while` loop (lines 7-26) groups actions/tasks into compound tasks by using the methods from the model until it finds a task T_0 such that $act(T_0) = \{a_1, a_2, \ldots, a_n\}$ (line 26, the plan is valid) or it constructs all possible tasks that decompose to a subset of actions in the plan (line 27, the plan is invalid). The sets $act(T)$ are represented using Boolean vectors I ($I(j) = 1 \Leftrightarrow a_j \in act(T)$). These vectors are used to check that each action is generated from one task only (line 19). Indexes b_j and e_j for task T_j describe values $start(T_j)$ and $end(T_j)$ respectively. They are used when checking the decomposition constraints.

Totally-Ordered HTNs

The parsing-based verification algorithm may generate an exponential number of pairs $(T, act(T))$, where T is a task and $act(T)$ is a subset of actions from the plan that can be generated from the task T. This is because actions from different tasks may interleave in the plan, and hence we must assume subsets $act(T)$ of actions from the plan when composing the tasks T. There is an exponential number of such sets with respect to the length of the plan. However, when the domain model is totally ordered, then the sets $act(T)$ form contiguous sub-sequences of actions (Figure 1).

Proposition 1. *For a totally ordered HTN domain model, each task decomposes to a contiguous sub-sequence of actions in the plan.*

Proof. Assume a pair of different tasks T and T' used in the decomposition of some goal task T_g to a sequence of actions such that T and T' are not descendants of each other. There must exists a common ancestor task T_a for tasks T and T' in the decomposition tree and a method $T_a \rightarrow T_1, ..., T_k \;\; [C]$ used for the decomposition. Let the task T be obtained from the sub-task T_i and T' be obtained from the sub-task T_j. As the domain model is totally-ordered, without loss of generality, we may assume that $T_i \prec T_j$ and hence $end(T_i) < start(T_j)$. As T is a sub-task of T_i, we know $end(T) \leq end(T_i)$ and similarly $start(T_j) \leq start(T')$. Together we

get $end(T) < start(T')$. Hence for any pair of non descendant tasks T and T', it holds either $end(T) < start(T')$ or $end(T') < start(T)$, which means that the tasks do not interleave in the plan. □

We can exploit this property when verifying plans for totally-ordered domain models as follows. Assume a decomposition method $T \rightarrow T_1, ..., T_k\ [C]$ in a totally-ordered domain model. Then it holds $\forall i \in \{1, \ldots, k-1\} : T_i \prec T_{i+1}$. We call these precedence constraints *direct precedences* to distinguish them from classical precedence relations. Note that it is easy to detect automatically, if the domain model is totally ordered, for example, by using a transitive closure of

Data: a plan $\mathbf{P} = (a_1, ..., a_n)$, an initial state S_0, and a set of decomposition methods (domain model); $TO = true$ if the domain is totally ordered,
Result: `true` if the plan can be derived from some compound task, `false` otherwise

1 **Function** VERIFYPLAN
2 **for** *i = 1* **to** *n* **do**
3 **if** $\neg(\text{pre}(a_i) \subseteq S_{i-1})$ **then**
4 **return false**
5 $S_i = (S_{i-1} \setminus \text{eff}^-(a_i)) \cup \text{eff}^+(a_i)$
6 $\mathbf{sp} \leftarrow \emptyset$; new $\leftarrow \{(A_i, i, i, I_i)\ |i \in 1..n\}$
 Data: A_i is a primitive task corresponding to action a_i, I_i is a Boolean vector of size n, such that $\forall i \in 1..n, I_i(i) = 1$, $\forall j \neq i, I_i(j) = 0$
7 **while** new $\neq \emptyset$ **do**
8 $\mathbf{sp} \leftarrow \mathbf{sp} \cup$ new; new $\leftarrow \emptyset$
9 **foreach** *decomposition method* R *of the form* $T_0 \rightarrow T_1, ..., T_k\ [\prec, \text{bef}, \text{btw}]$ *such that* $\{(T_j, b_j, e_j, I_j) | j \in 1..k\} \subseteq \mathbf{sp}$ **do**
10 **if** $\exists (i, j) \in \prec : \neg(e_i < b_j)$ **then**
11 **continue with the next method**
12 **if** $TO \wedge \exists i : \neg(e_i + 1 = b_{i+1})$ **then**
13 **continue with the next method**
14 $b_0 \leftarrow \min\{b_j | j \in 1..k\}$
15 $e_0 \leftarrow \max\{e_j | j \in 1..k\}$
16 **for** *i = 1* **to** *n* **do**
17 $I_0(i) \leftarrow \sum_{j=1}^{k} I_j(i)$;
18 **if** $I_0(i) > 1$ **then**
19 **continue with the next method**
20 **if** $\exists (p, U) \in \text{bef} : p \notin S_{\min\{b_j | j \in U\}-1}$ **then**
21 **continue with the next method**
22 **if** $\exists (U, p, V) \in \text{btw}\ \exists i \in \max\{e_j | j \in U\}, \ldots, \min\{b_j | j \in V\} - 1 : p \notin S_i$ **then**
23 **continue with the next method**
24 new $\leftarrow$ new $\cup \{(T_0, b_0, e_0, I_0)\}$
25 **if** $\forall k, I_0(k) = 1$ **then**
26 **return true**
27 **return false**

Algorithm 1: Parsing-based HTN plan verification

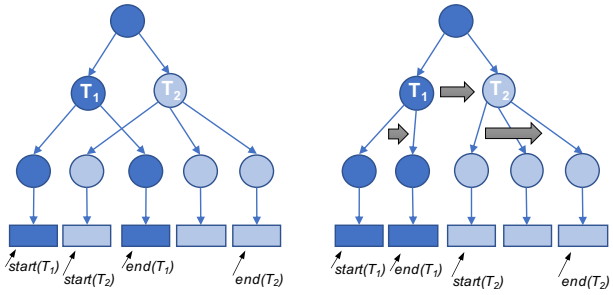


Figure 1: Task interleaving (left) vs. totally ordered (right).

precedence relations in the decomposition methods and verifying that sub-tasks in the method are totally ordered. The direct precedence relation $T_i \prec T_{i+1}$ means that the last action of task T_i is right before the first action of task T_{i+1}. This is a consequence of Proposition 1. Task T decomposes to a contiguous action sequence P. Each of its sub-tasks T_i also decomposes to a contiguous action sequence and these sub-sequences are ordered as $end(T_i) < start(T_{i+1})$. Together these sub-sequences must form the sequence P without any gap. Hence, the direct precedence relation imposes a more strict constraint

$$end(T_i) + 1 = start(T_{i+1}). \quad (1)$$

Note that the above claim also holds in the reverse order. Suppose we impose the above ordering constraint (1) for direct precedence relations in all decomposition methods. In that case, the tasks decompose to contiguous sequences of actions as no action can be inserted between any pair of directly following tasks.

The extended HTN plan verification algorithm (Algorithm 1) checks the direct precedence constraints for totally-ordered domain models at line 13. This extension gives the theoretical guarantee on the number of generated tasks.

Proposition 2. *Let t be the number of tasks in the totally-ordered HTN domain model and n be the number of actions in a plan. Then the extended HTN plan verification algorithm generates at most $O(t \times n^2)$ different pairs $(T, act(T))$.*

Proof. For totally ordered domain models, the sets $act(T)$ form contiguous sub-sequences of the plan. These sub-sequences are identified by the first and the last actions in the sequence, and hence there are at most $O(n^2)$ such sets. The same set of actions may be generated from different tasks; hence the maximal number of different pairs $(T, act(T))$ that the parsing-based verification algorithm may generate is $O(t \times n^2)$. □

Note that if the original verification algorithm is applied to totally-ordered domain models, then it may still generate an exponential number of pairs $(T, act(T))$ because the algorithm allows sets $act(T)$ to be arbitrary subsets of actions in the plan. The experimental study confirms this.

Empirical Evaluation

We compared the recent HTN plan verification algorithm (Barták et al. 2020) with its extended version that detects totally-ordered domain models and imposes constraints (1) to check the direct precedence constraints used in decomposition methods. Compared to previous evaluations, we have significantly increased the number of instances we consider. The International Planning Competition (IPC) 2020 has released an extensive set of plans[1] that were generated by the planners in the IPC on the IPC domains[2]. We are using the set of totally-ordered plans provided by the IPC, that is, all plans in our evaluation are totally-ordered. This set contains 10963 plans with an average length of 239 actions and a maximum length of 131071 actions.

Both the original verifier (Barták et al. 2020) and the modifications presented in this paper were implemented in C# 7 (from .NET 4.7). For running the program, we used mono in version 6.8.0.105 on a singularity container based on Ubuntu 20.10. We ran all experiments on an Intel Xeon Gold 6242 CPU (2.80GHz) with 5GB of RAM and a timeout of 10 minutes. The memory limit was never reached.

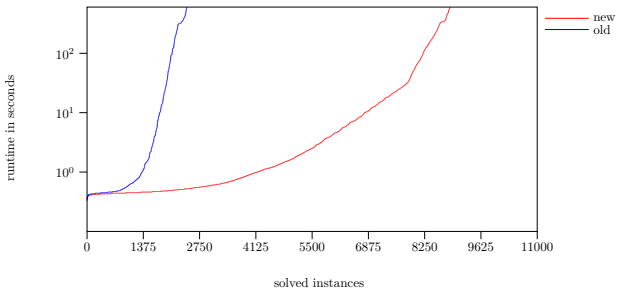


Figure 2: The number of solved problems per time.

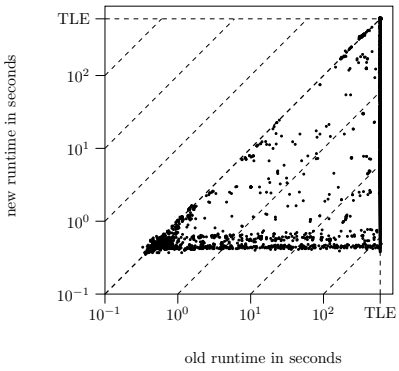


Figure 3: Direct comparison of runtimes.

The summary results are presented in Figure 2 showing the number of solved instances within a given time. The new approach solves a significantly larger number of instances (8870) than the original approach (2443). Any instance solved by the original approach was also solved by the new approach, while the new approach solved 6427 instances more. Figure 3 presents the direct comparison of both techniques using the same data. Each point represents one of the 2443 problem instances solved by both approaches. The runtimes of the algorithms define the coordinates of the point. Of these 2443 instances, the old approach is faster in 362 instances. Of these 362, only 159 have a runtime of more than one second. For these 362 instances, the old algorithm is faster than the new one by more than 10% in only 24 instances and at the most only 25% faster. The minor overhead of the new algorithm seems not to incur a significant disadvantage. For 210 instances, the runtime is identical, and for the remaining 1871 instances solved by both verifiers, the runtime of the new one is faster. The reduction in runtime on these 1871 instances is on average 46.36% with a maximum of 99.93%.

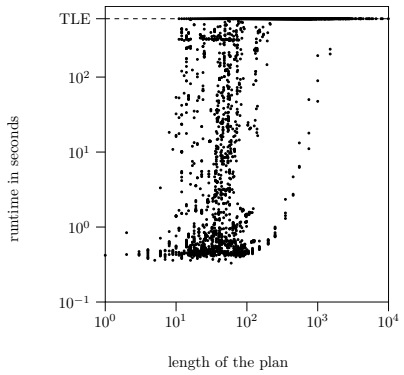


Figure 4: Runtime of the original algorithm as a function of plan length (omitting plans with more than 10.000 actions).

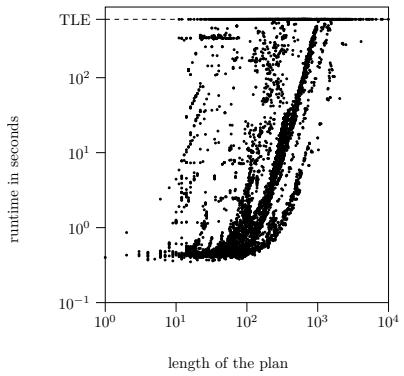


Figure 5: Runtime of the extended algorithm as a function of plan length (omitting plans with more than 10.000 actions).

Figures 4 and 5 show the dependence of runtime on plan length for the original and extended algorithm, respectively. Again, it is clearly visible that the new method solves a larger number of instances. The new approach can verify about one order of magnitude longer plans than the original algorithm. The longest verified plan for the old technique has 1500 actions, while for the new one has 4095 actions.

[1] https://github.com/panda-planner-dev/ipc-2020-plans
[2] https://github.com/panda-planner-dev/ipc2020-domains

Conclusions

We proposed extending the HTN plan verification algorithm to impose a more strict constraint describing direct precedence relations for totally-ordered models. The effect of this modification on the runtime of the algorithm is dramatic. The new algorithm verifies a much larger number of problem instances and also longer plans. As totally-ordered HTN domain models are frequent in practical applications, the method brings automated HTN plan verification closer to practical applicability on non-trivial plans and domains.

Acknowledgments. Research was supported by the joint Czech-German project registered under the number 21-13882J by the Czech Science Foundation (GAČR) and BE 7458/1-1 by Deutsche Forschungsgemeinschaft (DFG). S. Ondrčková is supported by SVV project number 260 575.

References

Alford, R.; Kuter, U.; and Nau, D. 2009. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. In *Proc. of the 21st Int. Joint Conf. on AI (IJCAI 2009)*, 1629–1634. AAAI Press.

Barták, R.; and Maillard, A. 2017. Attribute Grammars with Set Attributes and Global Constraints As a Unifying Framework for Planning Domain Models. In *Proc. of the 19th Int. Symposium on Principles and Practice of Declarative Programming (PPDP 2017)*, 39–48. ACM.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of Hierarchical Plans via Parsing of Attribute Grammars. In *Proc. of the 28th Int. Conf. on Automated Planning and Scheduling (ICAPS 2018)*, 11–19. AAAI Press.

Barták, R.; Ondrčková, S.; Maillard, A.; Behnke, G.; and Bercher, P. 2020. A Novel Parsing-based Approach for Verification of Hierarchical Plans. In *Proc. of the 32nd Int. Conf. on Tools with AI (ICTAI 2020)*, 118–125. IEEE.

Behnke, G. 2021. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning. In *Proc. of the 31st Int. Conf. on Automated Planning and Scheduling (ICAPS 2021)*, 25–35. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the Complexity of HTN Plan Verification and Its Implications for Plan Recognition. In *Proc. of the 25th Int. Conf. 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 *Proc. of the 27th Int. Conf. on Automated Planning and Scheduling (ICAPS 2017)*, 20–28. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-Ordered Hierarchical Planning through SAT. In *Proc. of the 32nd AAAI Conf. on AI (AAAI 2018)*, 6110–6118. AAAI Press.

Behnke, G.; and Speck, D. 2021. Symbolic Search for Optimal Total-Order HTN Planning. In *Proc. of the 35th AAAI Conf. on AI (AAAI 2021)*, 11744–11754. AAAI Press.

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 22nd European Conf. on AI (ECAI)*, 225–233. IOS Press.

de Silva, L.; Padgham, L.; and Sardina, S. 2019. HTN-Like Solutions for Classical Planning Problems: An Application to BDI Agent Systems. *Theoretical Computer Science* 763: 12–37.

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

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proc. of the 2nd Int. Joint Conf. on AI (IJCAI 1971)*, 608–620.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Proc. of the 21st European Conf. on AI (ECAI 2014)*, 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 *Proc. of the 26th Int. Conf. on Automated Planning and Scheduling (ICAPS 2016)*, 158–165. AAAI Press.

Howey, R.; and Long, D. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the Int. Planning Competition. In *Proc. of the ICAPS'03 Workshop on the Competition: Impact, Organization, Evaluation, Benchmarks*.

Lin, S.; and Bercher, P. 2021. Change the World – How Hard Can that Be? On the Computational Complexity of Fixing Planning Models. In *Proc. of the 30th Int. Joint Conf. on AI (IJCAI 2021)*. IJCAI.

Marthi, B.; Russell, S.; and Wolfe, J. 2007. Angelic Semantics for High-Level Actions. In *Proc. of the 17th Int. Conf. on Automated Planning and Scheduling (ICAPS 2007)*, 232–239. AAAI Press.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proc. of the 16th Int. Joint Conf. on AI (IJCAI 1999)*, 968–973.

Olz, C.; Biundo, S.; and Bercher, P. 2021. Revealing Hidden Preconditions and Effects of Compound HTN Planning Tasks – A Complexity Analysis. In *Proc. of the 35th AAAI Conf. on AI (AAAI 2021)*, 11903–11912. AAAI Press.

Sakai, I. 1962. Syntax in universal translation. In *Proc. of the 1961 Int. Conf. on Machine Translation of Languages and Applied Language Analysis*, 593–608.

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 29th Int. Conf. on Automated Planning and Scheduling (ICAPS 2019)*, 382–390. AAAI Press.

Vilain, M. 1990. Getting Serious About Parsing Plans: A Grammatical Analysis of Plan Recognition. In *Proc. of the 8th National Conf. on AI (AAAI)*, 190–197. AAAI Press.

Solving Hierarchical Auctions with HTN Planning

Antoine Milot,[1,2,3] Estelle Chauveau,[2] Simon Lacroix,[1] Charles Lesire[3]
[1] LAAS-CNRS, Université de Toulouse, CNRS, 7, Avenue du Colonel Roche, 31031 Toulouse, France
[2] Naval Research, Naval Group, 99 Avenue Pierre-Gilles de Gennes, 83190 Ollioules, France
[3] ONERA/DTIS, Université de Toulouse, 2 avenue Edouard Belin, 31055 Toulouse, France
antoine.milot@laas.fr, estelle.chauveau@naval-group.com, simon.lacroix@laas.fr, charles.lesire@onera.fr

Abstract

This paper presents a preliminary approach to solve the Multi-Robot Task Allocation problem through hierarchical auctions combined with the use of HTN planning. We present the global approach and the challenges arisen by partially-ordered HTNs through some examples. We finally outline some options to integrate such constraints in the allocation scheme.

Introduction

Deploying multi-robot systems for complex missions, such as post-catastrophic situation assessment or submarine mine-hunting, requires to reason about the tasks that the robots must perform, depending on their capabilities and the current situation. We then need to solve a Multi-Robot Task Allocation (MRTA) problem (Gerkey and Matarić 2004). Given a set of n robot $R = (r_1, \dots, r_n)$ and a set of s tasks $Q = (t_1, \dots, t_s)$, solving the MRTA problem consists in finding an allocation $A : Q \rightarrow R$, i.e. allocate each task $t \in Q$ to a robot $r \in R$.

When the environment is highly dynamic, tasks have to be reallocated regularly, due to the impossibility for some robot to achieve allocated tasks, or to the arrival of new tasks to achieve. The auction-based approaches have been extensively considered to approximately solve the MRTA problem in such environments (Dias et al. 2006).

While simple auction schemes only allocate tasks one after the other, hence not yielding optimal allocations, near-optimal allocation schemes need to resort to *combinatorial auctions*, i.e., auctions where each robot can bid over any subset of the tasks to allocate. However, solving the Winner Determination Problem (WDP), which is done by the auctioneer based on robots' bids, becomes untractable for combinatorial auctions. A good balance between expressing combinatorial auctions and solving the corresponding WDP has been proposed through *hierarchical auctions*, where subsets of tasks on which robots can bid are limited to nodes of the task decomposition.

Hierarchical auctions have been used to allow robots to bid on abstract tasks, which has the advantage for the bidders to account for local constraints on primitive tasks (Zlot and Stentz 2006; Liu et al. 2013; Khamis, Elmogy, and Karray 2011). However, in these approaches, greedy Breadth-First-Search (BFS) based algorithms have been proposed to solve the WDP. Consequently, they cannot handle causal constraints between tasks nor correctly manage ordering constraints.

In a former work, we laid the groundwork of an auction-based approach that allocates hierarchical tasks (Milot et al. 2021). We implemented a first prototype and tested it on totally-ordered coverage problems for multiple underwater robots. We obtained good performances in term of solution quality and computation time with respect to the state-of-the-art. Encouraged by the results of this proof of concept, we formalize in this paper the approach with causal and ordering constraints.

This approach relies on HTN planning (Bercher, Alford, and Höller 2019) to both estimate individual bids and solve the WDP. While integrating HTN planning in the hierarchical auction scheme is quite straightforward when HTN problems are totally ordered, managing partial-order problems is more challenging.

The next section summarizes related work. We then describe the general approach and detail the steps of the process for totally-ordered problems. Finally, we illustrate the current limits of the approach for partially-ordered tasks and we draw the first ideas to integrate these constraints into the proposed approach.

Related Work

Auction-based approaches to handle the MRTA problem have been explored for a long time (Dias et al. 2006), mainly because of their simplicity and their ability to handle dynamic events and unreliable communications (Otte, Kuhlman, and Sofge 2019).

The most basic scheme involves Single-Item (SI) auctions (Koenig, Keskinocak, and Tovey 2010): the *auctioneer* agent, which is responsible of the allocation, has only one item to allocate. This scheme then follows these steps: (1) the *announcement*, when the auctioneer opens the auction by broadcasting the information on the item for sale to the *bidders*; (2) the *bids estimation*, in which each bidder estimates the cost associated to the item and sends back its bid to the auctioneer; (3) the *Winner Determination*, where the auctioneer decides to which bidder the item is allocated; and (4) the *reward* announcement, in which the auctioneer announces which bidder won the auction. In the SI scheme, each bidder produces a single bid for the auctioned item,

and the winner determination simply consists of selecting the best bid.

A direct improvement of this scheme are Sequential Single-Item (SSI) auctions: the auctioneer announces a list of items for sale, and each bidder produces a bid for each item of this list. If some items remain not allocated after solving the WDP, the auctioneer starts a new round with the remaining items. The process goes on until all items are allocated or a stop criterion is reached. While SSI allows robots to prioritize some items over others (Kalra, Ferguson, and Stentz 2005; Dias, Ghanem, and Stentz 2005; Botelho and Alami 1999), it does not allow to express the dependencies between the bids. Indeed, a strong assumption of SI and SSI schemes is the independence of the bids, which results in allocating at most one item per robot at each round.

However, such dependencies may be mandatory to express in complex problems, when the tasks (i.e. the items) have temporal or causal constraints between them. Nunes and Gini (2015) have proposed a sequential auction scheme for temporally constrained tasks, where the auctioneer maintains a Simple Temporal Network (STN) of the items for sale. However, this approach is used for the allocation of tasks of an already computed STN, while in our approach we aim at solving both the allocation and the selection of the tasks to perform in order to fulfill a mission objective.

Zlot and Stentz (2006) proposed an auction scheme for *hierarchical tasks*, in which an item consists of an AND/OR tree that decomposes a complex task into sub-tasks. Bidders can then express bids on any node of this tree, and the auctioneer can decide to allocate a complete sub-tree to a robot in a single round. A direct benefit of this approach is to better interleave task decomposition and allocation. Consequently, by selling sub-trees, this approach allows bidders to take account of tasks' dependencies in their bids (e.g. by placing a more interesting bid on an abstract task than on the sum of its individual tasks). In addition, OR nodes in the tree induce bidders to make choices. Therefore, the MRTA problem to solve (by allocating nodes in the tree) becomes also a planning problem. This feature allows to handle difficult problems where choices are needed. Finally, by constraining hierarchically the possible sets of tasks, the hierarchical auctions reduce the burden of classic combinatorial auctions where a bid can be expressed on any subset of tasks.

This approach has been extended to allow bidders to buy and resell tasks to others, possibly proposing a new decomposition (Liu et al. 2013; Khamis, Elmogy, and Karray 2011). However, in these approaches, the WDP is solved by a greedy BFS-like algorithm, where task allocations are locally decided, without reasoning on the global task decomposition tree. While it provides an efficient algorithm in terms of computation time, the quality of the allocations are questionable. Moreover, these approaches do not consider temporal nor causal constraints between tasks, and due to the specific WDP algorithm, these constraints cannot be easily integrated in the approach.

Prerequisites

Notations

In this paper, we use the following notations, inspired from the ones used by Erol, Hendler, and Nau (1994); Höller et al. (2020). By $\mathcal{L}$, we denote a first-order language composed of finite sets of constants, predicate, primitive and compound task symbols, an infinite set of variable symbols, and an infinite set of labels denoted by $\mathbb{L}$. Given a set of terms $x_1, \dots, x_k$ issued from this language, and s a task symbol, we denote by $t = s(x_1, \dots, x_k)$ a *task* (also called *task instance*). A task t can be decomposed by a method $m = (t, tn)$, where $tn = (L, \prec, \alpha)$ is a task network where $L \subset \mathbb{L}$ is a set of labels, $\prec$ is a strict partial order over L and $\alpha : L \to X$ maps labels to the method sub-tasks. We then denote a planning domain $\mathcal{D}$ as $(\mathcal{L}, T_P, T_C, M)$ where $\mathcal{L}$ is the underlying language, T_P and T_C are the sets of primitive and compound tasks, and M the set of decomposition methods. Finally we write a planning problem as $\mathcal{P} = (\mathcal{D}, s_I, tn_I)$ where s_I is the initial state, and tn_I is the initial task network.

Solving a problem $\mathcal{P} = (\mathcal{D}, s_I, tn_I)$ consists in finding a *solution* task network tn such that tn is *primitive* and *executable* in s_I, i.e., there is a sequence of tn tasks, that respects the ordering constraints, in which the preconditions of a task are valid in the state resulting from applying the previous task.

Illustrative example

To illustrate our approach, we consider a *BorderDelivery* problem, inspired from the *transport* and *logistics* problems of the IPC2020 (Behnke et al. 2019). The goal of the *BorderDelivery* problem is to move two packages from an area *Ext* to an area *Storage* and check one of these packages in an area *Check* before bringing it back to *Storage*. To move the packages from *Ext* to *Storage*, robots can bring both packages at the same time or bring each package one by one. The locations of packages are included in a predicate *at(pkg, location)*. Thus, preconditions of the tasks to allocate verify this predicate and their effects change it, leading to causal relations between tasks.

The corresponding domain and problem formulated in Hierarchical Domain Definition Language (HDDL) are shown respectively on Listings 1 and 2. This minimal example allows to highlight the key points of our approach and to shine a light on challenges with partially ordered problems. We first explain the approach working under total order assumption.

General Approach

Our approach uses a SSI auctions scheme, similarly to (Zlot and Stentz 2006), but we use HTN planning at critical steps of the process, and then build our approach on HTN structures. This process is depicted in Figure 1.

Our approach relies strongly on a duality *global/local*. The global part corresponds to the multi-robot level, it consists in describing *what* to allocate and to *who*. This part is embodied by the task tree sent by the auctioneer and shared with all robots participating to the auction. On the other

```
(define (domain BorderDelivery)
   (:requirements :typing)
   (:types
     location locatable - object
     package - locatable)
   (:constants
     storage ext check - location
     package-0 package-1 - package)
   (:predicates
     (at ?x - locatable ?v - location))
   (:task random-check :parameters ())
   (:task store-packages :parameters (?p1
     ?p2 - package ?l1 ?l2 - location))
   (:method m-random-check
     :parameters (?p - package)
     :task (random-check)
     :ordered-subtasks (and
       (bring-new-package ?p storage check)
       (bring-new-package ?p check storage)
       ))
   (:method m-store-packages-one
     :parameters (?p1 ?p2 - package
       ?l1 ?l2 - location)
     :task (store-packages ?p1 ?p2 ?l1 ?l2)
     :ordered-subtasks (and
       (bring-new-package ?p1 ?l1 ?l2)
       (bring-new-package ?p2 ?l1 ?l2)))
   (:method m-store-packages-all
     :parameters (?p1 ?p2 - package
       ?l1 ?l2 - location)
     :task (store-packages ?p1 ?p2 ?l1 ?l2)
     :ordered-subtasks (and
       (bring-all-packages ?p1 ?p2 ?l1 ?l2)
       ))
   (:action bring-new-package
     :parameters (?p - package
       ?l1 ?l2 - location)
     :precondition (and
         (at ?p ?l1))
     :effect (and
         (not (at ?p ?l1))
         (at ?p ?l2)))
   (:action bring-all-packages
     :parameters (?p1 ?p2 - package
       ?l1 ?l2 - location)
     :precondition (and
         (at ?p1 ?l1)
         (at ?p2 ?l1))
     :effect (and
         (not (at ?p1 ?l1))
         (not (at ?p2 ?l1))
         (at ?p1 ?l2)
         (at ?p2 ?l2))))
```

Listing 1: HDDL description of the *BorderDelivery* domain.

hand, the local part corresponds to the *how* a particular robot can accomplish these tasks. Also, we assume that all multi-robot effects are included in the global part and local actions do not impact them.

An auction is initialized by the definition of an *Auction problem* that corresponds to the root task to be decomposed

```
(define (problem pb)
   (:domain BorderDelivery)
   (:htn
      :subtasks (and
         (t1 (store-packages package-0
            package-1 ext storage))
         (t2 (random-check)))
      :ordering (and (< t1 t2))
   (:init
      (at package-0 ext)
      (at package-1 ext)))
```

Listing 2: HDDL description of the *BorderDelivery* problem.

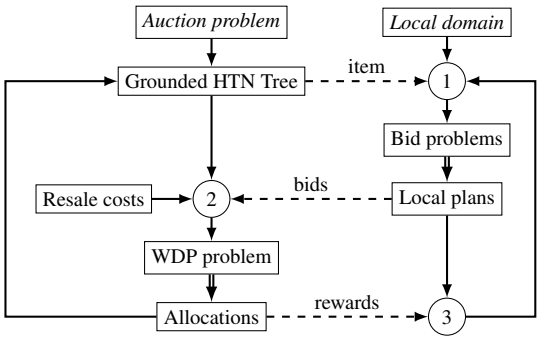


Figure 1: Protocol description. Rectangles represent information or data structures managed by the agents. Blocks on the left are managed by the auctioneer, blocks on the right by each bidder. Dashed arrows represent data exchanged between the auctioneer and the bidders. Circles indicate processes that aggregate information to build new structures. Double arrows indicate calls to a HTN planner.

and allocated. In order to start the allocation process, we first have to determine all the possible tasks to be allocated. To do so, we build a *Grounded HTN tree* $\mathcal{H}$ from the auction problem.

This *Grounded HTN tree* is a structure representing the grounded tasks and methods of the problem similarly to the Task Decomposition Graph (TDG) (Bercher et al. 2017). However, the TDG does not allow to differenciate task instances (i.e. a task symbol with associated parameters) that occur several times while in the *Grounded HTN tree* each task instance is labeled. This need for labeling the task instances comes from both ensuring that our robots reason over the same elements and that the mission is entirely fulfilled.

The auctioneer then sends $\mathcal{H}$ as an item for sale to the bidders. $\mathcal{H}$ represents the task decomposition of the *Multi-robot problem*. Each robot can bid on each task of the decomposition, depending on its ability to perform them.

In order to produce a bid on a labeled task in $\mathcal{H}$, the bidder will plan it to estimate a cost. The solution to this planning problem will be composed of the bidder's own *local* actions (which will increment the cost). To this aim, the bid-

der needs to express how a task in $\mathcal{H}$ can be accomplished regarding its own capacities: this is achieved thanks to HTN planning. The basic idea behind it is that we extend $\mathcal{H}$ with HTNs representing the bidder's local capacities and then use a HTN solver to estimate a bid. The merging of $\mathcal{H}$ with the bidder's *local domain* is done preserving the structure of $\mathcal{H}$ and the associated preconditions and effects on the multi-robot problem. This is accomplished by applying a protocol turning specific primitives tasks of $\mathcal{H}$ into abstract tasks.

Consequently, each bidder produces, for each task to estimate, an estimation problem. This problem is specifically built from the received item and its local domain for the task to estimate and is solved to determine its bid value (process ①). These problems are then solved by a HTN planner to produce the bid values that correspond to the cost of the local plan associated to the task.

Then, bids received from the bidders need to be integrated in order to find an allocation (i.e. a set of winning bids). To this aim, the auctioneer extends $\mathcal{H}$ by including bids as new decompositions of the concerned tasks, which produces a *WDP problem* (process ②). This problem is then solved by a HTN planner to decide which tasks of $\mathcal{H}$ are allocated to which robot. Finally, the result of the allocation is dispatched to the bidders.

Depending on the received bids, it may be possible to still have unallocated tasks at the end of the auction round. In this case $\mathcal{H}$ is modified to account for the allocated tasks. Then a new round is started by sending a new item for sale (with the updated HTN tree) to the bidders.

Hierarchical auctions on totally-ordered HTNs

When the auction problem is totally ordered, the process is sound. We detail the steps in this section. Then, we show the challenges raised when considering a partially ordered auction problem.

As we consider totally ordered problems, we need to add several constraints in our Border Delivery example. These are precedence constraints between every tasks. For example in our *BorderDelivery* problem, we have to store all packages before checking one. These precedence constraints are encoded in the corresponding decomposition methods.

Auction Initialisation

Our approach aims to solve an MRTA problem defined by an auction problem $\mathcal{P}_{auc} = (\mathcal{D}_{auc}, s_I, tn_I)$ formulated by the auctioneer. $\mathcal{D}_{auc}$ is a domain describing the high-level decomposition of tasks at the multi-robot scale. In fact, this domain focuses on the decomposition of tasks that can be allocated, i.e. describing *what* can be allocated without taking into account *how* a robot will accomplish the allocated tasks in terms of its proper actions (*goto, load, unload...*). s_I is the initial state determined by the auctioneer and tn_I is the initial task network of the problem. As a reminder, we consider that all multi-robot effects are included in this problem description and that local actions do not impact them.

Solving $\mathcal{P}_{auc}$ means solving the MRTA problem associated to this auction. However, as the decision scheme is decentralized, when robots bid on a task we need to determine to which instance of this task in the final multi-robot plan this bid is associated. Therefore, we cannot only reason on $\mathcal{P}_{auc}$ and we need to identify each task instance in the problem.

To do so, we use $\mathcal{P}_{auc}$ as an input of the auction scheme. From this auction problem, we build a *grounded HTN tree* $\mathcal{H}$ (algorithm 1). The grounded HTN tree describes hierarchical decomposition of tasks while labels bring unicity and identify them. During an auction round, robots rely on labels to share information relative to the grounded HTN tree.

Algorithm 1: *BuildGroundedHtnTree*

Input: $\mathcal{P}_{auc}$
Output: $\mathcal{H} = (V_T, V_M, E), labels$
$\mathcal{G} \leftarrow TDG(\mathcal{P}_{auc})$
if $\mathcal{G}$ *has cycles* **then return** error;
Let X be an empty First-In-First-Out list
Let l_{top} be a new unique label
$V_T \leftarrow \{(l_{top}, top)\}$, where top is the root task of $\mathcal{G}$
$labels \leftarrow \emptyset$; $V_M \leftarrow \emptyset$; $E \leftarrow \emptyset$
$X.push((l_{top}, top))$
while X *is not empty* **do**
  $(l, t) \leftarrow X.pop()$
  forall *method* $m \in \mathcal{G}$ *such that* $m = (t, (L, \prec, \alpha))$ **do**
    $V_M \leftarrow V_M \cup \{(l, m)\}$
    $E \leftarrow E \cup \{((l, t), (l, m))\}$
    forall $u \in L$ **do**
      Let v be a new unique label
      $V_T \leftarrow V_T \cup \{(v, \alpha(u))\}$
      $E \leftarrow E \cup \{((l, m), (v, \alpha(u)))\}$
      $labels \leftarrow labels \cup \{(l, u, v)\}$
      $X.push((v, \alpha(u)))$

This algorithm first builds a TDG from the auction problem. However, as the TDG does not allow to differenciate multiple occurences of a task instance the algorithm goes through the TDG in a breadth-first search way while labeling each task instance. In order to accomplish this translation from the TDG to the *grounded HTN tree*, the TDG must not have cycles, otherwise the algorithm will never end. We verify this condition with the TDG properties.

We then initialize $\mathcal{H}$ by adding the root task top, with a unique label l_{top} to the set of task vertices. Then, we expand $\mathcal{H}$ by creating, for each labeled task (l, t), and each decomposition method m of t in the TDG, a labeled method (l, m) in the set of method vertices. Finally, for each sub-task u of m, we create a new unique label v, and the corresponding labeled sub-task to $\mathcal{H}$, and store the label mapping to the $labels$ set. $labels$ will contain tuples (l_i, l_j, l_k) representing that sub-task with label l_j in the decomposition method of task labeled l_i has label l_k in $\mathcal{H}$. Figure 2 represents the resulting grounded HTN tree for a *BorderDelivery* auction problem.

At each round, the auctioneer sends $\mathcal{H}$ to each bidder, along with complementary information from the auction problem. An *item for sale* δ is then defined as a tuple

Figure 2: $\mathcal{H}$ for a *BorderDelivery* problem. Rounded rectangles are labeled tasks, hexagons are labeled methods.

$(\mathcal{H}_\delta, s_\delta, L_\delta)$, where $\mathcal{H}_\delta$ is a finite grounded HTN tree representing the hierarchical decomposition between labeled tasks and the associated precedence constraints (included in the decomposition methods); s_δ is a set of atomic formulas on the constants in $\mathcal{H}_\delta$; L_δ is the set of task labels in $\mathcal{H}_\delta$ that are *sellable*, i.e. on which robots can produce bids.

For example at the first round of the *BorderDelivery* problem, the auctioneer sends δ_1 with $\mathcal{H}_{\delta_1}$ (illustrated on Figure 2), s_{δ_1} which contains *at(pkg, location)* predicates specifying the initial locations of the packages, and $L_{\delta_1} = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7\}$.

Estimating bids with HTN planning

Once an item for sale δ is received, the bidder must compute a bid for each feasible labeled task among L_δ, i.e. each task executable by the bidder. The bid valuation corresponds to the cost of performing this task. Therefore, for each $l \in L_\delta$, we build a planning problem $\mathcal{P}_l = (\mathcal{D}_l, s_l, tn_l)$, and ask a HTN planner to solve it.

To compute this estimation, robots must indicate *how* they can perform primitive tasks of $\mathcal{H}_\delta$. They may indeed need to perform specific actions, like moving to the locations of packages, activating sensors or actuators to grab packages, etc. We consider that the descriptions of the tasks specific to each robot are defined in a *local problem*. This local problem has the following constraints: first, it must share the tasks that the robot can decompose locally with the auction problem (i.e., symbols and constants). Second, we reasonably consider that the local and auction problems do not share any predicate. It allows indeed to consider that a multi-robot task on the auction problem cannot depend on a predicate that could be validated only by a specific local action of a single robot. Conversely, that robot's local actions cannot depend on effects of multi-robot tasks.

The aggregation of the item and the local problem corresponds to step ① in Figure 1. It consists, for each label $l_\delta \in L_\delta$ on which the bidder wants to produce a bid, in extending $\mathcal{H}_\delta$ in the following way: each leaf task l of $\mathcal{H}$ that is either a descendent of l_δ, or was already allocated to the bidder in previous rounds, is replaced by an abstract task with exactly one ordered method, made of (in order) (1) a $start(l)$ action representing the beginning of l, and containing only the preconditions of l, (2) an abstract task that will be decomposed in the bidder local problem, and (3) an $end(l)$ action representing the end of l, and containing only the effects of l. Other primitive tasks are accounted a cost of 0. Essentially, $start(l)$ and $end(l)$ allow to insert the robot's local actions between the preconditions and effects of the task l that were defined in $\mathcal{H}$.

Figure 3 illustrates this process by showing the expansion of the task labeled l_4 from the *BorderDelivery* problem. Tasks in yellow correspond to the decomposition introduced earlier. Orange tasks correspond to the TDG built with a robot's local domain. For example, solving the bid estimation problem can lead to a sequence of actions such as $[start(l_4) \rightarrow get\text{-}to(Ext) \rightarrow \cdots \rightarrow end(l_4)]$.

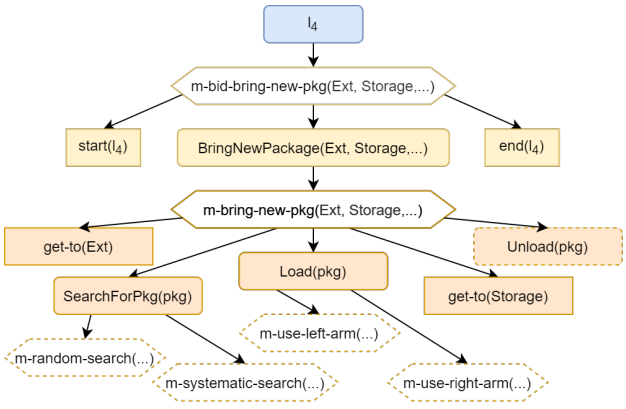


Figure 3: Illustration of the decomposition of labeled task l_4 for the bid estimation of the *BorderDelivery* problem. Hexagons represent methods, rounded rectangles represent compound tasks while sharp rectangles are primitive tasks, dotted lines represent nodes with hidden decompositions. Blue nodes represent elements from $\mathcal{H}$. The specific decomposition for the bid estimation is represented in yellow. Local decomposition nodes are represented in orange.

Moreover, the bidder builds s_l by merging s_δ with its proper initial state including its own local information (e.g. its current position, energy...). tn_l is defined by the root task of $\mathcal{H}_\delta$.

Finally, for each labeled task on which we want to esti-

mate a bid, we synthesize a domain and problem (including the decomposition mentioned earlier, and the initial states both from the item for sale and the local problem). We rely on the HDDL formalism to define these domain and problem. In this problem, primitive actions are either the start and end actions introduced earlier, which have a null cost, or bidder's local actions, which may have a non-unit cost. A HTN planner then solves this problem, producing both a local plan to complete the task and a cost associated to this plan. Actions start and end in the plan allow to check preconditions and trigger effects of the task.

Finally, the bidder returns to the auctioneer a set B containing triplets (l, c, P), where $l \in L_\delta$ is a task label, c the associated bid value, i.e. the cost returned by the planner, and P the set of $\mathcal{H}_\delta$ task labels present in the local plan returned by the planner.

Solving the WDP with HTN planning

At the end of an auction round, the auctioneer solves the WDP to determine the task allocations. Given L_δ the set of labels of the tasks for sale and $R = (r_1, \ldots, r_n)$ the set of bidders participating to the auction, we denote B_{r_i} the set of received bids from the bidder r_i bearing on labels in L_δ. The set of all received bids is denoted by $\mathcal{B} = \bigcup_{r_i \in R} B_{r_i}$. Solving the WDP consists in finding a set of winning bids $\mathcal{B}_w \subset \mathcal{B}$ such that each bidder wins at most one bid and each $l \in L_\delta$ is won by at most one bidder.

In order to determine this set of winning bids, the auctioneer needs to build $\mathcal{D}_{wdp}$ and $\mathcal{P}_{wdp} = (\mathcal{D}_{wdp}, s_{wdp}, tn_{wdp})$ a planning domain and problem dedicated to the WDP solving.

To do so, we complete $\mathcal{H}_\delta$ with the bids received from the bidders (step ② in Figure 1). We do this by adding to every task one method for every bid. Each of these methods corresponds to a unique action whose cost is the bid value. However, we must integrate the SSI scheme constraint on bid independence, that enforces that at most one task (whatever its abstraction level) can be allocated to each robot.

Encoding this property in the actions corresponding to the bids is quite straightforward. Each task on which a bid has been received must then also be decomposed into an action corresponding to not allocating this task (that we call *resell*). The pattern is quite similar to the one used for bidding: we add a new method for each task on which we have bids, decomposed with $start$ and end actions for primitive tasks, and an abstract $AllocateOrResell$ task that will itself be decomposed into one method per bid plus one resell method.

Figure 4 illustrates this decomposition for task with label l_4, for which robot r_1 bids. A new decomposition leading to the task allocation or reselling was added with the task network consisting in doing $start(l_4) \rightarrow AllocateOrResell(l_4) \rightarrow end(l_4)$. And $AllocateOrResell(l_4)$ can be decomposed as a resell action or an allocate action to robots that have bid on this task, here r_1 submitted a bid on task l_4.

Similarly to the estimation process, tn_{wdp} is defined by the root task of $\mathcal{H}_\delta$. However, the initial state s_{wdp} is the same as s_δ.

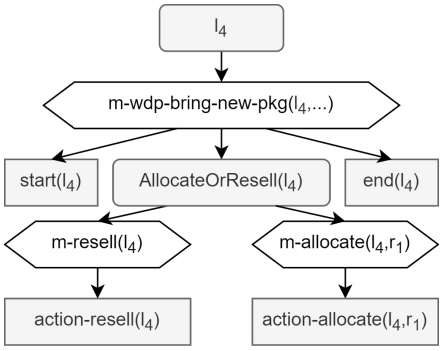


Figure 4: Decomposition of task l_4 integrating received bids for the WDP of the *BorderDelivery* problem. Hexagons represent methods, rounded rectangles represent compound task while sharp rectangles are primitive tasks.

As we cannot force the WDP to allocate exactly one task to each robot, we cannot set the cost of the $resell$ actions to 0: these tasks would systematically be resold when optimizing the allocation plan. Consequently, we need to define in a clever way the *resale costs* to have a sound and efficient allocation behavior. While these costs could be defined per domain, we have proposed two strategies to define them, based on the received bids on a task:

- a pessimistic strategy about the bids that will be received in future rounds for this task, encouraging to allocate the task at this round; the resale cost is then set as the maximum of the bid values for this task plus one;
- an optimistic strategy that hopes for better bids in the future; the resale cost is set as the minimum of the bids values for this task plus one.

In summary, the auctioneer creates, from $\mathcal{H}_\delta$ and the received bids, a new planning problem $\mathcal{P}_{wdp}$ corresponding to the WDP of this auction round. This problem, translated to a HDDL domain and problem, is then solved by a HTN planner, which returns a plan containing either an *allocation* action or a *resell* action for each task. The auctioneer then sends reward messages with winning bids for the allocated tasks. The winners keep traces of their reward and commit the local plan associated to the winning bid (process ③ in Figure 1). Finally, the auctioneer integrates the plans attached to the winning bids into $\mathcal{H}_\delta$[1], and a new round is started with the tasks to resell.

Auctioning with partially-ordered HTNs

The allocation protocol presented earlier is quite simple: a consistent HTN tree is updated all along the rounds with the plans corresponding to the allocated tasks, and is extended respectively by the bidders to integrate their own decomposition when estimating bids, and by the auctioneer to integrate the bids and solve the WDP. While the formulation requires

[1]this step, which also removes the deprecated decomposition from $\mathcal{H}$, is quite straightforward, and is hence not further detailed.

some modelling tricks, the process is sound and simple when tasks are totally ordered.

In this section, we present through an example why partially-ordered tasks cannot be managed by the previous approach, and we give some perspectives towards integrating such constraints.

Illustrative example

Again, we use the *BorderDelivery* problem as a support example. Without losing relevancy, we remove ordering contraints $l_1 \rightarrow l_2$ and $l_4 \rightarrow l_5$ because we do not need to bring all packages before checking one and we do not need to bring a precise package before the other, therefore the problem is now partially-ordered.

Let us consider an auctioneer that tries to allocate the $\mathcal{H}$ of Figure 2 in this partially-ordered case. The auctioneer sends the corresponding item to two robots, r_1 and r_2.

Let us also consider that at the first round, r_1 won task l_4 and r_2 won task l_5. The plans of r_1 and r_2 are then respectively (if we only look at primitive tasks in the multi-robot problem) $[\boldsymbol{l_4}]$ and $[\boldsymbol{l_5}]$.

At the second round, the auctioneer sends an updated $\mathcal{H}_2$, where method $(l_1, m\text{-}store\text{-}pkg\text{-}all)$ has been removed, l_4 and l_5 are already allocated, and l_2, l_6 and l_7 are sellable tasks. At this second round, r_1 wins l_6, for which it has computed the plan:

$$[l_5 \rightarrow \boldsymbol{l_6} \rightarrow \boldsymbol{l_4}] \quad (1)$$

where tasks in bold correspond to tasks executed by r_1 (either already allocated or being bid) and non-bold tasks represents tasks allocated to another robot (or not allocated tasks) that are necessary to the robot's plan, i.e. tasks whose effects are preconditions of robot's tasks.

At the third round, only l_7 is to sell. $\mathcal{H}_3$ integrates that both l_4 and l_6 have been allocated to r_1, and l_5 to r_2. Let us consider that r_2 wins l_7, with the plan:

$$[l_4 \rightarrow \boldsymbol{l_7} \rightarrow \boldsymbol{l_5}] \quad (2)$$

Therefore, at the end of the third round, all tasks have been allocated and the auction is finished. However, the resulting allocation cannot be executed. Both robots locally computed a plan involving a temporal constraint between one of their task and a task of the other robot. These constraints are specifically induced by a causal constraints on the *at* predicate, representing the location of the packages. We can indeed notice that the multi-robot plan issued from equations (1) and (2) contains a mutual dependency between l_4 and l_5.

Such a situation could not arise in totally-ordered problems: ordering constraints are already modeled in $\mathcal{H}$, and the plans computed by the robot cannot add new ordering constraints.

Conversely, in partially-ordered problems, each bidder may decide on its own to fix an order between tasks to solve the bid problem, but this new constraints is not forwarded to the auctioneer, and then to other bidders, then leading to possible inconsistencies in the ordering constraints of each local plan.

Problem statement

As we have seen in the previous example, inconsistencies may arise due to the lack of information on the bidders' local plans when solving the WDP and estimating successive bids. A worse situation may even happen: as a bidder decomposes primitive multi-robot tasks with its local domain, the resulting local plan can *interleave* these tasks, while they are primitive for the auctioneer, i.e. not decomposable.

We then have to not only integrate information about ordering constraints coming from bidders local plans, but also make the approach able to account for the interleaving of tasks, at any level of the hierarchy (as bidders can bid on abstract tasks), integrating causal relations with tasks allocated to other robots.

Solution perspectives

The current approach for totally-ordered problems has already some nice features and assumptions that will help integrating all the considered constraints. First, the constraint to allocate at most one task per robot per round eases the integration, as we will not have to consider simultaneous constraints coming from several bids of the same robot at the same time. The update of $\mathcal{H}$ to integrate allocated tasks will also ensure that constraints from previous allocations are enforced in the new bids. Finally, the proposed decomposition of primitive tasks with a $start$ and end action will help formulating constraints allowing the interleaving of tasks.

Therefore, to keep a sound problem, results (i.e. committed local plans) from the allocation of the previous round must be encoded into the new $\mathcal{H}$. By doing so, they will be taken into account during the bid estimation phase without need of further modification. Moreover, this encoding directly comes out from the WDP solution which must not deny the intended plans of the winners. Thus, the most challenging part is the WDP problem formulation (process ②) which must reflect the bidders' intentions in order to provide a consistent solution.

Consequently, we will integrate every local plan computed during the bid estimation into the WDP problem. More specifically, we will investigate whether integrating this local plan can be done when decomposing the task on which the bid is evaluated, in place of a simple action, as illustrated in Figure 4.

Discussion

In this paper, we proposed a preliminary approach to handle the Multi-Robot Task Allocation problem by solving hierarchical auctions with HTN planning. The approach relies on hierarchically linked tasks and auctions in order to interleave decomposition and allocation. Items for sale are HTNs and HTN planning is used to both formulate the bids and solve the Winner Determination Problem.

We presented a protocol that is sound for totally-ordered problems. However, partially-ordered problems raise some specific challenges, and we outlined some perspectives to handle them and support causal and temporal constraints.

Current work aim at improving the approach with the proposed perspectives. Finally, we want to demonstrate the applicability of the approach to online reparation problems.

References

Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Pellier, D.; Fiorino, H.; and Alford, R. 2019. Hierarchical Planning in the IPC. In *Workshop on HTN Planning (ICAPS)*.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*. Macao, China.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI*.

Botelho, S.; and Alami, R. 1999. M+: a scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Int. Conf. on Robotics and Automation (ICRA)*. Detroit, MI, USA.

Dias, M.; Zlot, R.; Kalra, N.; and Stentz, A. 2006. Market-Based Multirobot Coordination: A Survey and Analysis. *Proceedings of the IEEE* 94(7).

Dias, M. B.; Ghanem, B.; and Stentz, A. 2005. Improving cost estimation in market-based coordination of a distributed sensing task. In *Int. Conf. on Intelligent Robots and Systems (IROS)*. Edmonton, Canada.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *AAAI Conference on Artificial Intelligence (AAAI)*. Seattle, WA, USA.

Gerkey, B.; and Matarić, M. 2004. A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems. *The International Journal of Robotics Research* 23(9).

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *AAAI Conference on Artificial Intelligence (AAAI)*. New York City, NY, USA.

Kalra, N.; Ferguson, D.; and Stentz, A. 2005. Hoplites: A Market-Based Framework for Planned Tight Coordination in Multirobot Teams. In *Int. Conf. on Robotics and Automation (ICRA)*. Barcelona, Spain.

Khamis, A. M.; Elmogy, A. M.; and Karray, F. O. 2011. Complex Task Allocation in Mobile Surveillance Systems. *International Journal on Intelligent and Robotic Systems* 64(1).

Koenig, S.; Keskinocak, P.; and Tovey, C. 2010. Progress on Agent Coordination with Cooperative Auctions. In *AAAI Conference on Artificial Intelligence (AAAI)*. Atlanta, GA, USA.

Liu, Y.; Yang, J.; Zheng, Y.; Wu, Z.; and Yao, M. 2013. Multi-Robot Coordination in Complex Environment with Task and Communication Constraints. *International Journal of Advanced Robotic Systems* 10(5).

Milot, A.; Chauveau, E.; Lacroix, S.; and Lesire, C. 2021. Market-based Multi-robot coordination with HTN planning. In *Int. Conf. on Intelligent Robots and Systems (IROS)*.

Nunes, E.; and Gini, M. 2015. Multi-robot auctions for allocation of tasks with temporal constraints. In *AAAI Conference on Artificial Intelligence (AAAI)*. Austin, TX, USA.

Otte, M.; Kuhlman, M. J.; and Sofge, D. 2019. Auctions for multi-robot task allocation in communication limited environments. *Autonomous Robots* 44.

Zlot, R.; and Stentz, A. 2006. Market-based Multirobot Coordination for Complex Tasks. *The International Journal on Robotics Research* 25(1).

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

Solving POMDPs online through HTN Planning and Monte Carlo Tree Search

Robert P. Goldman
SIFT, LLC
319 1st Ave N., Suite 400,
Minneapolis, MN 55401, USA
rpgoldman@sift.net

Abstract

This paper describes our SHOPPINGSPREE HTN algorithm for online planning in Partially Observable Markov Decision Processes (POMDPs). SHOPPINGSPREE combines the HTN planning algorithm from SHOP3, extensions to SHOP3's representation to handle partial observability, and Monte Carlo Tree Search for efficient sampling in the problem space. This paper presents only the algorithm and initial notes on the implementation: this is work in progress.

1 Introduction

In this paper we describe SHOPPINGSPREE (named after a US TV game show and SHOP3), a technique for solving Partially Observable Markov Decision Processes (POMDPs) in an online fashion – that is, interleaving planning and execution – based on Monte Carlo Tree Search (MCTS), currently under development, building on the Hierarchical Task Network (HTN) planner, SHOP3. We describe a complete algorithm, but the implementation is still in very early stages: its implementation is still messy, made up primarily of patches to the existing version of SHOP3.However, the algorithm is promising because it enables us to combine HTN planning with sequential decision problems.

The key tenets of our approach are as follows: 1. We approach POMDPs as *games against nature*, where the system's objective is to execute an (approximately) optimal strategy against its environment. 2. The planning agent's "turns" in this game are the set of actions taken between observations. 3. The planning agent plans turn-by-turn ("online"), allowing us to avoid computing full policies, which can be exponentially large. 4. If myopic, turn-by-turn planning can be severely suboptimal. We use MCTS to provide lookahead and avoid myopic decision making. 5. To apply MCTS to POMDP planning, we combine planning in belief space with sampling in the base state space.

In this paper we briefly introduce POMDPs and the textbook "oil wildcatter" sequential decision problem. Again briefly, we summarize MCTS. Then we explain the extensions to SHOP3's Knowledge Representation (KR), and how to use it to represent a POMDP. Finally, we present our method for integrating HTN planning and MCTS, and conclude with notes on some limitations, mention of related work, and future directions.

A quick definition before we begin:

Definition 1 (POMDP) *A POMDP, $\mathcal{P} = \langle \mathcal{S}, s, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O} \rangle$ where 1. $\mathcal{S}$ is a finite set of states, $s \in S$ is the initial state;$\mathcal{A}$ is a finite set of actions; 2. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ is the state-transition function, assigning a probability distribution over successor states when an action, a is executed in state s; 3. $\mathcal{R}$ is the reward function, which may be defined over $\mathcal{S} \times \mathcal{A}$, and defines the reward, a real number reward received when a is executed in s. The reward of a finite trace is the sum of the rewards at each step in the trace. Equivalently, we use a cost function, in the work described here. 4. Ω is a set of observations that the agent may make; 5. $\mathcal{O} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \Pi(\Omega)$ is the observation function, which gives a probability distribution over the set of observations that may be received when the agent takes action a in state s_0 and the successor state (dictated by $\mathcal{T}$) is s_1 (Kaelbling, Littman, and Cassandra 1998, adapted).*

There are important subtypes of POMDP varying by time horizon. We address **only** *finite horizon* POMDPs.

A POMDP's solution is a *policy*: a mapping from belief states to action choices. We define an *optimal* policy as a policy that provides maximum expected utility. Note that the policy need not be explicitly computed. One way to think about POMDPs is as games against "nature," a stochastic opponent. That is the perspective we take here. Rather than computing a policy off-line and executing it, SHOPPINGSPREE computes its policy implicitly and on-line.

As a running example, we use the textbook "oil-wildcatter" decision problem (Raiffa 1968): The oil wildcatter needs to decide whether to drill or not. They don't know if their hole is **dry**, **wet**, or **soaking**. Their payoffs are given in Table 1. All payoffs are *net*: profit less $70,000 for drilling, if the wildcatter chooses to. For $10,000, the wildcatter can test the site's geological structure: no structure (NS), open (OS), or closed structure (CS). The structure is correlated with the likelihood that oil is present (Table 2). This problem may be formulated as a POMDP where $\mathcal{S} = \{\text{Dry}, \text{Wet}, \text{Soaking}\}$; $\Omega = \{\text{NS}, \text{OS}, \text{CS}\}$, and $\mathcal{A} = \{\text{test}, \text{drill}\}$, and $\mathcal{T}$ and $\mathcal{R}$ are from Tables 1 and 2. Expected rewards for some policies are given in Table 3.

2 Monte Carlo Tree Search (MCTS)

MCTS is a high-performance search method originally developed for game playing for games with extremely large

State	Act a_1	Act a_2
Dry (θ_1)	- $70,000	0
Wet (θ_2)	$50,000	0
Soaking (θ_3)	$200,000	0

Table 1: Monetary Payoffs for Oil Wildcatter problem (Raiffa 1968, pp. 35).

State	Seismic Outcome NS	OS	CS	Marginal Probability of state
Dry (θ_1)	.300	.150	.050	.500
Wet (θ_2)	.090	.120	.090	.300
Soaking (θ_3)	.020	.080	.100	.200
Marginal probability of seismic outcome	.410	.350	.240	1.000

Table 2: Joint and marginal probabilities for Oil Wildcatter problem (Raiffa 1968, pp. 35).

state spaces, such as Go. Our discussion here is heavily indebted to the excellent survey by Browne et al.(2012). Briefly, the job of MCTS is to estimate the value of each node in the top of the search tree, so that an agent can choose its actions by greedily taking the highest value choice. To do this, the algorithm must search the tree so that it can accurately estimate the value of early decisions on the eventual outcome (*e.g.*, estimate the value of an opening move in terms of its effect on the eventual outcome of a game). To search the tree, the algorithm must have a way of choosing a child at every node of the tree. MCTS splits its search into two phases: first by *tree policy*, then by *default policy*.

In large search spaces, states near the root of the search tree will be explored according to the tree policy, but since the search space prohibits complete exploration, when search reaches some depth threshold, the system will switch to the default policy. The tree policy is generally a choice that weights known estimates of child nodes – causing the system to *exploit* its knowledge of where value is to be found – against a term that weights parts of the tree that have not been visited often – causing the system to *explore* new parts of the search space. We use the popular UCT (Upper Confidence Bounds for Trees) (Kocsis and Szepesvári 2006) rule:

$$\arg\min_{a \in A} \frac{V(a)}{N(a)} + k\sqrt{\frac{2\ln N}{N(a)}} \quad (1)$$

where $V(a)$ is the mean value of node a, $N(a)$ is the visit count, and N is the visit count of the parent of a, and k is a constant. Currently, we use the original $k \equiv 1/\sqrt{(2)}$ (Kocsis and Szepesvári 2006). Depending on problem, other policy rules may be better than UCT and, in UCT, different values of k may be better. We leave this for future work.

The default policy will be some extremely cheap, largely random choice rule. At present, since we are working with small problems, SHOPPINGSPREE has no default policy.

Algorithm 1 General MCTS approach, reproduced from (Browne et al. 2012)

```
1: function MCTSSEARCH(s_0)
2:     create root node v_0 with state s_0
3:     while within computational budget do
4:         v_l ← TREEPOLICY(v_0)
5:         Δ ← DEFAULTPOLICY(s(v_l))
6:         BACKUP(v_l, Δ)
7:     end while
8:     return a(BESTCHILD(v_0))
9: end function
```

When a "rollout" has been completed by reaching a leaf node of the search tree, the value encountered at the leaf will be *backpropagated* to update the estimates of the value of nodes nearer the root of the tree, and their visit counts.

Finally, when some resource threshold is reached, the system will choose an action (or set of actions) to take, based on the highest expected value. In a game – including a game against nature like our oil wildcatter problem – thee the MCTS system will wait to receive the opponent's move (*e.g.*, the results of the seismic test), and then repeat the process. The process is summarized in Algorithm 1.

3 Knowledge Representation (KR)

The current version of SHOPPINGSPREE makesonly minimal extensions to the SHOP3 KR for domains and problems: hidden propositions, and uncontrollable actions and methods. Other than this, we use standard SHOP3 (Goldman and Kuter 2019) notation, which we review below.

SHOP3 primitive operators have a **head**, comprising an **operator name** and **parameters**. They also have **preconditions**, **add-list**, **delete-list**, and **cost function**. The add-list and delete-list are lists of literals, potentially containing parameter variables. The preconditions are more expressive than STRIPS or PDDL: supporting logical operators, finite quantification, and the invocation of arbitrary functions. We will make use of this expressive power below. The cost function can be an arbitrary function of the parameters and any variables bound in its preconditions. An example operator from the oil-wildcatter domain:

```
(:op (!do-drill)
  :precond (not (drilled))
  :add ((drilled))
  :delete ()
  :cost 70)
```

SHOP3 method definitions have a **head**, and **preconditions** as above and a **task network**. Note: variables in the head *and* in the preconditions are scoped over the task network. Task networks are made up of tasks and `:ordered` and `:unordered` networks. For example:

```
(:method (make-decisions)
 () ; empty preconditions
 (:ordered (decide-test)
           (decide-drill)
           (profit)))
```

Policy	Details	Expected Value	Rescaled EV
Don't test, don't drill	1×0	0	0.286
Don't test, drill	$0.5 \times -70 + 0.3 \times 50 + 0.2 * 200$	20	0.357
Test, drill iff CS	$-10 + (.09 \times 50) + (.1 \times 200)$	14.5	0.338
Test, drill if OS or CS	$-10 + (.09 \times 50) + (.1 \times 200) + (.12 \times 50) + (.08 \times 200)$	**36.5**	**0.416**

Table 3: Rewards for some policies, in thousands of dollars, and rescaled to between 0 and 1.

The above definition states that the `(make-decisions)` task can be rewritten into the task network, unconditionally, since its preconditions are empty.

SHOP3 domains may also have **axioms**, Prolog-style Horn clauses that are used when checking preconditions. The following axiom is used to compute the profit (`?p`) from drilling when the oil condition is `?o`:

```
(:- (drill-profit ?o ?p)
   ((= ?o dry) (= ?p 0))
   ((= ?o wet) (= ?p 120))
   ((= ?o soaking) (= ?p 270)))
```

SHOP3 permits a compressed representation allowing multiple Horn clauses with the same head (here `(drill-profit ?o ?p)`) to be combined.

Problems contain an initial state and task network:

```
(defproblem wildcatter ; problem name
  wildcatter ; planning domain name
  () ; initial state (empty)
  (oil-wildcatter)) ; initial task network
```

A SHOP3 planning problem is *solved* when its task network has been fully rewritten into a sequence of primitive actions that is executable (each action's preconditions are satisfied when it is executed). We will see, though, that generating plans is only a *subproblem* for SHOPPINGSPREE.

We have made minimal extensions to SHOP3's KR to model finite-duration POMDPs: 1. We allow propositions to be marked as `hidden` 2. We mark some methods as **stochastic**: these methods are allowed to read `hidden` propositions and invoke `random` in their preconditions. 3. We mark some operators as **uncontrolled**; these may read `hidden` propositions in their preconditions, and do not appear in the agent's history (see Section 4).

Here is an example of how these extensions are used in the oil wildcatter problem:

```
(:stochastic-method (init-oil~)
 (and (assign ?r (random 1.0d0))
      (oil-outcome ?r ?o))
 (:ordered (!init-oil~ ?o)))

(:op (!init-oil~ ?o)
 :add ((hidden (oil ?o)))
 :cost 0)

(:- (oil-outcome ?r ?o)
   ((< ?r 0.5) (= ?o dry))
   ((> ?r 0.5) (< ?r 0.8) (= ?o wet))
   ((> ?r 0.8) (= ?o soaking)))
```

`init-oil~` randomly samples from a categorical to find what the oil conditions then uses the uncontrolled operator `!init-oil~` to record the hidden literal. All uncontrolled tasks have the `~` character as suffix.

We represent the oil wildcatter problem as follows (full domain: https://pastebin.com/pUqLt2H6):

The top-level task is `oil-wildcatter`, which decomposes to `prepare-problem` and `make-decisions`.

`prepare-problem` decomposes to `init-oil~` followed by `init-test~`. `init-oil~`, is as described above.`init-test~` samples from a categorical (conditioned on the `oil`) and adds `(test-result` s`)`, for $s =$ `ns`, `os`, or `cs`. These are only revealed to the agent if it tests.

`make-decisions` expands to `test-decision` followed by `drill-decision`. Each of these expands to a decision to either perform or not perform the corresponding action. `!test` incurs the testing cost and reveals the value of o in `(hidden (test-result` o`))` by adding `(test-outcome` o`).drill` incurs the cost and accrues some income depending on o in `(hidden (oil` o`))`.

The way SHOPPINGSPREE handles partial observability is, in a way, the inverse of the one in Definition 1: state components default to being visible, and must be explicitly hidden, rather than needing to be explicitly observed.

4 Planning Algorithm

In this section, we describe how we have fused MCTS with SHOP3's HTN planning algorithm. As with a conventional planning problem, the input to SHOPPINGSPREE is a planning domain and problem (Section 3), but the notion of solution is quite different. The "solution" is a sequence of actions generated in an attempt to maximize the agent's utility (as defined in the domain and problem): see Alg. 1. At the moment, we have a very simple simulator, also based on the planning domain and state. In a real application one would connect the planning agent to effectors, and incorporate sensory inputs from the environment.

A simplified version of SHOP3's planning algorithm is given in Algorithms 2, and 3. At the top level, the "Find plans" procedure of Algorithm 2 is invoked with the initial state, the top-level task from the planning problem, and the empty set of bindings. For reasons of space, we do not discuass the basic SHOP3 algorithm in detail here. However, these pseudocode procedures were taken from our previous paper on SHOP3 (Goldman and Kuter 2019), interested readers can consult that paper for further details.

The first modification to the planning algorithm is to use the MCTS method for choosing options at nondeterministic choice points. These choice points occur at lines (5 in Alg. 2), (2 in Alg. 3), and (8 in Alg. 3). In each case, it is straightforward to adapt the MCTS bandit method to choose one of the alternatives from the finite set.

There is a complication, however: the scores for the MCTS algorithm must be tabulated at states that are *equiv-*

Algorithm 2 Simplified planning search algorithm.

```
1: procedure FIND PLANS(S, T, B)        ▷ state, tasklist, bindings
2:     if T = ∅ then
3:         return ()      ▷ No tasks: return empty action sequence.
4:     end if
5:     choose t ∈ T with no predecessors
6:     if t is primitive then
7:         o ← operator for t
8:         if o is applicable in S then
9:             S' ← result(o, S)
10:            T' ← T − t
11:            P ← FIND PLANS(S', T', B)
12:            return cons(o, P)
13:        else
14:            return FAIL
15:        end if
16:    else                                   ▷ t is a complex task
17:        < b, R' >= reduction(t, S)
18:        if b is FAIL then
19:            return FAIL
20:        else
21:            B' = apply(b, B)
22:            if B' is FAIL then
23:                ▷ Merge new bindings with incoming.
24:                return FAIL
25:            end if      ▷ Replace t with its expansion R' in T
26:            T' ← replace(t, R', T)
27:            return FIND PLANS(S, T', B')
28:        end if
29:    end if
30: end procedure
```

Algorithm 3 Task reduction procedure

```
1: procedure REDUCTION(t, S)                          ▷ task, State
2:     choose m a method for name(t)
3:     ▷ List of bindings from precondition query.
4:     b* = query(pre(m), S)
5:     if b* = ∅ then
6:         return FAIL
7:     else
8:         choose b ∈ b*          ▷ bindings from preconditions
9:         R ← task-net(m)
10:        R' ← apply(b, R)
11:        return b, R'
12:    end if
13: end procedure
```

alent with respect to knowledge, rather than at complete states. See line 3 of Alg. 4. This is necessary because the agent can only choose actions based on the state it observes, rather than based on complete knowledge. The oil wildcatter can only choose to drill or not based on the results of the test (if they have done it), not based on the full state of the world, including the `oil` predicate. On the other hand, when doing a rollout, the planner must take into account the full state in order to project the outcome and its value. Put differently, the *agent* can *choose* only based on the visible part of the state, and the value estimates it collects through MCTS. But the *MCTS sampling algorithm* must *sample* (project) based on the full state, whose evolution it simulates.

In SHOPPINGSPREE, "equivalent with respect to knowledge" is implemented by projecting SHOP3's state onto the set of visible propositions. The set of visible propositions are those propositions that are not `hidden` (see Section 3). In addition to this, the state index (b in Alg. 4) includes the *visible history* of the state. The visible history of the state is the path of actions from the initial state leading to this state, omitting the uncontrolled actions.

The visible history is an essential part of the table indexing because the POMDP policy that we are computing is not memoryless; MDPs have optimal policies that are memoryless (Puterman 1994), but POMDPs do not unless the state is taken to include the agent's belief state, as well as the world state. The tables that MCTS computes are approximations of the value function, not the agent's belief state.

Where Algs. 2 or 3 require a choice, the choice is made according to Alg. 4, which either initializes the choice table for the current choice and chooses an arbitrary alternative, or chooses the best decision, based on the current statistics, according to the MCTS rule chosen. For this reason, an MCTS rollout corresponds to the generation of a full plan, fitted to a particular outcome of the chance variables: in the case of the oil wildcatter problem, the `oil-state` and the outcome of a test (if the planner chooses to make one), given the `oil-state`. Because the plan is completed in a specific context in terms of the chance variables, it can be scored.

The score of each plan/rollout is backpropagated through the tree nodes of the search tree, and onto the corresponding statistics tables constructed as part of Algorithm 4. Per Equation 1, we increment the count of the action chosen at each table entry, and increment the cost entry with the cost of the full plan. If the search reaches a dead end – the current partial plan cannot be completed– we back-propagate $+\infty$.

The rollouts and backpropagation are the way that MCTS avoids the problems of myopic decision making we mentioned in the introduction. In the oil wildcatter problem, the planner can "see" that testing will provide information that will later be of use, rather than rejecting it because of the up-front cost. This kind of lookahead is more expensive and difficult under uncertainty than in classical planning.

The second modification is that this "planning algorithm" is not used as a conventional planner. Instead, it is essentially used as a way to populate the MCTS decision tables through planning and sampling. When it is time for plan execution, the system repeats the planning process, but this time, instead of using the tables to randomly generate a chosen action, the system takes the action dictated by the table: the one with the highest expected value (breaking ties randomly). Execution continues in this way until the system makes an observation, at which time the entire process repeats.

5 Conclusions

In this paper we have described an algorithm for HTN-based POMDP planning that combines the HTN planner, SHOP3, with Monte Carlo Tree Search. POMDP planning requires a combination of expressive power and sequential decision making in which observations and actions are interleaved, but where myopic planning and deterministic relaxations are

Algorithm 4 Choice method (Browne et al. 2012, Alg. 2, adapted)

```
C ← MCTSTable()
function CHOOSE(S, T, A)    ▷ state, choice type, alternatives
    b ← belief state for S
    if ⟨b, T⟩¬ ∈ C then                    ▷ new choice table entry
        C [⟨b, T⟩] ← new table for ⟨b, T⟩
        return random member of A
    end if
    if ∃c ∈ alternatives | c¬ ∈ C [⟨b, T⟩] then
        return c                                ▷ try untried option
    else
        return best element of C [⟨b, T⟩]
    end if
end function
```

insufficient to provide acceptable behaviors. Our work is still at an extremely early stage where we have developed the algorithm, but are still working to identify the best modeling techniques and to refine our method to achieve the best performance in the classes of problem that interest us.

Related work There is a great deal of work on applying MCTS to various sorts of planning. An obvious parallel to our work is the the application of MCTS to HTN by Wichlacz et al. (2020). The key difference here is that their work applies to classical HTN planning, not POMDPs.

SHOPPINGSPREE will not be competitive with other planning systems based on MCTS that aim to handle large POMDPs of standard structure (Silver and Veness 2010, *e.g.*,). Where we hope that SHOPPINGSPREE will shine is in problems with complex structures to the action space that will take advantage of HTN capabilities, where the sequential decision making is relatively simple, but bushy, and where we can profit from SHOP3's expressive power, allowing us to handle real-world complexities including object creation, numerical attributes, *etc.*.

Limitations A prominent limitation of this work is that it cannot handle situations where the HTN planner can get "off track" in the course of execution and not be able to complete a plan. Consider a case where the planner has chosen a method $M = T \rightarrow T_1, T_2, T_3$, to expand task T. Now imagine something goes wrong in the course of executing the subtasks of T_1. Since the planner has committed to M, it can no longer back up and consider alternatives, either alternatives to M, or alternatives to the current expansion of T_1. There are two ways to address this. The first is to ensure that the planning domain does not feature such dead ends. In our oil wildcatter example, as long as the discretization of the test results is complete, there are no dead ends.

An alternative to the no dead ends property is to support *plan repair*: the search space of the planner is expanded to consider repair operations when the execution of the current plan fails. We have developed an approach to plan repair for SHOP3 (Robert P. Goldman, Ugur Kuter, and Richard G. Freedman 2020) that we will incorporate in future work.

Acknowledgments This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under Contract No. FA8750-17-C-0184. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of DARPA, the Dept. of Defense, or the U.S. Government.

References

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1): 1–43. ISSN 1943-068X, 1943-0698. doi:10.1109/TCIAIG.2012.2186810. URL http://ieeexplore.ieee.org/document/6145622/.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: The Case of SHOP3. In *Proceedings of the 12th European Lisp Symposium*. Genova, Italy.

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence* 101(1-2): 99–134. ISSN 00043702. doi:10.1016/S0004-3702(98)00023-X. URL https://linkinghub.elsevier.com/retrieve/pii/S000437029800023X.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In Hutchison, D.; Kanade, T.; Kittler, J.; Kleinberg, J. M.; Mattern, F.; Mitchell, J. C.; Naor, M.; Nierstrasz, O.; Pandu Rangan, C.; Steffen, B.; Sudan, M.; Terzopoulos, D.; Tygar, D.; Vardi, M. Y.; Weikum, G.; Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *Machine Learning: ECML 2006*, volume 4212, 282–293. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-45375-8 978-3-540-46056-5. doi:10.1007/11871842_29. URL http://link.springer.com/10.1007/11871842_29.

Puterman, M. L. 1994. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. New York, NY: John Wiley & Sons, Inc.

Raiffa, H. 1968. *Decision Analysis: Introductory Lectures on Choices under Uncertainty*. Behavioral Science: Quantitative Methods. New York: Random House.

Robert P. Goldman; Ugur Kuter; and Richard G. Freedman. 2020. Stable Plan Repair for State-Space HTN Planning. In *Hierarchical Planning Workshop*. Nancy, France. URL https://hierarchical-task.net/HPlan/HPlan2020-paper4.pdf.

Silver, D.; and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. In Lafferty, J.; Williams, C.; Shawe-Taylor, J.; Zemel, R.; and Culotta, A., eds., *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc. URL https://proceedings.neurips.cc/paper/2010/file/edfbe1afcf9246bb0d40eb4d8027d90f-Paper.pdf.

Wichlacz, J.; Höller, D.; Torralba, A.; and Hoffmann, J. 2020. Applying Monte-Carlo Tree Search in HTN Planning. In *Proceedings SoCS*.

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

The Complexity of Flexible FOND HTN Planning

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

Abstract

Hierarchical Task Network (HTN) planning is an expressive planning formalism that has often been advocated as a first choice to address real-world problems. Yet only a few extensions exist that can deal with the many challenges encountered in the real world. One of them is the capability to express uncertainty. Recently, a new HTN formalism for Fully Observable Nondeterministic (FOND) problems was proposed and studied theoretically. In this paper, we lay out limitations of that formalism and propose an alternative definition, which addresses and resolves such limitations. We conduct a complexity study of an alternative, more flexible formalism and provide tight complexity bounds for most of the investigated special cases of the problem.

1 Introduction

Hierarchical Task Network (HTN) planning is a planning approach that focuses on problem decomposition. Compound tasks describe abstract activities, and the domain model describes how they can be carried out by exploiting decomposition methods, pre-defined recipes stating by which plans such compounds tasks may be implemented.

The goal is to find a plan – a sequence of primitive tasks that can be executed – which successfully implements the initially given initial compound tasks defining the planning problem. Because this task hierarchy may be exploited to encode expert knowledge and thus gives another means of modelling a problem, and because it may be used to also *exclude* undesired solutions, it has been used in many different practical scenarios (Bercher, Alford, and Höller 2019).

In particular when facing real-world problems, we may face challenges and limitations when modelling the world with a fully deterministic model. Often, the world may be dynamically changing (Patra et al. 2020), be only partially observable (Richter and Biundo 2017), or require reasoning over actions with non-deterministic outcomes (Kuter and Nau 2004; Kuter et al. 2005, 2009).

Most of these works in the realm of HTN planning and uncertainty focused on developing planners that produce classical policies to (non-hierarchical) Fully Observable Nondeterministic (FOND) problems. Task hierarchies were exploited as mere control knowledge, but solutions generated need not to be refinements of the initial compound tasks – which can be seen from the solution structure, which is still a simple policy, i.e. a state/action mapping, which is not complex enough to represent arbitrary long plans as solutions for HTN problems due to their undecidability (Erol, Hendler, and Nau 1996).

Recently, an extension to such FOND policies was proposed capable of capturing solutions to FOND HTN problems where solutions need to be refinements of an initial task network, just like in standard deterministic (i.e., FOD) HTN planning (Chen and Bercher 2021). For this FOND HTN formalisation, we studied the computational complexity of various standard HTN problems, as well as the impact of two ways when uncertainty is taken into account: during planning time (linearisation-dependent solutions), or during execution time (outcome-dependent solutions). The latter more flexible solution definition states that a primitive (partially ordered) plan is regarded a solution when after the execution of any (non-deterministic) action one is still able to continue executing the plan by picking an appropriate action depending on previous action outcomes until all actions of the plan are successfully executed. This definition assumes that one still needs to compute one such primitive plan (policy) before action outcomes may be taken into account. Thus we will denote this formalism as FOND^{FM} HTN, indicating that they use "fixed methods" in their solutions.

Contributions

In this paper we propose another more flexible FOND HTN formalisation where policies are no longer defined based on primitive plans like that of Chen and Bercher (2021), but allow a selection of decomposition methods for compound tasks in the policy, which therefore allows choice of decomposition methods depending on the outcome of executed tasks. We thus denote the novel formalism FOND^{MP} HTN which indicate that we use "method-based policies".

We begin by introducing the alternative formalisation followed by a comprehensive complexity study on the plan existence problem. Similar to the studies made by Chen and Bercher (2021), our results are all one class harder for most subproblems with restricted recursion in comparison to standard FOD HTN planning. We also propose search algorithms for solving FOND^{MP} HTN problems which are exploited in our proofs, and may also serve as decision procedures to be implemented in the future.

2 Formalism

The definitions of the FOND$^{\text{MP}}$ HTN planning domain are the same of that for FOND$^{\text{FM}}$ HTN planning by Chen and Bercher (2021) by extending definitions for deterministic HTN planning (Geier and Bercher 2011; Bercher, Alford, and Höller 2019) to include nondeterministic actions. Due to space constraints we will list Def. 2.1 to 2.4 by Chen and Bercher (2021) for a FOND$^{\text{MP}}$ HTN domain and problem and only provide additional definitions as required.

Definition 2.1. A *task network* $\mathtt{tn}$ is a tuple $\langle T, \prec, \alpha \rangle$ where
- T is a finite set of *task id symbols* or *labels*,
- $\prec \subseteq T \times T$ is a strict partial order on T,
- $\alpha : T \to \mathcal{N}$ maps a task id to some task name in the set of task names $\mathcal{N}$.

We also define an equivalence between two task networks which might have the same underlying structure but different task id symbols. Specifically, we say that two task networks $\mathtt{tn} = \langle T, \prec, \alpha \rangle$ and $\mathtt{tn}' = \langle T', \prec', \alpha' \rangle$ are *isomorphic* if there exists a bijection $\sigma : T \to T'$ between task id symbols where for all $t_1, t_2 \in T$, we have $(t_1, t_2) \in \prec$ iff $(\sigma(t_1), \sigma(t_2)) \in \prec'$ and $\alpha(t) = \alpha'(\sigma(t))$ for all $t \in T$. This definition of equivalence will be required for building well defined FOND$^{\text{MP}}$ HTN problems and solutions.

We also have notation for special task networks: let

$$\mathtt{tn}(a) = \langle \{t\}, \emptyset, \{(t, a)\} \rangle, \mathtt{tn}_\emptyset = \langle \emptyset, \emptyset, \emptyset \rangle$$

denote the task network for a single task name a and the trivial task network respectively.

Definition 2.2. An *HTN domain* $\mathcal{D}$ is a tuple $\langle \mathcal{F}, \mathcal{N}_P, \mathcal{N}_C, \delta, \mathcal{M} \rangle$ where
- $\mathcal{F}$ is a finite set of *facts*,
- $\mathcal{N}_P$ is a finite set of *primitive task names*,
- $\mathcal{N}_C$ is a finite set of *compound task names*,
- $\delta : \mathcal{N}_P \to \mathcal{A}$ is an action mapping,
- $\mathcal{M}$ is a finite set of *decomposition methods*,

with $\mathcal{N}_P \cup \mathcal{N}_C = \mathcal{N}$ disjoint and $\mathcal{A} \subseteq 2^{\mathcal{F}} \times 2^{2^{\mathcal{F}} \times 2^{\mathcal{F}}}$ denoting the set of nondeterministic primitive tasks. A *primitive task* or *action* is a tuple of preconditions and effects $a = (\text{pre}(a), \text{eff}(a))$ with $\text{eff}(a) = \{(\text{add}_i(a), \text{del}_i(a)) \mid 1 \leq i \leq n\}$ for n dependent on a and $\text{pre}(a), \text{add}_i(a), \text{del}_i(a) \subseteq \mathcal{F}$. However, for ease of notation whenever we have a deterministic action ($|\text{eff}(a)| = 1$) we will use $(\text{pre}(a), \text{add}(a), \text{del}(a))$ as to remove redundant brackets. The definitions of FOND$^{\text{MP}}$ HTN planning actions and non-hierarchical planning actions are equivalent so we also formalise the mechanisms for applying actions to states.

Define a set of states $S = 2^{\mathcal{F}}$ corresponding to subsets of $\mathcal{F}$. Let $\tau : \mathcal{A} \times \mathcal{S} \to \{\top, \bot\}$ denote *executability* of an action at a state where $\tau(a, s) = \top$ for $\text{pre}(a) \subseteq s$ and $\tau(a, s) = \bot$ otherwise. For ease of notation, we also define the executability function τ for primitive task names and id symbols in the obvious way by $\tau(n, s) = \tau(\delta(n), s)$ and $\tau(t, s) = \tau(\alpha(t), s) = \tau(\delta(\alpha(t)), s)$ for $n \in \mathcal{N}_P$ and $t \in T$ respectively. We also define an application function $\gamma : \mathcal{A} \times \mathcal{S} \to 2^{\mathcal{S}}$ where for $a \in \mathcal{A}, s \in \mathcal{S}$ we have $\gamma(a, s)$ undefined if $\tau(a, s) = \bot$ and otherwise we have

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

Similarly define γ on primitive task names and id symbols by $\gamma(n, s) = \gamma(\delta(n), s)$ and $\gamma(t, s) = \gamma(\alpha(t), s) = \gamma(\delta(\alpha(t)), s)$. Lastly, we say that $\mathtt{tn} = \langle T, \prec, \alpha \rangle$ is a *primitive task network* if all its tasks are primitive, meaning that for all $t \in T$, we have $\alpha(t) \in \mathcal{N}_P$.

Definition 2.3. Define $m = (c, \mathtt{tn}_m)$ with $c \in \mathcal{N}_C$ and $\mathtt{tn}_m = \langle T_m, \prec_m, \alpha_m \rangle$ to be a *(decomposition) method*. We can apply m to $\mathtt{tn}_1 = \langle T_1, \prec_1, \alpha_1 \rangle$ if there exists some $t \in T_1$ where $\alpha_1(t) = c$, and in this case we say m decomposes t in $\mathtt{tn}_1$ to generate a task network $\mathtt{tn}_2 = \langle T_2, \prec_2, \alpha_2 \rangle$ with

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

where $T_1' = T_1 \setminus \{t\}$ and $\mathtt{tn}_m'$ is a task network isomorphic to $\mathtt{tn}_m$ such that $T_1' \cap T_m' = \emptyset$. The $|_{T_1'}$ symbol denotes restriction on the map α and ordering $\prec$ in the canonical way to only tasks in T_1'. The requirement for T_1' and T_m' disjoint is such that $\prec_2$ is still partial and α_2 is well defined. We denote this method application by

$$\mathtt{tn}_1 \to_m^t \mathtt{tn}_2.$$

Definition 2.4. An *HTN problem* $\mathcal{P}$ is a tuple $\langle \mathcal{D}, s_I, \mathtt{tn}_I \rangle$ with $\mathcal{D}$ a FOND$^{\text{MP}}$ HTN domain, $s_I \in 2^{\mathcal{F}}$ an initial state and $\mathtt{tn}_I$ an initial task network.

With a FOND$^{\text{MP}}$ HTN problem in hand, we now provide explicit definitions for what a plan or solution means. Similar to FOND$^{\text{FM}}$ HTN planning, we employ policies to define a solution. The difference between the two formalisms lies in how decomposition plays into a solution. In FOND$^{\text{FM}}$ HTN planning, solutions are defined by fixing a sequence of decomposition methods to apply on the initial task network and then constructing a policy for the acquired primitive task network. On the other hand, we will integrate methods into our policy, meaning that methods may be applied at different times depending on nondeterministic task effects.

Although it appears that the latter idea is more flexible by integrating decomposition into online execution, it has a few drawbacks: policies can grow arbitrarily large and online execution is hard. This arises from how we define a policy to take as input a task network and state, where the set of task networks is possibly unbounded in contrast to outcome-dependent solutions of FOND$^{\text{FM}}$ HTN problems which define policies for primitive task networks only using a lookup table of a current state and previously executed tasks. The latter is always bounded by noticing that there are only a finite number of states and subsets of tasks to account for.

Definition 2.5. Let $\mathcal{D}$ be a FOND$^{\text{MP}}$ HTN domain. A *policy* π is a partial function $\pi : TN \times \mathcal{S} \to T \times \mathcal{M}'$ where *TN* is the set of all possible task networks, T is the union of the sets of tasks in the task networks of *TN* and $\mathcal{M}' = \mathcal{M} \cup \{\varepsilon\}$. Moreover, $\langle (\mathtt{tn}, s), (t, m) \rangle \in \pi$ for $\mathtt{tn} = \langle T, \prec, \alpha \rangle$ only if $t \in T$ and
- if t is primitive, $m = \varepsilon$, and
- if t is compound, $m = (\alpha(t), \mathtt{tn}')$ is a method of $\mathcal{D}$.

We also impose the condition on a policy that for all pairs $\langle(\mathsf{tn}_1, s_1), (t_1, m_1)\rangle, \langle(\mathsf{tn}_2, s_2), (t_2, m_2)\rangle \in \pi$, if $s_1 = s_2$, then tn_1 and tn_2 are not isomorphic. This condition is required to create a well defined notion of execution of a policy as we shall now describe.

Execution of a policy for FOND STRIPS planning is described as a reactive execution loop that executes actions based on a survey of the state of the world, which shall also be made explicit for FOND$^{\text{MP}}$ HTN planning for a given task network tn_I and state s_I in Algorithm 1.

Algorithm 1: Policy Execution Procedure

```
(tn, s) ← (tn_I, s_I);
while InstructionExists(π, tn, s) do
    (t, m) ← GetInstruction(π, tn, s);
    if m = ε then
        tn ← Remove(tn, t);
        Execute(t);
        s ← SenseCurrentState();
    else
        tn ← Decompose(tn, t, m);
```

The function $\texttt{InstructionExists}(\pi, \mathsf{tn}, s)$ returns true if tn is not the empty task network and there exists a task network tn' that is isomorphic to tn such that $\pi(\mathsf{tn}', s)$ exists. $\texttt{GetInstruction}(\pi, \mathsf{tn}, s)$ returns $\pi(\mathsf{tn}', s)$, assuming that $\texttt{InstructionExists}(\pi, \mathsf{tn}, s)$ is true. $\texttt{Remove}(\mathsf{tn}, t)$ returns the task network tn without task t with the canonical restrictions to $\prec$ and α as described in Def. 2.4 by Chen and Bercher (2021) and $\texttt{Decompose}(\mathsf{tn}, t, m)$ the task network we get when m decomposes t in tn. Lastly, $\texttt{SenseCurrentState}()$ returns the state of the world.

Having defined a mechanism to execute FOND task networks, we can now describe and formalise FOND$^{\text{MP}}$ HTN solution criteria. We will define weak, strong and strong cyclic solutions as is canonical to non-hierarchical nondeterministic planning (Cimatti et al. 2003) and also in YoYo, a planner which integrates HTN planning for solving such planning problems (Kuter et al. 2005, 2009). To formalise these concepts we will define the execution structure of a policy as a graph and use this graph structure to define our solutions.

Definition 2.6. Let $\mathcal{P}$ be a FOND$^{\text{MP}}$ HTN problem. Let the tuple $L = \langle \mathcal{U}, \mathcal{V}\rangle$ where $\mathcal{U} \subseteq \mathit{TN} \times \mathcal{S}$ and $\mathcal{V} \subseteq (\mathit{TN} \times \mathcal{S}) \times (T \times (\mathcal{M} \cup \{\varepsilon\})) \times (\mathit{TN} \times \mathcal{S})$ are minimal sets satisfying the conditions $(\mathsf{tn}_I, s_I) \in \mathcal{U}$, and if $(\mathsf{tn}, s) \in \mathcal{U}$ and $\pi(\mathsf{tn}, s) = (t, m)$ then

1. if t is primitive, for all $s' \in \gamma(t, s)$ we have $(\mathsf{tn} \setminus t, s') \in \mathcal{U}$ and $((\mathsf{tn}, s), (t, m), (\mathsf{tn} \setminus t, s')) \in \mathcal{V}$,
2. if t is compound, we have $(\mathsf{tn}', s) \in \mathcal{U}$ and $((\mathsf{tn}, s), (t, m), (\mathsf{tn}', s)) \in \mathcal{V}$ where $\mathsf{tn} \to^t_m \mathsf{tn}'$.

The *execution structure* induced by a policy π is the tuple $[L] = \langle [\mathcal{U}], [\mathcal{V}]\rangle$ where $[\mathcal{U}]$ is the set $\mathcal{U}$ quotient out by the relation $(\mathsf{tn}, s) \sim (\mathsf{tn}', s)$ iff tn and tn' are isomorphic and $[\mathcal{V}]$ is the collapsed relation corresponding to $[\mathcal{U}]$.

For ease of notation, we will omit the equivalence relation notation (i.e. the square brackets) for an execution structure. We can also view an execution structure L as a directed graph with nodes represented by elements in $\mathcal{U}$ and directed edges by elements in $\mathcal{V}$. Define (tn_I, s_I) to be an *initial node* and any $(\mathsf{tn}, s) \in \mathit{TN} \times \mathcal{S}$ to be a *terminal node* if it has no outgoing edges, and a *goal node* if $\mathsf{tn} = \mathsf{tn}_\emptyset$. We now proceed to define the three solution criteria.

Definition 2.7. Let $\mathcal{P}$ be a FOND$^{\text{MP}}$ HTN problem and tn a task network. Let π be a policy with execution structure $L = \langle \mathcal{U}, \mathcal{V}\rangle$. We say that π is

1. a *weak solution* if L is finite and there exists a terminal node of L that is a goal node,
2. a *strong cyclic solution* if every terminal node of L is a goal node,
3. a *strong (acyclic) solution* if L is finite and acyclic and every terminal node of L is a goal node.

Another way of interpreting the solution criteria is looking at how Algorithm 1 terminates: weak solutions sometimes terminate and if it does, it has a chance of terminating with an empty task network, strong solutions always terminate with an empty task network (hence the requirement for acyclic L), and strong cyclic solutions eventually terminate.

Practically, strong solutions are the most reliable as they guarantee the goal condition be met in a finitely many steps. This is followed by strong cyclic solutions which also guarantee that eventually we reach the goal condition or equivalently, we never fail. However, execution can be arbitrarily long. Finally, weak solutions are as their name suggests 'weak' in the sense that sometimes they do not even reach the goal condition. Thus, we can see that strong solutions are a special case of strong cyclic solutions which are in turn a special case of weak solutions.

Problem Classes

Given that standard HTN planning is undecidable (Erol, Hendler, and Nau 1996), studies have been made to find problem subclasses can be decided. We list the commonly studied subclasses here (Erol, Hendler, and Nau 1996; Alford et al. 2012; Alford, Bercher, and Aha 2015). We will define *stratifications* proposed by Alford et al. (2012) to help define the latter two.

Definition 2.8. An HTN problem $\mathcal{P}$ is *primitive* if tn_I is primitive. Note that sets $\mathcal{N}_C$ and $\mathcal{M}$ are now irrelevant.

Definition 2.9. An HTN problem $\mathcal{P}$ is *regular* if for its initial task network $tn_I = \langle T, \prec, \alpha\rangle$ and for all its methods $(c, \langle T, \prec, \alpha\rangle) \in \mathcal{M}$ it holds that there is at most one compound task in T, and if $t \in T$ is compound, it is the *last* task, meaning that for all $t' \in T$ with $t' \neq t$ we have $t' \prec t$.

Definition 2.10. A *stratification* on a set S is a total order $\leq$ on S. An inclusion-maximal subset $C \subseteq S$ is a *stratum* if for all $x, y \in C$ both $x \leq y$ and $y \leq x$ holds.

Definition 2.11. An HTN problem $\mathcal{P}$ is *acyclic* if no compound task can reach itself via decomposition. More formally, we can define a stratification on $\mathcal{N}_C$ in $\mathcal{P}$ with $c \leq c'$ if there exists a method $(c, \langle T, \prec, \alpha\rangle) \in \mathcal{M}$ and $\alpha(c') \in T$, and for all $c, c' \in \mathcal{N}_C$, if $c \leq c'$, then $c' \not\leq c$.

Definition 2.12. An HTN problem $\mathcal{P}$ is *tail-recursive* if we can define a stratification on $\mathcal{N}_C$ of $\mathcal{P}$ where for all methods $(c, \langle T, \prec, \alpha \rangle)$ it holds that if there exists a last compound task $t \in T$, then we have $\alpha(t) \leq c$, and for any non-last compound task $t \in T$, we have $\alpha(t) \leq c$ and $c \not\leq \alpha(t)$.

Note by definition that primitive, regular and acyclic problems are all special cases of tail-recursive problems. We also use the same definitions to describe decomposition methods. For example a regular method is a method whose task network has at most one compound task which has to be last.

3 Search Algorithms

In this section, we will describe two algorithms for determining plan existence of a given FOND$^{\text{MP}}$ HTN problem. The motivation for doing so is simple: to provide baseline algorithms for applications and to aid with membership proofs in our complexity proofs in Section 4. Although not optimal, they are very canonical in the sense that they are extensions of other baseline algorithms for planning problems.

Alternating Progression Search

The first algorithm extends progression search which is considered the canonical search algorithm for solving HTN problems (Alford et al. 2012; Höller et al. 2018; Höller et al. 2020) and also employed in efficient HTN planners such as SHOP, SHOP2 and SHOP3 (Nau et al. 1999, 2003, 2005; Goldman and Kuter 2019). We extend the algorithm by introducing 'universal' vertices to the graph, similarly to universal states of an ATM or AND nodes of an AND/OR-tree, to deal with nondeterminism.

Algorithm 2: Alternating Strong Progression Search

```
1  Procedure StrongPlanExistence (tn, s, M, V):
2      if tn = tn_∅ then return true;
3      if (tn, s) ∈ V then return false;
4      V ← V ∪ {(tn, s)};
5      guess a first task t in tn;
6      if t is a primitive task then
7          if not τ(t, s) then return false;
8          tn ← Remove(tn, t);
9          return for-all s' ∈ γ(t, s)
               StrongPlanExistence (tn, s', M, V);
10     else
11         guess a method m in M;
12         tn ← Decompose(tn, t, m);
13         return StrongPlanExistence (tn, s, M, V);
```

Algorithm 2 provides the procedure for determining plan existence. Given a FOND$^{\text{MP}}$ HTN problem $\mathcal{P} = \langle \mathcal{D}, s_I, \mathsf{tn}_I \rangle$ with $\mathcal{D} = \langle \mathcal{F}, \mathcal{N}_P, \mathcal{N}_C, \delta, \mathcal{M} \rangle$, the alternating procedure StrongPlanExist$(\mathsf{tn}_I, s_I, \mathcal{M}, \emptyset)$ determines if a strong solution for $\mathcal{P}$ exists. The meaning of the input variables $\mathsf{tn}, s, \mathcal{M}$ are straightforward. The product set V stores previously progressed task networks and visited states in order to detect cycles and deal with them.

The given ATM progression algorithm is an extension of the textbook progression algorithm used for classical HTN planning to our FOND$^{\text{MP}}$ HTN setting. Progression is a search algorithm which makes nondeterministic guesses for choosing whether to execute a random first primitive task or to decompose a compound task. By first task of a task network tn, we mean any task that has no predecessors in tn. After doing so, we remove the chosen task from the task network and change the state if the chosen task was primitive. If a solution exists, then by choosing the correct progression steps, we will end up with an empty task network and satisfy the solution criterion. To extend this concept to nondeterministic domains, we make use of universal states of alternating Turing machines to recursively check whether all possible progressed task networks from an executed nondeterministic action contribute to a solution.

Line 2 checks whether we have progressed away the task network and hence have reached an accepting state of the alternating computation tree. Line 3 checks whether we have visited the task network-state tuple before and enters a rejecting state. Line 4 then updates the previous task network-state tuples. Line 5 makes a nondeterministic choice[1] of a task t with no predecessors in tn.

The remainder of the algorithm performs the progression procedure depending on whether t is primitive or compound. If t is primitive, lines 7 to 9 checks whether t is executable at the progressed state s and if so proceeds to remove t from the progressed task network and then recursively calls the function "for all" possibly progressed states as given by $\gamma(t, s)$. The **for-all** statement represents entering a universal state for an alternating turing machine encoding. A more high level interpretation is that we return the logical conjunction of the StrongPlanExistence procedure for all possible progressed states. If t is compound, we guess a method for t and expand the task network at t with such method and proceed with the progression algorithm.

Note that this algorithm can be determinised by replacing nondeterministic choices with branching as described in Alg. 1 by Höller et al. (2018) and similarly replacing the **for-all** statements in the canonical way. In fact, the optimisation described in Alg. 2 in the same study for reducing branching from decomposition methods in this determinised algorithm can also be applied here. The reason we do not provide the deterministic version of the algorithm is to emphasise the usual tools (alternation) required to deal with nondeterministic tasks and for complexity analysis later.

Figure 1 provides a visualisation of the high level computation tree associated with the algorithm. We provide the abstract primitive problem it solves as follows. Let $\mathcal{D} = \langle \{1, 2\}, \{a, b, c\}, \emptyset, \delta, \emptyset \rangle$ with δ defined by $a \mapsto (\emptyset, \{(\{1\}, \emptyset), (\{2\}, \emptyset)\}), b \mapsto (\{2\}, \{1\}, \emptyset), c \mapsto (\{1\}, \{2\}, \emptyset)$. Then we define the problem by $\mathcal{P} = \langle \mathcal{D}, s_\emptyset, \mathsf{tn}_I \rangle$ where $s_\emptyset = \emptyset$ and tn_I is the task network of totally ordered primitive task names a, b, c.

Given that general HTN planning is undecidable, it is not necessarily the case that the following algorithm terminates though we will show later that the algorithm terminates for certain problem subclasses. We can also modify the algo-

[1] Using an oracle to find the right choice, contrary to deterministic search where we find the correct choice via branching.

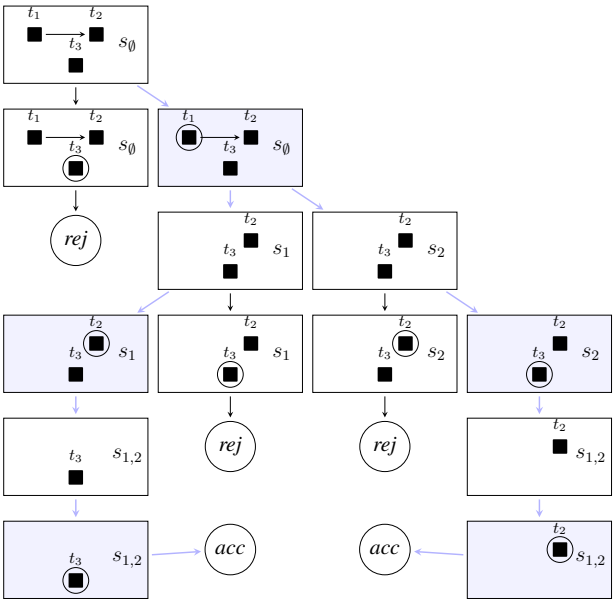


Figure 1: Visualisation of the alternating progression search algorithm. Denote $s_1 = \{1\}, s_2 = \{2\}, s_{1,2} = \{1, 2\}$.

rithm to allow for strong cyclic solutions by using the set V of visited problem subclasses but this will not be provided explicitly as it is not used for complexity proofs later.

Rectangular nodes in the figure represent search nodes consisting of the currently progressed task network and state. We omit the set V in the visualisation as there we do not have to worry about cycles for primitive problems. Black squares indicate primitive tasks and circles the selection of a task by line 5 of the algorithm. Blue nodes indicate universal states where we have to check that all children nodes are accepting, while the other rectangular nodes indicate existential states. Blue arrows indicate the subtree of the computation tree corresponding to a strong solution.

Bounded Graph Search

In addition to progression search, there is another search technique we can use to determine plan existence if we assume that the number of reachable task networks under progression and state combinations are bounded (e.g. primitive, acyclic, regular and tail-recursive problems). The main idea is that we can generate a bounded search space in the form of a graph (in contrast to a search tree) for a FOND$^{\text{MP}}$ HTN problem. Another way of interpreting this is that we compile a FOND$^{\text{MP}}$ HTN problem into a state transition system with initial and goal states and solve the compiled problem similarly to how Cimatti et al. (2003) generates the whole search space as a graph for a non-hierarchical FOND planning problem and uses such graph to solve the problem. Specifically, let $\langle S, A, I, G\rangle$ be a state transition system with S a set of states, $A \subseteq S \times 2^S$ a set of nondeterministic actions defined with an action defined as a tuple $(s_\alpha, \{s_1, \ldots, s_n\})$ which when applied in s_α can progress to any of the states s_1 to s_n. Next, we have $I \in S$ an initial state and $G \subseteq S$ a set of goal states. The definitions of strong and strong cyclic solutions are similar to that for propositional planning.

To compile a FOND$^{\text{MP}}$ HTN problem into such a system $\langle S, A, I, G\rangle$, we begin by letting S be the set of all possible *reachable task networks* and state tuples. Specifically, the set of reachable task networks TN_R for a problem is defined to be the set of task networks that can be obtained from the initial task network by applying a sequence of first tasks or methods, quotient out by isomorphism. To minimise the size of TN_R, we apply methods to compound tasks which have no predecessors. We will call S the set of *subproblems* given that they can be viewed as HTN problems with the same domain $\mathcal{D}$ and their task networks are part of a solution to the initial task network.

To consider an example, suppose we have a regular HTN problem. Then TN_R includes the initial task network $\mathtt{tn}_I$, the task networks for each method and all task networks that can be reached from $\mathtt{tn}_I$ by some number of primitive or compound task application. This is because under a canonical progression algorithm, we have at most one compound task in the current task network, meaning that all task networks must be some sub task network of a task network in method. Thus for regular problems, TN_R is bounded exponentially.

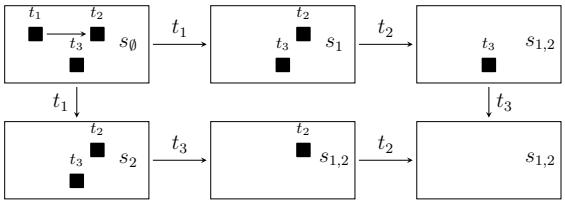


Figure 2: The whole search space of a FOND HTN problem.

Then $I = (\mathtt{tn}_I, s_I)$ and $G = \{(\mathtt{tn}_\emptyset, s) \mid s \subseteq \mathcal{F}\}$ denote the initial and goal states of the compiled problem. Then we define actions by looping through all $\sigma_\alpha = (\mathtt{tn}_\alpha, s_\alpha) \in S$ as follows. For each first task t in $\mathtt{tn}_\alpha$,

- if t is primitive, define an action (transition) $a = (\sigma_\alpha, \{\sigma_i = (\mathtt{tn}_\alpha \setminus \{t\}, s_i) \mid s_i \in \tau(t, s_\alpha)\})$,
- else for each method applicable to t, define an action $a = (\sigma_\alpha, \{\sigma_\beta = (\mathtt{tn}_\beta, s_\alpha)\})$ where $\mathtt{tn}_\alpha \rightarrow^t_m \mathtt{tn}_\beta$.

The main idea of such actions is that they connect HTN subproblems depending on if one can reach one subproblem from the other corresponding to an execution of some task.

Figure 2 illustrates the graph associated with the compiled HTN problem described in Section 3 on the alternating progression search. Rectangular nodes once again represent subproblems which are now the states of the compiled classical planning problem and directed edges representing actions and their effects.

To solve the system viewed as a non-propositional planning problem, we employ algorithms for weak, strong and strong cyclic planning in Sections 3 and 4 by Cimatti et al. (2003). All of them run in polynomial time with respect to the size of the graph as they search the graph a number of times to build up a solution.

Now we investigate the complexity of the algorithm by looking at the runtime of the two main steps: building the

graph and solving it. As mentioned in the previous paragraph, solving takes polynomial time with respect to the size of the graph. Building the graph is a bit more involved given that we have to check for graph isomorphism for equality of nodes. If checking for equality was only constant time, then the time it takes to build the graph is at least exponential in $\mathcal{F}$ as there are exponentially many reachable states, and bounded above polynomially by the size of TN_R. This is because we can build all $2^{|\mathcal{F}|} \cdot |TN_R|$ nodes first then for each node check which other node is reachable from it. Given that there are quadratically many directed edges between nodes, this means that building would take at least exponential time (polynomial with respect to the number of nodes which is exponential), in the order of $2^{|\mathcal{F}|} \cdot |TN_R|$.

Now, if we replace equality checking with graph isomorphism, we get complexity of order

$$2^{|\mathcal{F}|} \cdot |TN_R| \cdot f(\max_{\mathsf{tn} \in TN_R} |\mathsf{tn}|)$$

where $f(n)$ denotes the complexity for solving graph isomorphism for graphs with n vertices. A safe but loose upper bound for f is the exponential function with a brute force algorithm for checking task network isomorphism. In our membership proofs of certain HTN problem classes later in Section 4, the order of $|TN_R|$ will range from polynomial to double exponential with the upper bound on the size of a reachable task network $\max_{\mathsf{tn} \in TN_R} |\mathsf{tn}|$ being ranging from polynomial to exponential. This means that the complexity we will encounter for this algorithm varies between exponential and double exponential.

4 Complexity

This section will cover all the complexity results for our FOND$^{\text{MP}}$ HTN formalism. Due to space constraints, most hardness proofs will be given as sketches. We show that weak/primitive problems are equivalent to weak/primitive problems for FOND$^{\text{FM}}$ HTN planning and thus have the same complexities. For strong and strong cyclic problems we have that all acyclic, regular, and tail-recursive classes are made one step harder from their classical counterparts.

We begin by describing how weak and primitive FOND$^{\text{MP}}$ HTN and FOND$^{\text{FM}}$ HTN planning problems separately have the same semantic definitions which in turn means that the complexity for FOND HTN planning is equivalent as that for FOND$^{\text{FM}}$ HTN from which there exist results by Chen and Bercher (2021). Specifically, we show that the definitions for weak solutions are equivalent, and also for strong solutions when problems are primitive as both formalisms share the same definitions for domains and problems.

Proposition 4.1. *The definitions for primitive FOND^{MP} HTN and primitive FOND^{FM} HTN planning are equivalent.*

Proof. First observe a primitive FOND$^{\text{MP}}$ HTN policy no longer requires instructions for method applications. Thus, we only need to show that the policies consisting of only primitive task execution for the respective problems are equivalent. This can be noticed by viewing weak solutions for both formalisms as a sequence of tasks that can be executed for favourable nondeterministic effects. Given a sequence, a policy can be formed for either formalism. □

Corollary 4.2. *Let $\mathcal{P}$ be a partially (totally ordered) primitive FOND^{MP} HTN problem. Deciding whether $\mathcal{P}$ has a strong or strong cyclic solution is* PSPACE*-complete (in* P*).*

Proof. We notice that strong and strong cyclic solutions collapse for primitive problems as the same task network cannot be reached more than once due to the absence of methods. Then we get our complexity from Prop. 4.1 above and Thm. 4.8 and 5.1 by Chen and Bercher (2021). □

Proposition 4.3. *The definitions for weak FOND^{MP} HTN and weak FOND^{FM} HTN planning are equivalent.*

Proof. This can be realised by noticing that we can choose the methods corresponding to a trace of a weak FOND$^{\text{MP}}$ HTN solution for a weak FOND$^{\text{FM}}$ HTN solution. Conversely, we can construct a weak FOND$^{\text{MP}}$ HTN solution by first expanding the methods for a weak FOND$^{\text{FM}}$ HTN solution and then creating a policy corresponding to a FOND$^{\text{FM}}$ HTN policy for the expanded primitive task network. □

As a direct consequence of this, we have that the complexity for weak FOND$^{\text{MP}}$ HTN planning is equivalent to that of weak FOND$^{\text{FM}}$ HTN planning by using Thm. 4.1/4.2/4.4/4.5 by Chen and Bercher (2021).

For acyclic problems, we again exploit the fact that since we will never reach the same task network twice under progression due to the absence of recursion in compound task decomposition, strong and strong cyclic solutions collapse. Although the term acyclic was originally intended to describe acyclicity of compound task decomposition in the deterministic HTN setting, it also happens to be the case that FOND$^{\text{MP}}$ HTN solutions themselves are acyclic.

Theorem 4.4. *Let $\mathcal{P}$ be a totally ordered acyclic FOND^{MP} HTN problem. Deciding whether $\mathcal{P}$ has a strong or strong cyclic solution is* EXPTIME*-complete.*

Proof. Membership: we show that the progression algorithm described in Section 3 always terminates and requires only polynomial space. We exploit the fact that for acyclic problems we can never reach the same task network and state pair more than once during progression. This means that strong and strong cyclic solutions coincide and that eventually all search nodes will have an empty task network or a rejecting state. Furthermore, we do not need the variable V to store the progression history. This leaves us with variables $\mathsf{tn}, s, \mathcal{M}$. Clearly, s and $\mathcal{M}$ are polynomially bounded. The size of tn under progression is bounded in the same way as progression in the deterministic setting given that decomposition of compound tasks are equivalent. Thus, we can use Lemma 3.6 by Alford, Bercher, and Aha (2015) for totally ordered acyclic problems to get that tn is bounded polynomially. Hence, we have that totally ordered acyclic strong/strong cyclic FOND$^{\text{MP}}$ HTN planning is in APSPACE = EXPTIME (Chandra and Stockmeyer 1976).

Hardness: we will give a proof sketch on how to give a polynomial reduction of deciding whether an arbitrary alternating Turing machine (ATM) accepts an input string w_I

Hierarchy	Order	Classical/Weak		Strong		Strong cyclic	
primitive	total	P*/NP	[4.3]		P*		[4.2]
	partial	NP	[4.3]		PSPACE		[4.2]
acyclic	total	PSPACE	[4.3]		EXPTIME		[4.4]
	partial	NEXPTIME	[4.3]		EXPSPACE		[4.5]
regular	total	PSPACE	[4.3]	EXPTIME	[4.6]	EXPTIME	[4.6]
	partial	PSPACE	[4.3]	EXPTIME	[4.6]	EXPTIME	[4.6]
tail-recursive	total	PSPACE	[4.3]	EXPTIME	[4.7]	EXPTIME	[4.7]
	partial	EXPSPACE	[4.3]	2-EXPTIME	[4.8]	2-EXPTIME*	[4.8]
arbitrary	total	EXPTIME	[4.3]	semi-decidable*		semi-decidable*	
	partial	semi-/undecidable		semi-/undecidable		semi-/undecidable	

Table 1: Complexity results for FOND HTN planning. Classes marked $*$ are not complete where only membership is given. The first column collapses deterministic and weak HTN planning as most weak problems can be determinised and hence have the same complexity as deterministic planning (Chen and Bercher 2021). Undecidability of results arise from realising that standard HTNs are a special case of FOND HTNs and undecidable. Semi-decidability results arise from Section 3.

in space k. This will give us APSPACE = EXPTIME-hardness. The main idea of the reduction is that we exploit the fact that a polynomially space bounded ATM can be decided in an exponential number of steps given that there are at most exponentially many configurations $C = |Q| \cdot (|\Gamma| + 1)^k \cdot k$ using k space. We get $|Q|$ from the number of states, $(|\Gamma|+1)^k$ from all possible length k strings that can be constructed with the alphabet Γ plus a blank symbol, and k for the number of locations the tape head can be. We will define primitive tasks to mimic ATM transitions and use a task hierarchy to define a task network that can be decomposed into exponentially many tasks.

To model ATM configurations and transitions, we first define facts that represent the tape contents and ATM states with the same state variables as those in the PSPACE-hardness proof of non-hierarchical planning (Bylander 1994). We also define similar actions with the modification where given a $\forall$ state and an ATM transition, we create a nondeterministic task with the same number of corresponding effects, whereas in an $\exists$ state, we create a deterministic task modelling each effect. This enforces that at a $\forall$ state, all the next configurations must be accepting whereas at an $\exists$ state, we only have to choose one good effect.

To model exponentially many tasks, we construct compound tasks and methods in the same way as in Section 4 by Alford, Bercher, and Aha (2015). For ease of notation, let n be the smallest number such that $2^n \geq C$, the number of configurations shown to be exponential above. The main idea of the construction is that we define compound tasks

$$2^k \cdot \mathit{sim}$$

for $0 < k \leq n$, each with one method which decomposes it into a totally ordered task network with two tasks mapping to $2^{k-1} \cdot \mathit{sim}$. Next, we have $1 \cdot \mathit{sim}$ have one method for each primitive task n decomposing it to $\mathsf{tn}(n)$, and one method decomposing it to $\mathsf{tn}_\emptyset$. In this way, we can define an initial task network $\mathsf{tn}(2^n \cdot \mathit{sim})$ which can decompose into up to any number of tasks bounded exponentially to simulate an accepting ATM computation as required. □

Proving EXPSPACE-hardness using ATMs is not as straightforward any more as our reduction now has to be logarithmic. Thus, we can no longer define a fact for each tape cell which would cause a polynomial reduction and instead we will extend the NEXPTIME-hardness proof for deterministic acyclic HTN planning (Alford, Bercher, and Aha 2015) from the reduction of a nondeterministic Turing machine (NTM) to a reduction of an ATM. The idea of the original proof is that we do not define explicit facts to represent an NTM configuration but instead we only have one state and tape cell fact true at any time. Totally ordered primitive tasks are used to represent a witness and use synchronisation techniques to model and verify NTM transitions. This is because we can compactly represent k tasks in a task network with only a logarithmic number of defined tasks for acyclic problems.

Theorem 4.5. *Let $\mathcal{P}$ be an acyclic FOND^{MP} HTN problem. Deciding whether $\mathcal{P}$ has a strong or strong cyclic solution is* EXPSPACE*-complete.*

Proof. Membership: we again use our alternating progression algorithm from Section 3 and the fact that strong and strong cyclic solutions collapse for acyclic problems. Also similarly to the proof of membership of Theorem 4.4, we do not require the variable V although this is not necessary now as we will provide a time bound. We notice that the initial task network can be decomposed into a primitive task network with bounded size m^{k+1} where k is the maximum stratification of the compound tasks, and m is the size of the largest task network in the problem as shown in Corollary 3.2. by Alford, Bercher, and Aha (2015). Thus, the progression algorithm always terminates and determines if a solution exists within an exponential number of steps as the number of methods which can be applied is bounded exponentially and similarly with the execution of primitive tasks. So the problem is in AEXPTIME = EXPSPACE.

Hardness: we will give a proof sketch on how to give a logarithmic reduction of deciding whether an arbitrary ATM A accepts a string w_I in time k. We will extend the proof of NEXPTIME-hardness for deterministic acyclic problems in Thm. 6.1 by Alford, Bercher, and Aha (2015) which in-

volved a reduction from an NTM. The idea of the original proof is to represent a witness for an NTM or a sequence of strings $w_0, \ldots, w_k$ with exponentially many totally ordered tasks as described in the proof of Thm. 4.4 above. Each task asserts a fact representing a tape symbol at a tape cell, of which only one can be true at a given time. A second sequence of totally ordered tasks are synchronised with these tasks to check whether the ith character of strings w_j and w_{j+1} are equivalent. A third totally ordered sequence is used to keep track of the tape head and determine transitions. In this way, the transformed problem is able to generate any witness for the NTM with input w_I and only yields a solution if the witness is indeed a proof for the original problem.

The modification we describe is by introducing additional nondeterministic tasks to model ATM transitions. In the original proof, there exist deterministic *step*$_\exists$ tasks for each nondeterministic effect of each nondeterministic transition in the first totally ordered sequence of tasks. We can introduce additional nondeterministic *step*$_\forall$ tasks which model universal ATM transitions. To make this work, we have the additional ATM assumption that at a given universal state, all transitions step in the same direction in order for the synchronisation process to still work. This can be compiled away by introducing additional states and deterministic transitions which take an ATM state back to its intended position after every universal transition. The correspondence of solutions still holds as a strong solution holds iff a computation tree exists for A determining that the initial configuration is accepting. This comes from being able to dynamically choose the correct decompositions in the second and third task sequences for verifying a computation tree induced by the first sequence of tasks. Thus, the problem is AEXPTIME = EXPSPACE-hard and complete. □

We exploit the fact that FOND$^{\text{MP}}$ HTN planning is able to model non-hierarchical nondeterministic planning whose complexity we know. The idea of the reduction is that we can define a compound task which can decompose into arbitrarily many primitive tasks corresponding to actions for a non-hierarchical planning problem.

Theorem 4.6. *Let $\mathcal{P}$ be a regular FOND^{MP} HTN problem. Deciding whether $\mathcal{P}$ has a strong or strong cyclic solution is* EXPTIME*-complete. The decision is also* EXPTIME*-complete for totally ordered problems.*

Proof. Membership: we use the bounded graph search algorithm described in Section 3 and recall that the set of reachable task networks for regular problems is bounded exponentially. This is also true for the number of states such that the size of S and hence the size of the problem to solve is exponential. Since the subroutine to solve strong or strong cyclic plan existence is polynomial with respect to the size of the graph, the problem is in EXPTIME.

Hardness: we model non-hierarchical planning problems with regular FOND$^{\text{MP}}$ HTN problems in the same way described for the deterministic case (Erol, Hendler, and Nau 1996). The main idea is that we create a compound task *repeat* which has a method for every action in the original problem decomposing into a totally ordered task network with a task corresponding to such action followed by *repeat*. We also add a task *done* with precondition the goal condition. The only extension from the original proof is that we are able to model nondeterministic actions using nondeterministic tasks. The reduction mimics the mechanics of the original problem so strong and strong cyclic solutions correspond. It was shown by Rintanen (2004) that plan existence for both strong and strong cyclic planning is EXPTIME-complete by reduction from ATMs. Hence, the problem in question is EXPTIME-hard and complete. □

Theorem 4.7. *Let $\mathcal{P}$ be a totally ordered tail-recursive FOND^{MP} HTN problem. Deciding whether $\mathcal{P}$ has a strong or strong cyclic solution is* EXPTIME*-complete.*

Proof. Membership: we will again use the bounded graph search algorithm provided in the Search Algorithms section and show that it runs in exponential time. To do this, we show that the number of reachable task networks under progression is only exponential. First, we have from Lem. 3.6 by Alford, Bercher, and Aha (2015) that under progression of a totally ordered tail-recursive HTN problem $\mathcal{P}$, a task network is bounded polynomially by $m = k + r \cdot h$ for k initial tasks, r the largest number of tasks in any method for $\mathcal{P}$ and h the height of the stratification on compound tasks. Note that although we are in the nondeterministic setting now, the bounds calculated for deterministic HTN problem carry over as the decomposition mechanics are the same.

Thus, letting $n = |\mathcal{N}_P \cup \mathcal{N}_C|$ be the number of task names in $\mathcal{P}$, the number of reachable task networks is bounded exponentially by $\sum_{i=0}^{m} i^n \leq (m+1)m^n$.

The sum arises from counting the number of task networks of size i for $0 \leq i \leq n$ and the i^n from choosing any of n task names for each task in a totally ordered task network. Hence, the graph we build in the search algorithm has at most $n^m \cdot 2^{|\mathcal{F}|}$ nodes. Building and searching the graph takes exponential time, and thus so is the runtime of the algorithm.

Hardness: given that regular problems are a special case of tail-recursive problems and that totally ordered regular strong and strong cyclic HTN planning is EXPTIME-complete, we have EXPTIME-hardness for totally ordered tail-recursive strong and strong cyclic HTN planning. □

Theorem 4.8. *Let $\mathcal{P}$ be a tail-recursive FOND^{MP} HTN problem. Deciding whether $\mathcal{P}$ has a strong or strong cyclic solution is in 2-*EXPTIME*. Determining existence of a strong cyclic solution for $\mathcal{P}$ is* EXPSPACE*-hard and a strong solution is 2-*EXPTIME*-hard and hence complete.*

Proof. Membership: again we use the bounded graph search algorithm but now the upper bound for reachable task networks is higher given that there is no longer any total order assumption. From Lemma 3.4 by Alford, Bercher, and Aha (2015), now the size of a task network under progression is bounded exponentially by $m = k \cdot r^h$ with variables the same as described in Theorem 4.7. Thus, letting n be the number of task names, the number of reachable task networks is upper bounded by $|TN_R| \leq \sum_{i=0}^{m} i^n \cdot f(i)$, where $f(i)$ counts the number of directed acyclic graphs for i labelled vertices.

Again, i^n gives a loose upper bound for calculating the number of reachable non ordered task networks of size i, and the function f then gives us the number of possible partial orderings for size i task networks with names attached given that partial orderings are synonymous with directed acyclic graphs. We can provide a loose closed form upper bound for the number of DAGs by counting the number of directed graphs: $f(n) \leq \sum_{i=0}^{n^2} \binom{n^2}{i} = 2^{n^2}$. This follows by noticing that there are at most n^2 directed edges for n vertices and that there are $\binom{n^2}{i}$ ways of choosing i edges for building a directed graph with i edges. Thus the number of reachable task networks is bounded by $|TN_R| \leq \sum_{i=0}^{m} i^n \cdot 2^{i^2} \leq (m+1) \cdot m^n \cdot 2^{m^2}$. Since m is exponential, we have that the size of the graph is bounded double exponentially and hence the algorithm itself takes double exponential time to run.

Hardness: for strong cyclic problems, this follows from the fact that the deterministic version of the problem is EXPSPACE-complete (Alford, Bercher, and Aha 2015) and is a special case of nondeterminism. For strong problems, we extend the EXPSPACE-hardness proof for deterministic tail-recursive problems in the same fashion as described in Theorem 4.5 to get AEXPSPACE = 2-EXPTIME-hardness. □

5 Conclusion

In this paper we propose an alternate formalism for FOND HTN planning, with differences to previous work lying in how method decomposition is considered: at plan generation or execution. Our formalism provides more flexible solutions at the cost of plan execution complexity. Specifically, by integrating method choice into a plan we create a richer class of solutions but such solutions have two weaknesses: (1) policies for strong cyclic solutions are potentially unbounded (in contrast to non-hierarchical planning where policies are bounded by the number of reachable states), and (2) generally, policy execution is hard as we need to perform graph isomorphism checks to receive instructions.

We also provide basic search algorithms and many tight complexity results for existing HTN problem subclasses and show that problems with some restriction on recursion of compound tasks are only made at most one class harder when nondeterminism is introduced. This arises from the existence of natural extensions of algorithms and reductions for standard HTN planning using alternation.

References

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

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

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

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

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

Chen, D.; and Bercher, P. 2021. Fully Observable Nondeterministic HTN Planning – Formalisation and Complexity Results. In *ICAPS 2021*, 74–84. AAAI Press.

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.

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

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

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

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A Generic Method to Guide HTN Progression Search with Classical Heuristics. In *ICAPS 2018*, 114–122. AAAI Press.

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

Kuter, U.; and Nau, D. S. 2004. Forward-Chaining Planning in Nondeterministic Domains. In *AAAI 2014*, 513–518. AAAI Press / The MIT Press.

Kuter, U.; Nau, D. S.; Pistore, M.; and Traverso, P. 2005. A Hierarchical Task-Network Planner based on Symbolic Model Checking. In *ICAPS 2015*, 300–309. AAAI.

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

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

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.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI 99*, 968–975. Morgan Kaufmann.

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

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

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

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

Temporal Hierarchical Task Network Planning with Nested Multi-Vehicle Routing Problems — A Challenge to be Resolved

Jane Jean Kiam[1], **Pascal Bercher**[2], **Axel Schulte**[1]
[1]University of the Bundeswehr, Munich, Institute of Flight Systems
[2]The Australian National University, College of Engineering and Computer Science
jane.kiam@unibw.de, pascal.bercher@anu.edu.au, axel.schulte@unibw.de

Abstract

This paper focuses on presenting a complex real-world planning application based on a rescue mission. While temporal hierarchical planning seems to be a promising solution to such a class of problems, given its ability to consider experts' knowledge and dissect the search space, many major challenges of complex real-world planning problems are not addressed yet formally, i.e. recursive decomposition to achieve a goal state, optimization of utility functions defined for abstract tasks, and optimal allocation of tasks to multiple actors.

1 Introduction

Hierarchical planning has proven to be useful in solving many real-world planning problems (Bercher, Alford, and Höller 2019), ranging from web services (Sirin et al. 2004) to medical applications (Fdez-Olivares et al. 2019), robotics (Hayes and Scassellati 2016; Jain and Niekum 2018), and aviation (Benton et al. 2018), to name just a few. Its wide usability owes to its similarity with the task planning humans practise, which usually regards first the planning of tasks at a higher abstraction-level, followed by the refinement of each task down to an executable level. The hierarchical task network planning paradigm can therefore encode naturally known "recipes" for the decomposition of higher-level tasks. Furthermore, by doing so, problem solving is simplified, either by considering a lower-resolution state space at the higher abstraction levels, or by considering a limited search space[1]. This advantage has also been leveraged by Kaelbling and Lozano-Pérez (2011) and Patra et al. (2020), for example to develop hierarchical planners that manage to cope with non-deterministic and dynamic environments. The common driving idea behind is to plan at a more abstract level and leave the details (i.e. lower-level plan) be decided once the knowledge on the environment becomes "clearer". Given their ability to include experts' knowledge and reduce search space, hierarchical planners are promising to solve even more complex real-world planning problems. This paper presents one challenging scenario based on a large-scale rescue mission.

There are many formalisms centered around the idea of hierarchical planning, where "pure" HTN planning is only one, together with other extensions, such as HTN planning with task insertion, Hierarchical Goal Network planning, etc. (Bercher, Alford, and Höller 2019). One of such extensions of major importance for practical applicability of the formalism is featuring time – though only few formalisms and existing planning approaches support it. Works by Goldman (2006) (durative planning with SHOP2), Molineaux, Klenk, and Aha (2010) ($\mathrm{SHOP2}_{\mathrm{PDDL+}}$), Fdez-Olivares et al. (2006; 2019) (complex mission and multi-agent planning), Dvořák et al. (2014) and Bit-Monnot, Smith, and Do (2016) (the FAPE planner), as well as by Stock et al. (2015) (for robotics) are some of these exceptions. Furthermore, a unifying formalism featuring time does not yet seem to be available. In the context of the recent International Planning Competition (IPC) 2020 for HTN planning (Behnke et al. 2019), HDDL (Höller et al. 2020a), the proposed common standard for describing HTN problems, also does not support time (yet). The Action Notation Modelling Language (ANML) (Smith, Frank, and Cushing 2008) focuses on time, but it has not been integrated into HDDL; its semantics is also incompatible with recursion, which is an inherent feature of many HTN planning task models. Besides, its ability to model problems with the need for optimal allocation of tasks to actors, comparable to approaches used for solving multi-vehicle routing problems (MVRPs)[2] is unclear.

For practical real-world use cases we see the support of time as one of the major requirements. This will require both 1) an adequate and easy to use modelling language (potentially complemented with adequate modelling support), and 2) planning systems able to find solutions efficiently, which goes beyond just supporting time technically – it requires new or improved heuristics, or techniques, depending on the respective approach chosen to tackle the problem.

Another challenge that we foresee is how utility functions are being handled, and which kinds of plan metrics are supported. Furthermore, real-world use cases involve often more than one "executor" of the plan, which renders plan-

[1]Search space reduction is possible since HTNs allow to restrict search in directions anticipated by the expert, though in general no bound on the search space exists since the HTN framework allows to express undecidable problems (Erol, Hendler, and Nau 1996).

[2]In a MVRP, vehicles are assigned tasks at different locations (Beck, Prosser, and Selensky 2003). It is often implicitly implied that vehicles with the ability to perform the task are interchangeable, and that each task is assigned to only one vehicle (and will be a goal aimed to reach by the corresponding vehicle).

ning even more complex. Hierarchical planning can be beneficial in this case given its ability to simplify the problem by leveraging experts' knowledge and by dissecting the search space, a concept adopted by Kiam et al. (2019; 2021) to develop a domain-specific planner that computes plans for multiple Unmanned Aerial Vehicles (UAVs) tasked to monitor dissected ground locations in a dynamic environment.

In the following, a real-world temporal hierarchical planning problem (with nested MVRPs) based on a complex rescue mission will be presented. Challenges posed by this class of problem will be analysed and a more formal representation of the problem will be provided to facilitate a more thorough understanding of the problem, allowing thereby the scaling of complexity (i.e. introducing more tasks, actors, complex goal conditions, and utility functions).

2 A Real-World Problem Example

Hierarchical planning is convenient for describing real-world complex planning problems, especially problems that rely on operational rules or experts' knowledge. Rescue missions are a typical example in which leveraging hierarchical planning can be beneficial, as argued in some previous works (Biundo and Schattenberg 2001; Fdez-Olivares et al. 2006; Patra et al. 2020). The section presents a real-world planning problem based on a rescue mission, the challenges of which are either omitted or only partially considered in these works. Fdez-Olivares et al. (2006) consider in their SIADEX planner temporal reasoning. While this is not considered by Patra et al. (2020), the interaction between planning and acting is making it possible to consider the dynamics of the environment.

Figure 1a depicts an example use case of a realistic and complex rescue mission after a disaster (e.g. earthquake) involving multiple rescue teams of different capabilities. Marked in blue is a disaster-struck area[3]; different locations within the area that require rescue operations are marked in orange, while the different objects (e.g. buildings, clusters of victims, etc.) within a location are marked with stars, the sizes of which also indicate the complexity (or rather the duration) of the rescue tasks to be performed. The capabilities of the rescue teams as well as the team members are listed in Table 1. Note that the number of each type of team member (i.e. humans, robots, UAVs) varies in each team.

Part of the task network structure is depicted in Figure 1b. Due to the number of locations (`?l`) to attend to within a disaster area, and the number of objects or patients to cope with at each location, a time-dependent MVRP must be considered in the decomposition of the high-level task `clear-earthquake-disaster(?a)` to decide for the order the locations will be cleared, i.e. the order in which the `clear(?l)` tasks will be performed. Depending on the need, and on the best practices, different tasks are needed for each location (see the two example decompositions of `clear(?l)`, in which one location needs triage `triage(?tt ?l)` and medical aid `aid(?met ?l)` after having monitored the location with `monitor(?mot ?l)`, while the other requires only an infrastructure team to attend to structural damages `build(?it ?l)`). Additionally, as there are several objects within a location, decomposing the task `monitor(?mot ?l)` for example into more "refined" tasks to be performed by the team members (i.e. `patrol(?h ?o)` by a human `?h`, `drive-by(?r ?o)` by a robot `?r`, and `fly-over(?u ?o)` by a UAV) is again a time-dependent MVRP.

[3]There can be more than one disaster areas.

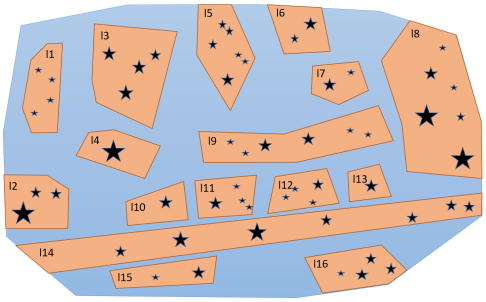
Rescue team	**Team members**
Monitoring, `?mot`	$\{h_1,...,h_H\ u_1, ...u_U, r_1,...,r_R\}$
Triage, `?tt`	$\{h_1,...,h_H, r_1,...,r_R\}$
Medical, `?met`	$\{h_1,...,h_H\}$
Infrastructure, `?it`	$\{h_1,...,h_H, r_1,...,r_R\}$

Table 1: Rescue teams and their heterogeneous capability, as well as the various team members, i.e. robots r_i, human h_i and UAVs u_i

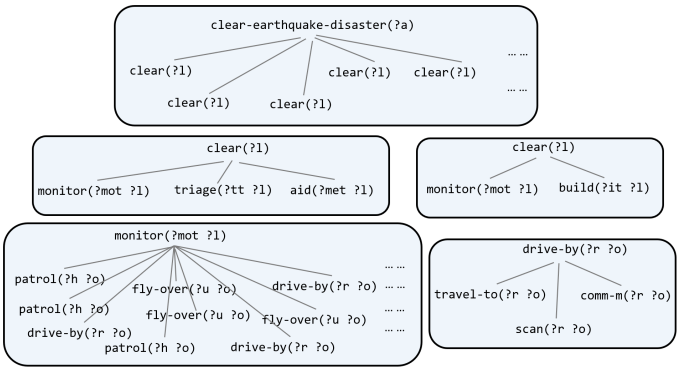
To solve such planning problems, the temporal aspect must be considered, so that the execution of concurrent tasks by different actors are possible, while managing the resources. Besides, such planning problems also pose several new challenges listed below, which, to the best of our knowledge, were either not considered before or do not yet have a straightforward or commonly accepted solution.

- The recursive decomposition into subtasks: The decompositions of the compound tasks, for instance `clear-disaster-area(?a)` and `monitor(?mot ?l)` terminate only if the subtasks altogether achieve the goals imposed by the compound tasks. For `clear-disaster-area(?a)`, the decomposition terminates once all locations `?l` within the area `?a` are cleared, while for `monitor(?mot ?l)`, the decomposition terminates after all objects `?o` within the location are attended to by either the human first responder `patrol(?h ?o)`, a robot `drive-by(?r ?o)` or an UAV `fly-over(?u ?o)`. One way to achieve this is by employing a recursive decomposition, if the number of locations is known at planning time. However, recursive decomposition is not yet very well studied or supported in temporal hierarchical planning. Furthermore, if the exact number of locations is unknown at planning time, such as in a framework where planning and acting interleave[4] (see following subsection), or if the knowledge on the number of locations changes due to updates of information, therefore requiring plan repair (see following subsection), such recursive decomposition is even less straightforward.
- Finding a plan that is optimal with respect to the underlying utility functions: Utility functions such as number of lives saved, cost of structural damages, cost of time, etc. can often only be defined intuitively by the domain expert at a higher abstraction level, i.e. `aid(?met ?l)` for the lives saved and `build(?it ?l)` for the cost of structural damages. However, the exact evaluation of the utility functions can only take place once the decomposi-

[4]This is considered by Patra et al. (2020). However, the temporal aspect is ignored.



(a) Illustration of an example scenario during a rescue mission with the polygons in orange within the blue disaster area depicting the locations where rescue teams are needed.



(b) Decomposition of abstract tasks

Figure 1: An example application for hierarchical planning with nested multi-agent routing problems.

tion is performed down to the level of the primitive tasks. The propagation of utility can therefore be challenging in the formalism for this class of hierarchical planning problems. Furthermore, determining heuristics capable of optimizing different utility functions can also be a challenging issue. The challenge is even more amplified if multiple objectives are to be considered, such as by Kiam, Besada-Portas, and Schulte (2021). Solving it using a hierarchical planning approach is feasible in a domain-dependent manner, but to the best of our knowledge, the feasibility of solving such a class of problems in a domain-independent fashion still remains unknown.

- The allocation of subtasks to actors: This is relevant for example in the decomposition of `monitor(?mot ?l)`, where the subtasks can be allocated to humans `patrol(?h ?o)`, robots `drive-by(?r ?o)` and UAVs `fly-over(?u ?o)`. The optimality of the allocation depends on the utility function; its implementation can be challenging with respect to the formalism or modelling language and the heuristics.

Besides rescue missions, maintenance of a building complex, managing large-scale construction work, or complex logistic problems have similar planning requirements, involving on the one hand hierarchical decomposition of tasks, and on the other, nested time-dependent MVRPs.

Flexible Planning Framework

The above describes only the planning domain and the planning problem in a static fashion. The knowledge of these is insufficient to solve the planning problem, as this must be solved within a framework that adopts a carefully engineered architecture capable of coping with the flow of information on the (dynamic) state space. Concretely, the planning problem can or must be solved in different manners:

- Offline planning: this mode of planning will force the computation of a complete plan with the knowledge assumed before plan execution. Although the plan will be suboptimal since the dynamics of the environment are not considered at planning time, this mode of planning is still useful as an initial overview for resource management.

- Planning and acting: Patra et al. (2020) worked on a planning architecture capable of interchanging between off- and online planning to take into account the dynamics of the environment observed in the course of "acting" (i.e. while carrying out the actions). By doing so, abstract tasks that need not be performed immediately do not have to be decomposed immediately. The decomposition of the task network follows a forward and top-down manner. This kind of framework is convenient when future tasks only need to be planned as an "anticipatory" measure, but detailed action plan is not required immediately.

- Plan and plan repair: This type of planning framework exploits full-fledged offline planning with the information on the state space known before plan execution, yet must be flexible enough to allow for plan repair as new information becomes available during plan execution. However, plan repair is not always straightforward in hierarchical planning, since the execution of a method's task network cannot be aborted and "skip" to the execution of another task network without reversing some effects incurred in the course of the execution of the aborted task network. The permission to "skip" to another task network is often unknown to the planner, as the modelling of all possible plan repair cases would require much more extensive expert knowledge. One possible solution would be to annotate which methods are merely "advice" on how to solve a task (and may be skipped without negative consequences), and which methods actually carry semantics so that a true refinement needs to be found (which implies that it cannot just be skipped). The former repair approach was for example exploited by Goldman, Kuter, and Freedman (2020), while the latter was described by Höller et al. (2020b); but we are not aware of any combination.

3 A More Formal Representation of the Planning Problem

In this subsection, we provide a first attempt for a more formal problem representation of the problem(s) outlined before. However, this does not yet incorporate the dynamics of the world. The planning problem is defined by a tuple $\mathcal{P} = (X_p, X_n, A, A', T_p, T_c, M, s_I, tn_I, G, U, \alpha, \alpha')$, where X_p and X_n are the sets of propositional and numeric state variables respectively, A is the set of actors performing primitive tasks (also called actions), A' is the set of actor-sets, which group actors in teams (e.g. a monitoring team `?mot` is an element of A'), $a' \in A'$ is a set of actors $\{a_1, ..., a_{|a'|}\}$ with $|a'|$ being the cardinality of the set a', $a_i \in A$, and T_p is the set of tuples $(t_p, \delta_{\min}, \delta_{\max})$ containing the primitive task (or action), as well as its minimum and maximum durations respectively, T_c is the set of compound (or abstract) tasks, M is the set of methods to decompose compound tasks into substasks, s_I is the initial state, tn_I is the initial task network, G is the set of goal conditions which define the state(s) to achieve within a time window or before a time instant, and U is the set of utility functions with respect to the (abstract or primitive) tasks. α and α' denote functions that map an actor $a \in A$ (and a set of actors $a' \in A'$ respectively) to a tuple $(t_p, \delta_{\min}, \delta_{\max}) \in T_p$ (and an abstract task $t_c \in T_c$ respectively), where t_p (and t_c respectively) are primitive task (and compound task respectively) that the actor a (and a' respectively) can perform.

A solution to the planning problem is a set of plans, i.e. $\pi = \{\pi^{a_1}, ..., \pi^{a_{|A|}}\}$, where each plan π^a is a partially or totally ordered set of actions with their associated time constraints for actor $a \in A$.

As mentioned in Section 2, depending on the availability of information, the knowledge about the environment can be updated before the planning loop terminates (i.e. in planning and acting), or after the termination of planning and during the plan execution (i.e. plan repair is required). In these cases, this formal representation may no longer be adequate.

4 Expected Features of Future Planners

In this section, expectations on the temporal hierarchical planners to be developed as solutions to the underlying challenges described in Section 2 and 3 will be discussed.

Modelling Language

In order to build on each other's domain-independent reasoning and planning methodologies, which can be based on heuristics, compilation-based or grounding techniques, a common formalism and modelling language are necessary. In view of this, HDDL was developed by Höller et al. (2020a) and used as input language (directly or translated further) for the HTN track of the IPC 2020. However, HDDL does not consider temporal planning aspects. Recent previous works by Fernandez-Olivares and Perez (2020), Bit-Monnot et al. (2020) and Stock et al. (2015) include temporal information; however, they employ different problem modelling languages. Fernandez-Olivares and Perez (2020) use the Hierarchical Planning Description Language (HPDL) for modelling the hierarchical planning problem coupled with a temporal reasoning engine, while Bit-Monnot et al. (2020) use ANML developed by Smith, Frank, and Cushing (2008), and Stock et al. (2015) use a customized domain modelling language that enables the expression of temporal constraints such as start time and duration of tasks.

Therefore, the underlying challenge with respect to the modelling language is to first properly define the formalism for this class of temporal hierarchical planning problems (as described in Section 2) in a more general sense, followed by the development of a syntax capable of taking into account specifically the challenging aspects listed in Section 2, namely the recursive decomposition into subtasks, consideration of utility functions, and the allocation of subtasks to actors (or sets of actors).

Planning System

In AI planning, the development of modelling languages is parallel to the development of the planning system. Besides the qualitative assessment on the adequacy of the modelling language to model the class of planning problems described in Section 2, as well as the correctness of the formalism, a systematic and quantitative assessment of the planning system ought to be considered too. The planning efficiency can be assessed by scaling the problems with respect to the number of tasks, the number of actors, as well as the depth of the task network. Besides, the plan quality can be assessed according to the defined utility functions. These can be linear and non-linear functions, or even more complex mathematical equations such as the Bellman equation considered by Kiam, Besada-Portas, and Schulte (2021), adapted from Boyan and Littman (2000).

Flexible Planning Framework

As described in Section 2, a planning problem can be solved in different manners. Besides the most straightforward offline planning framework, planners must also be compatible with a framework that requires interchanging of off- and online planning to cope with the dynamics of the environment, as well as plan repair, which requires extensive work on the merging of task networks, involving even interaction with human expert(s) to retrieve additional information on the problem models to perform a correct merging.

5 Conclusion

Besides a descriptive overview on a complex temporal hierarchical planning problem, this paper also describes the challenges related to such a class of problems, which must be solved with 1) a proper definition of the formalism, followed by the development of a rigorous problem modelling language, and 2) domain-independent planning systems that support the the modelling language and solve the problem efficiently. For wider usability, the planner must also cope with different planning frameworks, allowing an interleaving of planning and acting, as well as a plan repair.

Our main motivation is to make aware of the unresolved challenges in hierarchical planning that will require collective efforts if such a class of problems is to be solved in a

domain-independent fashion. One way to promote research interest is to gradually include subsets of the aforementioned challenges in future IPC tracks on HTN planning.

References

Beck, J. C.; Prosser, P.; and Selensky, E. 2003. Vehicle Routing and Job Shop Scheduling: What's the Difference? In *Proceedings of the 13th ICAPS*.

Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Pellier, D.; Fiorino, H.; and Alford, R. 2019. Hierarchical Planning in the IPC. In *Proceedings of the Workshop on the International Planning Competition*.

Benton, J.; Smith, D.; Kaneshige, J.; Keely, L.; and Stucky, T. 2018. CHAP-E: A Plan Execution Assistant for Pilots. In *Proceedings of the 28th ICAPS*.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *Proceedings of the 28th IJCAI*.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. 2020. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. *arXiv:2010.13121[cs.AI]* .

Bit-Monnot, A.; Smith, D. E.; and Do, M. 2016. Delete-Free Reachability Analysis for Temporal and Hierarchical Planning. In *Proceedings of the 22nd ECAI*.

Biundo, S.; and Schattenberg, B. 2001. From Abstract Crisis to Concrete Relief – A Preliminary Report on Combining State Abstraction and HTN Planning. In *Proceedings of the 6th European Conference on Planning (ECP)*.

Boyan, J. A.; and Littman, M. L. 2000. Exact Solutions to Time-Dependent MDPs. In *Proceedings of the 13th International Conference on Neural Information Processing Systems (NIPS)*.

Dvořák, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. A Flexible ANML Actor and Planner in Robotics. In *Proceedings of the 2nd Workshop on Planning and Robotics (PlanRob)*.

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

Fdez-Olivares, J.; Castillo, L.; García-Pérez, O.; and Palao, F. 2006. Bringing Users and Planning Technology Together. Experiences in SIADEX. In *Proceedings of the 16th ICAPS*.

Fdez-Olivares, J.; Onaindia, E.; Castillo, L.; Jordán, J.; and Cózar, J. 2019. Personalized conciliation of clinical guidelines for comorbid patients through multi-agent planning. *Artificial Intelligence in Medicine* 96: 167–186.

Fernandez-Olivares, J.; and Perez, R. 2020. Driver Activity Recognition by Means of Temporal HTN Planning. In *Proceedings of the 30th ICAPS*.

Goldman, R. P. 2006. Durative Planning in HTNs. In *Proceedings of the 16th ICAPS*, 382–385.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. Stable Plan Repair for State-Space HTN Planning. In *Proceedings of the 3rd ICAPS Workshop on Hierarchical Planning (HPlan)*.

Hayes, B.; and Scassellati, B. 2016. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *Proceedings of the 2016 IEEE International Conference on Robotics and Automation (ICRA)*.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020a. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Plan Repair via Model Transformation. In *Proceedings of the 43th German Conference on Artificial Intelligence (KI)*. Springer.

Jain, A.; and Niekum, S. 2018. Efficient Hierarchical Robot Motion Planning Under Uncertainty and Hybrid Dynamics. In *Proceedings of Machine Learning Research - The Conference on Robot Learning (CoRL)*, volume 87.

Kaelbling, L. P.; and Lozano-Pérez, T. 2011. Hierarchical task and motion planning in the now. In *Proceedings of the 2011 IEEE International Conference on Robotics and Automation (ICRA)*.

Kiam, J. J.; Besada-Portas, E.; Hehtke, V.; and Schulte, A. 2019. GA-Guided Task Planning for Multiple-HAPS in Realistic Time-Varying Operation Environments. In *Proceedings of the 2019 Genetic and Evolutionary Computation Conference (GECCO)*.

Kiam, J. J.; Besada-Portas, E.; and Schulte, A. 2021. Hierarchical Mission Planning with a GA-Optimizer for Unmanned High Altitude Pseudo-Satellites. *Sensors* 21(5).

Molineaux, M.; Klenk, M.; and Aha, D. 2010. Planning in Dynamic Environments: Extending HTNs with Nonlinear Continuous Effects. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. In *Proceedings of the 30th ICAPS*.

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2004. HTN planning for Web Service composition using SHOP2. *Journal of Web Semantics* 1(4): 377–396.

Smith, D.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Stock, S.; Mansouri, M.; Pecora, F.; and Hertzberg, J. 2015. Online task merging with a hierarchical hybrid task planner for mobile service robots. In *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)

Towards Robust Constraint Satisfaction in Hybrid Hierarchical Planning*

Tobias Schwartz, Michael Sioutis, Diedrich Wolter
University of Bamberg
An der Weberei 5
Bamberg
{tobias.schwartz, michail.sioutis, diedrich.wolter}@uni-bamberg.de

Abstract

Hybrid planning is essential for real world applications, as it allows for reasoning with different forms of abstract knowledge, such as time, space or resources. This unavoidably leads to a combinatorial explosion of the search space that has previously been tackled using a hierarchical task network (HTN) planning approach. Existing HTN planners mostly focus on finding a solution as fast as possible, with only recent work considering length-optimal solutions. In real world scenarios, it can easily happen that the environment changes before a plan is fully executed. We are motivated to conduct planning in such a way that the solution has the best chance of withstanding such changes in the environment. We call this ability the robustness of a solution. Defining robustness, however, is an inherently difficult challenge, as many different forms and notions exist. In this paper, we start the discussion by outlining a possible notion of robustness recently introduced in the light of Qualitative Spatial Reasoning (QSR) within the scope of hybrid hierarchical planning.

Introduction

We are interested in the question of how to do robust planning in the face of a dynamic execution environment. Existing planners are able to quickly derive a plan given a valid domain and problem file. In the real world, however, we deal with a high degree of uncertainty. Some events may take longer than anticipated or other (dependent) objects in the scene may have moved and thus are not available on the same conditions as during the initial planning phase.

Traditionally, classical planning is concerned with finding a sequence of actions that lead to a desired goal state (Ghallab, Nau, and Traverso 2016). Actions for this matter are defined in a specific domain file, stating at least their preconditions and effects. Intuitively, an action a is only applicable in a state s if all its preconditions are satisfied. However, in practical applications it is usually not enough to only state static preconditions and effects and indirectly impose an ordering constraint on the possible actions. Other aspects, such as a sense of time, the spatial characteristic of the scenario, physical restrictions in movements or resource limitations often have to be considered.

In order to reason about all those different classes of knowledge, the notion of *meta-CSP* has been introduced (Mansouri and Pecora 2016). It formulates an abstract high-level Constraint Satisfaction Problem (CSP) which is solved using a meta-reasoner combining dedicated reasoners for each of the different types of knowledge. All different viewpoints (e.g., spatial, temporal, or resource) can then be modeled within one reasoning framework by defining a constraint in the meta-CSP. One challenge in such a meta-CSP is the combinatorial explosion of the search space. To still use this approach in planning, a hierarchical planning approach has been considered to systematically reduce the scope and computational complexity (Stock et al. 2015).

Hierarchical Task Network (HTN) planning differs from classical planning in that it distinguishes between primitive and compound (or abstract) tasks. Instead of defining a specific goal state and let the planner find applicable actions leading to a plan, in HTN planning a (series of) compound task(s) is given and then decomposed into executable primitive actions. The planner here tries to find an applicable decomposition. Many extensions and varying realizations within the hierarchical planning framework have been considered (cf. Bercher, Alford, and Höller (2019)).

Like Stock et al. (2015), we are motivated to apply abstract reasoning in the context of HTN planning. Stock et al. (2015) transform the HTN planning problem into a CSP by encoding causal links of the HTN problem as so-called meta-constraints into the aforementioned meta-CSP. Note that this is similar to the combination of HTN planning with Partial Order Causal Link (POCL) planning (Schattenberg 2009; Bercher et al. 2016), which builds into many current hierarchical planners (Bercher 2021).

The meta-CSP formulation allows for abstract reasoning not only within the task network, but for example also about space, time and resource constraints. Reasoning about abstract spatial and temporal information is usually done using Qualitative Constraint Networks (QCNs) (Ligozat 2013; Dylla et al. 2017). In this context, Sioutis, Long, and Janhunen (2020) recently studied a notion of robustness, which concerns the perturbation tolerance of QCN solutions, i.e., their likelihood to resist a change in the environment.

*We would like to thank the anonymous reviewers for their insightful feedback. This research is partially supported by BMBF AI lab dependable intelligent systems.

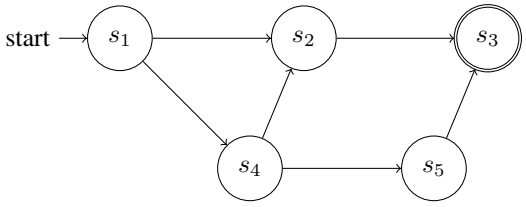


Figure 1: A train network with five stations and multiple paths leading from the start s_1 to the destination s_3

Robustness itself certainly is not a new concept, and can probably be traced back even to the first algorithms for problem solving, as with different methods to obtain a solution to a given problem, there was also the need to compare those solutions on a (usually robustness-related) basis (see for example Ginsberg, Parkes, and Roy (1998); Verfaillie and Jussien (2005)). Still, the work by Sioutis, Long, and Janhunen (2020) was the first time robustness as a measure was considered in QCNs. Likewise, to the best of our knowledge, the same holds for constrained reasoning in the context of hybrid hierarchical planning, where no measures of robustness have been established so far.

This paper contributes by pointing out the challenge of defining robustness in the scope of hierarchical planning, starting a discussion towards more robust solutions in HTN planning. We demonstrate our motivation and its usefulness in the context of a train routing problem. Following the work by Mansouri and Pecora (2016) and Stock et al. (2015) on modelling HTN planning in a meta-CSP, we establish similarities to QCNs and indicate how the notion of robustness from Sioutis, Long, and Janhunen (2020) may be applied. This may be seen as a first starting point for further work on robustness in hierarchical planning.

Robustness

Let us consider the simple train network depicted in Fig. 1, which, for the sake of our example, encodes the following planning task: Given n trains $t_1, \ldots, t_n$ with the goal to drive from station s_1 (start) to s_3 (destination). Driving a train from one station to another is a complex task on its own, where multiple signals have to be adhered, speed has to be adjusted accordingly, the tracks have to be monitored, etc. (see Cardellini et al. (2021) for a recent formulation of a similar problem in PDDL+). For the sake of the example, we are for now abstracting from the details of said driving process and only focus on the high-level decisions of which train has to drive to which station at what time. We are essentially reducing the problem such that given n trains in a specified order, we try to find a viable route for each train trough the network from the start to the destination. As additional constraint, trains may only traverse in the direction indicated by the arrow, i.e. reversing is not permitted.

Assuming a constant driving time between stations and enough distance between incoming trains, the fastest and thus likely preferred choice would be to use the direct path via only station s_2 for all trains. The problem with this route is that once started at s_1, there is no turning point and no alternative route to switch to. If there occurs any form of delay on a preceding train on this route, this unavoidably affects all other trains. Likewise, if a part of the track between s_2 and s_3 has to close temporarily, e.g., because of an animal accident, no alternative path exists.

Uncertainty in the plan execution is the reason why current work in railway planning increasingly focuses on robustness in the quality of solutions to a given planning problem, such as the one outlined above. Here, a plan may be considered robust if it is able to withstand such unexpected changes in the environment. Accordingly, robustness can be defined as the ability of a system to resist change (Verfaillie and Jussien 2005; Lusby, Larsen, and Bull 2018). Note, however, that based on the context also other notions of robustness exist. One can further differentiate between flexible and robust solutions to a problem (Verfaillie and Jussien 2005; Muise 2014). A flexible solution is anything that can quickly generate a new solution in case of change, whereas a robust solution has every chance to resist all possible changes given by a model. Hereby, it is not only important how possible changes are modeled (qualitative or quantitative, probabilistic or not), but also when those changes are available to a planner (before or during plan execution). Depending on those characteristics, different notions of robustness may be applicable.

For example, reacting dynamically to a sudden change in the environment, such as a technical disturbance that prevents a train from reaching its next station, requires the use of some form of execution monitoring (Fritz 2009; Muise 2014). Thereby, it is actively observed whether the executed plan remains *valid*, given what is known in the current state. Muise (2014) defines a plan in classical planning valid, iff the plan is executable in the initial state I and results in the goal state G. In contrast to such a sequential plan, the least-commitment approach followed in partial-order planning promises more flexibility during execution, as some ordering choices can be delayed until run-time. A partial-order plan (POP) is valid iff every linearization achieves the goal from the initial state, clearly a strong requirement if we are only interested in finding a possible way to achieve the goal. More practically, a POP is called *viable* iff there exists a linearization that achieves the goal, which may be efficiently found by exploiting state *relevance*. Robustness may then be achieved by actively monitoring the execution of the plan and always selecting the most relevant partial plan fragment for achieving the goal.

Similarly, in the context of hierarchical planning, Patra et al. (2020) apply a UCT-like Monte-Carlo tree search procedure called UPOM, an online planner for the Refinement Acting Engine (RAE) (Ghallab, Nau, and Traverso 2016), to guide the selection of a method instance in case multiple ones for a task exist. The RAE can not only accomplish tasks but, like execution monitoring, also react to external events.

While the outlined planning approaches can consider changing environments and to an end possess the capability to include robustness, they may only be able to really include one notion, e.g., probabilities or qualitative information might not be considered. Also, they do not directly incorporate robustness as a measure before plan execution.

Case Study: Meta-CSPs

In what follows, we elaborate on a notion of robustness in the context of hybrid hierarchical planning using meta-CSPs, drawing on prior work by Sioutis, Long, and Janhunen (2020) in the context of QCNs. We start by briefly presenting some frameworks that are relevant to our discussion and that we will be referring to throughout the paper.

Constraint Satisfaction Problems

We adopt the standard notation of a CSP from Russell and Norvig (2020). A CSP is a tuple (X, D, C) where

- X is a set of variables $\{x_1, \ldots, x_n\}$, where each variable $x_i \in X$ corresponds to exactly one domain $D_i \in D$,
- and C is a set of constraints that restrict possible value assignments to variables.

An assignment that does not violate any constraints is called a consistent assignment. Solving a CSP now is the task of finding a consistent, complete assignment for all variables. A CSP is often visualized using a constraint graph $G = (V, E)$, where the vertices V represent variables and the edges E define constraints between any two variables.

Qualitative Constraint Networks

Reasoning on infinite domains, such as space and time, is typically done using Qualitative Constraint Networks (QCNs) (Ligozat 2013; Dylla et al. 2017). Similar to a constraint graph for a CSP, a QCN is a network where the vertices represent spatial or temporal entities and the edges are labeled with qualitative spatial or temporal relations, based on a finite set of *jointly exhaustive and pairwise disjoint* relations, called the set of base relations B.

Hybrid Planning

Autonomous systems, such as robots, typically operate in dynamic environments. Planning in such an environment is particularly difficult as many different forms of knowledge, such as temporal, causal or spatial information and constraints, have to be considered. Motivated by those needs of real world applications, hybrid planning methods have been studied. Hybrid planning in this context describes a classical planner that instead of focusing on one particular type of constraint can reason with multiple classes of knowledge. Note that the notion of hybrid planning has also been used in the context of combining hierarchical with state based Partial Order Causal Link (POCL) planning (Schattenberg 2009; Bercher et al. 2016).

For hybrid reasoning, Mansouri and Pecora (2016) have proposed the use of a so-called *meta-CSP*. Here, fluents are used to represent causal, temporal, spatial or resource semantics. A fluent may be used to represent an action (e.g. "drive") or display the current situation ("a train is at the station"). Given a set of fluents $\mathcal{F}$ and a set of constraints $\mathcal{C}$ among the fluents in $\mathcal{F}$, we can define a constraint network as the pair $(\mathcal{F}, C)$. Note that this is similar to the notation used for a CSP, with fluents being the variables of a heterogeneous set of domains. Now, we can cast the problem of finding a feasible plan as a *meta-CSP*, i.e. a high-level CSP that captures the heterogenous information of the overall problem, and is defined as a collection of meta-constraints. A meta-constraint is a triple (M, Ξ, Δ), where $M = (\mathcal{F}, C)$ is a constraint network, Ξ is a set of meta-variables $\{\xi_1, \ldots, \xi_n\}$, each of which is a subnetwork of M, i.e. $\xi_i = (\mathcal{F}_i \subseteq \mathcal{F}, C_i \subseteq C)$, and $\Delta = \{\delta(\xi_1), \ldots, \delta(\xi_n)\}$ is a set of domains, one for each meta-variable.

Encoding a classical planning problem as a CSP has already been studied in the literature (Barták, Salido, and Rossi 2010). The intuition is that we can restrict the length of a possible plan and then subsequently increase this limit until a plan is found[1]. The planning problem can then be modeled as a series of boolean satisfiability (SAT) problems, where each SAT instance is the problem of finding a plan of a given length. Those instances can be encoded as CSP.

Mansouri and Pecora (2016) represent actions as operators, defined as the pair $(f, (\mathcal{F}, C))$, where $f = (A, \cdot, \cdot, u, \cdot)$ is a fluent indicating that action A is being executed, $\mathcal{F}$ describes the set of fluents including the set of precondition fluents $\mathcal{F}_p$ and both negative effect fluents $\mathcal{F}_- \in \mathcal{F}_e$ and positive effect fluents $\mathcal{F}_+ \in \mathcal{F}_e$; and, finally, C is a set of causal (CC), temporal (TC), spatial (SC), and symbolic (BC) constraints on $\mathcal{F} \cup \{f\}$. Consider following example based on the train problem outlined in Fig. 1, using the notations introduced by Stock (2016):

$$
\begin{aligned}
f &= (!driveTo(?t_1, ?s_2), [0, \infty], [0, 30], u(track1) = 1) \\
F_p &= \{f_1 = (At(?t_2, ?s_1), \cdot, \cdot)\} \\
F_- &= \{f_1\} \\
F_+ &= \{f_2 = (At(?t_3, ?s_3), \cdot, \cdot)\} \\
CC &= \{f_1 \text{ pre } f, f \text{ closes } f_1, f \text{ opens } f_2, f \text{ planned } f\} \\
BC &= \{S_1^{(f)} = S_1^{(f_1)}, S_1^{(f)} = S_1^{(f_2)}, S_2^{(f)} = S_2^{(f_2)}\} \\
TC &= \{I^{(f)} \; oi \; I^{(f_1)}, I^{(f)} \; fi \; I^{(f_2)}\}
\end{aligned}
$$

A certain task `driveTo` requests a train t_1 to drive from its current location (s_1) to the station s_2 and finish this task no later than at time 30, i.e. arrive within 30 minutes. Additionally, driving here requires a resource of track1. Also, we model a set of causal and temporal constraints between the fluents and define symbolic constraints stating for example that the symbolic variables $?t_1$, $?t_2$, and $?t_3$ represent the same train (i.e. t_1). All constraints are added to and considered within the general constraint network of the meta-CSP and, thus, allow for joined reasoning over all available information. A plan is then only feasible iff the constraint network is consistent regarding all sources of input, e.g. it is temporally, symbolically, causally and resource consistent. Sophisticated reasoners for each domain may be used.

While this framework allows for potentially straightforward addition of all kinds of knowledge, this comes at a high computational cost. An intuitive solution is thus to employ some heuristics. For example, Stock et al. (2015) propose a number of variable ordering heuristics to potentially boost efficiency in the CSP search.

[1] As Barták, Salido, and Rossi (2010) we here assume that a plan always exists.

Hierarchical Hybrid Planning with Meta-CSPs

Hierarchical planning extends classical planning by introducing a task hierarchy. Instead of only using the notion of applicable actions, it essentially differentiates between primitive and compound tasks. Primitive tasks are hereby comparable to the normal actions in classical planning. Compound tasks describe a more abstract notion of a set of actions. This grouping can impose additional restrictions that might not be easily achievable using only preconditions and effects of actions. For example, an imposed ordering constraint can be easily encoded in a compound task and drastically improve efficiency of the planner. In fact, ordering tasks according to a partial order can be seen as the motivation behind HTN planning, the most influential subarea of hierarchical planning (Bercher, Alford, and Höller 2019).

In what follows, we briefly recall the definitions for HTN planning as defined by Bercher, Alford, and Höller (2019). The basis for HTN planning is the so called *task network*, which essentially imposes a strict partial order on a finite set of *tasks* T. The HTN *planning domain* D is defined as a tuple (F, N_P, N_C, δ, M), where

- F is a finite set of facts,
- N_P and N_C are names of primitive and compound tasks, respectively,
- $\delta : N_P \to 2^F \times 2^F \times 2^F$ maps actions to primitive task names,
- and M is a finite set of decomposition methods.

Finally, a HTN problem P is a tuple (D, s_I, tn_I), where D is the planning domain, $s_I \in 2^F$ is the initial state, and tn_I is an initial task network. A *solution* to this problem is then a final task network tn_S which is reachable from s_I by only applying methods and compound tasks. In the process, all compound tasks need to be decomposed into primitive actions, such that tn_S does not contain compound tasks anymore. The enforced task hierarchy directly restricts the set of possible solutions to only those that can be obtained by task decomposition (Bercher, Alford, and Höller 2019).

Following the formulation by Bercher et al. (2016), compound tasks may have their own set of preconditions and effects. These may be modeled using *causal links* as seen in POCL planning. Informally, a causal link is used to impose a direct ordering between two tasks by linking the effects of the former to the preconditions of the latter task, such that no other task may be allowed to be ordered between them. In a hierarchical setting, we may pass down causal links to all appropriate subtasks.

Given a meta-CSP as described above, it is now possible to encode the ordering relations imposed by the task network as causal constraints into meta-constraints. Meta-variables can then include all unplanned tasks whose predecessor tasks, indicated by the ordering constraint, have already been planned (Stock et al. 2015). Finding applicable tasks, i.e., the planning process, is then modelled as a CSP search, such as backtracking search.

Robustness in Meta-CSPs

Following the aforementioned definition of robustness, in the context of HTN planning, we are interested in a configuration that given one ore more tasks, has the ability to retain its feasibility more than any other in the case where some of the facts in the world have changed. In other words, we are interested in performing all tasks using a plan that has higher chance than any other to remain viable after changes in the environment occur. We call such a plan a *robust plan*, a plan with the maximum ability of resisting and avoiding infeasibility. Therefore, a robust plan can be seen as a *proactive measure* that limits as much as possible the need for successive repairs and replanning, and hence can play an important role in environments that are prone to perturbation and unexpected change, such as real-life configurations.

To further detail how robustness and dynamic reasoning can play a role in planning, let us reconsider the simple train network from Fig. 1. For convenience let us define all three possible routes from s_1 to s_3 as $r_1 = [s_1, s_2, s_3]$, $r_2 = [s_1, s_4, s_2, s_3]$, and $r_3 = [s_1, s_4, s_5, s_3]$. We can accomplish the task of driving a train to a specific destination with hierarchical planning using an abstract task like `driveTo(?train, ?destination)`. In case there is no direct connection, the task may be decomposed recursively to build a path from start to finish.

As mentioned before, when converting the whole planning task into a meta-CSP, the ordering of the tasks is encoded via causal constraints, where only the configurations representing the possible routes in the network are satisfiable scenarios. Following Sioutis, Long, and Janhunen (2020), we can calculate the *similarity* between any one particular solution of the meta-CSP and all other satisfiable ones. A robust scenario is then one with maximum average similarity to all other scenarios. Intuitively, a robust scenario on average shares the largest set of constraints with each other satisfiable scenario. In contrast to execution monitoring (Fritz 2009; Muise 2014), such a notion of robustness can guide the planning process actively and thus acts as a *proactive* measure, whereas the former mostly describes a *reactive* method. In this regard, the outlined proactive robustness formulation may best be compared with the measure of *relevance* guiding the search of Muise (2014).

So far, the focus in HTN planning has been mostly on finding a solution as quickly as possible, and only recent work considered length-optimal plans (Behnke, Höller, and Biundo 2019). One promising direction in this regard is the UPOM planner (Patra et al. 2020), where different utility functions can be optimized. While currently only efficiency is considered, it may also be possible to directly integrate a measure of robustness.

While it is difficult to say with certainty which path any current hierarchical planner would follow in the outlined train routing example, it seems more likely that a majority might choose the shortest option (i.e. route r_1), not because it is the length-optimal plan, but rather because its decomposition depth is also smaller than any of the alternative routes. Unless specified otherwise with some constraints, subsequent trains all follow the same route. In terms of minimizing the travel time, this might be a preferred configu-

ration. From a collision avoidance perspective, it might be favorable to split traffic onto all available routes. And solving the problem conservatively, we might select only routes via station s_4, i.e. r_2 or r_3, since following a least commitment strategy it might be possible to only commit to one available task decomposition once the train actually reaches s_4. In case such online changes are not directly permitted, a potential plan repair by restarting the planning process from this configuration should yield similar results. It is difficult to judge which option should be implemented in a real scenario. The currently most likely behavior, however, in our opinion is the least suitable in terms of robustness.

Conclusion and Future Directions

We motivated the need for robust solutions in hierarchical planning using an example of a train routing problem. Robustness has not been studied in the context of hierarchical planning before and finding a suitable definition yet alone applying it in an actual planner poses a difficult challenge. To this end, robustness might be best understood as a metric that can be favored more or less, depending on the safety restrictions imposed by the planning environment and the likelihood that certain events may disrupt normal operation. Here it can also be useful to consider predicted knowledge from experts or machine learning systems. In our example of a train network, we may get the information that one track will be subject to buckling due to a heatwave (Nguyen, Wang, and Wang 2012), an information we clearly want to take into account when routing the trains.

As a starting point, we chose a previously proposed approach to hierarchical planning by Stock et al. (2015) based on an abstract CSP representation, called meta-CSP (Mansouri and Pecora 2016). Not only the task network is modeled in this meta-CSP, but it also allows to incorporate many other types of information, such as resource, temporal, and spatial constraints. Reasoning in temporal and spatial domains is usually done using QCNs (Ligozat 2013; Dylla et al. 2017). In this context, Sioutis, Long, and Janhunen (2020) recently studied a notion of robustness, which concerns the perturbation tolerance of QCN solutions, i.e. their likelihood to resist a change in the environment. Based on similarities between the way knowledge is represented in meta-CSPs and QCNs, we discussed the potential applicability of a similar notion of robustness in the scope of hierarchical planning. This work poses as a first step towards more robust solutions in the hybrid hierarchical planning framework, and as such hopefully sparks a lively discussion on how to best define, measure, and incorporate robustness in existing applications.

References

Barták, R.; Salido, M. A.; and Rossi, F. 2010. Constraint satisfaction techniques in planning and scheduling. *J. Intell. Manuf.* 21: 5–15.

Behnke, G.; Höller, D.; and Biundo, S. 2019. Finding Optimal Solutions in HTN Planning - A SAT-based Approach. In *IJCAI*.

Bercher, P. 2021. A Closer Look at Causal Links: Complexity Results for Delete-Relaxation in Partial Order Causal Link (POCL) Planning. In *ICAPS*.

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

Cardellini, M.; Maratea, M.; Vallati, M.; Boleto, G.; and Oneto, L. 2021. In-Station Train Dispatching: A PDDL+ Planning Approach. In *ICAPS*, 450–458. AAAI Press.

Dylla, F.; Lee, J. H.; Mossakowski, T.; Schneider, T.; van Delden, A.; van de Ven, J.; and Wolter, D. 2017. A Survey of Qualitative Spatial and Temporal Calculi: Algebraic and Computational Properties. *ACM Comput. Surv.* 50: 7:1–7:39.

Fritz, C. 2009. *Monitoring the Generation and Execution of Optimal Plans*. Ph.D. thesis, University of Toronto.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Ginsberg, M. L.; Parkes, A. J.; and Roy, A. 1998. Supermodels and Robustness. In *AAAI/IAAI*.

Ligozat, G. 2013. *Qualitative Spatial and Temporal Reasoning*. ISTE Ltd and John Wiley & Sons, Inc.

Lusby, R. M.; Larsen, J.; and Bull, S. 2018. A survey on robustness in railway planning. *Eur. J. Oper. Res.* 266: 1–15.

Mansouri, M.; and Pecora, F. 2016. A robot sets a table: a case for hybrid reasoning with different types of knowledge. *J. Exp. Theor. Artif. Intell.* 28: 801–821.

Muise, C. 2014. *Exploiting Relevance to Improve Robustness and Flexibility in Plan Generation and Execution*. Ph.D. thesis, University of Toronto.

Nguyen, M. N.; Wang, X.; and Wang, C.-H. 2012. A reliability assessment of railway track buckling during an extreme heatwave. *Journal of Rail and Rapid Transit* 226: 513–517.

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

Russell, S. J.; and Norvig, P. 2020. *Artificial Intelligence - A Modern Approach, Fourth Edition*. Pearson Education.

Schattenberg, B. 2009. *Hybrid planning & scheduling*. Ph.D. thesis, University of Ulm, Germany.

Sioutis, M.; Long, Z.; and Janhunen, T. 2020. On Robustness in Qualitative Constraint Networks. In *IJCAI*.

Stock, S. 2016. *Hierarchische hybride Planung für mobile Roboter*. Ph.D. thesis, University of Osnabrück, Germany.

Stock, S.; Mansouri, M.; Pecora, F.; and Hertzberg, J. 2015. Online task merging with a hierarchical hybrid task planner for mobile service robots. In *IROS*.

Verfaillie, G.; and Jussien, N. 2005. Constraint Solving in Uncertain and Dynamic Environments: A Survey. *Constraints* 10: 253–281.